M. Teresa Higuera-Toledano
Andy J. Wellings *Editors*

Distributed, Embedded and Real-time Java Systems

Springer

Distributed, Embedded and Real-time Java Systems

M. Teresa Higuera-Toledano • Andy J. Wellings
Editors

Distributed, Embedded and Real-time Java Systems

Springer

Editors
M. Teresa Higuera-Toledano
Universidad Complutense de Madrid
Facultad de Informática, DACYA
Calle del Profesor Gaecía
Santesmas
28040 Madrid
Spain

Andy J. Wellings
Department of Computer Science
University of York
Heslington
York
United Kingdom

ISBN 978-1-4419-8157-8 e-ISBN 978-1-4419-8158-5
DOI 10.1007/978-1-4419-8158-5
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011945437

Preface

The Java language has had an enormous impact since its introduction in the last decade of the twentieth century. Its success has been particularly strong in enterprise applications, where Java is one of the preeminent technology in use. A domain where Java is still trying to gain a foothold is that of real-time and embedded systems, be they multicore or distributed. Over the last 5 years, the supporting Java-related technologies, in this increasingly important area, have matured. Some have matured more than others. The overall goal of this book is to provide insights into some of these technologies.

Audience

This book is aimed at researchers in real-time embedded systems, particularly those who wish to understand the current state of the art in using Java in this domain. Masters students will also find useful material that founds background reading for higher degrees in embedded and real-time systems.

Structure and Content

Much of the work in real-time distributed, embedded and real-time Java has focused on the Real-time Specification for Java (RTSJ) as the underlying base technology, and consequently many of the Chapters in this book address issues with, or solve problems using, this framework.

Real-time embedded system are also themselves evolving. Distribution has always been an important consideration, but single processor nodes are rapidly being replaced by multicore platforms. This has increased the emphasis on parallel programming and has had a major impact on embedded and real-time software development. The next three chapters, therefore, explore aspects of this

issue. In Chap. 1, Wellings, Dibble and Holmes discuss the impact of multicore on the RTSJ itself and motivate why Version 1.1 of the specification will have more support for multiprocessors. Central to this support is the notion of processor affinity.

RTSJ has always remained silent on issues of distributed system as other work in the Java Community has been addressing this problem. Unfortunately, this work has been moribund for several years and is now considered inactive. It is, therefore, an appropriate time to re-evaluate the progress in this area. In Chap. 2, Basanta-Val and Anderson reviews the state of the art in distributed real-time Java technology considering both the problems and the solutions.

As embedded real-time system grow in size, they will become more complex and have to cope with a mixture of hard and soft real-time systems. Scheduling these systems, within the imposed time constraints, is a challenging problem. Of particular concern is how to integrate aperiodic events into the system. The accepted theoretical work in this area revolves around the usage of execution-time servers. In Chap. 3, Masson and Midonnet consider how these servers can be programmed in the RTSJ at the application level.

Irrespective of size, embedded systems have imposed constraints, be they power consumption, heat dissipation or weight. This results in the resources in the platform being kept to a minimum. Consequently, these resources must be carefully managed. Processors and networks are the two of the main resource types, and have been considered in Chaps. 1–3. Memory is the other main resource, and it is this that is the topic of Chap. 4 through Chap. 6. Real-time garbage collection is crucial to the success of real-time Java and the advent of multicore has added new impetus to that technology to produce parallel collectors. In Chap. 4, Siebert explains the basic concepts behind parallel collectors and reviews the approaches taken by the major RTSJ implementations.

RTSJ introduced a form of region-based memory management as an alternative to the use of the heap memory. This has been one of the most controversial features of the specification. For this reason, Higuera-Toledano, Yovine, and Garbervetsky (in Chap. 5) evaluate the role of region-based memory management in the RTSJ and look at the strengths and weaknesses of the associated RTSJ scoped memory model. The third aspect of memory management for embedded systems is how to access the underlying platform's physical memory – in order to maximize performance and interact with external devices. This has always been a difficult area and one that has been substantially revised in RTSJ Version 1.1. In Chap. 6, Dibble, Hunt and Wellings considers these low-level programming activities.

Part and parcel of developing embedded systems is deciding on the demarcation between what is in software and what is in hardware. From a Java perspective, there are two important topics: hardware support for the JVM and how applications call code that has been implemented in hardware. These two issues are addressed in Chaps. 7 and 8. In Chap. 7, Schoeberl reviews the approaches that have been taken to support the execution for Java programs. This includes important areas like support for the execution of Java byte code and garbage collection. In contrast, Whitham and

Audsley in Chap. 8, focus on the interaction between Java programs and functional accelerators implemented as coprocessors or in FPGAs.

Embedded systems are ubiquitous and increasingly control vital operations. Failure of these systems have an immediate, and sometimes catastrophic, impact on society's ability to function. Proponents of Java technology envisage a growing role for Java in the development of these high-integrity (often safety-critical) systems. A subset of Java augmented with the RTSJ, called Safety Critical Java (SCJ) is currently being developed by the Java Community Process. This work is approaching fruition and two of the most important aspects are discussed in Chaps. 9 and 10. Most embedded and real-time system directly or indirectly support the notion of a mission. In the RTSJ, this notion can be implemented but there is not direct representation. SCJ provides direct support for this concept. In Chap. 9, Nielsen and Hunt discuss the mission concept and show how it can be used. Paramount to high-integrity applications is reliability. Although Java is a strongly typed language run-time exceptions can still be generated. Recent versions of the language have added support for annotations, which essentially allow added information to be passed to the compiler, development tools supporting static analysis, and the JVM itself. In Chap. 10, Tang, Plsek and Jan Vitek explore the SCJ use of annotations to support memory safety, that is the absence of run-time exceptions resulting from violations of the SCJ memory model.

Up until now, the topics that have been addressed have been concerned with the underlying technology for real-time and embedded Java. The remainder of the book focuses on higher level issues. Real-time and embedded system designers have to be more resource-aware than the average Java programmer. Inevitably, this leads to complexities in the programming task. The RTSJ has introduced abstractions that help the programmer manage resources; however, some of these abstraction can be difficult to use (for example, scoped memory). In Chap. 11, Plsek, Loiret and Malohlava argue that this can make the use of the RTSJ error prone. They consider the role of RTSJ-aware component-oriented frameworks that can help provide a clearer separation between RTSJ-related issues and application-related issues.

One well-known Java framework for component-based service-oriented architectures is the Open Services Gateway initiative (OSGi). Although this initiative has wide support from industry, it lacks any support for real-time applications. In Chap. 12, Richardson and Wellings propose the integration of OSGi with the RTSJ to provide a real-time OSGi framework.

In the final chapter of this book, Holgado-Terriza and Videz-Aivar address the issue of building a complete embedded application that includes both software and hardware components from a reusable base.

Acknowledgements

The editors are grateful to Springer who gave us the opportunity to produce this book, and to all the authors for agreeing to contribute (and for reviewing each others chapters).

We are also indebted to Sitsofe Wheeler for his help with latex.

Andy J. Wellings
M. Teresa Higuera-Toledano

Contents

Chapter 1
Supporting Multiprocessors in the Real-Time Specification for Java Version 1.1

Andy J. Wellings, Peter Dibble, and David Holmes

Abstract Version 1.1 of the Real-Time Specification for Java (RTSJ) has significantly enhanced its support for multiprocessor platforms. This chapter discusses the rationale for the approach adopted and presents details of the impact on the specification. In particular, it considers the new dispatching and processor affinity models, issues of cost enforcement and the affinity of external events.

1.1 Introduction

Multiprocessor platforms (be they multicore or multichip systems)[1] are becoming more prevalent and their role in the provision of a wide variety of real-time and embedded systems is increasing. In particular symmetric multiprocessor (SMP) systems are now often the default platform for large real-time systems rather than a single processor system.

[1]The term *processor* is used in this chapter to indicate a logical processing element that is capable of physically executing a single thread of control at any point in time. Hence, multicore platforms have multiple processors, platforms that support hyperthreading also have more than one processor. It is assumed that all processors are capable of executing the same instruction sets.

A.J. Wellings (✉)
Department of Computer Science, University of York, York, UK
e-mail: andy@cs.york.ac.uk

P. Dibble
TimeSys, Pittsburgh, PA, USA
e-mail: peter.dibble@timesys.com

D. Holmes
17 Windward Place, Jacobs Well, QLD 4208, Australia
e-mail: dholmes@ieee.org

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_1,

Since its inception, the Real-Time Specification (RTSJ) has preferred to remain silent on multiprocessor issues. It allows vendors to support multiprocessor implementations, but provides no guidelines on how the specification should be interpreted in this area. Furthermore, it provides no programming abstractions to help developers configure their systems. The latest version of the RTSJ recognizes that this position is no longer tenable. However, multiprocessor, and multicore architectures in particular, are evolving at a bewildering rate, and Operating System Standards (in particular the suite of POSIX Standards) have yet to catch up. Consequently, it is difficult to determine the level of support that the specification should target. This is compounded by the requirement to stay faithful to the guiding principles that have delimited the scope of the RTSJ. These include [65]:

- Predictable execution as the first priority in all tradeoffs.
- Write once, run anywhere but not at the expense of predictability (later rephrased to "write once carefully, run anywhere conditionally").
- Address current practice but allow future implementations to include enhanced features.
- Allow variations in implementation decisions.

This chapter is structured as follows. In Sect. 1.2, the motivations for providing more direct support for multiprocessors in the RTSJ is given along with the constraints imposed by the RTSJ guiding principles. In the majority of cases, the RTSJ run-time environment executes as middleware on top of an underlying operating system. The facilities provided by these operating systems constrain further the possible support the specification can provide. Section 1.3 reviews the support that is typical of current operating systems in widespread use. These motivations and constraints are used to generate, in Sect. 1.4, a list of multiprocessor requirements for the RTSJ Version 1.1. Then, in Sect. 1.5, the full support for multiprocessor systems in the RTSJ version 1.1 is discussed. Finally, conclusions are drawn. An appendix details the new API.

1.2 Motivations and Constraints

Whilst many applications do not need more control over the mapping of schedulable objects[2] to processors, there are occasions when such control is important. Here the focus is on symmetric multiprocessor (SMP) systems, which are defined to be identical multiprocessors that have access to shared main memory with a uniform access time. The impact of non uniform memory architectures is left for future versions of the specification, as are heterogeneous processor architectures or systems with homogeneous processor architectures with different performance characteristics.

[2]The RTSJ uses the generic term schedulable object where other languages and operating systems might use the term real-time thread or task.

1.2.1 Motivation: To Obtain Performance Benefits

The primary motivation for moving from a single processor to a multiprocessor implementation platform is to obtain an increase in performance. Dedicating one CPU to a particular schedulable object will ensure maximum execution speed for that schedulable object. Restricting a schedulable object to run on a single CPU also prevents the performance cost caused by the cache (TLB or other non-memory related context associated with the schedulable object) invalidation that occurs when a schedulable object ceases to execute on one CPU and then recommences execution on a different CPU [251]. Furthermore, configuring interrupts to be handled on a subset of the available processors allows performance-critical schedulable objects to run unhindered on other processors.

1.2.2 Motivation: To Support Temporal Isolation

Where an application consists of subsystems of mixed criticality levels (non-critical, mission-critical and safety-critical) running on the same execution platform, some form of protection between the different levels is required. The strict typing model of Java provides strong protection in the spatial domain. The Cost Enforcement Model of the RTSJ (including processing group parameters) provides a limited form of temporal protection but is often very coarse grained. More reliable temporal protection is obtainable by partitioning the set of processors and allowing schedulable objects in each criticality level to be executed in a separate partition. For example, placing all non-critical schedulable objects in a separate partition will ensure that any overruns of these less critical schedulable objects cannot impact a high-criticality subsystem.

1.2.3 Constraint: Predictability of Scheduling

The scheduling of threads on multiprocessor systems can be:

1. Global – all processors can execute all schedulable objects.
2. Fully partitioned – each schedulable object is executed only by a single processor; the set of schedulable objects is partitioned between the set of processors.
3. Clustered – each schedulable object can be executed by a subset of the processors; hence the schedulable objects set may be partitioned into groups and each group can be executed on a subset of the processors.

In addition to the above, there are hybrid schemes. For example, semi partitioned schemes fixed the majority of schedulable objects to individual processors but allow the remainder to migrate between processors (sometime at fixed points in their executions). The situation can become even more complex if the platform can vary the number of processors available for execution of the program.

Although the state of the art in schedulability analysis for multiprocessor systems continues to advance [26, 121], it is clear that for fixed priority scheduling, neither global nor fully-partitioned systems are optimal. There is no agreed approach and the well-known single processor scheduling algorithms are not optimal in a multiprocessor environment. Cluster-based scheduling seems essential for systems with a large number of processors. However, within cluster, hybrid approaches allow greater utilization to be achieved over global scheduling within cluster [121].

Hence, it is not possible for the RTSJ to provide a default priority-based multiprocessor scheduler that takes care of the allocation of schedulable objects to processors. And even if it was, it would require a very specialized real-time operating system to support it.

1.2.4 Constraint: Predictability of Resource Sharing

Given the discussion in Sect. 1.2.3, it is hardly surprising that the sharing of resources (shared objects) between schedulable objects, which potentially can be executing on different processors, is problematic. The essence of the problem is how to deal with contention for global resources; how to keep the data coherent and how to bound the time that a schedulable object waits for access to it.

Over the last 40 years, many different approaches have been explored. One approach that has maintained its popularity over the years (and which has provided the inspiration for the Java model) is the monitor. A monitor encapsulates a shared resource (usually some shared variables) and provides a procedural/functional interface to that resource.

In Java, a monitor is an object with the important property that the methods that are labeled as synchronized are executed atomically with respect to each other. This means that one synchronized method call cannot interfere with the execution of another synchronized method. The way this is achieved, in practice, is by ensuring that the calls are executed in mutual exclusion. There are several different ways of implementing this, for example, by having a lock associated with the monitor and requiring that each method acquires (sets) the lock before it can continue its execution. Locks can be *suspension-based* or *spin-based*.

From a multiprocessor real-time perspective, the following typically apply.

- Suspension-based locking – if it cannot acquire the lock, the calling schedulable object is placed in a priority-ordered queue and waits for the lock. To bound the blocking time, priority inversion avoidance algorithms are used.
- Spin-based locking – if it cannot acquire the lock, the calling schedulable object busy-waits for the lock to become available. To bound the blocking time, the schedulable object normally spins non-preemptively (i.e., at the highest priority) and is placed in a FIFO or priority-ordered queue. The lock-owner also runs non-preemptively, and consequently is prohibited from accessing nested locks and must not suspend whilst holding a lock.

Of course, optimizations are possible, such as allowing the lock to only be acquired when contention actually occurs, or initially spinning for the lock before suspending [172]. On a single processor system it is also possible to turn off all hardware interrupts whilst holding the lock, and therefore prohibit any scheduling preemptions.

Mutual exclusion is a very heavy-weight mechanism to maintain data consistency. It reduces the concurrency in the system by blocking the progress of schedulable objects, and causes the well-known problems of deadlock, starvation, priority inversion etc.

This has led to the development of non-blocking approaches. An approach is non-blocking if the failure or suspension of any schedulable objects cannot cause the failure or suspension of any other schedulable object [172]. Non-blocking algorithms are usually classified as either *lock-free* or *wait-free*. From a multiprocessor real-time perspective, the following typically apply [79].

- Lock-free – access to the shared object is immediate but, following completion of the desired operation, the schedulable object must be able to detect and resolve any contention that may have occurred. Often resolving the conflict requires retrying the operation. When contention occurs, some schedulable object is guaranteed to make progress. However, it is not defined which one. Software transactional memory is an example where a lock-free approach is possible. Bounding the retries is a major challenge for the real-time algorithm designer.
- Wait-free – access to the shared resource is immediate and is guaranteed to complete successfully within finite time. Typically, specific wait-free data structures are highly specialized and are difficult to design and prove correct. Examples include, a wait-free queue or a wait-free stack. Generic wait-free read-write access to a single shared object can be supported (e.g. using Simpson's algorithm [379] for a single reader and single writer) but at the cost of replicating the data. Hence, data read may be stale if a writer is currently updating it.

Quoting from Brandenburg et al. [79] who give the following conclusions from an in-depth study on real-time synchronization on multiprocessors:

- when implementing shared data objects, non-blocking algorithms are generally preferable for small, simple objects, and wait-free or spin-based (locking) implementations are generally preferable for large or complex objects;
- Wait-free algorithms are preferable to lock-free algorithms;
- with frequently occurring long or deeply-nested critical sections, schedulability is likely to be poor (under any scheme);
- Suspension-based locking should be avoided for global resources in partitioned systems[3];
- using current analytical techniques, suspension-based locking is never preferable (on the basis of schedulability or tardiness) to spin-based locking;

[3]This is a paraphrase from the original which focused on a particular partitioning scheme.

- if such techniques can be improved, then the use of suspension-based locking will most likely not lead to appreciably better schedulability or tardiness than spinning unless a system (in its entirety) spends at least 20% of its time in critical sections (something we find highly unlikely to be the case in practice).

Of course, the choice of what approach to adopt is constrained by the data structures needed. There is no simple general approach to taking a particular data structure and producing a non-blocking access protocol.[4] Furthermore, it should be emphasized that this study excluded external devices with long access times for which suspension-based locking is essential.

1.3 Operating System Support

Although multiprocessors are becoming prevalent, there are no agreed standards on how best to address real-time demands. For example, the RTEM operating system does not dynamically move threads between CPU. Instead it provides mechanisms whereby they can be statically allocated at link time. In contrast, QNX's Nutrino [311] distinguishes between "hard thread affinity" and "soft thread affinity". The former provides a mechanism whereby the programmer can require that a thread be constrained to execute only on a set of processors (indicated by a bit mask). With the latter, the kernel dispatches the thread to the same processor on which it last executed (in order to cut down on preemption costs). Other operating systems provide similar facilities. For example, IBM's AIX allows a kernel thread to be bound to a particular processor.[5] In addition, AIX enables the set of processors (and the amount of memory) allocated to the partition executing an application to be changed dynamically.

Many multiprocessor systems allow interrupts to be targeted at particular processors. For example, the ARM Corex A9-MPCore supports the Arm Generic Interrupt Controller.[6] This allows a target list of CPUs to be specified for each hardware interrupt. Software generated interrupts can also be sent to the list or set up to be delivered to all but the requesting CPU or only the requesting CPU. Of course, whether the Operating System allows this targeting is another matter.

Safety critical system software must be predictable enough that most failures can be detected analytically or through systematic testing. This rules out the non-determinism implicit in flexible scheduling across multiple processors. It may even rule out interactions among threads on different processors. Currently there is limited use of general multiprocessor systems in safety critical systems. Traditionally,

[4]This can be contrasted to simply taking the current access protocol and surrounding it calls to an appropriate lock to support a blocking-based access protocol.

[5]See http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.basetechref/doc/basetrf1/bindprocessor.htm.

[6]See http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0375a/Cegbfjhf.html.

where multiprocessors are required they are used in a distributed processing mode: with boards or boxes interconnected by communications busses, and bandwidth allocation, and the timing of message transfers etc carefully managed. This "hard" partitioning simplifies certification and testing since one application cannot affect another except through well-defined interfaces.

However, there is evidence that future safety-critical systems will use SMP. For example, the LynxSecure Separation Kernel (Version 5.0) has recently been announced. The following is taken from their web site[7]:

- Optimal security and safety – the only operating system to support CC EAL-7 and DO-178B level A.
- Real time – time-space partitioned real-time operating system for superior determinism and performance.
- Highly scalable – supports Symmetric MultiProcessing (SMP) and 64-bit addressing for high-end scalability.

This work has been undertaken by Intel and LynuxWorks to demonstrate the MILS (Multiple Independent Levels of Security) architecture.[8]

In the remainder of this section, we briefly review the work that has been performed by the POSIX and Linux communities. We then consider Solaris as an example of the support provided by a vendor-specific OS.

1.3.1 POSIX

POSIX.1 defines a "Scheduling Allocation Domain" as the set of processors on which an individual thread can be scheduled at any given time. POSIX states that [286]:

- "For application threads with scheduling allocation domains of size equal to one, the scheduling rules defined for SCHED_FIFO and SCHED_RR shall be used."
- "For application threads with scheduling allocation domains of size greater than one, the rules defined for SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC shall be used in an implementation-defined manner."
- "The choice of scheduling allocation domain size and the level of application control over scheduling allocation domains is implementation-defined. Conforming implementations may change the size of scheduling allocation domains and the binding of threads to scheduling allocation domains at any time."

With this approach, it is only possible to write strictly conforming applications with real-time scheduling requirements for single-processor systems. If an SMP platform is used, there is no portable way to specify a partitioning between threads and processors.

[7] http://www.lynuxworks.com/rtos/secure-rtos-kernel.php.

[8] See http://www.intel.com/technology/itj/2006/v10i3/5-communications/6-safety-critical.htm.

Additional APIs have been proposed [272] but currently these have not been standardized. The approach has been to set the initial allocation domain of a thread as part of its thread-creation attributes. The proposal is only in draft and so no decision has been taken on whether to support dynamically changing the allocation domain.

1.3.2 Linux

Since Kernel version 2.5.8, Linux has provided support for SMP systems [136]. Partitioning of user processes and threads is obtained via the notion of CPU affinity. Each process in the system can have its CPU affinity set according to a CPU affinity mask [251]. A process's CPU affinity mask determines the set of CPUs on which its thread are eligible to run.

```
#include <sched.h>

int sched_setaffinity(pid_t pid,
  unsigned int cpusetsize, cpu_set_t *mask);

int sched_getaffinity(pid_t pid,
  unsigned int cpusetsize, cpu_set_t *mask);

void CPU_CLR(int cpu, cpu_set_t *set);

int CPU_ISSET(int cpu, cpu_set_t *set);

void CPU_SET(int cpu, cpu_set_t *set);

void CPU_ZERO(cpu_set_t *set);
```

A CPU affinity mask is represented by the `cpu_set_t` structure, a "CPU set", pointed to by the mask. Four macros are provided to manipulate CPU sets. `CPU_ZERO` clears a set. `CPU_SET` and `CPU_CLR` respectively add and remove a given CPU from a set. `CPU_ISSET` tests to see if a CPU is part of the set. The first available CPU on the system corresponds to a cpu value of 0, the next CPU corresponds to a cpu value of 1, and so on. A constant `CPU_SETSIZE (1024)` specifies a value one greater than the maximum CPU number that can be stored in a CPU set.

`sched_setaffinity` sets the CPU affinity mask of the process whose ID is pid to the value specified by mask. If the process specified by pid is not currently running on one of the CPUs specified in mask, then that process is migrated to one of the CPUs specified in mask [251].

`sched_getaffinity` allows the current mask to be obtained. The affinity mask is actually a per-thread attribute that can be adjusted independently for each of the threads in a thread group.

Linux also allows certain interrupts to be targeted to specific processors (or groups of processors). This is known as SMP IRQ affinity. SMP IRQ affinity is controlled by manipulating files in the /proc/irq/ directory.

As well as being able to control the affinity of a thread from within the program, Linux allows the total set of processors (its cpuset) allocated to a process to be changed dynamically. Hence, the masks given to `sched_setaffinity()` must be moderated by the kernel to reflect those processors that have been allocated to the process. Cpusets are intended to provide management support for large systems.

The real-time scheduler of the PREEMPT_RT patchset uses cpusets to create a *root domain*. This is a subset of CPUs that does not overlap with any other subset. The scheduler adopts an active push-pull strategy for balancing real-time tasks (tasks who priority range from 0 to (MAX_RT_PRIO-1)) across CPUs. Each CPU has its own run queue. The scheduler decides [169]:

1. Where to place a task on wakeup.
2. What action to take if a lower-priority task is placed on a run queue running a task of higher priority.
3. What action to take if a low-priority task is preemted by the wake-up of a higher-priority task.
4. What action to take when a task lowers its priority and thereby causes a previously lower-priority task to have the higher priority.

Essentially, a *push* operation is initiated in cases 2 and 3 above. The push algorithm considers all the run queues within its root domain to find one that has a lower priority task (than the task being pushed) at the head of its run queue. The task to be pushed then preempts the lower priority task.

A *pull* operation is performed for case 4 above. Whenever the scheduler is about to choose a task from its run queue that is lower in priority than the previous one, it checks to see whether it can pull tasks of higher priority from other run queues. If it can, at least one higher priority tasks is moved to its run queue and is chosen as the task to be executed. Only real-time tasks are affected by the push and pull operations.

1.3.3 Solaris

The Solaris operating system provides facilities similar to those of Linux, though the details of the API's and the low-level semantics differ. The `processor_bind` system call can be used to bind one or more threads (or Lightweight Processes – LWPs – as they are known on Solaris) to a particular processor.

```
#include <sys/types.h>
#include <sys/processor.h>
#include <sys/procset.h>

int processor_bind(idtype_t idtype, id_t id,
                   processorid_t processorid,
                   processorid_t *obind);
```

Solaris allows threads to be grouped according to process,"task", "project", "process contract" and zone, as indicated by the `idtype_t`, which in turn dictates what kind of `id_t` is expected. The same system call can be used to query existing bindings, or to clear them, by using the special `processorid` values of `PBIND_QUERY` and `PBIND_NONE` respectively. The `obind` parameter is used to return the previous binding, or the result of a query.

The binding of threads to multiple processors is done through the Solaris processor set facility. Processor sets are similar to Linux's CPU sets, but rather than being hierarchical they form a flat, disjoint, grouping. Threads will not run on processors in a processor set unless the thread, or its process, are explicitly bound to that processor set. Programs can be run on a specific processor set using the *psrset* $-$ *e* $<$ *set*# $><$ *program* $>$ command-line utility. When a process is bound to a processor set in this way, its threads are restricted to the processors in that set – the `processor_bind` system call can then be used to further restrict a thread to a single processor within that. An unbound process can only run on an unbound processor, so there must always be at least one unbound processor in a system. The `psrset` command is an administrative tool for creating and configuring processor sets, as well as a means to execute programs on a particular set – there is a corresponding programmatic API for this as well. The `pset_bind` system call can then be used within a program to bind one or more threads to a different processor set to that of the process. For example, you might give all NHRTs a dedicated processor set to run on.

```
#include <sys/pset.h>

int pset_bind(psetid_t pset, idtype_t idtype,
              id_t id, psetid_t *opset);
```

A `psetid_t` is a simple integer that names a particular processor set. The special values `PS_QUERY` and `PS_NONE` indicate a request to query, or clear, the current processor set binding, respectively. The `opset` parameter is used to return the previous binding, or the result of a query.

Processors are managed by the `psradm` utility (or the equivalent programmatic API) which allows for processor to be taken offline (they won't execute any threads, but might field I/O device interrupts), or to be declared `no-intr` (they will execute threads but won't field any I/O device interrupts). Some interrupts will always be handled by a specific processor, such as per-cpu timer expirations.

Effective processor set management requires extremely detailed knowledge of the underlying hardware and how the operating system identifies individual "processors". On multi-core systems, for example, you need to understand how cores are numbered within a CPU and/or within a chip, and what resources (e.g. L2 cache) might be shared between which cores.

1.4 Requirements for RTSJ Version 1.1

Section 1.2 has summarized the motivations for providing more explicit support for multiprocessor systems in the RTSJ, along with the constraints that must be imposed to maintain predictable scheduling. From this, the following requirements are distilled.

- It shall be possible to limit the dispatching of a schedulable object to a single processor – this is in order to support the motivations given in Sects. 1.2.1 and 1.2.2 and to reflect the constraints given for schedulability in Sect. 1.2.3.

  This is impossible to achieve in Version 1.02 of the specification. Although, the `java.lang.Runtime` class allows the number of processors available to the Java Virtual Machine (JVM) to be determined by the `availableProcessors()` method, it does not allow Java threads to be pinned to processors.
- Global scheduling of schedulable objects across more than one processor shall be possible – this is primarily to reflect the constraints given for schedulability in Sect. 1.2.3, and also because this is the default position for RTSJ version 1.0.2 implementations.
- Spin-based queue locking protocols for shared objects shall not be precluded – this is to reflect the predictability constraints given in Sect. 1.2.4.

  The Java intrinsic synchronization approach is built on suspension-based locking. The RTSJ, with its support for priority ceiling emulation, potentially allows a no-lock implementation of Java monitors on a single processor but only for constrained synchronized methods and statements (i.e. ones that do not self suspend). Implementing non-locking Java monitors is difficult to support in general. The RTSJ wait-free queue facility provides an alternative mechanism for schedulable objects to communicate.
- The impact of the Java-level handling of external events should be constrainable to a subset of the available processors – this is to reflect the motivations given in Sect. 1.2.1.

  In the RTSJ all external events are associated with *happenings*. Whilst it may not be possible to constrain where the first-level interrupt code executes, it should be possible to control where the Java code executes that responds to the interrupt.

1.5 The RTSJ Version 1.1 Model

The support that the RTSJ provides for multiprocessor systems is primarily constrained by the support it can expect from the underlying operating system. The following have had the most impact on the level of support that has been specified.

- The notion of processor *affinity* is common across operating systems and has become the accepted way to specify the constraints on which processor a thread can execute.

  RTSJ 1.1 directly supports affinities. A *processor affinity set* is a set of processors that can be associated with a Java thread or RTSJ schedulable object. The internal representation of a set of processors in an `AffinitySet` instance is not specified, but the representation that is used to communicate with this class is a `BitSet` where each bit corresponds to a logical processor ID. The relationship between logical and physical processors is implementation defined.

  In some sense, processor affinities can be viewed as additional release or scheduling parameters. However, to add them to the parameter classes requires the support to be distributed throughout the specification with a proliferation of new constructor methods. To avoid this, support is grouped together within the `ProcessorAffinitySet` class. The class also allows the addition of processor affinity support to Java threads without modifying the thread object's visible API.
- The range of processors on which global scheduling is possible is dictated by the operating system. For SMP architectures, global scheduling across all the processors in the system is typically supported. However, an application and an operator can constrain threads and processes to execute only within a subset of the processors. As the number of processors increase, the scalability of global scheduling is called into question. Hence, for NUMA architectures some partitioning of the processors is likely to performed by the OS. Hence, global scheduling across all processors will not be possible in these systems.

  The RTSJ supports an array of *predefined affinity sets*. These are implementation-defined. They can be used either to reflect the scheduling arrangement of the underlying OS or they can be used by the system designer to impose defaults for, say, Java threads, non-heap real-time schedulable objects etc. A program is only allowed to dynamically create new affinity sets with cardinality of one. This restriction reflects the concern that not all operating systems will support multiprocessor affinity sets.
- Many OSs give system operators command-level dynamic control over the set of processors allocated to a processes. Consequently, the real-time JVM has no control over whether processors are dynamically added or removed from its OS process.

  Predictability is a prime concern of the RTSJ. Clearly, dynamic changes to the allocated processors will have a dramatic, and possibly catastrophic, effect on the ability of the program to meet timing requirements. Hence, the RTSJ assumes that the processor set allocated to the RTSJ process does not change during its execution. If a system is capable of such manipulation it should not exercise it on RTSJ processes.

In order to make the RTSJ 1.1 fully defined for multiprocessor systems, the following issues needed to be addressed.

1. The dispatching model – version 1.02 of the specification has a conceptual model which assumed a single run queue per priority level.
2. The processor allocation model – version 1.02 of the specification provides no mechanisms to support processor affinity.
3. The synchronization model – version 1.02 of the specification does not provide any discussion of the intended use of its monitor control policies in multiprocessor systems.
4. The cost enforcement model – version 1.02 of the specification does not consider the fact that processing groups can contain scheduling objects which might be simultaneously executing.
5. The affinity of external events (happenings) – version 1.02 of the specification provides no mechanism to tie happenings handlers to particular processors.

The remainder of this section considers each of the above issues in turn. An appendix gives the associated APIs.

1.5.1 The Dispatching Model

The RTSJ dispatching model specifies its dispatching rules for the default priority scheduler. Here, the rules are modified to address multiprocessor concerns.

1. A schedulable object can become a running schedulable object only if it is ready and the execution resources required by that schedulable object are available.
2. Processors are allocated to schedulable objects based on each schedulable object's active priority and their associated affinity sets.
3. Schedulable object dispatching is the process by which one ready schedulable object is selected for execution on a processor. This selection is done at certain points during the execution of a schedulable object called *schedulable object dispatching points*. A schedulable object reaches a *schedulable object dispatching point* whenever it becomes blocked, when it terminates, or when a higher priority schedulable object becomes ready for execution on its processor.
4. The schedulable object dispatching policy is specified in terms of *ready queues* and schedulable object states. The ready queues are purely conceptual; there is no requirement that such lists physically exist in an implementation.
5. A ready queue is an ordered list of ready schedulable objects. The first position in a queue is called the head of the queue, and the last position is called the tail of the queue. A schedulable object is ready if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of schedulable objects of that priority that are ready for execution on that processor, but are not running on any processor; that is, those schedulable objects that are ready, are not running on any processor, and can be executed using that processor. A schedulable object can be on the ready queues of more than one processor.

6. Each processor has one running schedulable object, which is the schedulable object currently being executed by that processor. Whenever a schedulable object running on a processor reaches a schedulable object dispatching point, a new schedulable object is selected to run on that processor. The schedulable object selected is the one at the head of the highest priority nonempty ready queue for that processor; this schedulable object is then removed from all ready queues to which it belongs.

In a multiprocessor system, a schedulable object can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready schedulable objects, the contents of their ready queues are identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains.

1.5.2 The Processor Allocation Model

The requirement is for a minimum interface that allows the pinning of a schedulable object to one or more processors. The challenge is to define the API so that it allows a range of operating system facilities to be supported. The minimum functionality is for the operating system to allow the real-time JVM to determine how many processors are available for the execution of the Java application.

The number of processors that the real-time JVM is aware of is represented by a `BitSet` that is returned by the static method `getAvailableProcessors()` in the `ProcessorAffinitySet` class. For example, in a 64 processor system, the real-time JVM may be aware of all 64 or only a subset of those. Of these processors, the real-time JVM will know which processors have been allocated to it (either logical processors or physical processors depending on the Operating System). Each of the available processors is set to one in the bit set. Hence, the cardinality of the bit set represents the number of processors that the real-time JVM thinks are currently available to it. The length of the bitset is the total number of processors the JM is aware of.

An affinity set is a set of processors which can be associated with a thread (be it a Java thread or a real-time thread), an asynchronous event handler or processing group parameters instance. For instances of thread and asynchronous event handlers, the associated affinity set binds the thread or asynchronous event handler to the set of processors.

The processor membership of an affinity set is immutable. The processor affinity of instances of `Thread` (including real-time threads and no-heap threads) and bound asynchronous event handlers can be changed by static methods in `ProcessorAffinitySet` class. The change in affinity has an impact on the scheduling of the thread no later than the next dispatching point.

The processor affinity of un-bound asynchronous event handlers is fixed to a default value, as returned by the `getHeapSoDefault` and `getNoHeapSoDefault` methods.

An affinity set factory only generates usable `AffinitySet` instances; i.e., affinity sets that can be used with `set(ProcessorAffinitySet, BoundAsyncEventHandler)`, `set(ProcessorAffinitySet, Thread)`, and `set(ProcessorAffinitySet, ProcessingGroupParameters)`. The factory cannot create an affinity set with more than one processor member, but such affinity sets are supported. They may be internally created by the RTSJ runtime, probably at startup time. The set of affinity sets created at startup is visible through the `getPredefinedSets(ProcessorAffinitySet[])` method.

The affinity set factory may be used to create affinity sets with a single processor member at any time, though this operation only supports processor members that are valid as the processor affinity for a thread (at the time of the affinity set's creation.)

In many cases, the initial affinity set of an entity is determined by that of its creator, but there are times when this is not possible:

- A Java thread created by a schedulable object will have the affinity set returned by `getJavaThreadDefaultAffinity`.
- A schedulable object created by a Java thread will have the affinity set returned by `getHeapSoDefaultAffinity`, or `getNoHeapSoDefaultAffinity`, depending on whether or not it uses the heap.
- An unbound asynchronous event handler will have the affinity set returned by `getHeapSoDefaultAffinity`, or `getNoHeapSoDefaultAffinity`, depending on whether or not it uses the heap.

Every Java thread, schedulable object and processing group parameters instance is associated with a processor affinity set instance, either explicitly assigned, inherited, or defaulted.

An appendix summarizes the `ProcessorAffinitySet` class API.

1.5.3 The Synchronization Model

RTSJ Version 1.1 does not prescribe a particular implementation approach for supporting communication between threads/schedulable objects running on separate processors. As mentioned in Sect. 1.2.4, the state of the art is still immature in this area and, consequently, it would be premature to attempt to provide standardized support. However, it is clearly an important area. In this section, we, therefore, summarize the available options.

1.5.3.1 Using the RTSJ Monitor Control Policies

Both priority inheritance and priority ceiling emulation (PCE) protocols are appropriate for SMP systems, although the algorithm used for setting of ceiling priority values may change [312].

To obtain the best schedulability, the approach advocated in the literature, when applied to Java, requires that synchronized methods and statements (that can be called from threads/schedulable objects running on more than one processor) do not suspend holding the lock; furthermore, nested synchronized calls are prohibited. If these constraints hold, then a form of priority ceiling emulation can be used. The ceiling of the shared object is set to be the highest in the system, so that the code effectively executes non-preemptively. The access protocol requires a calling thread/schedulable object to non-preemptively spin on a FIFO (or priority-ordered) queue when the lock is unavailable. Where locks are only accessed locally (if some form of partitioning is used), the normal RTSJ rules for setting ceiling apply, and priority inheritance can also be used. However, if the programmer fails to classify the shared objects correctly, the blocking that occurs when a thread tries to access a already-locked object may become unbounded.

For nested locks, the possibility of deadlock is reintroduced in multiprocessor systems, even with priority ceiling emulation. Consequently, the programmer must adopt a strategy for preventing it from occurring. One such approach is to use the Flexible Multiprocessor Locking Protocol (FMLP) [60]. Applying the FMLP approach to an RTSJ program requires the programmer to:

- Partition the shared objects into those with *short* and *long* access methods – short accesses should not self suspend for any reason, while long accesses may self suspend; if any method in an object self suspends then the object is a long-access object;
- Short-access objects are accessed via queue-based spin-locks as described above;
- Long-access objects are accessed via suspension-based priority inheritance locks (the default RTSJ policy);
- Nested accesses between shared objects are grouped together into *shared resources* – no short-access shared object is allowed to access a long-access shared object;
- For each resource, a single lock is defined (at the application level) – this lock must be acquired by the application before any of the resource objects are accessed; for short resources this is a spin-based lock, for long resources this is a priority inheritance lock.

With the above approach, the resource lock is used to guarantee the deadlock freedom property (it is a deadlock-prevention mechanism). Note that blocking only occurs on access to the resource lock. The separation of the resources between those that are short and long allows fast access for the common simple case.

Note also that short shared object accesses should not perform any operations that might cause it to block waiting for the garbage collector.

1.5.3.2 Using the RTSJ Wait-Free Queues

The RTSJ goes to great length to support no-heap real-time threads and to ensure that garbage collection delays cannot impact on time critical code. Part of this support package is the provision of no-wait read and write queues. It is important to stress that this motivation is slightly different from the usual motivation for "wait-free" queues given in the real-time literature (see Sect. 1.2.4), where the concern is mainly for concurrent access.

In the RTSJ, the `WaitFreeReadQueue` class is intended for single-reader multiple-writer communication. Only the read access is wait-free. If an application wants to have multiple readers, it has to supply its own synchronization protocol between the readers to ensure only one of them attempts access at any point in time. Writers are not wait-free. A reader is generally an instance of `NoHeapRealtimeThread`, and the writers are generally regular Java threads or heap-using schedulable objects. Communication is through a bounded buffer of objects that is managed first-in-first-out. The `write` method appends a new element onto the queue. It is synchronized, and blocks when the queue is full. It may be called by more than one writer, in which case, the different callers will write to different elements of the queue. The read method removes the oldest element from the queue. It is not synchronized and does not block; it will return null when the queue is empty.

A corresponding wait-free write queue that is non-blocking for producers is also provided. The `WaitFreeWriteQueue` class is intended for single-writer multiple-reader communication. A writer is generally an instance of `NoHeapRealtimeThread`, and the readers are generally regular Java threads or heap-using schedulable objects.

On a multiprocessor systems, although the write end of a read queue (and the read end of a write queue) are synchronized, they are short (using the definition given above). However, they block when the queue is full/empty.

In summary, the RTSJ wait-free queues are appropriate for their intended use in a multiprocessor environment. However, they are not intended to solve the general problem of non-blocking communication.

1.5.3.3 Using the `java.util.concurrent.atomic` Package

The `java.utils.concurrent.atomic` package[9] provides a set of classes that support lock-free, thread-safe programming on single variables such as integers, booleans, longs and object references. Atomic variable are also, by definition, volatile and have the same memory semantic [172].

[9] http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/atomic/package-summary.html.

The specifications of the methods in the related classes enable implementations to employ efficient machine-level atomic instructions (where available, such as compare and swap instructions).

The JSR 282 expert group anticipates that, as more experience is accrued with multiprocessor real-time Java implementations, real-time concurrency utilities (using `java.util.concurrent.atomic` package) will become available. For example, Java versions of real-time wait-free algorithms such as Simpson's algorithm [379].

1.5.4 The Cost Enforcement Model

The RTSJ manages the CPU time allocated to a schedulable object via two complementary mechanisms. The first is via the `cost` parameter component of the `ReleaseParameter` class, which essentially represents the worst-case execution time of a schedulable object. In version 1.02 of the RTSJ there was an optional cost enforcement model that, if supported, required a schedulable object to consume no more than this value for each of its releases. In version 1.1, the model has been extended to also provide a softer approach where the application is simply informed if a cost overrun has occurred. It is also now mandatory to provide mechanisms to measure the execution-time of a schedulable object (albeit, within some granularity appropriate for the underlying platform).

In an SMP systems (assuming the processors execute at approximately the same speed), these approaches easily scale from the single processor model (because a schedulable object can only be executing on one processor at a time). Clearly, in a heterogenous architecture where processors may differ in speed, setting an appropriate value for the execution time is problematic. This issue is beyond the scope of this chapter.

The second, more problematic, mechanism is that provided by `ProcessingGroupParameters`,[10] which allows a group of schedulable objects to be allocated a utilization. Whilst conceptually, this could be applied to multiple threads from the same group executing in parallel with global scheduling, it is not clear that the general model should be applied to a partitioned system. Indeed, the implementation challenge of detecting budget overrun may make the whole approach untenable. Instead, it is necessary to place a constraint that all schedulable objects allocated to a processing group must be allocated to the same processor.

The approach adopted in version 1.1 of the RTSJ is that real-time threads and bound asynchronous event handlers that have processing group parameters are members of that processing group, and their processor affinity is governed by the intersection of the processing group's affinity and the schedulable object's

[10] Support for processing groups is an optional facility in the RTSJ.

affinity. The affinity set of a processing group must have exactly one processor. The intersection of a non-default PG affinity set with the schedulable object's affinity set must contain at most one entry. If the intersection is empty, the affinity is set to the default set.

The manipulations of affinities for processing groups is provided by static methods in the `ProcessorAffinitySet` class. This class also controls the default affinity used when a processing group is created. That value is the set of all available processors. (Which permits each member of the processing group to use the affinity set it would use if it were in no processing group.) The processor affinity of the processing group can subsequently be altered with the `set(ProcessorAffinitySet,ProcessingGroupParameters)` method.

1.5.5 Affinity and Happenings

In version 1.1 of the RTSJ, the model for interfacing with the external environment has been significantly enhanced (see Chap. 6). In particular, the RTSJ notion of *happenings* has been extended so they can be handled at various levels.

From an affinity perspective, the following definitions and points apply:

- A first-level interrupt handler is the code that the platform executes in response to the hardware interrupts. A first level interrupt is assumed to be executed at an execution eligibility (priority) and has an affinity both dictated by the underlying platform (which may be controllable at the platform level).
- On a stand-alone JVM, a handler is also needed. This is the code that the JVM executes as a result of being notified by the first-level interrupt handler that the interrupt is targeted at the RTSJ application. When the JVM is hosted on an Operating System, the equivalent level is the JVM code that executes in response to a signal being received. This code can now be associated with a schedulable object and can have application-defined affinity and execution eligibility.
- If appropriate, the Java-level handler can find one or more associated asynchronous events and fire them. This then releases the associated RTSJ asynchronous event handlers. These can have application-defined affinity and execution eligibility.

1.6 Conclusions

As the complexity of real-time and embedded systems continues to grow, there is little doubt that they will in future rely on multiprocessor/multicore platforms. As with all resources, it is necessary to allow the real-time programmer to have the ultimate control over their use. Hence, facilities must be made available at the

operating system level and then via language-level APIs. The RTSJ tries not to stand in the way of multiprocessor implementations. However up until now, it did not facilitate control over the resources in a portable way. This chapter has considered how to address these concerns and has defined the semantics of Version 1.1 of the RTSJ.

Acknowledgements The authors gratefully acknowledge the contributions of the other members of the JSR 282 Expert Group especially Ben Brosgol, James Hunt and Kelvin Nilsen. We particulary thank Kelvin Nilsen for his comments on a early draft of this chapter.

The first author gratefully acknowledges the discussions that he has had with Alan Burns, Ted Baker and Sanjoy Barauh on multiprocessor scheduling issues.

This work has been supported in part by EU JEOPARD project 216682, Sun Microsystems, and Timesys.

Appendix: The APIs

In this appendix the `ProcessorAffinitySet` is abridged. Full details of the possible exception conditions are not given.

```
package javax.realtime;
public class ProcessorAffinitySet {

  public static ProcessorAffinitySet generate(java.util.BitSet bitSet)
    // Returns an Affinity set with the affinity BitSet bitSet and
    // no associations.

  public static ProcessorAffinitySet get(
                BoundAsyncEventHandler handler)
    // Return the affinity set instance associated with handler.

  public static ProcessorAffinitySet get(
                ProcessingGroupParameters pgp)
    // Return the affinity set instance associated with pgp.

  public static ProcessorAffinitySet get(java.lang.Thread thread)
    // Return the affinity set instance associated with thread.

  public static java.util.BitSet getAvailableProcessors()
    // This method is equivalent to getAvailableProcessors(BitSet)
    // with a null argument.

  public static java.util.BitSet getAvailableProcessors(
                                 java.util.BitSet dest)
    // In systems where the set of processors available to a
    // process is dynamic (e.g., because of system management
    // operations or because of fault tolerance capabilities),
    // the set of available processors shall reflect the processors
    // that are allocated to the RTSJ runtime and are currently
    // available to execute tasks.
```

```
public static int  getCurrentProcessor()
  // Return the processor presently executing this code.

public static ProcessorAffinitySet getHeapSoDefault()
  // Return the default processor affinity set for heap-using
  // schedulable objects.

public static ProcessorAffinitySet getJavaThreadDefault()
  // Return the default processor affinity for Java threads.

public static ProcessorAffinitySet getNoHeapSoDefault()
  // Return the default processor affinity for non-heap using
  // schedulable objects.

public static ProcessorAffinitySet getPGroupDefault()
  // Return the processor affinity set used for SOs where the
  // intersection of their affinity set and their processing
  // group parameters' affinity set yields the empty set.

public static int  getPredefinedSetCount()
  // Return the minimum array size required to store references
  // to all the pre-defined processor affinity sets.

public static ProcessorAffinitySet[]   getPredefinedSets()
  // Equivalent to invoking getPredefinedAffinitySets(null).

public static ProcessorAffinitySet[] getPredefinedSets(
                                     ProcessorAffinitySet[] dest)
  // Return an array containing all affinity sets that were
  // pre-defined by the Java runtime.

public java.util.BitSet getProcessors()
  // Return a BitSet representing the processor affinity set
  // for this AffinitySet. Equivalent to getProcessors(null).

public java.util.BitSet getProcessors(java.util.BitSet dest)
  // Return a BitSet representing the processor affinity set
  // of this AffinitySet.

public boolean isProcessorInSet(int processorNumber)
  // Ask whether a processor is included in this affinity set.

public static boolean isSetAffinitySupported()
  // Return true if the set(ProcessorAffinitySet, *) family
  // of methods is supported.

public static void set(ProcessorAffinitySet set,
                       BoundAsyncEventHandler aeh)
  // Set the processor affinity BoundAsyncEventHandler aeh to set.

public static void set(ProcessorAffinitySet set,
                       ProcessingGroupParameters pgp)
  // Set the processor affinity of
```

```
    // ProcessingGroupParameters pgp to set.

  public static void set(ProcessorAffinitySet set,
                         java.lang.Thread thread)
    // Set the processor affinity of a java.lang.Thread
    // thread or RealtimeThread to set.
}
```

Chapter 2
Using Real-Time Java in Distributed Systems: Problems and Solutions

Pablo Basanta-Val and Jonathan Stephen Anderson

Abstract Many real-time systems are distributed, i.e. they use a network to interconnect different nodes. The current RTSJ (*Real-time Specification for Java*) specification is insufficient to address this challenge; it requires distributed facilities to control the network and maintain end-to-end real-time performance. To date, the real-time Java community has produced the DRTSJ (*The Distributed Real-Time Specification for Java*), though a wide variety of other distributed real-time Java efforts exist. The goal of this chapter is to analyze these efforts, looking for the problems they addressed and their solutions, and defining (when possible) their mutual relationships.

2.1 Introduction

The next generation real-time systems are likely to be cyber-physical [73, 246], meaning that they are more distributed and embedded than before and have a coupled interaction with physical systems. From the point of view of real-time Java, it is not sufficient to rely solely on centralized timeliness specifications like the RTSJ (*Real-time Specification for Java*) [65]. Programmers require solutions that allow interconnection of these isolated virtual machines to offer real-time support on a myriad of networks and applications. The kinds of networks that should be addressed include, among others, automotive networking, industrial networks, aviation busses, and the Internet. Each system has different domain challenges.

P. Basanta-Val (✉)
DREQUIEM Lab. Universidad Carlos III de Madrid, Avda. de la Universidad 30
Leganés-Madrid, 28911, Spain
e-mail: pbasanta@it.uc3m.es

J.S. Anderson
Virginia Tech., ECE Department, Blacksburg, VA 24061, USA
e-mail: andersoj@anderson.org

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_2,

The evolution of centralized and distributed real-time Java specifications has been conducted at different rates. Whereas there are commercial implementations for centralized systems based on the RTSJ, distributed systems still lack this kind of support.

On the one hand, centralized systems have an important infrastructure; they can use to develop their real-time Java applications using a common specification: the RTSJ (Real-Time Specification for Java). Designers can build their systems using Oracle's Java RTS implementation [64, 399], IBM's WebSphere RT [213], Aicas' Jamaica [372] or Apogee's Aphelion [16]. Each implementation extends the normative model in one or more ways. For instance, many of the aforementioned commercial implementations support real-time garbage collection, one feature that it is not mandatory in the RTSJ; many have plug-ins for common IDEs. There are also experimental platforms that include JTime [410] (the RT-JVM implemented as the reference implementation for the RTSJ), and free source-code implementations like JRate [108, 111], OVM [153], and Flex [327].

On the other hand, distributed real-time application practitioners suffer from a lack of normative implementations and specifications on which they may develop their systems. Most efforts toward a distributed real-time specification address one of two different technologies: Java's RMI (*Remote Method Invocation*) [398] and OMG's RT-CORBA (*Real-Time Common Object Request Broker Architecture*) [344]. Neither process is complete (i.e. no specification and implementation that developers may use have emerged).

For RMI, a natural choice for pure Java systems, the primary work has been drafted as a specification under the JSR-50 umbrella. The name given to the specification is the DRTSJ (*Distributed Real-Time Specification for Java*) [10, 225]. Unfortunately, the work is unfinished, inactive, and requires significant effort to produce implementations similar in quality and performance to those available in the RTSJ. In a wider context, some partial implementation efforts have been carried out in some initiatives (e.g. RT-RMI-York [72], RT-RMI-UPM [408], and DREQUIEMI-UC3M [43]) to test different architectural aspects; however, current implementations cannot be used in their current form in commercial applications.

RT-CORBA [344] was mapped to RTSJ [13, 219, 220] to take advantage of the real-time performance offered by the real-time virtual machine. The mapping is far from trivial because many features of RTSJ have to be combined with the real-time facilities of CORBA which offer similar support. The main implementation supporting RTSJ and RT-CORBA mapping [139, 314], RTZen, has produced a partial implementation which cannot be used in its current form to develop real-time applications, and requires additional implementation effort to produce a commercial product.

Apart from these two control-flow efforts there are other options for distributed real-time Java based on emerging standards like DDS [292] (*The Data Distribution Service*). For instance, the integration of distributed Java with the real-time distribution paradigm is being lively investigated in the context of the iLAND

project [24, 166, 168]. Many practical distributed real-time systems implemented with RTSJ include DDS products like RTI's DDS [336] or PrismTech's Open Splice [308].

The rest of this chapter explores previous work in differing depth depending on importance and the amount of publically accessible information. Before entering into the discussion of the main work carried out in distributed real-time Java, the chapter analyzes different Java technologies (Sect. 2.2) that may be used to support Java distribution. After that, it focuses the attention on DRTSJ and other real-time RMI (RT-RMI) approaches (Sect. 2.3). Then, extra support for distributed real-time Java that ranges from the use of specific hardware to the use of formal methods and multi-core infrastructure are discussed (Sect. 2.4). The chapter continues with a set of technologies that augment distributed real-time with additional abstractions and new services (Sect. 2.5). The chapter ends with conclusions and future work (Sect. 2.6).

2.2 Approaches to Distributed Real-Time Java

The section explores different alternatives for distributed real-time Java from a technological perspective identifying a set of candidate technologies. The leading effort, DRTSJ, is included as part of the discussion. For each category, the authors analyze advantages and inconveniences stemmed from the use of each technology.

Although there are many distributed abstractions and categories, from the point of view of real-time systems, they may be classified at least in the following types [226, 429, 430]:

- *Control-flow*. This category includes systems that use method invocation in simple request-response fashion. Java's RMI (*Remote Method Invocation*), remote invocations or any RPC (*Remote Procedure Call*) belong to this category. The basic idea of them is to move both application data and the execution point of the application among remote entities.
- *Data-flow*. This category includes the publish/subscribe model that is supported in technologies like the OMG's DDS [292]. The goal of DDS is the movement of data without an execution point among application entities following publisher/subscriber patterns which communicate through shared topics.
- *Networked*. Comprises systems that exchange asynchronous or synchronous movement of messages, without a clear execution point among their entities. IPC and message passing mechanisms included in many operating systems fall in this category. Java's JMS (*Java's Message System*) [395] could be included in the list too.

The list of distributed models for Java is more extensive, including: (1) *mobile objects*, (2) *autonomous agents*, (3) *tuple-spaces*, and (4) *web-services* (see [61]).

A comprehensive distributed real-time Java need not provide all of them; any (networked, data-flow, or control-flow) is powerful enough to develop a straightforward technology.

The impact of the three models on Java is quite different and it is in an uneven state. Control flow solutions, like DRTSJ and other RT-RMI technology, are the most addressed approaches because they were integrated previously in other technologies like RT-CORBA.

Data flow approaches are another alternative for building a real-time technology (with technologies like DDS). However, despite significant practical use, they have not been addressed by the real-time Java community in depth. Rather, the community has focused mainly on control-flow abstractions.

The use of networked solutions provides flexibility but requires cooperation from higher-level abstractions which should instead deal with end-to-end timeliness. Nevertheless, networked solutions are one of the most used approaches with several specific proprietary frameworks.

Other Java's technologies like Jini [146], JavaSpaces [401], and Voyager [319] offer augmented distribution models that enhance current infrastructures but they are silent about real-time issues. In many cases, these technologies require a predictable end-to-end communication model on which to build their enhanced models. For instance, Jini offers distributed service management (with three stages `discovery`, `join`, and `leasing`) in which communications rely on RMI. JavaSpaces offers an abstraction similar to object oriented databases (without persistence) with `read`, `write`, and `take` primitives which have to be accessed from remote nodes. Voyager supports mobile code and agents that may migrate in the platform at discretion among several virtual machines that communicate agents.

In the three cases described, the challenge is to characterize the augmented abstraction from the point-of-view of a real-time infrastructure. For instance, in one possible RT-Jini [167] the services could publish their real-time characterization which could be used in the discovery process by clients that require access to remote services. However, other alternative RT-Jini approaches may offer bounded discovery processes and enhanced composition.

Among these, three distributed technologies are of special interest for distributed real-time Java: RT-CORBA, RMI and DDS. The rest of this section analyses pros and cons of using each one of these three technologies when they are used as basic support for distributed real-time Java.

2.2.1 Main Control-Flow Approaches to Distributed Real-Time Java

Assuming a control flow abstraction, an implementation for distributed real-time Java may chose among two primary technologies: RT-CORBA and RMI.

2.2.1.1 RT-CORBA and RTSJ

The RT-CORBA standard enhances CORBA for real-time applications. The specification is divided into two parts: RT-CORBA 1.0 for static systems and RT-CORBA 2.0 for dynamic environments. One way to provide distributed real-time Java is to map these two models to RTSJ, as has been done previously for many other programming languages. For instance, a similar mapping has been carried out before for Ada and C/C++.

Defining a mapping from RT-CORBA to RTSJ is not as simple as it seems at a first glance. One may think that it is a simple process but it requires some kind of reconciliation between RT-CORBA and RTSJ in order to match and map abstractions for real-time parameterization, scheduling parameters and underlying entities.

For instance, using RT-CORBA with RTSJ, the programmer has two alternatives to control concurrency: one given by the RT-CORBA's mutex and another by using the `synchronized` statement and monitors included in RTSJ.

Choosing one or another alternative is equivalent to select one of the following target infrastructures:

1. RTSJ with a remote invocation facility given by RT-CORBA.
2. RT-CORBA enhanced with the predictability model of RTSJ.

The first choice corresponds to using the `synchronized` statement and the second to use a RT-CORBA's mutex (and probably discard the `synchronized` statement of Java).

Both approaches have implications from the point-of-view of the programming abstraction and architecture. The first choice distorts the abstract programming model of RT-CORBA, which is trimmed for a particular purpose. The second alternative is more conservative and maintains RT-CORBA interoperability among different programming languages and platforms, partially removing some support given by Java and the RTSJ.

In the second case, the price paid for it is high because many good properties of RTSJ have to be discarded. One example of this is the dual-thread model based on `RealTimeThreads` and `NoHeapRealtimeThreads` [65]. RT-CORBA only defines one kind of thread whereas RTSJ has two threads: real-time and non-heap threads.

The RTZen project [139] followed the second choice improving the internals of the ORB using RTSJ facilities. The project has generated also specific tools and programming patterns that may be used to build real-time systems. Among the main contributions carried out to distributed real-time Java are:

- A CORBA architecture that is compatible with the RTSJ [242, 314]. This architecture removes the garbage collector from the internals of the middleware using regions inside its internal architecture. However, it still maintains the GC's interference in the endpoints of the communication.

- Programming patterns [313] which are used in the implementation of a middleware and in other general programming abstractions.
- Tools for configuring applications [174]. They allow designers to select a subset from the whole middleware and reduce the application footprint.
- A component model (i.e., COMPARES [174, 175, 208]) for distributed real-time Java which relies on the RT-CORBA's resource model. It is based on the CCM (CORBA's Component Model) [285].

2.2.1.2 RMI and RTSJ

RMI [398] is the main distributed object abstraction for Java. It offers remote invocation and other services like connection management, distributed garbage collection, distributed class downloading and a naming service. Some of these services, like naming and connection management, are also available in the CORBA world; while others (class downloading and distributed garbage collection) have a clear Java orientation. This extra characterization reduces the development time in comparison to other alternatives.

The Java orientation of these services brings in advanced features like system deployment, automated update and avoids remote object memory leaks in distributed real-time systems. However, all these services are sources of unbounded interference and their interaction with the real-time Java virtual machine and runtime libraries must be better specified in order to avoid unwanted or unpredictable delay. This indeterminism has induced many researchers to avoid or forbid their use in specific environments like high-integrity applications.

Another interesting feature of RMI is the lack of a real-time remote invocation model that may produce a seamless integration with RTSJ. The remote objects may be characterized by using the model provided by RTSJ which includes scheduling, release, and processing-group parameters. From the point of view of a base technology for distributed real-time Java, a RMI-and-RTSJ tandem should be more synergic than a RTSJ-and-RT-CORBA solution.

It should be noted that the RMI-and-RTSJ in tandem approach is close to the first choice in the integration of RTSJ-and-RT-CORBA. In both cases the communication technology (i.e., RMI and RT-CORBA) is modified in order to be adapted to the model proposed by RTSJ. However, even in this case, RMI is closer to Java than RT-CORBA because RMI extends the Java's object model with genuine services like distributed garbage collection and class downloading, which are not available from CORBA.

2.2.2 A Data-Flow Approach: DDS and RTSJ

Alternatively, distributed real-time Java may be implemented with DDS-and-RTSJ in tandem, defining a distributed real-time technology based on these two technologies.

DDS offers another alternative to build real-time systems under the publish-subscribe paradigm with certain guarantees in message transmission and reception using *topics* and providing a quality-of-service model for messaging. However, it is unclear how to provide end-to-end Java abstractions on top of a publisher-subscriber abstraction efficiently. Such systems today require substantial developer intervention and explicit context management in application code to provide end-to-end abstractions and mechanisms for resource management and timeliness assurance.

One solution is to build simple request-response abstractions on DDS, similar to those available in RMI, or use isolated communication channels – as proposed in ServiceDDS [128] and the iLAND project [168, 214] – to replicate a synchronous communication. However, neither ServiceDDS nor the iLAND project has defined integration of RTSJ as its main goal. Both consider RTSJ to be a suitable programming language for use in their developments; however, the integration between RTSJ and DDS is not its primary goal.

Another solution is to provide optimized end-to-end models to run under the real-time publisher-subscriber paradigm. The authors consider this an open challenge in the distributed real-time Java context.

2.3 Distributed Real-Time Java

This section considers the main approaches defined for distributed real-time Java. It first describes the leading approach, DRTSJ, and then concentrates on other RT-RMI frameworks and approaches that are related to DRTSJ.

2.3.1 DRTSJ: The Distributed Real-Time Specification for Java

The leading effort is DRTSJ, its main goal is to enhance RTSJ with trans-node real-time end-to-end support, using as distribution mechanism RMI. In its initial position papers [10, 225, 226, 429, 430], DRTSJ is defined to have two goals:

- The primary aim is to offer end-to-end timeliness by guaranteeing that an application's trans-node has its timeliness properties explicitly employed during resource management. Provide for and express propagation, resource acquisition/contention and storage and failure management in the programming abstraction.
- The second goal is to enhance the programming model with control flow facilities that model real-time operations composed of local activities. The activity-control actions include: *suspend*, *abort*, *changes propagation*, *failure propagation*, *consistency mechanisms*, and *event notification*. The result of this activity is distributable real-time threads.

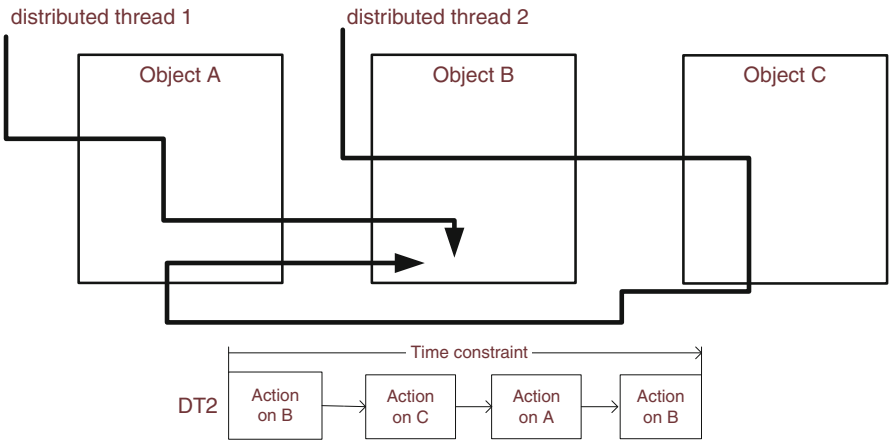


Fig. 2.1 Distributable threads used as an end-to-end flow abstraction

Among the general guiding principles of DRTSJ are:

- Bring distributed real-time to Java.
- Distributed objects and operations.
- Maintain the flavor of RTSJ.
- Not dictate the use of an RTSJVM for every node, nor any specific real-time transport.
- Provide coherent support for end-to-end application properties.

2.3.1.1 Distributable Real-Time Threads

The foundational abstraction for DRTSJ is the distributable thread, which is a single logical thread with a globally unique identity that extends and retracts through local and remote objects (see Fig. 2.1). Distributable threads extend and retract performing remote invocations (RPC's) to remote objects via RMI, and maintain an effective distributed stack.

It is noteworthy that distributable threads may have end-to-end time constraints, typically deadlines. However, other policies may be enforced in each segment through specific schedulers. Distributable threads are expected to carry their timeliness contexts (e.g., deadlines, priorities, or more complex parameters) with them as they extend and retract through the distributed system in the same manner as local threads.

Non-trivial distributed systems must consider themselves to be continually in a state of (suspected or real) partial failure; therefore fault detection, consistency, and

recovery policies must be provided. DRTSJ anticipates providing these recovery policies, but also anticipates application-specific policies which provide suitable engineering trades between consistency, performance, and timeliness.

Specification

The DRTSJ's expert group has drafted an internal specification for DRTSJ. It deals with the changes required in the current RTSJ infrastructure, new scheduling policies, new programming entities and asynchronous communications. Although describing the whole model is out of the scope of this chapter, the authors highlight several low-level details that clarify some key issues:

- It requires changes to RTSJ classes and some modifications to Java in order to support distributable threads, specifically marking some RTSJ classes as `Serializable`.
- It does not include any particular new scheduler. Its default behavior is based on the priority based scheduler currently included in RTSJ.
- It defines `DistributableThread` as its main abstraction.
- It is silent regarding use of the memory model offered by RTSJ, specifically scoped or immortal memory.
- In the past, it provided distributed `AsyncEventHandlers` as a lightweight mechanism to transfer failures. This mechanism is not currently considered defined as a part of the specification.

Software Suite

The JSR-50 expert group also described a software suite that consists of a modified RTSJ virtual machine with support for:

- *Distributable threads* with support for real-time performance.
- *Thread integrity mechanisms* like orphan detection and elimination policies which detect partial failures.
- *An optional meta-scheduler* component which allows the use of arbitrary scheduling policies.

Specific Problems and Solutions

Some of the core members of the JSR-50 Expert Group carried out a series of contributions to distributed real-time Java which are focused on the definition of scheduling algorithms for distributable threads:

- *Implementation of Distributable Threads via local proxy threads*, solving the ABA deadlock problem by servicing all local distributable thread segments using a single thread with re-entrant state management.
- *Scheduling algorithms for distributed real-time Java based on distributable threads*. Their contributions to this topic [10,11,27,27,316–318] use distributable threads as the main programming abstraction. The list of algorithms developed includes DUA-CLA [11], and ACUA [160]. DUA-CLA is a consensus-driven utility accrual scheduling algorithm for distributed threads, which detects [317] system failures and proposes recovery mechanisms for distributable threads [115, 318]. ACUA [160] is a distributed scheduling algorithm designed under a partially synchronous model; it allows for probabilistically-described message delays.
- *Distributable Thread Integrity policies*, sometimes called Thread Maintenance and Repair (TMAR). Building on precursor work from the Alpha and Mach projects, a number of timely thread failure and consistency management protocols have been proposed [173, 317].

2.3.1.2 Different Integration Levels

The position paper for DRTSJ was refined in [429] defining three integration levels (L0, L1 and L2). Each level requires some support from the underlying system and offers some benefit to the programmer.

- **L0** is the minimal integration level. The level considers the use of RMI without changes; this is its main advantage. In L0, applications cannot define any parameters for the server-side and a predictable end-to-end remote invocation is not feasible. The remote invocation should be used during initialization or non time-constrained phases of the distributed real-time application.
- **L1** is the first 'real-time' level. L1 requires mechanisms to bind the transmission and reception of messages and some kind of real-time remote invocation. To achieve this goal, DRTSJ would extend the remote object model with real-time characterization, and modified development tools. In L1, the remote invocation to a remote method creates objects that may be invoked from remote clients offering a predictable real-time remote invocation. For L1, additional changes are required in the serialization of some classes to be transferred through the network. As a result of this design, client and server do not require a shared clock (i.e., using clock synchronization) and failures in the server are propagated to the client as a result of the remote invocation (i.e. synchronously).
- **L2** offers a transactional model for RMI. The transactional model is based an entity called distributable real-time threads; these entities transfer information from one node into another. L2 requires clock synchronization among RMI nodes and changes in class serialization in some Java classes. The model also offers asynchronous events that provide an asynchronous mechanism and distributed asynchronous transfer-of-control. Using this support the server may notify changes to clients asynchronously.

2.3.2 Architectures for RT-RMI

Another set of works have targeted RT-RMI from a different angle producing a real-time version for RMI similar to RT-CORBA without having distributable threads as the main entities. This section introduces three frameworks: DREQUIEMI, a previous framework designed at the University of York, and a third work carried out at Universidad Politcnica de Madrid.

2.3.2.1 DREQUIEMI

Researchers at Universidad Carlos III de Madrid worked on extensions for distributed real-time Java [30, 43]. The resulting framework is named DREQUIEMI. DREQUIEMI integrates common-off-the-shelf techniques and patterns in its core architecture. The major influences to the model are RMI, the RTSJ, RT-CORBA, and some time-triggered principles. The reference architecture designed could be used to deploy distributed real-time threads on it and offers support for DRTSJ's L1. It also incorporates some elements taken from L2.

Its goal is to detect and analyze sources of unpredictability in a distributed Java architecture based on RTSJ and RMI. In DREQUIEMI many of the explored issues deal with a triple perspective: the impact on the programmer, the infrastructure and the run-time benefit stemmed from its adoption.

The current architecture has five layers (see Fig. 2.2):

- *Resources*. Collectively, they define the foundations for the architecture which includes the list of resources that participate in the model. The model takes from real-time Java two key resources: memory and processor and from RMI the network resource.
- *Infrastructure*. These three resources are typically accessed through interfaces (via an RTOS and a RT-JVM) shaping an infrastructure layer. In DREQUIEMI, the access to the infrastructure is given through a set of primitives. It is important to highlight that to be compliant with the model, the current RTSJ requires the inclusion of classes for the network. Fortunately, these classes are already available in standard Java (and in other packages), so that they may be added to a new network profile.
- *Distribution*. On top of this layer, the programmer may use a set of common structures useful for building distributed applications. There is one new type of element corresponding to each key resource, namely: *connection pool*, *memory-area pool*, and *thread pool*.
- Common services. There are four services:
  - A stub/skeleton service,
  - a DGC (Distributed Garbage Collection) service,
  - a naming service, and
  - a synch/event service.

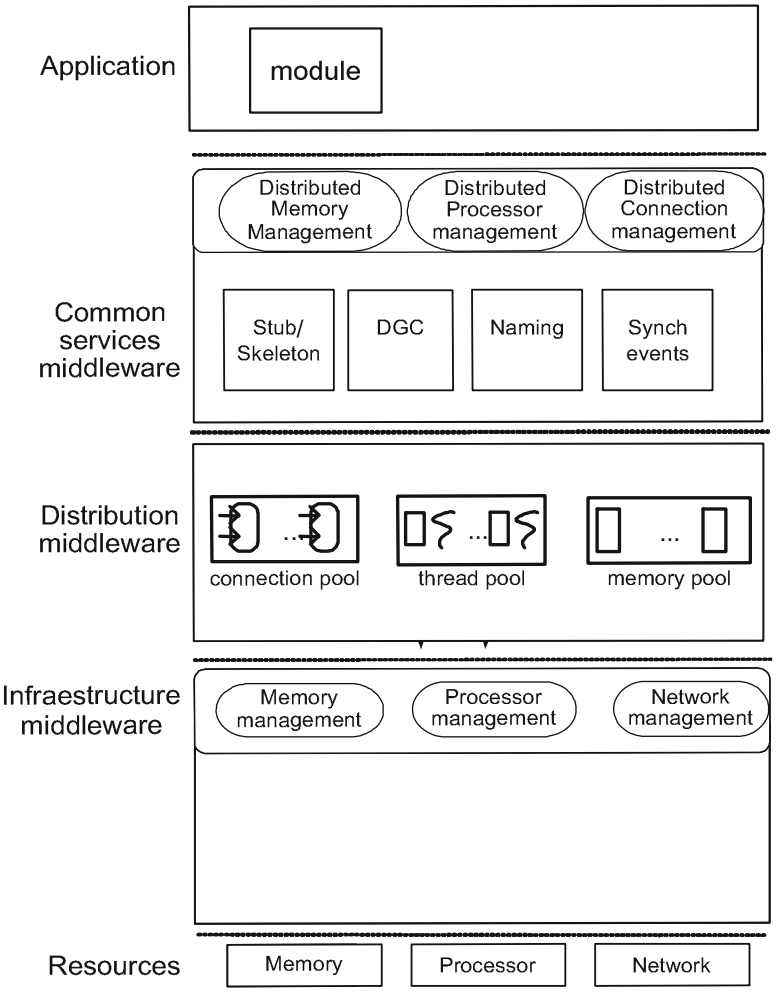


Fig. 2.2 DREQUIEMI's architecture for real-time distributed systems

The first allows the carrying out of a remote invocation while maintaining a certain degree of predictability.

The DGC service eliminates unreferenced remote objects in a predictable way; that is, introducing a limited interference on other tasks of the system.

The naming service offers a local white page service to the programmers, enabling the use of user-friendly names for remote objects.

The synch/event service is a novel service (not included currently in RMI) for data-flow communications.

- *Application*. Lastly, the specific parts of the application functionality are found at the uppermost layer, drawn as modules. The modules are based on components,

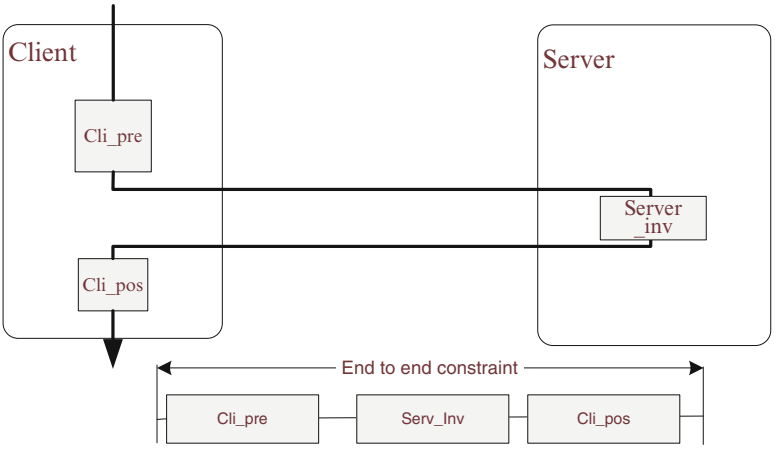


Fig. 2.3 End-to-end constraints in DREQUIEMI

promote reuse of pieces of other applications (see [44]) and may be supported by other augmented technologies.

The architecture defines two management levels:

- One centralized manager which handles memory, CPU and network resources at a centralized level (i.e. the real-time virtual machine).
- Another distributed manager which handles how these resources are assigned to distributed applications.

Scheduling and End-to-End Performance

From the point of view of end-to-end programming abstractions, in DREQUIEMI, there is no first class main entity such as DRTSJ's distributable threads. Instead, a real-time thread with certain deadline that invokes a remote method in a remote object establishes an end-to-end requirement for the application which comprises three stages (see Fig. 2.3):

1. The local client-side pre invocation (*Cli_pre*).
2. The server-side invocation (*Server_inv*).
3. The client-side post invocation (*Cli_pos*).

The characterization is enough to use well-known end-to-end scheduling models with precedence rules. Currently, the types of techniques addressed are based on end-to-end response time analysis for fixed priority-based scheduling (e.g., Direct Synchronization (DS), Release Guards (RG) and Sporadic Servers (SS) [181, 391, 392, 411]).

This model is also flexible enough to support other abstractions like the distributable threads described in the previous section on DRTSJ. However, its inclusion in DREQUIEMI requires extensions to the architecture to support new services similar to those defined for DRTSJ.

Specific Problems and Solutions

Using the DREQUIEMI framework, the following general distributed real-time Java issues have been addressed:

- *No-heap remote objects*. On this architecture [36, 39, 49] the authors designed a model, which is called no-heap remote object, to leave out the penalty of the garbage collector from the end-to-end path of the remote invocation. The model requires changes on the internals of the middleware and certain programming constraints called the no-heap remote object paradigm. From the point of view of distributed real-time Java, the no-heap remote object is useful to offer end-to-end real-time communications that do not suffer garbage collector interactions.
- *Non-functional information patterns*. Another problem addressed in DREQUIEMI is how to transfer non-functional information in distributed real-time Java. The authors identified different alternatives, evaluated their complexity and proposed a simple framework for priority-based systems [35, 45].
- *Asynchronous remote invocations*. The authors also analyzed the remote invocation model and proposed an asynchronous remote invocation which can be used to offer efficient signaling [42]. This mechanism avoids blocking at the client and requires changes in the transport protocol of RMI, namely JRMP.
- *Time-triggered integration*. Some real-time systems – like TTA (the Time Triggered Architecture) – use time-triggered communications to avoid network collisions and offer predictable communications. The authors developed a time triggered approach compatible with the unicast nature of RMI in [32] and [31]. By using it, programmers may increase the predictability of their communications.
- Protocol optimizations. The communication protocol of RMI, namely JRMP, has been optimized to offer efficient multiplexing facilities that reduce the number of open connections necessary to carry out a remote invocation [47, 48].

As a result of implementing RT-RMI with RTSJ, DREQUIEMI has revealed three optimizations for the RTSJ, namely: `RealTimeThread++` [34,41], `Extended-Portal` [40], and `AGCMemory` [37, 38]. The first extension generalizes the concurrency model of RTSJ proposing a unique type of thread that may be used by all applications; it also includes enhanced synchronization protocols (*Memory Ceiling Protocol* (MCP) and *Memory Inheritance Protocol* (MIP)). `Extended-Portal` offers an API mechanism able to violate the assignment-rule of RTSJ in a safe way, thus reducing the complexity of having to implement RT-RMI. Lastly, `AGCMemory` enhances the region model of RTSJ with a new type of region that enables partial object recycling.

2.3.2.2 A RT-RMI Framework for the RTSJ

Some members of the real-time systems group of York University are involved in the DRTSJ expert group and they carried out relevant work on how to develop a real-time RMI framework for DRTSJ's L1. In [69] and [72] they addressed the definition of a real-time RMI framework for RTSJ. The goal in this work is to build a model for the real-time remote invocation for RMI using the support given by RTSJ. The authors also defined a set of open issues related to the support for RMI.

For the remote invocation issue, the authors defined solutions for the client, the server and the network. The client-side remains almost unchanged while the server suffers several modifications. The authors proposed new classes to handle incoming remote invocations. At the server, the extra support added allowed defining scheduling parameters and a thread-pool which were attached to each remote object. The implementation considered assumes a TCP/IP network with a RSVP management infrastructure.

The framework supports propagation of parameters between client and server. The authors defined three policies to generate client-side parameters. These parameters are transferred to the server in each remote invocation. The set of parameters considered includes scheduling parameters and the relationship with the garbage collector.

The authors also considered memory issues regarding the implementation. At the client-side the implementation is easier than at the server, which requires the use of different types of memory.

Specific Problems and Solutions

The list of open issues described by York includes important issues related to certain L1 features and others stemmed from L2 requirements:

- *Class downloading*. RMI can download classes transparently from other virtual machines. However, this support introduces an important interference on the application that may have to wait for one of these classes to conclude its downloading. This is an important cost that should be removed from the remote invocation or at least be under application control.
- *Registry*. RMI has a naming mechanism that helps find other objects available in the system. However, the indirection given by this service is also another source of indeterminism and its integration in a real-time platform should be clarified.
- *Distributed garbage collector.* RMI extends the garbage collection support from a local virtual machine to multiple machines which recycle remote objects using a distributed garbage collector algorithm. However, the integration of this mechanism in a distributed architecture should be clarified in order to reduce its interference in real-time applications.
- *Object serialization*. RMI uses a general purpose object serialization model to marshal and unmarshal remote object parameters. From the computational point

overview, the mechanism is complex and its overhead for the remote invocation could be quite high.
- *Asynchronously interrupted exceptions* (i.e., AIEs). The support defined for L2 requires that one thread may raise an asynchronously interrupted exception in another. However, this action consumes network, memory, and CPU resources that have to be modeled and integrated in the core of RMI.
- *Asynchronous event handlers*. L2 allows a thread to raise an event that is handled in another node, requiring extra resources to carry out such an action. As in the previous case (AIEs) the platform should model and integrate the interaction of this interference in the distributed model.

2.3.2.3 The UPM Framework for RT-RMI

Another relevant contribution has been done by the Universidad Politecnica de Madrid (UPM) [408]. UPM worked in profile definition for RT-RMI and also addressed certain support issues related to improving the predictability and efficiency of RMI for real-time systems.

The authors considered that not all distributed real-time applications fit in the same distributed real-time Java profile; different applications may require different RT-RMI profiles.

Three different environments are considered:

- *Safety critical systems*. This type of system refers to systems where deadline misses can cost human lives or cause fatal errors. In this type of system, the behavior has to be correct and highly deterministic.
- *Business-critical systems*. This kind of profile refers to soft real-time systems whose anomalous behavior may have financial costs. In this kind of systems efficiency and robustness are extra-functional requirements.
- *Flexible business-critical systems*. The last profile is similar to the previous one but with a great deal of flexibility (e.g. future ambient intelligent systems and mission critical systems with survivability).

Four features define each profile:

- *Computational model*. The computational model refers to the abstraction used to design programs. In all cases, the abstraction chosen is a linear model described with an end-to-end predictable behavior. This end-to-end representation is used by the three models adjusting their computational features.
- *Adaptation required on RMI*. Each profile defines specific classes and hierarchies of objects for specific domains which have to be supported inside the core of the middleware.
- *Concurrency model in the remote invocation*. This feature refers to the rules that govern the handling of the remote invocation at both client and server.
- *Memory considerations*. Finally, each profile has different memory requirements and can use certain types of memory during its execution.

The first profile (RMI-HRT) is based on preemptive priority-driven scheduling and requires the use of priority ceiling protocols. It also considers two execution phases: initialization and mission; the first initializes the system and the second has real-time constraints. The profile defines the use of immortal memory for initialization and scoped memory in mission. An additional constraint introduced by the model in RMI is that clients cannot share the same reference to a remote object and dynamic allocation of objects is not allowed. The profile forbids the use of some dynamic mechanisms of RMI, removing among others: class downloading, firewalls, distributed garbage collection and security. RMI requires specific classes to handle real-time remote invocations and the interface compiler should be extended. The concurrency model defines handler threads at the server and a blocking thread at the client. The use of memory is constrained to regions – the garbage collector is disabled – and only one nesting level is allowed.

The second profile (RMI-QoS) has also end-to-end constraints. However, it does not have to rely on a worst-case response time analysis as RMI-HRT does. RMI-QoS accepts certain deadline misses if they fit in the quality-of-service pattern. The profile relies on the full-blown RTSJ and targets at soft-real-time applications. Its computational model has to support local and global negotiations. The computational model is also more complex than the previous; it includes: server initialization, reference phase, negotiation, and data transactions. Another key difference is that clients may share references to remote objects. The support given by the virtual machine has been extended with negotiations and specific classes to code remote objects. At the server side, the concurrency model has three threads: listener, acceptor and handlers. The memory restrictions placed on this model are fewer than in RMI-HRT.

The third profile considers a support based on OSGi with bundles that manage certain resources. Some features of RTSJ like the asynchronous interrupted exception are not considered in the framework. The profile may use real-time garbage collection to reduce the development cost of its applications.

Specific Problems and Solutions

A specific contribution has been done on the serialization currently available for RMI [407]. The authors distinguish two cases: one that transfers primitive types and another for references to Java objects. In the case of objects, the authors detected the problem of having acyclic structures which may require a complex analysis when they are used in a hard real-time system. The new mechanisms proposed have an impact on the serialization, which is extended with additional classes.

Many of the previous results described in the profiles have their origin in a previous article that defines solutions to make RMI real-time [124]. In this work the author based his solution on reservation protocols currently used in the Internet. The author also identified some indeterminism sources and proposed solutions for the client and server-side. As a result of his approach, the resulting RT-RMI architecture is more predictable.

2.3.3 Real-Time Component Models for RT-RMI

The list of RT-RMI related initiatives finishes with a component model designed at the Texas A&M University. As a part of a high mobility hard real-time component [321, 322] they proposed an approach for a real-time remote invocation server-centric service. The framework uses the total bandwidth server (TBS) algorithm to guarantee the allocation of CPU to tasks performed at the server side. The authors focus their analysis on CPU and are silent on network and memory management issues.

The impact of this work on distributed real-time Java is high. DRTSJ and RT-RMI may use the technique described, or another similar, to control and limit the CPU consumed at the server side.

2.4 Supplements for DRTJava

There are some supplements that enhance DRTJava with general support intended to offer enhanced support or simply better performance. The list included in this section refers to three functionalities: ability to use embedded and hardware platforms, the use of specific networks, and the use of formal models. They can be used in combination with many of the techniques described previously. For each of them the section explores key contributions carried out in each work and their specific contribution to distributed real-time Java.

2.4.1 Embedded Distributed Real-Time Java

Some pieces of work considered the issue of producing solutions for restricted performance and tight memory resources by using low-level communication facilities (e.g., [376, 377, 404]). Their work complements DRTJava with an embedded orientation and specific protocol communications.

One of these hardware optimized platforms is described in [405]. The platform is built on FemtoJava which was expanded with a RTSJ-based support. In this framework different Java nodes are connected using APICOM. This API provides communication via a network interface by using several services that establish connections, exchange messages, establish a logical local address, and broadcast messages. The model described in APICOM supports two main communication paradigms: point-to-point and publisher-subscriber.

Another alternative analyzed by the authors is the use of time-triggered models in their framework which could run on CAN [96] networks or the IEEE 802.154 wireless standard. These two communication facilities were integrated in hardware providing an interface to communicate Java nodes more efficiently.

2.4.2 Real-Time Networking

Implicitly, DRTSJ requires some predictability from the network: the messages exchanged among different nodes have to be time-bounded in order to offer bounded end-to-end response-time. Several authors have carried out different pieces of work that may contribute to this goal.

In order to enhance DRTSJ's communications there are two main approaches: local-area network integration and Internet communication facilities. The first includes integration of techniques such as time-triggered protocols (like the TTA [238]) and the second improves the predictability in packet transmission in open IP-based networks.

For the Internet approach, some pieces of work [46] addressed the definition of a technique to mark packets with a priority which is enforced in special routers using tools designed for Linux, once it is properly configured for hard real-time systems.

Another mechanism that has been analyzed in distributed real-time Java is RSVP, the resource reservation protocol of the Internet. In [124] the author defined a set of changes in RMI to use this transport protocol as part of his solutions for distributed real-time Java.

2.4.3 Multi-core Support for Distributed Real-Time Java

Many current infrastructures are multi-core systems with several processors that interconnect one each other using shared memory or have some kind of special bus that interconnect multiple cores. As a result of this change, current techniques defined for centralized real-time Java are being extended to profit from a multi-core infrastructure (see [284] or [426]). The changes have effects on current centralized infrastructures and scheduling algorithms that have to be redefined in order to consider multiple processors in a single chip and different core hierarchies.

From the perspective of distributed real-time Java two issues should be highlighted:

- The first is related to the scheduling techniques that both infrastructures share. To some extent, both models consider multiple nodes with precedence constraints on the activation of several tasks. The main difference is the possibility of having or not having shared memory. Systems like DRTSJ do not consider nodes that may have shared memory while traditional multi-core systems do have shared memory.
- The second important issue is that very probably future DRTJava implementations will run on multi-core infrastructures. So that, the internal scheduling algorithms that support DRTSJ should be designed with a multi-core infrastructure in mind. For instance, current end-to-end techniques defined for distributed DRTJava are silent on how to integrate a multi-core infrastructure as part of their infrastructures.

2.4.4 Models Based on CSP Formalism

As an alternative to the programming model defined by the DRTSJ, applications may use CSP (*Communicating Sequential Processes*). CSP is a formal description language for describing concurrent systems that can be used to describe distributed real-time Java systems written in CSP. Some researchers have produced an alternative model for distributed real-time Java entirely based on this formalism [204]. CSP supports sequential, parallel and alternative constructs that communicate using channels. The framework allows the use of priorities attached to processes so that users may use classical scheduling algorithms to assign priorities to concurrent entities. In this model, special channels transfer data among nodes using arbitrary communication frameworks like TCP/IP and CORBA.

DRTJava may profit from two characteristics of CSP: it may be modeling its behavior using CSP [206] and also may be modifying its communication model in order to include this abstraction within its core. In the first case, the choice enables the use of verification tools typically associated with CSP. In the second, the programming model is augmented with additional communication mechanisms which interconnect different threads and provide an additional programming abstraction not included currently in DRTSJ.

2.5 Augmented Technologies: Benefits and Approaches

In relationship to the approaches defined in the previous sections, there are other real-time Java technologies that can be part of cyber-physical infrastructures [44]. The current list of candidates includes names such as RT-OSGi, RT-EJBs, and RT-Jini. Each technology complements and augments distributed real-time Java in a different aspect, offering different enhanced facilities to RT-RMI. In addition, RT-RMI may be a requirement in the development of these technologies. For each technology, the rest of this section outlines its goals and the role played by distributed real-time Java in meeting them.

2.5.1 RT-OSGi

The first candidate is RT-OSGi [180] (real-time open services gateway initiative). OSGi offers life cycle management to deploy, start, stop and undeploy bundles automatically at runtime. These bundles may have interactions with local and remote entities. The real-time prefix should control local resources by assuming an underlying RTSJ framework and using DRTSJ in remote communications as required. In addition, the bundles have standard services and may define system dependencies.

Some specific contributions to the definition of this technology include: a hybrid model (C and Java) [180] which declares a real-time component model; and several platform optimizations to provide centralized isolation [325] and admission control [324]. So far, the integration of distributed real-time support in RT-OSGi has been addressed only partially in some preliminary works [33].

Readers interested in the integration of the RTSJ and OSGi are referred to Chap. 12 of this book, which considers in greater depth this issue.

2.5.2 RT-EJBs

In Java, Enterprise Java Beans (EJBs) [393] define a component model for distributed Java applications with persistence, naming and remote communication services. Internally, EJBs may use RMI to carry out a remote invocation. Unfortunately, the model lacks a real-time (i.e., RT-EJBs) specification that provide a quality-of-service framework and enhanced services, similar to those proposed for the CORBA's component model (CCM) [285].

Some interesting steps have been given in terms of characterizing a container model for RTSJ services [414] and other RTSJ-based component models (e.g., the one described in [302]). In both approaches, the container runs on a real-time Java virtual machine that offers real-time predictability. In addition to the centralized support, the container may introduce a distribution service based on distributed real-time Java technology.

Readers interested in component-oriented development for real-time Java are referred to Chap. 11 of this book, which considers in greater depth this issue.

2.5.3 RT-Jini

Jini offers the possibility of registering, finding and downloading services in a centralized service. When services are distributed, the application may use remote invocations to communicate remote virtual machines. A real-time version (RT-Jini) could improve the basic support in several ways including [167]: quality-of-service parameterization of services, predictable execution (which could be built upon RTSJ and DRTSJ) and predictable lookup and composition. Some initial steps in service characterization [167] and composition [151,152] have been given in CoSeRT [167].

In addition, some previous work on real-time tuples that use real-time Java [66] may help to refine the RT-Jini's model.

2.6 Conclusions

This chapter has covered the current state of distributed real-time Java, addressing different technologies and giving the status of each the main. It has also focused on DRTSJ, the leading effort towards a distributed real-time Java technology. It also introduces other related efforts that may be considered as alternative approaches, such as RT-CORBA alternatives or predictable infrastructures based on RT-RMI and the use of a DDS-and-RTSJ in tandem. For all of them, the chapter outlines primary issues addressed highlighting general contributions to distributed real-time Java.

For the support required to implement distributed real-time Java, the chapter addressed the use of hardware infrastructures, real-time networking, multi-core infrastructures and formal models that help validate applications. All these techniques may enhance distributed real-time Java in different ways.

Lastly, the chapter considered the use of distributed real-time Java as some kind of support to other upcoming real-time technologies. The list of outlined technologies includes RT-Jini, RT-EJBs and RT-OSGi.

A final note should be written regarding the next step to be given in distributed real-time Java. To date, there is no publicly-available distributed real-time Java reference implementation nor a specification. The next efforts should be driven in this direction, i.e. producing a public reference implementation for distributed real-time Java. To achieve this goal, much of the work described in this chapter (e.g, the distributable threads and the architectures for RT-RMI) may be helpful. This support is also crucial for other augmented Java technologies that require end-to-end predictable communications as part of their specification.

Acknowledgements This work was supported in part by the iLAND Project (100026) of Call 1 of EU ARTEMIS JU and also in part by ARTISTDesign NoE (IST-2007-214373) of the EU 7th FP programme. We also thank Marisol García-Valls (mvalls@it.uc3m.es) for her comments on a previous draft of this chapter.

Chapter 3
Handling Non-Periodic Events in Real-Time Java Systems

Damien Masson and Serge Midonnet

Abstract Most real-time systems consist of a mixture of hard and soft real-time components. Hard real-time tasks are typically periodic, whereas soft real-time tasks are usually non-periodic. The goal of real-time scheduling is to minimize of the response times of soft tasks while guaranteeing the periodic tasks' deadlines. This chapter presents the mechanisms provided by the Real-Time Specification for Java to help program this mix of hard and soft components. An approach where support is provided at the application level (user-land) is proposed and evaluated. APIs that unify the various approaches and permit their use by a non specialist are defined.

3.1 Introduction

Real-time system theory has historically focused on fully periodic task systems. This came from the need to have a known activation model to ensure the respect of timing constraints. Modern applications, however, cannot be uniquely composed of fully periodic tasks. Some parts of the work, like user feedback, control, or switches to another operating mode are quintessentially aperiodic. There are two main approaches for dealing with such events. The choice of a particular approach will depends both on the nature of the non-periodic event being handled, and on system design issues. In the first approach, if it is possible to bound the maximal

D. Masson (✉)
Université Paris-Est, LIGM UMR CNRS 8049, ESIEE Paris, Cité Descartes BP 99, 93162 Noisy-le-Grand Cedex, France
e-mail: d.masson@esiee.fr

S. Midonnet
Université Paris-Est, LIGM UMR CNRS 8049, Université Paris-Est Marne-la-vallée, Cité Descartes, Bât Copernic, 5 bd Descartes, Champs sur Marne 77454 Marne-la-Vallée Cedex 2, France
e-mail: serge.midonnet@univ-paris-est.fr

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_3,

arrival frequency of an event, or if some releases of this event can be dropped, the designer can choose to serve them just as regular periodic tasks. In this case, we assume that in the worst case, the event will arrive with a maximum frequency, also called minimal inter-arrival time. The traffic will then be called *sporadic*. In the second approach, when using the maximal frequency to find the worst case period brings too much pessimism, or when it is not appropriate from a design point of view, there is no solution other than to consider that this kind of work cannot have strict timing constraints, since the task activation model is not known. Then the real-time scheduler's role will consist of trying to process this traffic as fast as possible. Indeed, the system performance can be increased by the fast treatment of these tasks, whereas the response times of periodic tasks is of no importance provided that their deadlines are respected. This second approach requires the joint scheduling of hard periodic tasks with soft aperiodic events. The objective is to ensure that the deadlines for periodic tasks are met and, at the same time, to minimize the response times of the aperiodic event handlers.

One solution is to schedule all non-periodic tasks at a lower priority (assuming that the tasks are scheduled using a preemptive fixed priority policy). This policy is known as *Background Servicing (BS)*. Although it is very simple to implement, it does not address the problem of the non periodic response times minimization. The periodic task servers were introduced in [248]. A periodic task server is a periodic task, for which classical response time determination and admission control methods are applicable (with or without modifications). This particular task is in charge of servicing the non-periodic traffic using a limited capacity. Several types of task server can be found in the literature. They differ in the way the capacity is managed. We can cite the *Polling Server* policy (PS), the *Deferrable Server* policy (DS) and the *Priority Exchange* policy (PE) first described in [248] and developed in [381]. Finally the *Sporadic Server* policy (SS) was proposed later in [382].

The use of the task servers corresponds to a kind of resource reservation on the system for the aperiodic traffic. Another approach was proposed in [247]: the *Static Slack Stealer* (SSS). It consists of storing precomputed data which is used online to compute the *slack-time*: the maximal amount of time available at instant t to execute aperiodic tasks at the highest priority without endangering the periodic tasks. The approach was extended to systems with hard real-time sporadic tasks in [315]. This was originally considered optimal in terms of minimizing the aperiodic tasks response times but it was later proved in [409] that such an optimal schedule cannot be obtained by a non clairvoyant algorithm. Moreover, the time and memory complexities of the slack stealing approach are much too high for it to be usable. In his PhD thesis work [118], Davis proposes the *Dynamic Slack Stealer* (DSS), a dynamic algorithm to compute the exact available slack-time. Since this algorithm still suffers from high time complexity, he also proposed two algorithms to compute a lower bound on available slack-time: the *static approximate slack stealer* (SASS) and the *Dynamic Approximate Slack Stealer* (DASS). The first is based on the static exact algorithm while the second is dynamic. The dynamic one has the advantage

of allowing *gain time*[1] to be assimilated in the slack-time and then transparently reallocated for aperiodic traffic.

The remainder of this chapter is organized as follow: first we review the reservation approaches and slack stealing approaches in Sect. 3.2. In Sect. 3.3 discusses the mechanisms that are provided by the RTSJ to support the programming of non periodic activities. Section 3.4 explains how these mechanisms can be used to implement event managers. Section 3.5 then undertakes a performance analysis of the implemented event managers. A unified API to facilitate the use of the event managers in a real-time application and improve its genericity and re-usability is also presented.

Moreover, the classical notations commonly encountered in real-time systems scheduling theory are summarized in Sect. 3.1.1 for a reader who is unfamiliar with this domain, thus avoiding any possible ambiguity.

3.1.1 Assumptions and Notations

We consider a process model of a mono processor system, Φ, made up of n periodic tasks, $\Pi = \{\tau_1, ..., \tau_n\}$ scheduled with fixed priorities. Each $\tau_i \in \Pi$ is a sequence of requests for execution characterized by the tuple $\tau_i = (r_i, C_i, T_i, D_i, Pi)$, where r_i is the instant of τ_i first release, C_i is the worst case execution time (WCET) of the request, T_i is the task period i.e. the amount of time between two consecutive requests, D_i is the deadline of a request relative to its release and P_i is the priority of the task, which can be arbitrary set (the priorities are not related to periods nor deadlines). Since we focus on this paper on the cases where the execution time is constant over all requests, we will denote the WCET the *cost* of the request. We restrict the study to the case $D_i \leq T_i$. The highest priority is 1 and tasks are ordered according to their priority, so τ_i has a higher priority than τ_{i+1} and more generally we have: $P_1 < P_2 < ... < P_n$. The system also has to serve an unbounded number p of aperiodic requests, $\Gamma = \{\sigma_1, ..., \sigma_p\}$. A request $\sigma_i \in \Gamma$ is characterized by a release date r_i and by an execution time C_i. Finally we assume that all tasks are independent and so we do not consider any blocking factor nor release jitter. If this work can be extended to the case where periodic tasks can share resources and possibly to the case with jitter on the periodic activations, it cannot be extended to the case where aperiodic share resources (with other aperiodic tasks or with periodic tasks). This work can also be extended to the case where priorities are associated to aperiodic tasks.

[1] If a periodic task has a worst case execution time greater than its mean execution time, most of its executions generate reserved but unused time called *gain time*.

3.2 Approaches to Handling Non Periodic Tasks

As we have seen in the introduction, there are two general approaches to handling non periodic traffic: the reservation approach (as identified in [248]) and the slack stealing approach.

3.2.1 Reservation Approaches

We choose to present this approach through two algorithms: the Polling Server (PS) and the Deferrable Server (DS). First, these two policies are the simplest and the most natural ones. Secondly the DS is a natural extension of the PS. Finally, the DS illustrates that the feasibility analysis may need to be changed. We also briefly present the Sporadic Server policy since it is the best efficient one.

3.2.1.1 Polling Server (PS)

A PS is a periodic task τ_s, with a priority P_s, a period T_s and a capacity C_s. From a feasibility analysis point of view, it is absolutely identical to a normal periodic task with a WCET of C_s. This task has access to a queue where the aperiodic events are enqueued when they are released. The queuing policy (*first in first out*, *last in first out*, any other order) is independent from the general principle of the PS and does not interfere with the hard tasks feasibility analysis. When the server is periodically released, its capacity is reset to its initial value. If the job queue is not empty, it begins to serve jobs, consuming its capacity, until its capacity falls to zero. If there are no more job to serve, its capacity falls instantaneously to zero. The principal draw back of this algorithm is that; if an aperiodic job is released just after a periodic activation of the server, say at a time $t = \alpha T_s + \epsilon$ with α an arbitrary positive integer and if the queue is empty at that time, the service of this job have to be delayed at least until the next activation ($(\alpha + 1)T_s$) since the server has just lost its capacity. Figure 3.1 illustrates the handling of aperiodic events with this policy.

3.2.1.2 Deferrable Server (DS)

The DS was proposed to address this PS weakness. The DS algorithm is quite similar to the PS one, except that it keeps its capacity even if the queue is empty. The result of this propriety is that the DS can be woken up at any moment, and can preempt a lower priority task. Figure 3.2 presents the same example as Fig. 3.1 but where the handling of aperiodic events is carried out by a DS.

Although the DS decreases the average response times of aperiodic events served, it also violates one hypothesis of the feasibility analysis theory which is that a periodic hard real-time task cannot suspends itself. Consequently, the interference

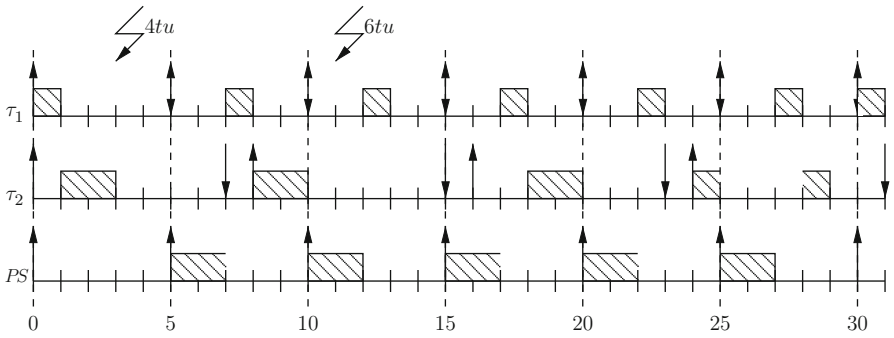


Fig. 3.1 τ_1 and τ_2 are two periodic tasks. At time 3 and 11, two aperiodic tasks arrives in the system. This is the result of their handling by the PS policy. The period and the capacity of the server was chose arbitrarily (but keep the system feasible) and are respectively of 5 and 2 time unit. It runs at the highest priority

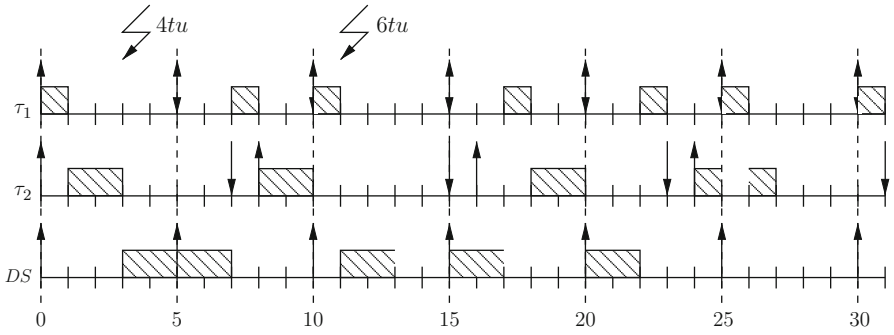


Fig. 3.2 We observe that the first aperiodic request now has a response time of 4 time units (against 9 tu with a PS). The second one now completes in 11 tu against 16 tu. The server runs at the highest priority

of the DS on the other tasks of the system is not the same as that of a periodic task with the same parameters. To illustrate this, an example is presented in Fig. 3.3. The direct consequence is that a number of systems feasible with a PS will not be feasible with a DS set with the same parameters. The cost of obtaining lower average response times for aperiodic tasks is a lower feasibility bound on the periodic ones.

3.2.1.3 Sporadic Server (SS)

The SS was originally introduced by Sprunt, Sha, and Lehoczky [382]. Conceptually, a SS is a task server with an execution time it can consume at each period. This budget is replenished according a specific rule. If there are no jobs in the queue at the time the server would be activated, the server activation is deferred until a job arrives.

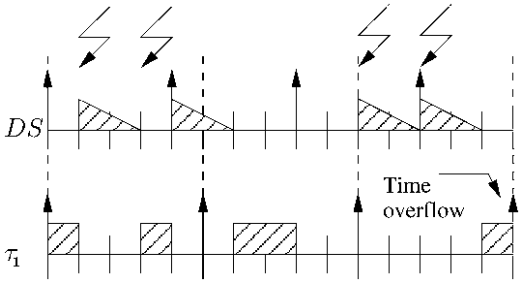


Fig. 3.3 We have in this example a DS with $C_s = 2$, $T_s = 4$ and a periodic hard real-time task τ_1 with $C_1 = 2$ and $T_1 = 5$. If the system is analyzed as if the DS was a regular periodic task, it is found feasible. However, we can see that if an aperiodic request is released at time 10 when the DS has kept its capacity, and another one at time 12 when the DS just retrieved its full capacity, τ_1 cannot complete its periodic request before its relative deadline

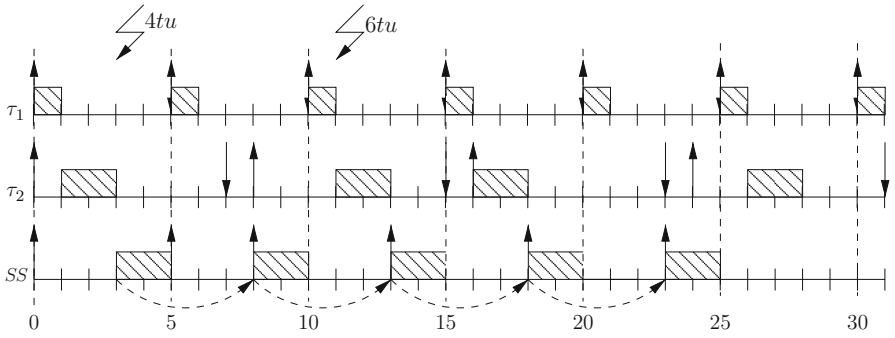


Fig. 3.4 The ideal sporadic server. At instant $t = 3$ the SS becomes active, the RT is equal to 8 time units i.e. the current time plus the period of the server. The server state is active when the priority of the current task is higher or equal of the priority of the SS. At instant $t = 5$ the SS becomes non active, the RA is the set to 2 time units i.e. the server budget consumed

To implement a SS we must process: the *Replenishment Time* (RT) i.e. the instant the capacity of the server will be replenished and the *Replenishment Amount* (RA) i.e. the capacity amount of the replenishment (the amount consumed between the server activation and the RT). Figure 3.4 illustrates a high priority SS behavior. The jobs will be served earlier than if they were served by a PS. This property is the same for a DS but the SS better preserves its execution budget than a PS or a DS. The activation model is a periodic model and thus a SS can be analyzed like a periodic task with the same cost and period.

Unfortunately the algorithm described in the original publication [382] violated the above assertion. The replenishment policy if implemented leads the server to consume more processor time than the equivalent periodic task. Several propositions for correcting this defect have been published [55, 253] and very recently in [383] where some experimentations show that the server's processor utilization was

significantly higher than that of a periodic task with same period and cost. Authors in [383] describe: "the premature replenishment" effect and the "budget amplification" effect. We must understand both effects before implementing a sporadic policy. For these reasons, although this policy is theoretically the best one and is the one supported by the POSIX specification, we choose to consider the simpler and more natural PS and DS algorithms.

3.2.2 Slack-Stealing Approaches

The general idea behind slack stealer algorithms is to compute at a time t, where there is an aperiodic pending request, a value, called the slack-time. This value corresponds to the maximal amount of time active tasks can be suspended without missing any deadlines. This time can then be used to handle aperiodic traffic. The first proposed approach was based on a table computed off line. This solution requires a large amount of memory to hold the tables. Moreover, it was impossible to integrate in the slack-time the time gained from periodic tasks that may complete before their WCET (gain time) and dynamic fluctuations of the system like release jitter were prohibited. It is so essential to compute slack-time on-line.

Note that when there is not possible to suspend all the tasks without missing any deadlines (i.e. when $S(t) < 0$), it is still possible to suspend only a subset of tasks with a lower range of priorities (if it exists k such that $\forall_{i \geq k} S_i(t) > 0$), and to schedule aperiodic tasks at a middle priority. However, for the sake of clarity, and since it cannot be prove that such an approach increases the average response times (see [409]), we will consider only the computation of the *system slack-time*, i.e. the amount of time all periodic tasks can be suspended. This corresponds to scheduling aperiodic requests at the highest priority.

Figure 3.5 illustrates the schedule obtained with a slack stealer on the example shown in Figs. 3.1, 3.2 and 3.4.

3.2.2.1 Dynamic Slack Stealing (DSS)

We consider an aperiodic request, J_a, released at time t. DSS is an algorithm to schedule this request. This algorithm relies on the determination at time t, and for each priority level, of the available slack-time, $S_i(t)$, which is the maximum amount of time the task τ_i can be delayed without missing its deadline. This value is equal to the number of unused time units at priorities higher or equal to i between t and the next τ_i deadline. The length of this interval is noted $d_i(t)$. Then, in order to serve J_a at the highest priority while guaranteeing the periodic tasks deadlines, we need to have a positive value for all $S_i(t)$. The number of "stolable" time units i.e. immediately available is then $S(t) = \min_{\forall i} S_i(t)$.

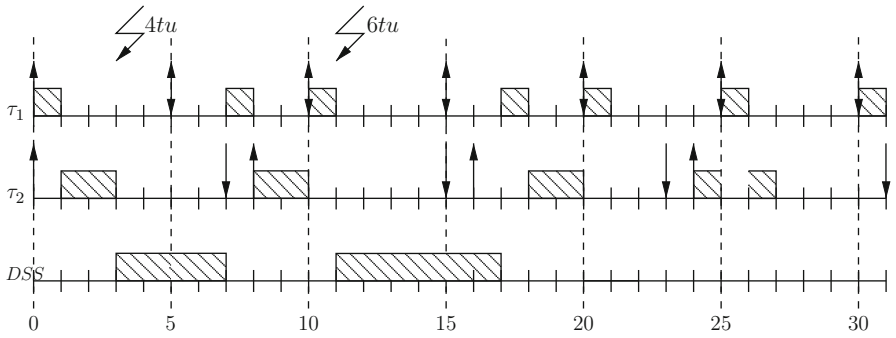


Fig. 3.5 Aperiodic requests are now served with a dynamic slack stealer. At time 3, both τ_1 and τ_2 have complete. We so have $S_1(3) = (10-3)-1 = 6$ and $S_2(3) = (15-3)-2-(2\times 1) = 8$, which implies $S^t(3) = 6 \geq 4$: the aperiodic task can be executed. At time 11, we have $S^t(11) = \min(8, 8) = 8 \geq 6$: again the task can be executed. The first request so completes by 4 tu (the same as with DS) and the second completes by 6 tu (against 11 with DS). However, server period and capacity was chosen arbitrarily on previous examples

To compute the $S_i(t)$ values, the interval between t and the next τ_i deadline that we denote $[t, t + d_i(t))$ is viewed as a succession of *i-level* busy periods[2] and *i-level* idle periods.[3] Then, $S_i(t)$ is the sum of the *i-level* idle period lengths. Equations to compute the end of a busy period starting at time t and the length of an idle period starting at time t can be derived from the classical feasibility analysis theory. These equations can thus be applied recursively until $d_i(t)$ is reached to determine $S_i(t)$.

With this methodology, slack-time has potentially to be recomputed from scratch at any instant, which is obviously not practicable. In order to reduce the complexity of the computation, we have to define slack-time at time t' relatively to slack-time at time t. There are two general cases to study:

None of the hard periodic tasks ends its execution in $[t, t')$. Then there are three possibilities for the processor activity between t and t': the processor can be idle, or it can be executing soft aperiodic requests (stealing slack-times) and it can be executing hard periodic tasks. For the two first possibilities, the slack-time is reduced by $(t' - t)$ for all priorities. However, if the processor is executing the hard real-time task τ_i, then the system is idle for higher priorities $k < i$, and the slack-time is reduced by $(t - t')$ only for these priorities.

One periodic hard real-time task, τ_i ends its execution at time $t'' \in [t, t')$. Then all *i-level* idle times present in $[t, d_i(t))$ will be also present in $[t, d_i(t) + T_i) = [t, d_i(t''))$, which is the new interval to consider for the *i-level* slack-times computation. Therefore, the τ_i termination can only increase $S_i(t)$ but never

[2]Periods where the processor is servicing priorities higher or equal to i

[3]Processor idle periods or periods where processor serves priorities lower than i

decrease it. Consequently, $S_i(t)$ has to be recomputed each time τ_i ends a periodic activation.

So, assuming that there is a time t where the $S_i(t)$ was up to date for all tasks, the DSS algorithm to compute $S(t'')$ is:

1. If none of the periodic hard real-time tasks ends in $[t, t')$
   a. If the processor is idle or executing soft aperiodic requests

$$\forall j : S_j(t') = S_j(t) - (t - t') \tag{3.1}$$

   b. If the processor is executing hard periodic task τ_i

$$\forall j < i : S_j(t') = S_j(t) - (t - t') \tag{3.2}$$

2. If hard real-time task τ_i ends at time $t'' \in [t, t')$, $S_i(t'')$ has to be computed using the recursive analysis described at the beginning of this section. The recurrence can be started with the previous value ($S_i(t'') = S_i(t)$).

A practical implementation of DSS computes all i-level slack-time values at time 0, and then updates them at each context switches. When an aperiodic task is release, the system slack-time can be obtained in a linear time complexity, by computing the minimum over the i-level slack-time values. However this algorithm is not directly usable because of the time complexity of the recursive computation of the $S_i(t)$ to perform at each task ends. That is why this part can be replaced by the computation of a lower bound. We will present two approximation algorithms in Sect. 3.5.3.

3.2.3 Conclusions

From a feasibility analysis point of view, hard real-time systems are composed by periodic tasks. If a task is not periodic, its arrival frequency must be bounded and perhaps enforced to respect this bound. However, some over tasks, even if they are not critical for the system, can improve its performances or its QoS. But it happens that these tasks arrival frequencies cannot be bounded, or at the price of a high pessimism in the analysis. That is why scheduling algorithms that try to minimize the response times of such task while guaranteeing deadlines for periodic tasks are needed. Two approaches exist: the reservations of time slots through task servers, and the collect of future unused time slots through a slack stealer mechanism.

3.3 Non Periodic Events Within the RTSJ

The aim of the Real-time Specification for Java is to design API and virtual machine specifications for the writing and the execution of real-time applications programmed with the Java language. This effort was started under the first Java Specification Request (JSR-01) that led to the RTSJ v1 and continues nowadays under the 282nd (JSR-282) which focus on the future RTSJ v1.1. Unfortunately, RTSJ does not propose the necessary scheduling mechanisms to handle the aperiodic traffic as described in the preceding Section. Although the specification allows an implementation to provide a customized scheduler, in which the necessary mechanisms could be integrated, many RTJVM delegates scheduling to the underlying operating system and the others, the proprietary ones, usually do not permit any modifications or upgrade. Thus, it is important to offer these mechanisms as RTSJ additional classes. That is why this section will review the existing tools in the RTSJ API to program the mixed traffics, Sect. 3.4 will show their use to implement the algorithms presented in Sect. 3.2 as user-land tasks and Sect. 3.5 evaluates the incidence on their performances.

3.3.1 RTSJ Schedulable Object

To generalize the `Thread` facility of regular Java programs, RTSJ proposes the notion of *Schedulable Object* (SO) through the interface `Schedulable`. A SO is an object which can be scheduled by the scheduler. A suitable context for the execution of such an object is created using scheduling and release parameters attached to each SO. The `Schedulable` interface proposes getter and setter methods to access and modify scheduling parameters (class `SchedulingParameters`) and release ones (class `ReleaseParameters`).

Two classes directly implement this interface in the RTSJ: `RealtimeThread` and `AsyncEventHandler`. The first one also inherits from `java.lang.Thread`. It is designed to be used in place of it in a real-time context. The second one models a SO that is not necessary bound to a system thread. The `java.lang.Runnable` logic that it encapsulates can be either executed in a dedicated thread or a thread pool can be used, depending on the VM implementation. In both case, the execution is possible only if the AEH was bound to one or many `AsyncEvent`. Then the execution is triggered by a call to the method `fire()` of one of these AE. The `RealtimeThread` class is appropriate for modeling periodic hard real-time traffic and the classes `AsyncEvent/AsyncEventHandler` best fit requirement for modeling aperiodic event traffic.

```
public class SimplePeriodic {

  public static void main(String[] args) {

    PriorityParameters prioP = new PriorityParameters(15);

    RelativeTime offset = new RelativeTime(3000, 0); // Start Offset
    RelativeTime period = new RelativeTime(5000, 0); // Period
    RelativeTime cost = new RelativeTime(200, 0); // Cost
    RelativeTime deadline = new RelativeTime(1000, 0); // Deadline

    PeriodicParameters periodicP = new PeriodicParameters(
        offset, period, cost, deadline, null, null);
    // no Cost Overrun handler and no Deadline Miss handler

    RealtimeThread pso1 = new RealtimeThread(prioP, periodicP) {
      public void run() {
        do {
          System.out.println("Simple Periodic Task 1");
        } while (waitForNextPeriod());
      }
    };
    AsyncEventHandler pso2 = new AsyncEventHandler() {
      public void handleAsyncEvent() {
        System.out.println("Simple Periodic Task 2");
      };
    };
    pso2.setSchedulingParameters(prioP);
    pso2.setReleaseParameters(periodicP);
    PeriodicTimer pt = new PeriodicTimer(offset, period, pso2);

    pso1.start();
    pt.start();
  }
}
```

Listing 3.1 Writing a periodic task

3.3.2 Periodic Traffic: Schedulable Object with PeriodicParameters

There is no mechanism in regular Java language to program a real periodic behavior. RTSJ brings this facility to SO through an appropriate `ReleaseParameters` subclass: `PeriodicParameters`. The periodic behavior can be enforced for `RealtimeThread` through the blocking method `waitForNextPeriod()`, and with a binding to a `PeriodicTimer` for an `AsyncEventHandler`.

Listing 3.1 presents the two ways to write a periodic task with RTSJ. In both cases, the `PeriodicParameters` are used to indicate that the SO will be released with a constant inter-arrival time of 5 s (`period` parameter), that an offset

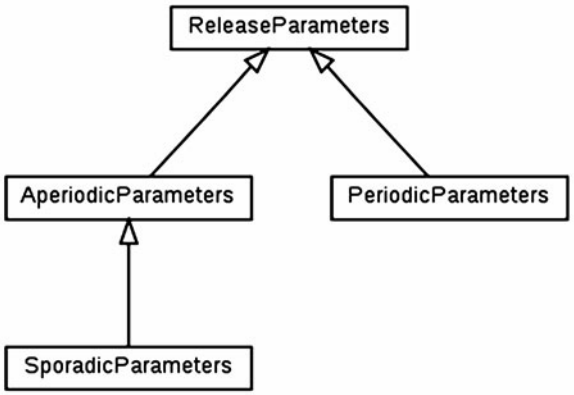


Fig. 3.6 RTSJ release parameters class diagram

of 3 s will be assigned for the first activation (`start`), and that each instances should completes before 1 s is elapsed after its release (`deadline`). The `cost` parameters is the maximal CPU time budget allowed for an instance and is used to decide the system feasibility. Two handlers can also be set to be triggered respectively if the task overruns its cost and missed its deadline. These handlers are SO and so also require scheduling and release parameters. Note that the correct trigger of the cost overrun handler necessitates that the VM monitor the CPU time consumption of each task, which is not a mandatory feature to respect the RTSJ specification. For `RealtimeThread` SO, the `waitForNextPeriod()` method returns true if the previous instance has respected its deadline. Otherwise the value depends on the scheduler. 'The Reference Implementation' base scheduler returns false and de-schedule the faulty task, then the deadline missed handler can be used to reschedule the task.

3.3.3 Non Periodic Traffic: AsyncEventHandler with Other ReleaseParameters

As shown in Listing 3.2, the simplest way to use an `AsyncEventHandler` is to associate the handler to an event with the method `addHandler()` (line 10) and to trigger the event with the method `fire()` (line 11). The application will release the handler after approximately 800 ms. In this state, the application cannot be qualified of real-time. As recapitulated by Fig. 3.6, two subclasses of `ReleaseParameters` can be used to specify real-time behavior of the asynchronous handler: `AperiodicParameters` and the `SporadicParameters`.

```
public class SimpleAEH {
  public static void main(String[] args)
        throws InterruptedException {
    AsyncEventHandler aeh = new AsyncEventHandler() {
      public void handleAsyncEvent() {
        System.out.println("An␣Asynchronous␣Event␣Handling");
      }
    };
    AsyncEvent ae = new AsyncEvent();
    ae.addHandler(aeh);
    ae.fire();
    Thread.sleep(800);
    ae.fire();
  }
}
```

Listing 3.2 AE/AEH example

3.3.3.1 AperiodicParameters

The `AperiodicParameters` specifies the cost, the deadline and optionally to set a cost overrun handler and a deadline missed handler to the SO. The following lines can be added to example of Listing 3.2:

```
aeh.setReleaseParameters(
  new AperiodicParameters(
    new RelativeTime(100, 0), // cost
    new RelativeTime(2000, 0), // deadline
    null, // no overrunHandler
    null); // no deadlineMissHandler
  ));
```

3.3.3.2 SporadicParameters

The `SporadicParameters` allows a Minimal Inter-arrival Time (MIT), between two releases of the handler, to be specified. The following lines can be added to Listing 3.2:

```
aeh.setReleaseParameters(
  new SporadicParameters(
    new RelativeTime(6000, 0), // MIT
    new RelativeTime(100, 0), // cost
    new RelativeTime(2000, 0), // deadline
    null, // no overrunHandler
    null); // no deadlineMissHandler
  ));
```

The application can also specify the behavior when the MIT parameter is not respected, for example:

```
sp.setMitViolationBehavior(SporadicParameters.mitViolationSave);
```

Four policies are supported by the RTSJ for the occasion when an attempt is made to release the handler before its MIT : `Except`, `Ignore`, `Save` and `Replace`. `Except` causes the `MITViolationException` to be thrown, `Ignore` leads to dropping of the faulty activation, and `Save` and `Replace` will lead the handler to postpone its execution. `Save` enqueues as many as possible releases while `Replace` just saves the last release.

3.3.4 The ProcessingGroupParameters Approach

The class `ProcessingGroupParameters` (PGP) models an object quite similar to `ReleaseParameters`, but shared by several SO. It allows in particular to assign a common time budget to several SO. This effectively sets up a logical execution-time server. Unfortunately, the desirable behavior when the budget is consumed, which is to not schedule the others SOs until the budget is replenished, is not mandatory because it relies on the capacity to monitor CPU consumption of the JVM. Moreover, this behavior is underspecified [90] and in any case does not allow to choose a particular policy for the consumption of the shared budget. A contribution to extend the PGP semantic for multiprocessors systems and to extend it in order to write task servers has been led in [431]. However this approach relies on modifications on the specification and is not implementable under currently available RTJVMs.

3.3.5 Conclusions

RTSJ brings us the ability to program both periodic and aperiodic traffics. Mixed traffic can be programmed, but there are no mechanisms able to enforce the deadlines of periodic tasks while trying to favor the aperiodic tasks executions. The PGP were an attempt to answer this question, but are under-specified and need a closer attention from the expert group in charge of future version revisions.

3.4 How to Write Event Manager with RTSJ

In the current RTSJ, there are two approaches that can be taken to implement event managers. The first is to integrate the algorithms presented in Sect. 3.2 into the RTSJ's scheduling framework. This, however, will be very dependent on the

mechanisms provided by the run-time environment. The second approach is to implement the algorithms at the application level (as user-land tasks). This is the more portable approach and, therefore, will be considered in this chapter.

We explore in this section the writing of the event manager mechanism as user-land task with RTSJ and expose the inherent limitations of the implementation of these algorithms. We illustrate the section with code skeletons that demonstrates how to write servers and slack-stealers.

3.4.1 Limitations to Write Event Managers with RTSJ

If one does not want to rely on a specific RTJVM, the event manager (server or slack stealer) must be a regular SO with no specific privileges. In such a context, the monitoring of its capacity consumption is only possible when the event manager runs at the highest priority compared to other SO. Moreover, the assumption is that the RTJVM runs on an RTOS and this is not shared with other real-time applications. Then, the monitoring can be performed using timers and time measurement after and before asynchronous event associated logic. As a consequence, only one unique event manager can be used in the system.

Another important draw back is the impossibility in Java to stop a thread from another one. This is addressed in the RTSJ by a semantic extension of the Java exception mechanism. If an executing context, which is explicitly marked to be interruptible, receives an exception of the type `AsynchronouslyInterruptedException`, the context can be properly exited—making it the programmer's responsibility to leave the system in a consistent state. However it only permits to stop a thread, not to resume it later. So, if the event manager is implemented in user-land, the aperiodic tasks must be scheduled only at the highest priority and in a non preemptive mode.

It directly implies two restrictions: there can exist only one unique manager in the system, and a task with an important cost cannot be dispatched on several manager instances.

```
public class MyPollingServer extends RealtimeThread {
  private final PriorityQueue<Interruptible> aperiodicQueue;
  private final Timed timer;
  private final RelativeTime capacity = new RelativeTime();
  private final RelativeTime elapsed = new RelativeTime();
  private final AbsoluteTime before = new AbsoluteTime();
  private final AbsoluteTime after = new AbsoluteTime();
  private final RelativeTime zero = new RelativeTime(0, 0);

  public MyPollingServer(PeriodicParameters release) {
    super(new PriorityParameters(PriorityScheduler.MAX_PRIORITY-1));
    this.aperiodicQueue = new PriorityQueue<Interruptible>();
    this.timer = new Timed(release.getCost());
  }
```

```
  @Override public void run() {
    while (true) {
      capacity.set(getReleaseParameters().getCost());
      while (!aperiodicQueue.isEmpty()
          && capacity.compareTo(zero) > 0) {
        timer.resetTime(capacity);
        Clock.getRealtimeClock().getTime(before);
        if (timer.doInterruptible(aperiodicQueue.peek())) {
          aperiodicQueue.poll();
          Clock.getRealtimeClock().getTime(after);
          after.subtract(before, elapsed);
          capacity.subtract(elapsed, capacity);
        } else break;
      }
      waitForNextPeriod();
    }
  }
  public void addEvent(Interruptible logic) {
    aperiodicQueue.add(logic);
  }
}
```

Listing 3.3 A simple Polling Server example

3.4.2 How to Write a Server

Listings 3.3 and 3.4 present respectively a possible implementation of a Polling Server and the corresponding implementation of an aperiodic task for this server. Since PS is a regular periodic task, it can be implemented as a subclass of `RealtimeThread`, with a constructor taking explicitly an instance of `PeriodicParameters`. We use the class `PriorityQueue` to model the aperiodic queue. A public method that adds handler code (an instance of `Interruptible`) to the queue is called when an aperiodic task, modeled by an `AsyncEventHandler` is released. In other words, the method `handleAsyncEvent()` of the aperiodic task will call the method `addEvent()`. The use of an `Interruptible` instance to model the aperiodic task logic allows the use of the class `Timed`, a subclass of `AsynchronouslyInterruptedException`, to automatically stop the execution if the capacity is reached. Indeed this class encapsulates a timer, if this timer expire, the exception is thrown. Note that system maximal priority is used when enqueuing the aperiodic tasks, while the polling server itself runs at the system maximal priority minus one. The whole mechanism so reserves two priorities in the system.

Writing a DS is just as simple. The main difference is that a DS can be waken up at any time, and not only periodically. The use of an `AsyncEventHandler` instead of a `RealtimeThread` is, therefore, preferable. A `PeriodicTimer` can then be use for its periodic wakeup in order to refill its capacity, and it can also be bound to an event fired each time that an aperiodic task is released,

```
public class AperiodicTask {
  public AperiodicTask(final HighResolutionTime release,
      final MyPollingServer server, final Interruptible logic) {
    AsyncEventHandler handler = new AsyncEventHandler() {
      @Override public void handleAsyncEvent() {
        server.addEvent(logic);
      }};
    handler.setSchedulingParameters(new PriorityParameters(
        PriorityScheduler.MAX_PRIORITY));
    timer = new OneShotTimer(release, handler);
  }
  public void start() {
    timer.start();
  }
  private final OneShotTimer timer;
}
```

Listing 3.4 Corresponding aperiodic task

in order to enqueue the logic and possibly to serve it. It has to be noted that to integrate the server in the feasibility analysis, it is not sufficient to assign it `PeriodicParameters`. We have to inherit the `PriorityScheduler` class, to override the feasibility analysis related methods, and to use this new scheduler as the default scheduler.

3.4.3 How to Write a Slack Stealer

When managing the capacity of a simple server like the PS or the DS, one only needs to monitor the server time consumption, but the consumption of all tasks in the system has to be known to compute slack-time. Indeed, as described in Sect. 3.2.2.1, it is possible to only recompute $S_i(t)$ when task τ_i ends a periodic instance. Then, at each context switches, all $S_i(t)$ values have to be updated in function of the previously executing task time consumption and priority.

This behavior, the consumption time monitoring, is missing from the RTSJ, but has been proposed by the JSR-282: *"7. Add a method to Schedulable and processing group parameters, that will return the elapsed CPU time for that schedulable object (if the implementation supports CPU time)"* [131].

We note that even with RTSJ 1.1 this feature will not be mandatory. However, the monitoring can be done at user-land with a mechanism that permits to add code just before the start of a periodic instance and just after its end. This is possible by overriding the `RealtimeThread` class as shown by Listing 3.5. Methods `runBeforePeriodic()` and `runAfterPeriodic()` contains the code to add and should be executed in a mutual exclusion scope.

```
public class MyRealtimeThread extends RealtimeThread{
  //...
  @Override
  public static boolean waitForNextPeriod(){
    runAfterPeriodic();
    boolean returnValue =
      RealtimeThread.waitForNextPeriod();
    computeBeforePeriodic();
    return returnValue;
  }
  //...
}
```

Listing 3.5 `WaitForNextPeriod()` modifications

Before proceeding, we can highlight here a strong limitation of this design: it supposes that all periodic tasks in the system (even tasks from which we do not want to steal slack-time) are coded using this subclass of `RealtimeThread`.

A slack stealer can then be coded with three SO:

- An `AsyncEventHandler` bounded to an `AsyncEvent` fired by method computeBeforePeriodic(),
- An `AsyncEventHandler` bounded to an `AsyncEvent` fired by method computeAfterPeriodic(),
- An `AsyncEventHandler` bounded to an `AsyncEvent` fired when an aperiodic task is released.

The two first AEH are responsible to maintain up to date the i-level slack-time values while the third one manages the aperiodic handler queue. The class `AperiodicTask` presented by Listing 3.4 can be adapted and reuse: in fact we just need to replace the field server by the third AEH above cited. A `Timed` can be used to ensure that aperiodic task does not consume more than the computed slack-time. The issue is that slack-time computation does not have negligible cost. The solution is to include this cost in the WCET of each periodic task since all costly operations are performed by handler of events fired by methods `computeBeforePeriodic()` and `computeAfterPeriodic()`.

Listing 3.6 presents a possible skeleton for a DSS implementation in user-land with RTSJ.

```
public class MyDSS extends AsyncEventHandler{

  private final PriorityQueue<Interruptible> aperiodicQueue;
  private final Timed timer;

  public final AsyncEvent beforePeriodicEvent;
  public final AsyncEvent afterPeriodicEvent;
  private final AsyncEventHandler beforePeriodicHandler;
  private final AsyncEventHandler afterPeriodicHandler;
```

```
  public MyDASS() {
    super(
      new PriorityParameters(PriorityScheduler.MAX_PRIORITY-1,...)
    );
    aperiodicQueue = new PriorityQueue<Interruptible>();
    timer = new Timed(zero);
    beforePeriodicEvent = new AsyncEvent();
    afterPeriodicEvent = new AsyncEvent();
    beforePeriodicHandler=
      new AsyncEventHandler(
        new PriorityParameters(PriorityScheduler.MAX_PRIORITY-1,...)
        ){
          public void handleAsyncEvent(){...}
    };
    afterPeriodicHandler=
      new AsyncEventHandler(
        new PriorityParameters(PriorityScheduler.MAX_PRIORITY-1,...)
        ){
          public void handleAsyncEvent(){...}
    };
    beforePeriodicEvent.addHandler(beforePeriodicHandler);
    afterPeriodicEvent.addHandler(afterPeriodicHandler);
  }

  private final RelativeTime capacity = new RelativeTime();
  private final RelativeTime elapsed = new RelativeTime();
  private final AbsoluteTime before = new AbsoluteTime();
  private final AbsoluteTime after = new AbsoluteTime();
  private final RelativeTime zero = new RelativeTime(0, 0);

  @Override public void handleAsyncEvent() {
      capacity.set(getSlackTime());
      while (!aperiodicQueue.isEmpty()
          && capacity.compareTo(zero) > 0) {
        timer.resetTime(capacity);
        Clock.getRealtimeClock().getTime(before);
        if (timer.doInterruptible(aperiodicQueue.peek())) {
          aperiodicQueue.poll();
          Clock.getRealtimeClock().getTime(after);
          after.subtract(before, elapsed);
          capacity.subtract(elapsed, capacity);
        } else
          break;
      }
  }
  public void addEvent(Interruptible logic) {
    aperiodicQueue.add(logic);
  }
  private RelativeTime getSlackTime(){...}
}
```

Listing 3.6 A (partial) implementation of DSS in user-land

3.4.4 Conclusions

In this section we have shown how to write event managers as user-land tasks with the tools provided by the RTSJ. However, this implementation brings some limitations. The next section will explain these limitations, propose solutions, evaluate the performances of obtained mechanisms and propose an API that permits the use of event managers in a transparent way.

3.5 Performance Limitations and Specific Solutions

3.5.1 Introduction

This section presents solutions to reduce the user-land implementation impact of the event manager policies. These solutions are evaluated through simulations. We develop an event-base simulator, written in Java, and available on GNU General Public License at [260].

3.5.1.1 Simulations Methodology

We measure the mean response time of soft tasks with different aperiodic and periodic loads. First, we generate groups of periodic task sets with utilization levels of 30%, 50%, 70% and 90%. The results presented in this section are averages over a group of ten task sets. For the same periodic utilization, we repeat generations over a wide range of periodic task set composition, from systems composed by 2 periodic tasks up to systems composed by 100 periodic tasks. The periods are randomly generated with an exponential distribution in the range [40–2560] time units. Then the costs are randomly generated with an uniform distribution in the range [1-period] In order to test systems with deadlines less than periods, we randomly generate deadlines with an exponential distribution in the range [cost-period]. Priorities are assigned assuming a deadline monotonic policy. Non feasible systems are rejected, the utilization is computed and systems with an utilization level differing by less than 1% from that required are kept.

For the polling server, we have to find the best period T_s and capacity C_s pair. We try to maximize the system load U composed by the periodic load U_T and the server load U_S. To find a lower bound $C_s^{\min}$ of C_s, we first set the period to 2560 (the maximal period). We then search the maximal value for C_s in $[1, 16]$ in order to keep a feasible system. This maximal value is our lower bound on C_s. Then we seek the lower possible T_s value in $[C_s^{\min}/(1 - \sum C_i/T_i), 2560]$. For each T_s value tested, we try decreasing values of C_s in $[C_s^{\min}, T_s(1 - \sum C_i/T_i)]$. Note that it is possible to find a capacity lower than the maximal aperiodic tasks cost. In such cases, since we have to schedule the aperiodic tasks in one shot, we have no solution but to

background scheduling the tasks with a cost greater than the server capacity. For the deferrable server, the methodology is similar, except that since the server has a bandwidth preservation behavior, we do not try to minimize the period and we can search the maximal C_s value in $[1, 2560]$.

Finally, we generate groups of ten aperiodic task sets with a range of utilization levels (plotted on the x-axis in the following graphs). Costs are randomly generated with an exponential distribution in the range [1–16] and arrival times are generated with an uniform distribution in the range [1–100,000]. Our simulations end when all soft tasks have been served.

3.5.2 Balancing the Limitations

The main limitation of servers and slack stealers implemented in user-land is the fact that aperiodic tasks cannot be preempted. This will directly result in a loss of server capacity or of slack-time. To avoid abusive use of asynchronous interruptions (and CPU time consumption waste), it is possible to restrict the schedule of an aperiodic task to the case where there is enough capacity/slack-time to let it complete in one-shot. But starvation situation can still happens, when there is capacity or slack-time available, but when the next aperiodic task to schedule has a bigger execution time than this capacity/slack-time.

3.5.2.1 Queuing Policy

A first solution to limit the impact on the performances is to use an appropriate queuing policy for aperiodic tasks. We conducted many simulations and experiments, all of which led to the conclusion that the best strategy to minimized the average response time is to sort the aperiodic queue by increasing WCET. We called this policy *Lowest Cost First* (LCF). This result confirms a well-known result from general message queuing theory, and is not surprising: the shorter a task is, the higher is its probability to complete in one-shot. However, if LCF policy queuing profits all algorithms, the benefits are counter-balanced by the non preemptive limitation.

Figure 3.7 illustrates this phenomenon in the case of the PS. Moreover, changing the queue policy is not sufficient to resolve the starvation issue.

3.5.2.2 BS Duplication

The only solution to resolve the starvation issue is to duplicate each aperiodic handler. One copy is then executed with background policy, and the other one is enqueued in the server or the slack stealer. The first replica which completes interrupt the other one.

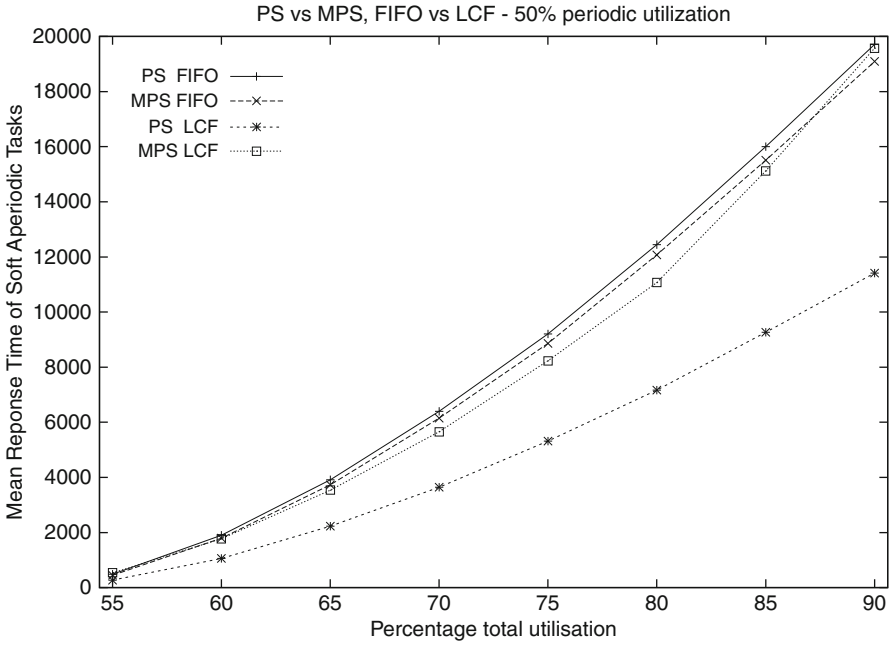


Fig. 3.7 Regular PS vs User-Land PS; queue policy: FIFO; Periodic Load: 70%; BS Duplication: No. MPS designates the user-land PS and PS the regular algorithm. We can see the important performances gap between PS-LCF and the other policies

This strategy offers very good results, the range of response time, which was between 1,000 and 25,000 time units is now between 70 and 200, as presented by Fig. 3.8. However, it has to be noted that it is possible to duplicate aperiodic tasks only with the assumption that they cannot share resources (because in that case the replica should run in mutual exclusion). Extensive simulations was performed and similar results were obtained for a wide range of periodic and aperiodic load, and for all policy tested. Interested readers can consult [262, 265].

The ideal solution should be to immediately start aperiodic tasks in BS when they are released. Then when the event manager can handle it, raise its priority, and again drop its priority to the background when the manager run out capacity/slack-time. However, the dynamic priority change is not possible with all actual RTJVM, even if it should be according to the specification.

3.5.3 Slack-Time Approximations and RTSJ

In the case of slack stealers, a factor of performance degradation is not directly linked to the RTSJ: it is the algorithmic cost of the slack-time computation. To pass

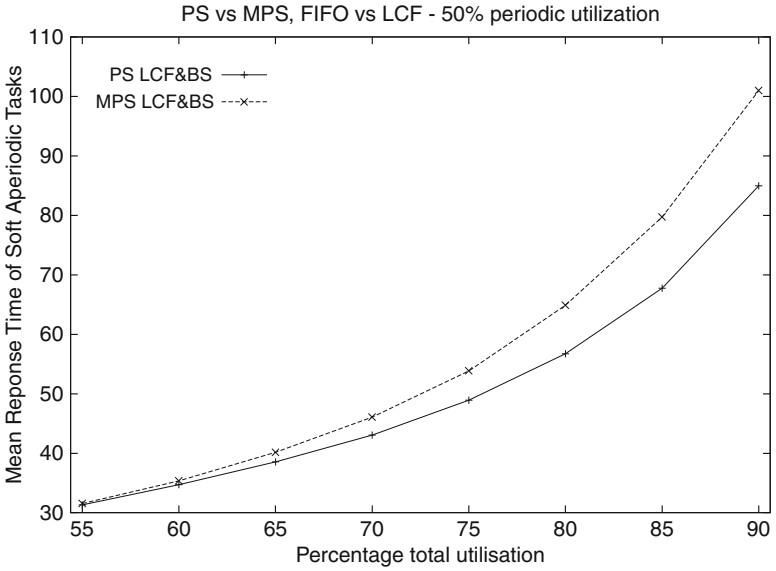


Fig. 3.8 User-land MPS with BS duplication against Regular PS with BS duplication

through this obstacle, it was proposed to use a lower bound instead of the exact value. We explain here the approximation method proposed by Davis in [118] and then an another method we developed specifically for an RTSJ implementation.

3.5.3.1 Dynamic Approximate Slack Stealer (DASS)

Since $S_i(t)$ is the sum of the i-level idle period lengths in the interval $[t, t + d_i(t))$, Davis proposes to estimate this quantity by computing a bound on the maximal interference the task τ_i can suffer in this interval. A bound on this interference is given by the sum of the interferences from each task with a higher priority than τ_i. Then (3.3) gives the interference suffered by a task τ_j from a task τ_i in an interval $[a, b]$.

$$I_i^j(a,b) = c_i(t) + f_i(a,b)C_i + \min(C_i, (b - x_i(a) - f_i(a,b)T_i)_0) \tag{3.3}$$

The function $f_i(a,b)$ returns the τ_i instance number which can begin and complete in $[a,b]$. It is given by (3.4).[4]

$$f_i(a,b) = \left\lfloor \frac{b - x_i(a)}{T_i} \right\rfloor_0 \tag{3.4}$$

[4]The notation $(x)_0$ means $max(x, 0)$

The function $x_i(t)$ represents the first activation of τ_i which follows t. Then the interference is composed of the remaining computation time needed to complete the current pending request, by a number of entire invocations given by $f_i(a, b)$, and by a final partial request.

A lower bound on the $S_i(t)$ value is given by the length of the interval minus the sum of the interferences from each task with a higher or equal priority than τ_i. It is recapitulated by (3.5).

$$S_i(t) = \left(d_i(t) - t - \sum_{\forall j \leq i} I_j^i(t, d_i(t)) \right)_0 \tag{3.5}$$

3.5.3.2 Minimal Approximate Slack Stealer (MASS)

The algorithm MASS and its integration with RTSJ API was initially proposed in [263, 264]. It consists on an alternative way to compute a lower bound on $S_i(t)$. While it is possible to adapt DASS to be implemented in user-land, MASS was designed to fit this requirement.

The idea behind MASS, is to simplify operations that have to be performed when a periodic tasks begins its instance. We decompose $S_i(t)$ in two parts to be able to compute them separately. First we consider the available work at a given priority before the next deadline. This quantity is denoted $\bar{w}_i(t)$. Second we consider the work demand at a given priority, denoted $\bar{c}_i(t)$. Equation 3.6 then gives us the slack-time in the system.

$$S(t) = \min_{\forall i \in hrtp} S_i(t) = \min_{\forall i \in hrtp} (\bar{w}_i(t) - \bar{c}_i(t)) \tag{3.6}$$

The $\bar{c}_i(t)$ values can still have to be maintained at each context switches, but the $\bar{w}_i(t)$ can be only update then a periodic instance completes.

3.5.3.3 DASS and MASS Comparative Example

We propose here a numerical example to better understand the differences between DASS and MASS. The studied system is composed of three tasks. The task parameters and the normal execution of the system when there is no aperiodic event to deal with are given by Fig. 3.9. We focus on the computation of slack-time for priority level 3.

With DASS

At time instant 0, $S_3(0)$ is equal to τ_3's deadline from which we subtract its cost and the interference of τ_1 and τ_2 in the time interval $[0, 14]$. We have $S_3(0) =$

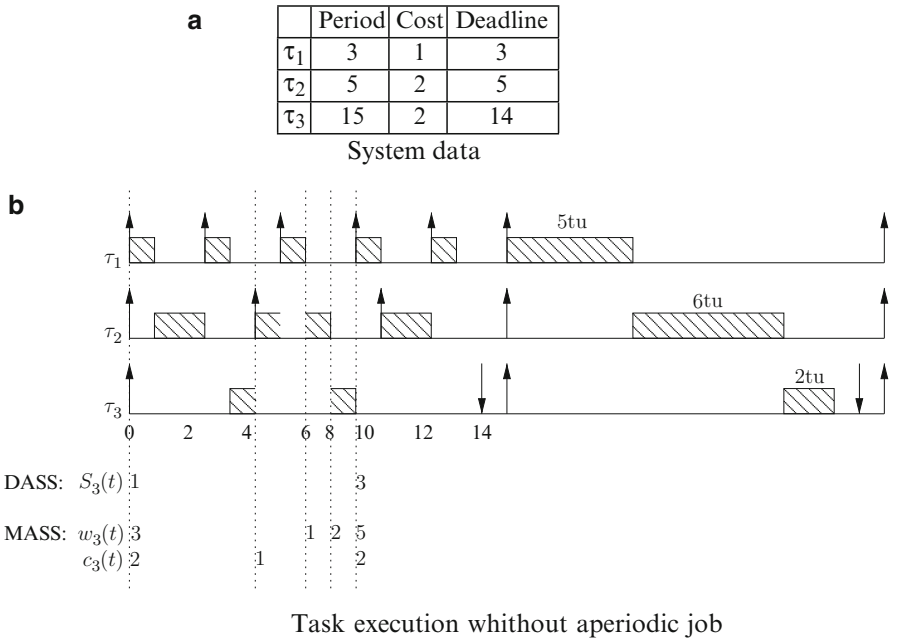


Fig. 3.9 Slack time computation at priority level 3 with DASS and MASS

$14 - 5 * 1 - 3 * 2 - 2 = 1$. It is the number of unused times slot at priority level 3 in this interval when the system does not accept any aperiodic job. The value does not change until time instant 9 since only tasks with a lesser or equal priority are executed. It has to be noted that at time instants 3 and 5, it is necessary to subtract 2 tu to $S_1(t)$ and 1 tu to $S_1(t)$ and $S_2(t)$. That is why the time complexity of start-task computation is in linear time complexity.

At time instant 9, τ_3 ends the execution of its first instance. The time interval considered for the computation of $S_3(t)$ is now $[t, 29]$ and not $[t, 14]$. The duration of this interval at time instant $t = 9$ is 20 tu. We subtract from this value the interference of 4 activations of τ_2, of 7 activation of τ_1 and of 1 activation of τ_3. We have so $S_3(9) = 20-8-7-2 = 3$. $S_3(9)$ is recomputed from scratch without used an information already available: there is 1 unused time unit before time instant 14.

With MASS

The $\bar{w}_3(0)$ value is equal to the τ_3 deadline from which we subtract the interferences of τ_1 and τ_2 on the interval $[0, 14]$. We so have $\bar{w}_3(0) = 14 - 6 - 5 = 3$. Since $\bar{c}_3(0) = 2$, it gives us $S_3(0) = 1$, i.e. the same value obtained with DASS.

At time instants 1 and 3, tasks τ_1 and τ_3 end an instance. $\bar{w}_3(t)$ is then decreased by the elapsed time, but immediately incremented by the cost of ending tasks, and so is constant until time instant 7.

This is not the case for $\bar{c}_3(t)$, which is updated at time instant 5. Indeed, τ_2 starts then an instance, and the previously executing task, τ_3 is updated. We have $\bar{w}_3(5) = 3$ and $\bar{c}_3(5) = 1$, which gives us $S_3(5) = 2$. This value is incorrect and is greater the correct one ($S_3(5) = 1$). However this error has no effect since slack-times are not evaluated when a task ends its execution. The next evaluation will occur at time instant 7.

At this time, $\bar{w}_3$ and $\bar{w}_2$ are decreased by 3 tu, because its corresponds to the elapsed time since the last end of a periodic task, and increased by the cost of τ_1, ie only 1 tu. We so have $S_3(7) = 3 - 3 + 1 - 1 = 0$, which a lesser value than the one obtained with DASS.

At time 8, τ_2 also ends a periodic execution. $\bar{w}_3(8)$ is so decreased by 1 tu and increased by 2 tu. We so have $S_3(8) = 1 + 2 - 1 - 1 = 1$, which is the same value than the one obtained with DASS.

Finally, at time 9, τ_3 ends an instance. The interference it suffers within time interval $[9, 15]$ from τ_1 and τ_2 is already integrated to $\bar{w}_3$, since the interference has been bounded with the activations number multiplied by the cost, and since the next τ_1 or τ_2 activation is not before time 15. The value of $\bar{w}_3$ is then increased by one period ($29 - 14 = 15$) since we now consider the next deadline, and decreased by the interference of τ_1 and τ_2 in the time interval $[15, 30]$. We have $S_3(9) = (2 + 15 - 6 - 5) - 2 = 3$.

Remarks on the Interference Computation

In this example, the interference computation of τ_1 and τ_2 on τ_3 is not hard because when the τ_3 deadline is reached all the started instances are ended. The interference caused by a task is a multiple of its activation number. If it was not the case, DASS should compute the exact interference, that is why the constant in the needed computation is high. With MASS, only an upper bound is used. This approximation permits to only consider the tasks activation at time instant greater than or equal to the deadline.

3.5.3.4 Discussion on the Differences

With the DASS algorithm, it is some times possible to obtain a more accurate bound. However we see that the gap between the two bounds (the one obtained with DASS and the one obtained with MASS) is fill in each time a periodic task ends an execution. At this expense, the bound is obtained with a much lesser greedy algorithm. Since the overhead of the algorithm has to be included in the worst case execution time of each periodic task, it result in a lower schedulability bound for

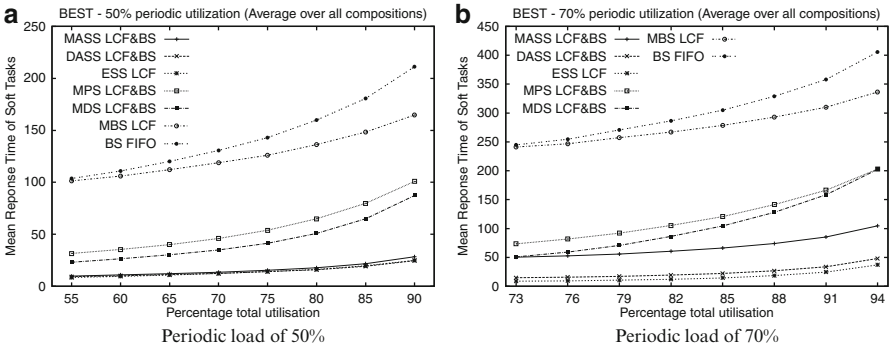


Fig. 3.10 Performances of all user-land policies compared with an exact slack-stealer (ESS) ones

systems using DASS than for systems using MASS. It is so precisely when a more accurate bound should be necessary that DASS could lead to make the system non feasible while it could had been with MASS.

3.5.4 Performances Evaluation

3.5.4.1 Simulations

Even with the limitations described above, we can show through simulations and benchmarks that the performances obtained are quite good. Figure 3.10 shows that even when traffic load is high, MASS performance are not so far from an ideal slack stealer, while this policy is not implementable. We note that when the load increase, DASS tends to deliver better response times than MASS. The same trends are observed when at equal periodic load, the number of periodic task is higher. However, the simulations do not take into account the algorithm overheads.

3.5.4.2 Executions on JamaïcaVM

The DASS and MASS implantations that we propose only differ by the added code at the beginning and at the end of each instance of each periodic task. Consequently, we have measured the time taken to execute this code with the two algorithms. To perform these measurements, we used the jamaïca virtual machine on a real-time linux free kernel 2.6.9 on an 1,22 GHz Intel(R) Pentium(R). Since the time complexity of the added code is constant or linear in the number of tasks, we have performed the measurements with the number of tasks varying between 2 and 25. Each task executes five instances. The period of each task is randomly generated in the interval $[5, 10]$ and its cost in the interval $[2, T_i]$. The first generated task has

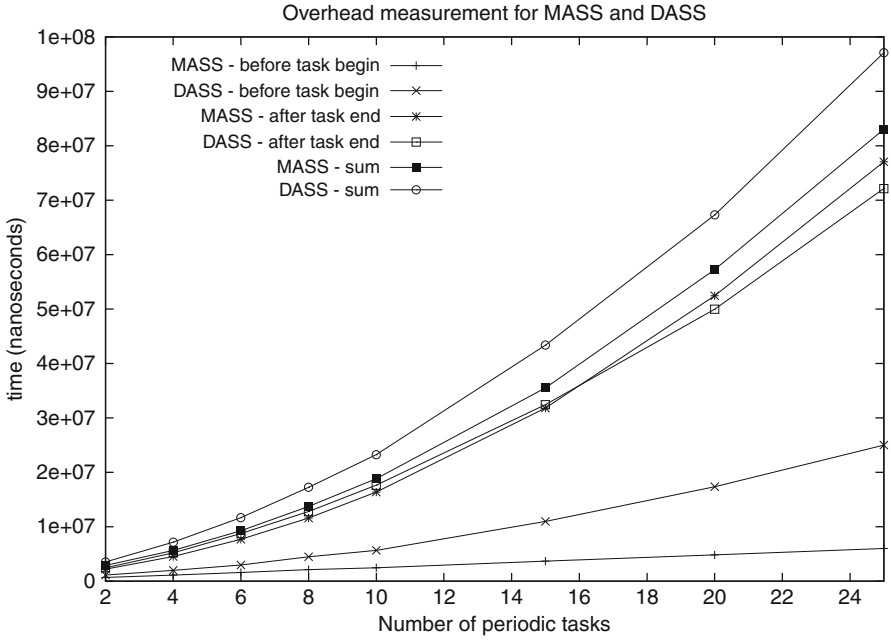


Fig. 3.11 Comparative overheads of MASS and DASS

the lowest RTSJ priority (11), the second 12 and so on. Figure 3.11 presents the obtained results. The cumulated time taken in the two methods is more important for DASS. The time passed in the added codes increases with the number of tasks and so the difference between the times obtained with the two algorithms. Even for a low number of task (2), these costs are measurable and so have to be integrated in the feasibility analysis. We can see that until ten tasks, the time passed in the added cost at the end of the instances is more important for DASS than for MASS. However this changes when there is more tasks. This is due to the fact that the code added for DASS includes more instructions for each loop passage. Even if the two algorithms have the same complexity, MASS has to loop on all the task whereas DASS only consider tasks with higher priorities. The total overheads in time for each instance is more important with DASS than with MASS. So the feasibility bound of a system will be lower with DASS than with MASS. So even if it does not perform as well as DASS, MASS can be used with more systems than DASS. This justify the use of MASS against DASS on systems with a high periodic load, even if theoretical simulations tend to show that DASS outperforms MASS. Indeed, this is precisely when DASS can be theoretically better than MASS that its overhead is more important. In practice, even if MASS offers higher response time, it results in more systems being schedulable.

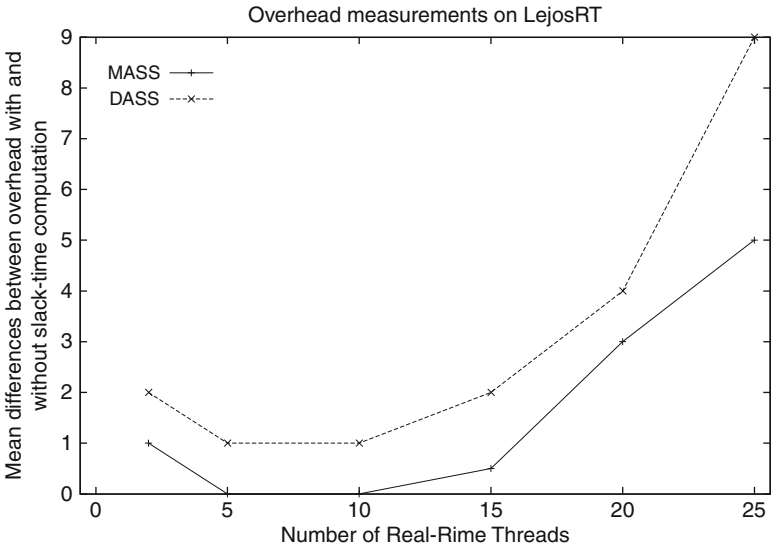


Fig. 3.12 System overheads with DASS and MASS depending on task number

3.5.4.3 Executions on lejosRT

LejosRT [244] is an open source project forked from Lejos, which is an alternative firmware for the Lego Mindstorms NXT brick. We implement DASS and MASS on the LejosRT scheduler, and perform overhead measurements. We consider a program with n real-time threads with the same release parameters (period and deadline) but with n different priorities. Moreover we shift start times in order to start first the thread with the lowest priority and last the one with the highest priority. That permits to maximize the number of preemptions. The period of the thread with the lowest priority is set shorter than the other one in order to have this thread preempted twice by each other ones. We measure the needed time to execute two instances for each thread. The system load is then maximal (100%). Figure 3.12 presents the results obtained for a range of thread number from 2 to 25. Each test was performed several times and the values reported in Y axis are the average differences between the time execution of the program on LejosRT without any slack computation and with the concerned algorithm (respectively DASS and MASS). On the X axis is reported the number of thread. We can note that the MASS overhead is always lower than the DASS one. This is the expected result, that this experiment confirmed on real implementations. Moreover, the more thread we have, the higher the measured difference is.

Many tests and simulations were performed and published in previous publication. The interested reader can refer to [265] and consult more curves on-line at [262].

3.5.5 Event Manager API and Its Relations with RTSJ API

In order to make accessible the advanced mechanisms presented in this chapter and their RTSJ implementation, we proposed an API to unified these approaches. The main goal is to allow a programmer to develop the application independently of the event manager policy, and even to easily change this policy. The sources are available on-line at [261].

3.5.5.1 Description

Figure 3.13 presents a class diagram with the proposed classes and their relationships with RTSJ classes.

AbstractUserLandEventManager

This class represent the SO which executes the *manageable* code. Since the user-land implementation design requires that there is an unique manager in the system, it is possible to use the singleton design pattern to obtain the current instance of an event manager. This abstract class provides the implementation for all common issues between servers and slack stealer in user-land: the manageable objects queuing policy, the asynchronous interruption mechanism, the BS duplication mechanism and the singleton instance of the event manager.

Two abstract subclasses are proposed: `AbstractUserLandTaskServer` and `AbstractUserLandSlackStealer`. The first should implement common issues to all task server base mechanism, in fact, all these issues are already handled by `AbstractUserLandEventManager` but we kept the class for the strong typing. The only method that remains abstract is a method `performStart()` use to start the real underlying SO (e.g. a `RealtimeThread` set with `PeriodicParameters` for a PS). The second however offers implementations for specific issues with slack stealer mechanisms: the slack-time updates and the manager wake-up. The only two methods that remain abstract are one method to initialize date structures, called at construction time, and one method to compute an approximation of the slack-time when called.

ManageableAsyncEvent/ManageableAsyncEventHandler

Based on the model of the pair `AsyncEvent/AsyncEventHandler`, we propose a pair of class `ManageableAsyncEvent/ManageableAsyncEventHandler`. A `ManageableAsyncEvent` (MAE) inherits from `AsyncEvent`, since there is no reason to justify that an `AsyncEvent` could not be associated with both regular `AsyncEventHandler` and `ManageableAsyncEven-`

Fig. 3.13 API proposal for event management

tHandler. However, a ManageableAsyncEventHandler (MAEH) is not an AsyncEventHandler, and is not even a SO, since it is destinate to be executed within an event manager, and not schedule by a scheduler.

The MAE class is easy to implement. We add a list of MAEH and we override the method fire(): the new method first call the fire() method of the super class (in order to release hypothetic regular AEH), and then call the method handleAsyncEvent() of each MAEH into the list. Finally we add a public method addManageableAsyncEventHandler() to add MAEH in the list above cited.

The MAEH object purpose is to encapsulate an interruptible object that represents the logic of the handler. When its method handleAsyncEvent() is called (by the method fire() of an associated MAE), the event manager singleton instance is retrieved and the interruptible is added to its queue.

To implement the BS duplication policy, MAEH must encapsulate an AE, an AEH and an AsynchronouslyInterruptibleException. The AE and the AEH are used to start a duplication in background and the exception to interrupt the code either of the AEH or of the MAEH. In handleAsyncEvent() method, after having added the interruptible to the manager singleton queue, the AE corresponding for the BS duplication is fired. Finally, a method that permits to interrupt the duplicated logic must be provided. Note that we alternatively could choose to inherit from AEH, but we preferred to allow the possibility of turning off the BS duplication mechanism, and in such a case, there is no reason for an MAEH to be type-compatible with an AEH.

3.5.5.2 Comparison with Other Proposals

Note that this API proposal is not in contradiction with the proposition developed in [431]: extending the semantic of the class ProcessingGroupParameters. Giving a particular subclass of ProcessingGroupParameters, say for example PollingServerPGP, then, adding this kind of PGP to a ManageableAsyncEventHandler may automate the instantiation of a PollingServer, the setting or the replacement of the current event manager singleton and the association between the MAEH and the manager.

3.6 Conclusions

This chapter has studied the problem of scheduling mixed traffic. The goal of mixed real-time scheduling is to minimize the aperiodic tasks response times, while guaranteeing the deadlines of periodic ones. We have presented two algorithmic approaches: the reservation of time slots in task servers, and the use a priori of future unused time slots by slack stealers. We called both *event managers*. We also have pointed out the fact that the RT-JVM specification does not address this issue.

There are two approaches to use these *event managers* with RTSJ: to write a scheduler or to implement their behavior within a user-land task. The first one will be dependent of the runtime context while the second will suffer from design and performance limitations. However, if it exists limitations in the design (only one manager, only servicing at the highest priority), the ones for the performances are relative in regard to the modularity and portability the user-land design facilitates. Indeed, we proposed an API that supports an event manager, and allows changes to the event manager policy, without having particular knowledge of how these advanced algorithms work.

The proposed design however suffers from a weakness: the integration within the feasibility analysis design offered by the RTSJ. Indeed, if one wants to use a mechanism such as the Deferrable Server which necessitates to adapt the feasibility analysis, one has to write a subclass of `PriorityScheduler` and override the feasibility analysis method. The feasibility can alternatively be performed through an independent package. This proposition is reinforced by the observation that an exact feasibility analysis is not always what the system designer wants. Sometimes, a simple acceptance test, or the use of a sufficient but not necessary condition are preferable. But that is another story.

Chapter 4
Parallel Real-Time Garbage Collection

Fridtjof Siebert

Abstract With the current developments in CPU implementations, it becomes obvious that ever more parallel multicore systems will be used even in embedded controllers that require real-time guarantees. When garbage collection is used in these systems, parallel and concurrent garbage collection brings important performance advantages in the average case. In a real-time system, however, guarantees on the GC's performance in the worst case are required.

This chapter explains the basic concepts for parallel real-time garbage collectors, the major real-time Java virtual machine implementations and their approach to parallel real-time memory management. Two implementations, Metronome-TS and JamaicaVM will be presented in more detail since these two represent two typical instances of the time-based and work-based approaches of running the garbage collector.

A discussion at the end of this chapter will compare the fundamental differences in the presented approaches. A final section on related work puts the presented implementations into a wider context on current garbage collection research.

4.1 Introduction

The biggest problems for the application of Java in time critical systems are caused by the use of automatic garbage collection which is used to provide safe memory management. The *garbage collector* (GC) is required since it forms the basis of the safety and security mechanisms.

F. Siebert (✉)
aicas GmbH, Haid-und-Neu-Str. 18, 76131, Karlsruhe, Germany
e-mail: siebert@aicas.com

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_4,

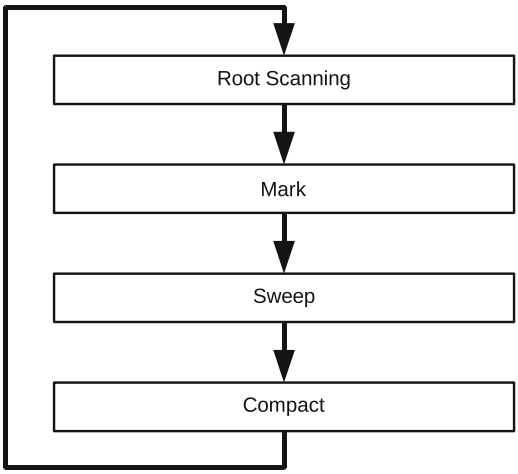


Fig. 4.1 Phases of a cyclic garbage collector

The GC automatically detects memory that cannot be referenced by the application program and hence can be freed and reused for future allocations. This provides a solution to the problem of memory leaks due to forgotten *free()*s and dangling references that can cause accesses to memory regions that were freed too early.

4.1.1 Garbage Collector Phases

The garbage collector in a Java system is typically implemented as a cyclic process that consists of several phases. This process is illustrated in Fig. 4.1. The four phases of a *mark-sweep-compact* garbage collector are explained in the following paragraphs.

GC-Phase 1: root scanning

In this phase, all objects on the Java heap that are referred to by references stored outside of the heap, e.g., references in local variables, are marked, such that their memory will not be reclaimed during the current garbage collection cycle.

GC-Phase 2: mark

Once all objects referenced from outside of the heap have been marked, the mark phase continues to mark objects that have not been marked yet and that are referenced by already marked objects. Consequently, a wave of marked objects advances through the heap during the mark phase until there are no remaining unmarked objects referenced by an already marked object.

GC-Phase 3: sweep

Once the mark phase has finished, no unmarked objects are referenced by any marked objects, hence no unmarked objects can be reached through any sequence of references originating in the root references. The unmarked objects can no more be accessed by the application and their memory can be freed.

During the sweep phase, the memory of the remaining unmarked objects is consequently added to the free list.

GC-Phase 4: compact

After the sweep phase, the memory of unused objects has been freed. However, the released memory might be scattered through the heap in small chunks that cannot be used for larger allocation requests. It is hence necessary to defragment, i.e., to move all allocated memory such that the free memory forms a contiguous range available to new allocations of objects of arbitrary sizes.

4.1.2 Incremental GC

The basic incremental mark- and sweep GC algorithm that enables interleaving of GC work with application work and hence control GC pauses was described by Dijkstra et al. in 1976 [137]. This algorithm maintains three sets of objects that are distinguished by the colours *white*, *grey* and *black*. A GC cycle starts with all objects being *white*, and the objects reachable from root pointers are marked *grey*. It then proceeds with the mark phase as long as there are *grey* objects by taking a *grey* object *o*, marking all *white* objects referenced by *o grey*, and marking *o* itself *black*. A write-barrier ensures that modifications of the object graph performed by the application will not result in the GC missing reachable objects. When the *grey* set is empty, all *white* objects are known to be garbage, so their memory will be reclaimed in the following sweep phase. In this phase, *black* objects are converted back to *white*, such that the next cycle can start with all allocated objects being *white*.

4.1.3 GC Execution

The whole garbage collection process needs to be executed while the actual Java application is running. The garbage collector is hence typically run as a separate thread, but an incremental garbage collection might also be performed within the application threads.

4.1.4 Real-Time GC on Parallel Systems

Implementing a garbage collector for Java on a current parallel system such that it does not prevent the Java application from performing real-time tasks with predictable timing is particularly hard. The rest of this chapter will present the basic implementation choices and will give details on some existing parallel real-time Java VM implementations.

4.2 Parallel GC Basics

Parallel garbage collection can refert to very different approaches and solutions to the main problems. Before presenting actual implementations in the following chapter, this chapter will define terminology to classify parallel GCs and then present fundamental problems that a parallel GC has to address.

4.2.1 Terminology

In the memory management community, the term *real-time* has often been used in a way that is much more relaxed than its use in the real-time community. In the real-time community, a real-time system is a system that has hard deadlines that must not be missed. It is in this strict sense that real-time is understood within this paper here: proven upper bounds on worst-case performance are needed.

The literature categorises GCs as incremental, concurrent or parallel. These categories are not disjoint, e.g., a parallel garbage collector may also be concurrent.

An *incremental* GC (Fig. 4.2) is a GC that can operate incrementally, i.e., it can perform a full garbage collection cycle in parts and allow the application to perform work in between these incremental garbage collection steps. A *concurrent* GC (Fig. 4.3) can operate in parallel to the application. A concurrent garbage collector must therefore be incremental to an extreme extent such that only very simple primitive operations, such as *compare-and-swap*, must be atomic with respect to the application. Finally, a *parallel* GC (Fig. 4.4) is a GC that can use several processors simultaneously to perform its work in parallel. A parallel GC may be non-concurrent, i.e., it may prevent the application from running concurrently with the garbage collector. Also, a concurrent GC can be non-parallel, i.e., it could be restricted to a single processor that performs garbage collection work sequentially, but concurrently with the application. A parallel and concurrent GC is illustrated in Fig. 4.5, it uses several CPUs in parallel while it runs concurrently with the application.

The most flexible approach to parallel garbage collection is a parallel incremental GC as illustrated in Fig. 4.6. Here, the GC work is split into very small incremental

Fig. 4.2 An incremental garbage collector intertwines GC work (*dark*) and application work (*light*)

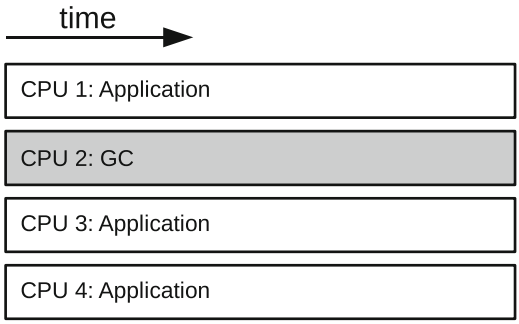


Fig. 4.3 A concurrent garbage collector uses one or several CPUs dedicated to GC work (*CPU 2*), while other CPUs are available for concurrent application work (*CPU 1, CPU 3*, and *CPU 4*)

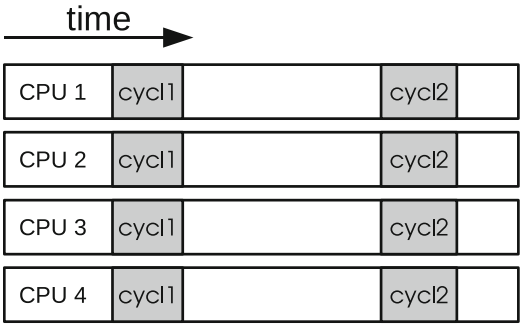


Fig. 4.4 A parallel garbage collector: All available CPUs perform a garbage collection cycle working in parallel (*cycl1* and *cycl2*), while the application is stopped

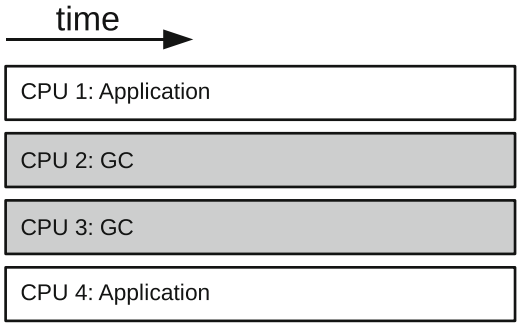


Fig. 4.5 A parallel concurrent garbage collector with two CPUs dedicated to parallel GC work (*CPU 2* and *CPU 3*) that is performed concurrently with the application code running on two different CPUs (*CPU 1* and *CPU 4*)

steps that may execute concurrently with the application, parallel to other incremental GC steps on other CPUs and intertwined with application code on the same CPU.

For a GC to be real-time, it is required that the performance of the application threads is predictable, in particular, there must not be any unbounded pre-emption due to GC work and the GC must ensure that it reclaims memory fast enough to keep up with the application's allocation requests such that the application never needs to wait on an allocation for the GC to free sufficient memory.

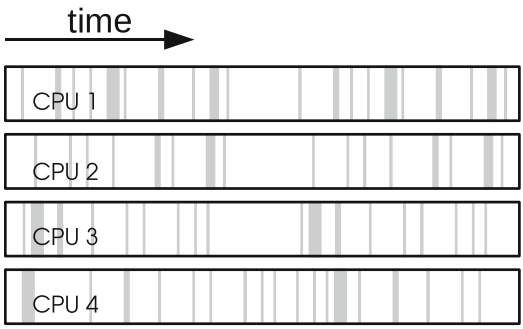


Fig. 4.6 A parallel incremental garbage collector

4.2.2 GC Scheduling

There are two main approaches to schedule the work of an incremental GC. One approach is work-based, i.e., for every allocation performed by the application, some garbage collection work will be performed in a way to ensure that sufficient GC progress is made to satisfy all allocation requests. Alternatively, time based GCs need to be scheduled such that the thread or threads assigned to perform garbage collection work receive enough CPU time to keep up with the worst case allocation rate of the application. Work-based GCs are easier to configure since they do not require an analysis of the application's worst-case allocation rate, while time based GCs permit faster memory allocation since no GC work is required at allocation time.

4.2.3 Dealing with Fragmentation

A particularly difficult aspect of a parallel *Real-Time Garbage Collector* (RTGC) is to ensure that memory fragmentation will not result in failure to allocate new objects. The classic approach to defragment the heap by a compaction phase typically requires stopping all application code, which is not an option in a real-time system. Since Java objects can be arrays of arbitrary sizes, even moving a single object in an atomic step may cause long pauses.

Current parallel RTGC implementations use very different approaches to deal with fragmentation. The main techniques are as follows:

4.2.3.1 Ignoring Fragmentation Issues

The chances that fragmentation actually leads to significant memory loss are relatively low in practice [229]. Hence, a simple solution is to plainly ignore fragmentation issues and not defragment the heap. As long as the application's allocation patterns are simple enough, e.g., only objects of a particular kind are allocated, fragmentation will not cause problems.

However, this approach is very dangerous: for non-trivial applications, no upper bound for the memory lost due to fragmentation can be given.

4.2.3.2 Incremental Compaction

Defragmenting the heap incrementally reduces the pause times caused by the compaction phase. If the maximum array and object size is limited, an upper bound for atomically moving single objects can be given, making this applicable in real-time systems.

4.2.3.3 Concurrent Defragmentation

There exist implementations that use incremental moving of objects, while specific read- and write-barries in the application code ensure that concurrent accesses to a partially moved object will not result in corrupted data. One possibility is to use forwarding pointers [83] in each object that points to the new copy of an object while it is moved, and that points to the object itself while there exists only one copy. A write-barrier then requires updating the object and the copy reachable through the forwarding pointer whenever a modification to the object state is made.

On a parallel system, ensuring that all possible interleavings of accesses to a partially moved object always result in a consistent view of the object is difficult and requires additional mechanisms.

4.2.3.4 Arraylets

The problematic aspect when defragmenting the heap are arrays since they can be of arbitrary sizes, while “normal” Java objects are typically very small. Moving these small “normal” Java objects atomically will introduce only small pauses, but moving arrays has to be avoided.

The idea of Arraylets [23] is to represent arrays non-contiguously in memory, but instead, to split them up into Arraylets that are small enough to be moved incrementally. Arraylets have a typical size of a few kBytes. Array accesses then need to go through one or two additional indirections before the actual array data can be accessed.

4.2.3.5 Fixed-Size Blocks

The use of very small fixed-size blocks, typically 32 bytes per block, to represent all Java objects is the most extreme approach that fully avoids the need to defragment the heap [370]. Instead, every object and array is built as a graph of these fixed-size blocks.

The representation of larger objects and arrays as such graphs has to be chosen carefully since it is performance critical. Java objects are typically very small such that for most Java objects, only one or two blocks are required to store their fields. A singly linked list of blocks is an efficient representation for these objects. In contrast, arrays may be very large. For arrays, a tree representation with a large branching factor (e.g., eight references in every inner node) enables fast traversal in $log_8(l)$ memory accesses for an array of length l.

Since fixed-size blocks completely avoid the need to move objects, no provisions to deal with changing object addresses, e.g. an additional redirection through handles or updating of object addresses, need to be made. The garbage collector itself is simplified significantly, not only since no compaction phase is needed, but also since the small blocks provide a natural unit of garbage collection work: One step of garbage collection work is the marking or sweeping of a single block. The garbage collector itself does not need to know about the graph structure of Java objects, only the structures of single blocks need to be exposed to the GC.

4.2.4 Theoretical Limitations

To make efficient use of the processors available to a parallel garbage collector, the garbage collection task must be parallelisable. An important part of the implementation is the load-balancing that ensures that no processors assigned to perform GC work are idle.

Unfortunately, there are theoretical limits to the parallelisation of a mark-and-sweep garbage collector [374]. The theoretical limit to the parallel execution of the mark phase depends on the shape of the object graph on the heap. The reason for this is that linear data structures such as lists cannot be traversed in parallel. In the extreme case of a heap containing only a singly linked list of objects, any parallel mark-and-sweep garbage collector will be forced to perform a non-parallel mark phase traversing this list.

In general, the maximum depth of the object graph gives the limit on the parallisation of the garbage collector. In case of a large linked list, the depth of the heap corresponds to the length of this list. The length of such a linked list is limited by this depth. To ensure good parallel garbage collector work, the application must ensure that the depth of the object graph is limited and small compared to the total heap size.

4.3 Parallel GC Implementations

4.3.1 IBM J9 with Metronome-TS

The real-time J9 virtual machine by IBM uses a time-based scheduling approach called Metronome [21, 23]. For parallel systems, this approach was extended in

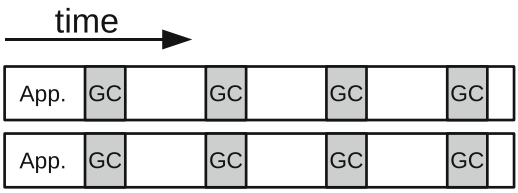


Fig. 4.7 The Metronome garbage collector has fixed time slots for GC work

the Metronome-TS garbage collector presented in 2008 [22]. The basis of this technology is a scheduling mechanism for garbage collection called tax-and-spend.

Tax-and-spend provides a general mechanism which works well across a variety of application, machine, and operating system configurations. Tax-and-spend subsumes the predominant pre-existing RTGC scheduling techniques. It allows different policies to be applied in different contexts depending on the needs of the application. Virtual machines can co-exist compositionally on a single machine.

4.3.1.1 Metronome

The basis of the Metronome-TS collector is the Metronome collector. Metronome is an incremental garbage collector that is somewhat parallel. It is run in pre-reserved time slots as shown in Fig. 4.7. This collector is incremental and partially parallel, but it is not concurrent.

Metronome is a mark-and-sweep garbage collector that does not perform compaction for large objects since moving a large object may not be possible within the time allocated for one incremental step of GC work. Instead, Metronome splits large arrays up into Arraylets, i.e., a large array is no longer contiguous in memory. Instead, one or several additional indirections are required when accessing an array to access the corresponding arraylet. The size of an arraylet may still be relatively large, it is limited by the length of an incremental garbage collection step, which is typically in the millisecond range.

4.3.1.2 Metronome-TS

The tax-and-spend scheduling requires a garbage collector that is both incremental and concurrent, i.e., it falls into the parallel and incremental category shown in Fig. 4.6. This category permits a mixture of dedicated CPUs for garbage collection and interleaved garbage collection and application work on other CPUs as shown in Fig. 4.8. The granularity for the incremental steps of this collector is down to 200 μs.

The Metronome-TS garbage collector automatically starts stealing application CPU time when the system load is so high that background GC work would not be sufficient. For this, application threads regularly check if GC work is required and preempt allocations or program code at safe points to perform garbage collection

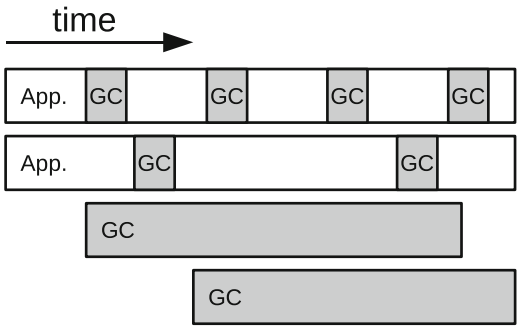


Fig. 4.8 The Metronome-TS garbage collector may combine incremental GC work and concurrent GC work

work. Safe points in the program code are also used to perform specific tasks such as stack scanning or synchronization actions via a callback into the threads at the corresponding GC phase. Priority-boosting is used in case waiting for such a callback to be performed results in a priority inversion.

Metronome-TS uses a snapshot-at-beginning write barrier [444] to simplify consistency during the mark phase and detection of the end of the mark phase. Phase changes are performed by the last-man-out algorithm. This means, that a global counter for the number of active worker threads is updated whenever a thread starts or finishes a quantum of GC work. If a thread finishes its mark phase activity, and there is no GC work left and the counter of worker threads drops to zero, this last thread will perform the phase change.

4.3.2 Fiji

Pizlo et al. recently presented different approaches for concurrent defragmenting RTGCs [299]: Their CHICKEN collector aborts moving an object in case of a concurrent modification, while their CLOVER collector detects writes by using a marker value for fields of objects that are obsolete when they have been moved. Their fine-grained synchronization results in µs response times.

An interesting approach to deal with fragmentation was presented by Pizlo in 2010 [300]: To reduce the overhead for memory defragmentation, arrays are built out of spines and small, fixed-size blocks that contain the array element data. Any access to an array element has to use an additional indirection through the spine. Defragmentation of the fixed-size blocks used for array element data is not required, instead, defragmentation is required only for the spines. Moving the spines, however, is much easier compared to moving the whole arrays since the spines are immutable once an array has been allocated and the amount of memory used for spines is significantly smaller than the total memory of the arrays.

4.3.3 SUN/Oracle RTS

The Sun Java Real-Time System (Java RTS) is Sun's commercial implementation of the Real-Time Specification for Java (JSR-001). It is based on the concurrent HotSpot VM.

The parallel garbage collector employed by Java RTS is a fine-grained concurrent collector. The approach taken to handle fragmentation is that fragmentation issues are largely ignored: no heap compaction is performed. To reduce chances of high memory loss due to fragmentation, heuristics on the selection of a suitable memory range taken from the free lists are used.

Consequently, the parallel garbage collector can give good real-time performance, but it suffers from a high risks that memory allocations may fail unpredictably due to fragmentation.

4.3.4 Atego AonixPerc-Ultra-SMP

AonixPerc-Ultra-SMP is an SMP version of Atego AonixPerc-Ultra [14]. It uses a time-based incremental garbage collector that has been extended to take advantage of the available CPUs. Typical response time of 1 ms are claimed. The concurrent GC in AonixPerc-Ultra-SMP performs collection of unused objects by multiple processors while Java application threads continue to operate concurrently. This enhances the ability of the GC to pace the garbage collection rate to the applications memory allocation rate.

4.3.5 JamaicaVM Parallel RTGC

The JamaicaVM parallel RTGC is a generalization of a fine-grain single-CPU RTGC. This section first describes the basis single-CPU GC before it explains the parallel version.

4.3.5.1 The JamaicaVM Single-CPU RTGC

JamaicaVM is based on an incremental mark and sweep collector as described by Dijkstra et al. in 1976 [137]. The scheduling of Java threads is limited to *synchronisation points* that are added automatically by the VM [369]. The GC does not deal with Java objects or Java arrays directly and does not know about their structure. Instead, it works on single fixed-size blocks.[1] [370]

[1]The typical size of these blocks is 32 bytes or eight 32-bit machine words.

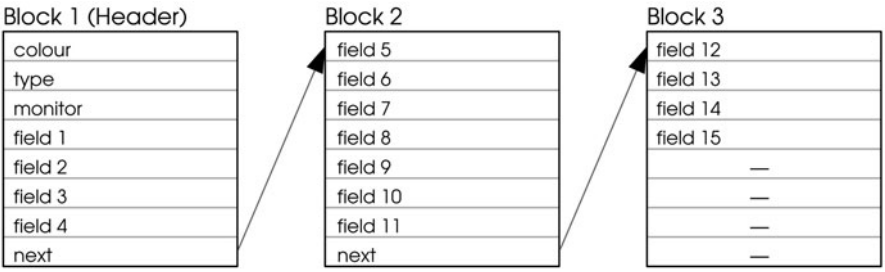


Fig. 4.9 Java object implemented as list of fixed-size blocks. The blocks are linked via a field *next*

Java objects or arrays that do not fit into a single fixed-size block are represented as a graph of fixed-size blocks that may be non-contiguous in memory. Figure 4.9 illustrates the structure of a Java object with 15 words of field data that is spread over three blocks. Using this object layout avoids the need to defragment the heap and therefore simplifies the implementation of the RTGC significantly. The GC no longer needs to move objects around, update references and deal with changing object addresses.

The price for this simplification is additional overhead when accessing fields of objects that are spread over several blocks. Arrays need to be implemented as trees, which leads to an access time that is logarithmic in the array size but small and limited in practice.

Constant Time Root Scanning

Scanning the stacks to find references typically requires stopping the corresponding thread, which can cause pauses that are too long for time-critical applications.

The solution employed by JamaicaVM that was originally published in [371] is to ensure that all root references that exist have to be present on the heap as well whenever the GC might become active. The compiler and VM generate additional code to store locally used references to a separate root array on the heap to ensure this.

Mark and Sweep Steps

With the elimination of an explicit root scanning phase and the lack of a need for a compaction phase due to the object model based on fixed-size blocks, the GC work is reduced to the mark and the sweep phases. These phases work on single blocks, i.e., one step in the mark phase means scanning the reference of one *grey* block and one step in the sweep phase means checking one block's colour and reclaiming its memory in case it is *white*.

Consequently, one GC cycle consists of very small and uniform incremental mark or sweep steps. The total number of mark and sweep steps is limited by the amount of allocated memory (free memory ranges are ignored by the sweep phase).

4.3.5.2 Towards a Parallel Collector

The existing single-CPU GC has been extended to enable parallel and concurrent execution [375]. This means, first, that several application threads may execute in parallel and perform memory related operations such as memory accesses, write barrier code or memory allocation in parallel. Also, it means that one or several threads may perform GC work in parallel to the mutator threads. The GC work must still be divisible into small and uniform steps to be able to measure the amount of GC work performed if such a collector is to be employed as a work-based collector. Finally, the number of work steps required to finish a complete cycle must be limited.

Scheduler and Synchronisation Primitives

As for the single-CPU VM, the parallel VM uses synchronisation points to restrict thread switches. However, the number of *RUNNING* threads has been increased to a run-time constant n, which must be less than or equal to the number of logical hardware CPUs available on the system. Any thread that is not in state *RUNNING* is stopped at a synchronisation point, i.e., thread switches are restricted to these points. However, all threads that are in state *RUNNING* may execute completely in parallel. For OSes that provide APIs to select CPU affinity, these APIs are used to enforce that each *RUNNING* thread runs on a distinct CPU. The underlying OS scheduler may, however, pre-empt a *RUNNING* Java thread to execute threads not attached to the VM.

Having control over the set of *RUNNING* threads makes it possible to assign a CPU-local data structure to each *RUNNING* thread. This CPU-local data structure can be accessed by the thread that is running on this CPU exclusively without synchronisation with other threads as long as no synchronisation point is executed. Different parts of the parallel GC implementation make use of these CPU-local data structures. In particular there are CPU-local *grey* sets, CPU-local free lists, and CPU-local sweep sets.

Colour Encoding

Any block on the heap needs to be marked with a marking colour, *white*, *grey*, or *black*, such that all blocks are grouped in three disjoint sets of *white*, *grey*, and *black* blocks. Important operations on blocks are shading of *white* blocks (as required by

the write-barrier or during the mark phase), obtaining one *grey* block from the grey set (required to perform a mark step on this block), and determining that the *grey* set is empty (which means that the mark phase is finished).

Similar to the single-CPU GC, one word per block is reserved for the colour such that *grey* blocks can be stored in a singly linked list. For parallel mutators and a parallel GC, using only one single list for the *grey* set, however, would result in an important bottleneck since write barrier code and mark phase work would access this list in parallel.

Instead, it was chosen to maintain several CPU-local linked list such that each one represents a subset of the *grey* blocks. Since each CPU is allowed to access only its local *grey* list, no synchronisation is required here. However, it has to be ensured that several CPUs that shade a block in parallel will not result in an inconsistent state.

Write-Barrier Code

The single-CPU collector uses an incremental-update write-barrier, i.e., whenever a reference to a *white* object *a* is stored into another object, *a* is added to the *grey* set [294]. This write barrier has the advantage that memory that becomes unreachable during a GC cycle might be reclaimed during this cycle, such that memory may be reclaimed early and less memory needs to be marked.

However, this approach is not feasible when several mutators run in parallel. One mutator *m1* might read a reference to an object *b* from the heap, and another mutator *m2* might overwrite this reference before *b* was added to the *grey* set. Then, *b* could only be found to be reachable from the roots of *m1* by re-scanning these roots. Rescanning the roots at the end of the mark phase, however, makes termination of the mark phase difficult.

Therefore, the incremental-update write-barrier was replaced by a snapshot-at-beginning write-barrier [444] that shades deleted references that refer to *white* objects. This write-barrier ensures the weak tricolour invariant (see [294]):

> All white objects pointed to by a black object are reachable from some *grey* object through a chain of white objects

Using this snapshot-at-beginning barrier ensures that all memory reachable at the beginning of a GC cycle and all memory allocated during the cycle will survive the current GC cycle. Compared to the original approach, some objects may therefore be reclaimed later, but the worst-case behaviour is the same since the incremental-update write-barrier gives no guarantee that objects that become unreachable will be reclaimed during the current cycle.

Free Lists

A central aspect of a parallel GC and a parallel memory allocation routine is the data structure used to store free memory. Since JamaicaVM uses graphs of fixed-size

blocks to represent objects of arbitrary sizes, there is no need to distinguish different size classes of free objects. An efficient data structure to hold the set of available fixed-size blocks is sufficient.

The operations required on the set of free blocks are allocation of a single block by a mutator thread and the addition of a block during the GC's sweep phase (see Sect. 4.3.5.3). It has to be possible for several threads running on different CPUs to perform these operations in parallel.

To enable parallel access, CPU-local free lists are used. Allocating a block from the CPU-local free list or adding a block to this list can be performed without any synchronisation with other CPUs. The maximum number of blocks stored in a CPU-local free list is limited by a constant *MAX_FREELIST_SIZE*.[2] Whenever a block is allocated and the CPU-local free list is empty, or when a block is freed and the CPU-local free list contains this maximum number of elements, one set of such objects is filled from or spilled to a global list.[3] This global list uses synchronisation via compare-and-swap and a retry in case the operation failed.

4.3.5.3 Phases of the GC Cycle

This section describes the phases of one cycle of the parallel RTGC. All CPUs that perform GC work are always in the same phase; the value of a global variable *gcPhase* gives the current phase. A GC cycle always starts with a *flip*, which is followed by the *mark* and *sweep* phases. For Java, a *finalization* phase is also required to take care of Java objects with finalizers.

It has to be ensured that several CPUs can perform single steps simultaneously in each phase, but it also has to be ensured that phase changes will be performed properly even if several CPUs simultaneously detect that the current GC phase is finished and a new phase should start. Therefore, compare-and-swap is used to perform the phase changes. Only one CPU will be successful in performing the phase change, any other CPU that is not successful performing a phase change can continue with the GC work for the new phase, no retry is required.

GC Cycle Start: *flip*

The start of each new GC cycle happens after the last cycle's sweep phase has finished. So, a thread that has detected that the sweep phase has finished will initiate the start of a new cycle. The initial GC phase is the *flip*. It is a very short operation that is performed by the first CPU that succeeds on the compare-and-swap operation that changes the phase from *sweep* to *flip*.

---

[2] In the current implementation, this constant is set to 64 blocks.

[3] Repeated freeing and allocating of one block by one CPU could cause repeated fills for an empty CPU-local set followed by spills of the just filled set. To avoid this, two CPU-local free lists are used, one only for freeing and the other one for allocating.

At the beginning of a GC phase, all allocated objects are marked *black*, there are no *grey* or *white* objects. The single root object will be marked *grey* and a 'flip' will be performed, i.e., the meanings of *black* and *white* will be exchanged. However, before the mark phase can start, it has to be ensured that all CPUs are aware of the fact that the meanings of *black* and *white* have been exchanged. Therefore, the thread that performs the flip will wait for all other CPUs to execute the next synchronization point. Only then the mark phase will start.

Mark Phase

One step during the mark phase is simple: A *grey* block b needs to be obtained using the *getGrey* function. Then, all blocks referenced from b that are still *white* must be shaded using the *shade* function and b itself must be marked *black*.

Problematic is the case if *getGrey* fails to return a *grey* block. This might mean either that the *grey* set is empty and the mark phase is finished, or it might mean that there are temporarily not enough elements in the *grey* set to allow all CPUs that are assigned to perform GC work to scan blocks in parallel. The current CPU can make no progress in this case, it can only stall until the *grey* set contains sufficient elements again. The number of these stalls during the mark phase is limited by $(n-1) \cdot d$ on a system with n CPUs performing mark work in parallel and for a heap depth of d [374]. The heap depth d is the maximum distance of any reachable block on the heap from the root reference of the heap graph. d depends on the application, a parallel mark phase is possible only for mutators that create a heap graph with $d \ll heapsize$.

It is important to distinguish the case of a stall from the case that the mark phase is finished. Therefore, after each stall, it is checked whether any other CPU performed any shading. If no other CPU shaded any new blocks, then the *grey* set must be empty and the mark phase is finished.

Sweep Phase

The purpose of the sweep phase is to add all blocks that remained *white* after the mark phase to the free lists. To enable parallel sweeping, the heap is partitioned into *sweep sets*[4] of s contiguous blocks. A CPU that performs sweep work will exclusively work on its CPU-local sweep set, such that no particular synchronisation with other CPUs that work on different sweep sets is required.

Only when the CPU-local sweep set is empty, the next set will be taken from a global pool of memory ranges that need to be swept. Accesses to this global pool are performed using compare-and-swap, and a retry is required in case the compare-and-swap failed. The number of retries on an n processor system, however, is limited by $n-1$ if the sweep set size s is large enough.

[4] In the current implementation, one sweep set has 128 blocks.

The end of the sweep phase is reached when the global pool of sweep sets is exhausted. However, at this point, some CPUs might still perform sweep steps on their CPU-local sweep sets. The GC phase cannot terminate before all CPU-local sweep sets have been fully processed. Therefore, at this last phase of the sweep phase, any CPU that has to perform GC work but has an empty local sweep set will perform sweep steps for other CPUs that are not done with their CPU-local sweep set yet. The code to increment the position in the current sweep set uses a compare-and-swap operation, but no retry is required since the counter can only increase and a failed compare-and-swap means that some other CPU has increased the counter already.

Finalisation Phase

The finalisation phase is required in a GC for Java to ensure that unreachable objects that define a *finalize* method will be scheduled for finalisation. The JamaicaVM GC dedicates a specific phase after the mark phase to finalisation. The single-CPU GC keeps a doubly-linked list of objects that need finalisation and, during the finalisation phase, the GC iterates over the list. For every node with an object that became unreachable, that object will be added to a list of finalisable objects that are dealt with by the finalizer thread.

The parallel GC cannot simply iterate and modify such a list. Instead, each CPU that performs finalisation work will obtain exclusive access to one element in the list using a single compare-and-swap operation. If this compare-and-swap was successful, the object will be added to the list of reachable objects that will be checked again in the next GC or the list of objects that require finalisation.

4.4 Discussion

The main differences in the approaches taken by the different real-time garbage collectors for parallel systems lie in the granularities of garbage collector work, in the way fragmentation is dealt with and in the technique used to schedule the garbage collection work.

4.4.1 Garbage Collector Work Granularity

There are extreme differences in the granularity of garbage collection work. The most expensive single step of work that needs to be performed by the garbage collector defines the granularity of garbage collector work. It is at this granularity that garbage collector work can be scheduled, this granularity defines how fine-grained switches between garbage collection and application work can be.

Typically, the most expensive single steps are moving a large object (in a compacting garbage collector) or scanning the stack of a thread that is in a deep recursion. Collectors that do not move objects or that scan stacks incrementally can therefore operate at a finer granularity. Typical values for the granularity range from less than 1 µs (e.g., JamaicaVM), via hundreds of µs (e.g., IBM Metronome) to hundreds for ms for non-realtime GCs.

4.4.2 Fragmentation

There are fundamentally different approaches to dealing with fragmentation. Using fixed-size blocks and no memory compaction as used in JamaicaVM is one extreme. This approach requires some overhead when accessing arrays, while it fully avoids heap compaction and all problems that moving objects may cause.

Partially compacting schemes such as IBM's Metronome or FijiVM avoid compaction where it would harm real-time performance.

Approaches that ignore fragmentation such as Java RTS may achieve good real-time behaviour, but cannot guarantee that an application will not fail allocating memory due to fragmentation. Such a system is not acceptable in any environment that requires software to be reliable.

4.4.3 Garbage Collector Scheduling

Garbage Collector scheduling is about deciding when to run the garbage collector. The scheduling can only be done at the granularity of the garbage collection work described above.

In a time-based scheme, time slots are allocated for garbage collection work such that sufficient garbage collection progress is guaranteed to keep up with the application's allocation rate and memory usage. An important feature of the time-based approach is that it can guarantee that a given percentage of the CPU time is available to the applications (MMU, minimum mutator utilization).

The work-based approach instead couples garbage collection work with allocation. Consequently, some garbage collection work is done whenever memory is allocated. This approach cannot guarantee a certain MMU since a burst of memory allocation will result in a burst of garbage collection work. However, a burst of allocation in one thread does not inhibit other threads (e.g., at higher priorities or on other CPUs) from making progress. The work-based approach automatically punishes a thread that performs memory allocation by requiring this thread to pay for the allocation by doing garbage collection work. Threads that perform little or no memory allocation consequently only perform little or no garbage collection work.

For the correct configuration of a time-based approach, it is required to know the maximum memory usage[5] and the maximum allocation rate in units such as MBytes per second (see Chap. 6 on regions for techniques to determine the maximum memory usage). With these values, the heap size and the garbage collector time slots can be configured to ensure that the garbage collector will always catch up with the application's memory demand.

The configuration in a work-based scheme is somewhat simpler: The allocation rate is not required since the work-based scheduling of garbage collection work automatically results in more garbage collection work during times with a high allocation rate. Consequently, all that is required is knowledge about the maximum heap usage of the application. With this, the heap size can be set such that the amount of garbage collection work required for an allocation is acceptable, while larger heap sizes result in less garbage collection work per unit of memory allocated.

On a multi-core system, combinations of time-based and work-based schemes may make sense: CPUs that would otherwise be idle could be used to perform continuous garbage collection work. As long as this garbage collection work is sufficient for the current memory usage and allocation rate of the application, no additional garbage collection work on allocation is required. Only in case it is not sufficient, the work-based approach could be used as a fallback.

4.5 Related Work

Early work on concurrent and incremental mark and sweep garbage collection dates back to the seventies with important contributions from Dijkstra [137] and Steele [384]. The first incremental copying collector was presented by Baker [24].

One of the earliest implementations of an incremental and parallel GC is the Multilisp implementation by Halstead [184]. This is a parallel version of Baker's copying garbage collector that uses processor-local old-spaces and new-spaces to enable parallel garbage collection work and parallel memory allocation. However, this approach did not address the problem of load balancing at all. Load balancing between different mutator processors was proposed by Endo [148]. In Endo's approach, each process maintains a local subset of the *grey* set. These local subsets are divided into two subsets: a local mark stack and a stealable mark set. When a processor would stall due to empty local mark sets, it will attempt to steal *grey* objects from another processor's stealable mark set.

Attanasio et al. made a comparison of the scalability of different parallel GC implementations (generational, non-generational, mark-and-sweep, semi-space copying) [20].

Endo, Taura and Yonezawa [149] presented an approach to predict the scalability of parallel garbage collection in the average case. Their approach takes the memory

[5] Amount of reachable memory.

graph and specifics of the hardware such as cache misses into account. Their result is an estimate of the scalability. More recent work gives an upper bound for the worst-case scalability of a parallel marking garbage collector [374].

Blelloch and Cheng have presented a theoretical bound on time and space for parallel garbage collection [59] and refined their approach [100] to become practically implementable by removing the fine granularity of the original algorithm and adding support for stack scanning, etc. The original scanning of fields one at a time prevented parallel execution. Their new approach scans one object at a time, resulting in parallel scanning of objects. Since scanning of large objects would prevent parallelism, the authors split up larger objects into segments that can be scanned in parallel.

A parallel, non-concurrent GC with load balancing via *work stealing* was presented by Flood et al. [162]. Each processor has a fixed-size work-stealing queue. If a processor's work-stealing queue overflows, part of its content is dumped to a global overflow set. Processors with an empty local queue first try to obtain work from the overflow set before they resort to stealing. This technique was then applied to parallel copying and mark-compact GCs resulting in a speedup factor between four and five on an eight processor machine.

The parallel, incremental and concurrent GC presented by Ossia et al. [290] employs a low-overhead work packet mechanism to ensure load balancing for parallel marking. In contrast to previous balancing work, all processors compete fairly for the marking work, i.e., there is no preference for a processor to first work on the work packets it generated.

Other fully concurrent on-the-fly mark-and-sweep collectors have been presented by Doligez and Leroy [141], Doligez and Gonthier [140], and Domani et al. [142, 143].

The possibility to implement the *grey* set by reserving one word per object such that all *grey* objects could be stored in a linked list as published earlier [298, 369].

A different approach to parallel garbage collection was presented by Huelsbergen and Winterbottom [212]. Their approach is basically a concurrent mark-and-sweep GC in which the mark and the sweep phases run in parallel. However, the mark phase itself is not parallel in this approach, so it does not suffer from the stalls discussed in this paper.

Being able to give an upper bound of the scalability of the garbage collector enables one to give an upper bound on the total work required to perform one garbage collection cycle. This work can then be used to schedule the GC such that it reclaims memory fast enough to satisfy all allocation requests. There are two main approaches to schedule the GC work: work-based scheduling [24, 368], or time-based approaches [23].

Performing GC work in a work-based scheme or a time-based scheme based on the allocation rate of an application was presented by Baker [24] and Henriksson [190], respectively. Schoeberl schedules the GC as an ordinary application thread [350].

Pizlo et al. recently presented different approaches for concurrent defragmenting real-time GCs [299] by either aborting moving an object in case of a concurrent

modification or by using a marker value to detect reads for obsolete fields. A non-blocking real-time GC implemented in hardware was presented recently by Schoeberl and Puffitsch [360].

4.6 Conclusions

Maintaining real-time guarantees in a garbage collected system is hard. The main user-visible difference between current implementations are in the granularity of GC work, in the way fragmentation is dealt with, and in the way the garbage collection work is scheduled.

A reliable implementation must be able to give real-time guarantees under all circumstances. This is not the case for all current implementations, for example implementations that do not limit the memory lost due to fragmentation can lead to unpredictable allocation failure.

The task of scheduling the garbage collector work become more complex on a multi-core system due to the fact that there are more possibilities. It has to be seen for which applications an explicit integration of GC work in the scheduling analysis brings advantages over a work based scheme that does require special care during scheduling analysis apart from the fact that allocation time includes GC work.

Chapter 5
Region-Based Memory Management: An Evaluation of Its Support in RTSJ

M. Teresa Higuera-Toledano, Sergio Yovine, and Diego Garbervetsky

Abstract Memory management is a source of unpredictability in the execution time of Java programs. This is because garbage collection introduces possibly unbounded blocking pauses to threads, which is unacceptable in real-time systems. To cope with this problem, the *Real-Time Specification for Java* (RTSJ) adopts a region-based approach which relies on memory scopes that are automatically freed upon termination of their lifetime. This allows the turning off of garbage collection during the execution of critical tasks, thus ensuring real-time guarantees. This chapter explains the RTSJ memory model and proposes improvements to its suggested implementation. It also discusses a static analysis-based approach to ensure that memory scopes are correctly use and dimensioned at compile time.

5.1 Introduction

Embedded real-time systems must be reliable and predictable, and should behave deterministically even when the environment changes. Thus, such software systems have been typically programmed in assembly, C or Ada. However, current trends in

M.T. Higuera-Toledano (✉)
Universidad Complutense de Madrid, Ciudad Universitaria Madrid 28040 Spain
e-mail: mthiguer@dacya.ucm.es

S. Yovine
CONICET, Departamento de Computación, FCEN, Universidad de Buenos Aires, Ciudad Universitaria, Buenos Aires, Argentina
e-mail: syovine@dc.uba.ar

D. Garbervetsky
Departamento de Computación, FCEN, Universidad de Buenos Aires, Ciudad Universitaria, Buenos Aires, Argentina
e-mail: diegog@dc.uba.ar

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_5,

the embedded and real-time software industry are leading practitioners towards the use of Java, which has unpredictable behavior in terms of execution time, scheduling and memory management. Much of this unpredictability is caused by the built-in garbage collection, i.e., the automatic reclaiming of heap-allocated storage after its last use by a program. Implicit garbage collection has always been recognized as a beneficial support from the standpoint of promoting the development of robust programs. However, this comes with overhead regarding both execution time and memory consumption, which makes (implicit) *Garbage Collectors* (GCs) poorly suited for small-sized embedded real-time systems.

In real-time systems, the GC must ensure memory availability for new objects without interfering with real-time constraints. In this sense, current Java collectors present a problem because threads may be blocked while the GC is executing. Consequently, automatic garbage collection [231] has not been traditionally used in real-time embedded systems. The reason is that temporal behavior of dynamic memory reclaiming is extremely difficult to predict. Several GC algorithms have been proposed for real-time embedded applications (e.g., [191, 203, 328, 370]).

However, these approaches are not portable (as they impose restrictive conditions on the underlying execution platform), do require additional memory, and/or even if statistically fast, do not really ensure predictable execution times.

An appealing solution to overcome the drawbacks of GC algorithms, is to allocate objects in regions (e.g., [413]) which are associated with the lifetime of a computation unit (typically a thread or a method). The stack discipline used by regions provides better spatial-temporal locality exploited by cache memories, also this discipline can be less space demanding than generational collectors, providing less fragmentation than heap allocation. In the region-based memory model, the store consists of a stack of regions, which makes it interesting from a real-time point of view, because region-based memory management are more time predictable than garbage collector techniques. We can find in [412] a description of the main technical developments, together with some thoughts about how this model work taken into account the interaction about theory and practice.

Both techniques, regions-based memory management and classical collection techniques can co-exits and cooperate to obtain optimal memory management. This approach is adopted, for instance, by the *Real-Time Specification for Java* (RTSJ) [65], where some real-time tasks can allocate and reference objects within the heap, whereas others (critical tasks) are not allowed to allocate nor reference objects within the heap, instead allocates objects within *scoped memory* regions which are freed when the corresponding unit finishes its execution. Real-time guarantees can then be provided by turning off the GC during the execution of critical tasks, since is not required for these tasks.

The RTSJ region-based approach defines an API which can be used to explicitly and manually handle allocation of objects within a program, while their deallocation is automatic. To avoid memory regions cycles, RTSJ introduce the *single parent rule* which keep longer life for region ancestors. In order to avoid dangling inter-region pointers, RTSJ imposes restricted assignments rules that keep possibly longer-lived

objects from referencing objects that may have a shorter life. However, on the current RTSJ memory model, the memory assignment rules by themselves do not prevent the dangling reference problem. Care must be taken when objects are mapped to regions in order to avoid dangling references. Since scoped memory regions can be nested, the full implication of the memory assignment rules must be considered and studied in detail. Thus, programming using such APIs is error-prone, mostly because determining objects lifetime is difficult.

Given that one of the requirements for RTSJ is that there should be no changes in the Java language and that Java base line compilers can be used to compile RTSJ programs, these rules must be enforced at run-time [67, 202]. This solution adversely affects both the performance and predictability of the RTSJ application. The success of the RTSJ may well hinge on the possibility to offer an efficient and time-predictable implementation of scoped memory regions. A RTSJ region-based implementation requires to maintain the root-set of the GC collecting the heap, and the reference-counters of the GC collecting scoped regions. In addition, to ensure that garbage collection within the heap complies with real-time constraints, read/write barriers has to be introduced depending on the incremental technique employed. Then, memory management in RTSJ requires run-time checks for (1) each memory access, (2) each assignment, (3) each time a task enter/exit a region, and (4) for each time a task is created/destroyed.

This chapter focuses on the data structure and the algorithms used to support and improve the suggested implementation of the RTSJ memory model. The RTSJ suggested implementation of some memory management rules require stack-based algorithms. The scope stack is determined by the control flow, and the depth of the stack is only now at run-time. The scope stack introduces a new source of indeterminism, given that in real-time systems the worst case execution time must be determined and bounded. Another consequence of the RTSJ memory management rules is the higher overhead that its implementation introduces, which makes the system highly inefficient. In order to make deterministic the application behaviour, we give a new definition for one of the RTSJ memory management rules: the single parent rule. The definition of the single parent rule proposed here simplifies the scoped memory hierarchy, avoiding the algorithms related with the stack exploration. This hierarchical model enables performing run-time checks in constant time, making RTSJ applications time predictable. The redefinition of this rule requires new and complex algorithms for the *Memory Region* (MR) collector. Moreover, some of the suggested algorithms to enforce these run-time checks are based on a scope stack (e.g., the maintaining of the scoped region reference-counters), which introduces a high overhead. Thus, it seems interesting to use static analysis to transfer as much work as possible to the compiler.

Clearly, without appropriate compile-time support, region-based memory management may lead to unexpected runtime errors. An appealing alternative to programming memory management directly using the RTSJ API consists in automatically transforming a standard Java program into another one relying on RTSJ functionality, such that objects are allocated in regions whenever possible.

Some techniques have been proposed to address this problem by automatically mapping sets of objects with regions. Such an approach requires analyzing the program behavior to determine the lifetime of dynamically allocated objects. In [126] this is done based on profiling, while [101, 164] rely on static *pointer and escape analysis* [58, 104, 171, 339–341, 432] to conservatively approximate object lifetimes.

A semi-automatic tool-assisted end-to-end approach to perform the translation of Java applications into RTSJ-compliant applications has been presented in [165]. The tool integrates a series of compile-time analysis techniques to help identifying memory regions, their sizes, and overall memory usage. First, the tool synthesizes a scoped-based memory organization where regions are associated with methods. Second, it infers their sizes in parametric form in terms of relevant program variables. Third, it exhibits a parametric upper-bound on the total amount of memory required to run a method.

5.1.1 Chapter Organization

In this chapter, Sect. 5.2 presents an in depth description of the RTSJ memory model regarding its current definitions and suggested implementation techniques. Section 5.3 introduces a basic approach, which includes write barriers to detect whether the application attempts to create an illegal assignment, and a description of how scoped regions are supported. Section 5.4, presents a hardware-based solution to improve the performance of write barriers, which reduces the write barrier overhead for intra-region assignments to zero, and proposes a specialized hardware to support the stack-based algorithm, which makes negligible the time-cost to detect illegal assignments across scoped regions. Section 5.5 presents an algorithm for escape analysis, which primary purpose is to produce useful information to allocate memory using a the RTSJ approach. It uses this simple static analysis algorithm coupled with region-based memory management to provide a semi-automatic environment where all memory operations run in predictable time. Finally, Sect. 5.6 provides a summary and some research directions.

5.2 The RTSJ Memory Model

Implicit garbage collection has always been recognized as a beneficial support from the standpoint of promoting the development of robust programs. However, this comes along with overhead or unpredictability regarding both execution time and memory consumption, which makes (implicit) garbage collection poorly suited for small-sized embedded real-time systems. However, there has been extensive research work in the area of making garbage collection compliant with real-time requirements. Main results relate to offering garbage collection techniques that

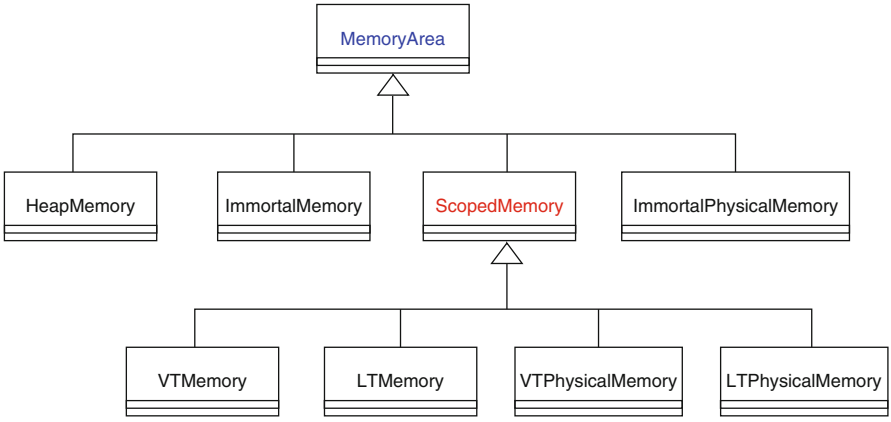


Fig. 5.1 An abridged `MemoryArea` hierarchy in RTSJ

enable bounding the latency caused by the execution of garbage collection. Proposed solutions may roughly be classified into two categories:

1. *Incremental garbage collection* (e.g., [25]) enables the interleaved execution of the garbage collector with the application.
2. *Region-based memory allocation* (e.g., [170]) enables grouping related objects within a region.

Note that the two above collection strategies are complementary: incremental garbage collection may be used within some regions in order to limit their size, while the use of regions allows reducing the runtime overhead due to garbage collection. The RTSJ introduces memory regions and allows the implementation of real-time compliant garbage collectors to be run within regions except within those associated with hard timing constraints.

RTSJ makes distinction between three main priority levels: (1) *critical tasks* that cannot tolerate preemption latencies because the GC, (2) *high-priority* tasks that cannot tolerate GC unbounded preemption latencies, and (3) *low-priority* tasks that tolerate the collector pauses. In RTSJ, critical tasks are instances of the `NoHeapRealtimeThread` class, high-priority tasks are instances of the `RealtimeThread` class, and low-priority are instances of the `Thread` class.

In the above context, a non-real-time collector introduces unpredictable pauses that are not tolerated by critical and high-priority tasks. Real-time collectors eliminate this problem but introduce a high overhead. Also, bounded pauses are not tolerated by critical tasks. An alternative approach is to use regions within which allocation and deallocation is customized. Each memory region is then managed so as to embed objects that are related regarding associated lifetime and real-time requirements. Such a facility is enabled by the RTSJ specification [65] (see Fig. 5.1): The `MemoryArea` abstract class supports the region paradigm in the RTSJ specification [65] through three kinds of regions: (1) *immortal memory*, supported

by the ImmortalMemory and the ImmortalPhysicalMemory classes, that contain objects whose life ends only when the JVM terminates; (2) (nested) *scoped memory*, supported by the ScopedMemory abstract class that enables grouping objects having well-defined lifetimes and that may either offer temporal guarantees (i.e., supported by the LTMemory and LTPhysicalMemory classes) or not (i.e., supported by the VTMemory and VTPhysicalMemory classes) on the time taken to create objects; and (3) the *conventional heap*, supported by the HeapMemory class.

The ImmortalPhysicalMemory, LTPhysicalMemory and VTPhysicalMemory classes support regions with special memory attributes (e.g., *dma shared swapping*). But, whereas the ImmortalPhysicalMemory objects end with the application, the LTPhysicalMemory and VTPhysicalMemory objects have the same restrictions imposed to scoped MRs.

Objects allocated within immortal MRs live until the end of the application and are never subject to garbage collection. Objects with limited lifetime can be allocated into a scoped region or the heap. Garbage collection within the application heap relies on the (real-time) GC of the JVM. Scoped regions get collected as a whole once it is no longer used. Since immortal and scoped MRs are not garbage collected, they may be exploited by critical tasks, specially LTMemory and LTPhysicalMemory objects, which guarantee allocation time proportional to the object size.

There is only one object instance of the HeapMemory and the ImmortalMemory classes in the system, which are resources shared among all threads in the system and whose reference is given by calling the instance() method. In contrast, for the ImmortalPhysicalMemory and ScopedMemory subclasses, several instances can be created by the application.

In LTMemory object allocation has linear time guaranties (i.e., the time needed to allocate n bytes is bounded by a polynomial function $f(n) \leq C_n$ for a constant $C > 0$), while the execution time of an object allocation within VTMemory region need not complete in linear time. Since time predictability is crucial when dealing with real-time applications, *jRate* [112] provides a new type of scoped memory, called CTMemory, which guaranties constant time object allocation. Another extension of the RTSJ MemoryArea hierarchy is the AGCMemory [37, 38], which extends ScopedMemory by providing special rules to allow partial region deallocation. Particularly, This extension enables the destruction of floating garbage created during the execution of Java methods.

5.2.1 Using Memory Regions in RTSJ

An application can allocate memory into the system heap (i.e., the HeapMemory object), the immortal system memory region (i.e., the ImmortalMemory object), several scoped memory regions, and several immortal regions associated with physical characteristics. Several related threads, possibly real-time, can share a

```
import javax.realtime;

class  Allocator implements Runnable {
     public void run() {
           HeapMemory.instance().newArray(Integer, 10);
           int[]  v = new int[20];
       }
}

class RegionUseExample {
    public static void main (String[] args)  {
           ScopedMemory myRegion =  new VTMemory(1024, 2*1024);
           RealtimeThread task = new RealtimeThread(
             null, null,
             new MemoryParameters(1024, 0),
             myRegion, null, new Allocator());
           task.start();
       }
}
```

Fig. 5.2 Using memory regions in RTSJ

memory region, and the region must be active until at least the last thread has exited. The `MemoryArea` abstract class provides methods like `enter(Runnable)` allowing real-time threads to execute within a determined scope.

The `enter()` method is the mechanism to activate a region. This method associates a memory area object to the thread during the execution of the `run()` method of the object passed as parameter. Another way to associate a region to a real-time thread is through a `MemoryArea` on its constructor. Allocations outside the active region can be performed by the `newInstance()` or the `newArray()` methods.

Commonly, regions are used explicitly in the program code as shown in Fig. 5.2, which shows a real-time thread, which allocates an array of 10 integers in the heap, and another of 20 integers in the memory region called `myRegion`.

Scoped regions can be nested. By using nested `enter()` calls, a real-time thread may enter multiple scopes, which dynamically builds up a hierarchy of memory scopes. The misuse of these methods (i.e., when a user tries to enter a `ScopedMemory` that is already accessible) throws the `ScopedCycleException`. A safe region implementation requires that a region gets deleted only if there is no *external* reference to it. This problem has been solved by using a *reference-counter* for each region that keeps track of the use of the region by threads, and a simple reference-counting GC collects scoped memory regions when their counter reaches 0. Before cleaning a region, the `finalize()` method of all the objects in the region must be executed, and it cannot be reused until all the *finalizes* execute to completion.

Note that in this hierarchy there is no way to keep scoped memory region alive unless a real-time thread is active within it. This could be achieved by the *wedge thread* pattern introduced in [297]. The RTSJ Version 1.1 (i.e., the JSR-282) [131]

extends the `MemoryArea` hierarchy introducing the concept *pinned scope*, which allows a scope to stay alive even when there is no active real-time thread within it [133]. In order to do that RTSJ-282 introduces the `PinnableRegion` interface [129]. This allows applications to explicitly *pin* a scope, thereby keeping the scope alive even though there is no active thread within it (e.g., by maintaining a a sleeping real-time thread or an idle timer).

5.2.2 Illegal References

The lifetime of objects allocated in scoped regions is governed by the control flow. Strict assignment rules prevent the creation of dangling pointers (i.e., references from an object to another one with a potentially shorter life). The following conditions are checked before the assignment is executed:

- Objects within the heap or an immortal region cannot reference objects within a scoped region.
- Objects within a scoped region cannot reference objects within a non-outer[1] scoped region.

Illegal assignments must be checked when executing instructions that store references within objects or arrays. These rules must be enforced dynamically, checking they at every memory scope with a longer lifetime does not hold a reference to an object allocated in a memory scope with a shorter lifetime. This means that objects within the heap memory and immortal memory (i.e., the *primordial area*) cannot hold references to objects allocated within a scoped memory area, nor an objects within a scoped memory area can hold a reference to an object allocated in an inner (more deeply nested) scope. The `IllegalAssignment()` exception is thrown on attempted violations of the memory access rules, whereas the `MemoryAccessError` exception is thrown when the application tries to access an object within a non-allowed scope (e.g., whether a `NoHeapRealtimeThread` tries to access an object within the heap). Specific actions (e.g., low level code) need to be included for detecting an attempt to make an illegal assignment when updating an object reference, throwing the exception in case of violation of some rule. In order to dynamically enforced safety, these rules check that every task has associated a region-stack containing all scoped MRs which the thread can hold. The MR at the top of the stack is the active region for the task whereas the MR at the bottom of the stack is the most outer scoped region for the task. When the task does not use any scoped region, the region-stack is empty and the active region is the heap, which is the default active region, or an immortal MR. Both, the active region and the region-stack associated with the task change when executing the `enter()` method.

[1] We consider as non-outer region all scoped regions that are not entered by a considered real-time thread before to enter the considered region (i.e., scoped regions which are not bellow the considered region in the scoped stack of a given control flow).

The RTSJ specification does not explicitly provide an algorithm to enforce the assignment rules, but there are some different approaches [53, 109, 202].

5.2.3 The Single Parent Rule

To support scoped memory regions, RTSJ propose a mechanism based on using a scope tree containing all region-stack in the system and a reference-counting collector. Every scoped region is associated with a reference counter that keeps track of the use of the region by tasks, while every tasks is associated with a stack that keeps track of the scoped region that can be accessed by the task. In order to avoid MR cycles in the region stack associated to a task, RTSJ defines the single parent rule as follows:

> *If a scoped region is not in use, it has no parent. For all other scoped objects, the parent is the nearest scope outside it on the current scoped region stack. A scoped region has exactly zero or one parent.*

The single parent rule guarantees that once a task has entered a set of scoped MRs in a given order, any other task will have to enter them in the same order. When the reference-counter of a scope region is zero, a new nesting (parent) for the region will be possible. This requirement assures that a parent scope will have a lifetime that is at least that of any of its child scopes.

To enforce the single parent rule, a compliant JVM has to check every attempt to enter a memory area by a task to ensure that the single parent rule is not violate. At this time, if the scope region has no parent, then the entry is allowed. Otherwise, the task entering the scoped region must have entered every proper ancestor of it in the scope stack. The suggested RTSJ algorithm that perform this test has a time complexity of $O(n)$.

5.3 Memory Management Solutions for RTSJ

A MR implementation must ensure that objects within the heap or the immortal memory cannot reference objects within a scoped region, and objects within a scoped region cannot reference objects within another scoped region that is neither non-outer nor shared. In this section, we use write barriers to detect illegal inter-region assignment at run-time, and a stack-based algorithm to check whether a scoped region is outer to another one. This section addresses the problem associated with the inter-region references introduced by MRs:

- Interaction of the real-time GC within the heap and MRs (i.e., external roots).
- References from objects within a memory area to a non-outer one (i.e., illegal assignment).
- Accesses to object within the heap made by critical tasks (i.e., memory access errors).

Since the current RTSJ memory model introduces a complex programming model, some researchers have produced new programming paradigms and patterns in order to reduce the complexity of programming using memory regions. In [110] we found some adaptations of classical patterns to the RTSJ memory, and in [54] a copying-based mechanism to perform automatic object copies from one region to another avoiding some of the limitations imposed by the assignment rules. In [109] we could find a mechanism to violate the assignment rule, which introduces the idea of having an easy navigation for RTSJ portals by using weak references. Here the RTSJ portal is extended to support a region-based distributed middleware. The solution presented in [71], uses both, the weak references mechanism and the reflection support of the Java API, as mechanism to violate the assignment rules. Other works present the use of a copy-based mechanism in order to avoid the maintenance of dangling pointers [63, 134, 313].

5.3.1 The Basic Collector Strategy for Objects Within the Heap

There are some important considerations when choosing a real-time GC strategy. Among them are space costs, barrier costs, and available compiler support. Copying GCs require doubling the memory space, because all the objects must be copied during GC execution. Non copying GCs do not require this extra space, but are subject to fragmentation. The age distribution of SPECjvm98 suite applications[2] confirms the generational hypothesis that most objects die young, but this effect is not as high as for other programming languages (e.g., Smalltalk). In Java, 40% of objects are still alive after 100 KBytes allocation. In contrast, only 6.6% of Smalltalk objects survive 140 KBytes allocation [135]. Java can, therefore, benefit from generational collection, which allows collecting younger objects more aggressively. However, the *new* generation should be substantially larger than for Smalltalk or ML.

We specifically consider an incremental non-copying collector based on the *tri-color* algorithm [137]. Since this algorithm allows the application to execute while the GC has been launched, *barriers* are used to keep the state of the GC consistent. The basic algorithm is as follows: an object is colored *white* when not reached by the GC, *black* when reached, and *grey* when it has been reached, but its descendants may not be, i.e., they are white. The collection is completed when there are no more grey objects. All the white objects can then be recycled and all the black objects began white.

Grey objects make a wavefront, separating the white (unreached) from the black (reached) objects, and the application must preserve the invariant that no black objects have a pointer to a white object, which is achieved using read barriers in the *treadmill* GC [25]. To coordinate the application and the GC, we can use write barriers (i.e., coordinate actions when pointers updates) instead of read barriers

[2] http://www.spec.org/osg/jvm98.

(i.e., coordinate actions when objects accesses). This decision is motivated by the fact that write barriers are more efficient than read barrier [450], and that the resulting collector can further be easily extended to generational, distributed, and parallel collection [438].

The GC algorithm used in Jamaica JVM [373] uses write barriers, and provides real-time guaranties by working with blocks (i.e., the Java objects are round up and fragmented in memory blocks of the same size). While the collector scans or sweeps an object it can not be preempted by the application. Blocks simplifies the incremental GC algorithm because it performs in a small and bounded unit works, which can be exactly calculated and measured in order to establish an upper bound. An important implementation decision is to distinguish header blocks from inner blocks. Since write barriers deals only with the object header, an efficient implementation for header blocks is required, while tolerating a less efficient implementation for inner blocks. Note in Java, all object accesses are through the object header. Another improvement of the Jamaica VM consists to grey any white block which reference is written (i.e., it does not mater the color of the block storing the reference).

5.3.2 Memory Regions and External Roots

Since objects allocated within regions may contain references to objects in the heap, the GC must scan regions for references to objects within the heap. Thus, the collector must take into account the external references, adding them to its reachability graph. To facilitate this task, we introduce a fourth color (e.g., *red*) meaning that the object is allocated outside the heap [201]. This color allows us to detect when the `v` object must be added to the *root-set* of the collector, where the root-list is updated in a similar way to how generational collectors maintain *inter-generational pointers* (i.e., by using write barriers).

We must check the inclusion of an external root for the collector when executing instructions that copy references or store references within other objects (or arrays), this mechanism correspond to write barriers already used in the basic collector strategy:

- The `astore` bytecode stores a reference into a local variable (i.e., `v = u`).
- The `aastore` bytecode stores a reference `u` into an array `v` of references (i.e., `v[i] = u`).
- The `putfield` bytecode causes a `v` object to reference a `u` object (i.e., `v.f = u`).
- The `putstatic` bytecode causes a `v` object to reference a class variable (i.e., `v.f = u`).

Note that class variables (i.e., `static` fields) are never subject to garbage collection. Then, we must execute the write barrier code to check illegal references only when executing the `astore`, `aastore`, and `putfield`, bytecodes.

```
if (region(u) = scoped)
        if (region(v) ≠ scoped) illegalAssignment()
        else nestedRegions(v, u);
```

Fig. 5.3 Memory regions write barrier code executed for each `v = u` or `v.f = u` statements

The Jamaica VM [373] allows time-bounded scanning for the GC roots by copying the references that are locally within the thread stacks or in the processor registers (i.e., the root-set) within a *root-array*, which is scanned incrementally. We must also introduce the referenced object (i.e., the greyed object) in this list when executing the write barrier code. This solution introduces some overhead that must be characterized and bounded. Whenever a reference within the root-array is not used anymore (e.g., when the MR is collected), it must be removed, ensuring its reclamation by the GC when it becomes garbage.

5.3.3 Nested Scoped Memory Regions and Illegal References

In order to keep track of the currently active MR of each schedulable object (e.g., a real-time thread), RTSJ uses a stack associated which it. Every time a real-time thread enters a MR, the identifier of the region is pushed onto the stack. When the real-time thread leaves the MR, its identifier is popped of the stack. The stack can be used to check illegal assignments among scoped MR[3]:

- A reference from an object `v` within a scoped region `S` to a object `u` within a scoped region `T` is *allowed* when the region `T` is *below* the region `S` on the stack.
- All other assignment cases among scoped regions (i.e., the region `T` is *above* the region `S` or it is not on the stack) are *forbidden*.

To maintain the safety of Java and avoid dangling references, objects in a scoped region can only reference objects within an outer or same region, within the heap, or within immortal memory, which must be checked when executing assignments instructions (i.e., by using write barriers). Since class variables are allocated within the `ImmortalMemoryArea`, which is considered as the outer region, and references to object allocated within it are always allowed, we must execute the write barrier code to check illegal references only when executing the `astore` and `aastore` and `putfield` , bytecodes (i.e., as for the GC).

Then, the MR to which the object `u` belongs must be outside of the MR to which the object `v` belongs. Figure 5.3 shows the write barrier pseudo-code, that we must introduce in the interpretation of the aforementioned bytecodes, where we denote as `v` the object that makes the reference, and as `u` the referenced object. The region

[3] Illegal assignments are pointers from a non-scoped MR (i.e., heap or an immortal MR) to a scoped one, or from a scoped region to a non-outer scoped region. Pointers to a non-scoped region are always allowed.

to which an object belongs must be specified in the header of the object. Then, when an object/array is created by executing the `new` or `newarray` bytecode, it is associated with the scope of the active region. The `nestedRegions(v, u)` function, throws the `IllegalAssignment()` exception when the region to which the `u` object belongs is not outer to the region to which the `v` object belongs and returns *true* when the region to which the `u` object belongs is outer or equal to the region to which the `v` object belongs.

Following this we describe the algorithm of the `nestedRegions(v, u)` function, which requires two steps:

1. The region-stack of the active task is explored, from the top to the bottom, to find the MR to which the `v` object belongs. If it is not found, this is notified by throwing a `MemoryAccessError()` exception[4].
2. The region-stack is again explored, but this time we take the MR found in the previous step as the top of the stack. Then, we start the search from the region to which the `v` object belongs, and the objective is to find the MR to which the `u` object belongs (i.e., the region to which the object `u` belongs must be outer to the region to which the object `v` belongs). If the scoped region of `u` is not found in the new region-stack, this is notified by throwing an `IllegalAssignment()` exception. If it is found, the `nestedRegions(v, u)` returns `true`.

Note that this write barrier code take into account that references from scoped regions to the heap or immortal memory regions are always legal, and the opposite is always illegal, which means that the `nestedRegions(v, u)` function scans a region-stack keeping track of scoped regions that the real-time thread can hold [202]. Since intervening entries into the heap or immortal regions do not affect the explored region-stack and the ancestor relation among scoped regions is defined by the nesting of the regions themselves, an alternative solution is to follow the scope's parent link to discover if the target reference is in an ancestor scope of the source [109].

The complexity of an algorithm which explores a stack is $O(n)$, where n is the depth of the stack, and the execution-time can be bounded by the maximum nested scoped levels. Note that the scope stacks for an application, particularly their depth, are not decidable at compile-time, which may make stack-based algorithms incur unpredictable time-overhead.

An efficient and time-predictable technique based on the type hierarchies concept, also called *Displays*, allowing to execute the `nestedRegions(v, u)` function in constant time has been also presented in [109]. Since the parent-child relation changes as the application runs and scoped memory areas are entered and exited, the associated type hierarchy is not fixed, the typing information associated with each scoped region has to be created and destroyed, respectively. However, this changes at well-defined points (i.e., when the reference count of a scoped region becomes zero or goes from zero to one).

[4]This exception is thrown upon any attempt to refer to an object in an inaccessible `MemoryArea`.

```
if (color(u) ≠ red) {
      if (type(τ) = critical) memoryAccessError();
      if (color(v) = red) updateRootSet(v, u);
      if (color(u) = white) greyObject(u);
}
```

Fig. 5.4 Garbage collectors write barrier code executed for each `v = u` or `v.f = u` statements

Another time-predictable solution using name-codes and binary masks to support write barrier checks based on the idea that regions are parented at creation time has been presented in [40]. This solution gives a memory model less flexible than the RTSJ one. However the advantages that it presents are determinism and simplicity in both its use and implementation.

5.3.4 Critical Tasks and Memory Access Errors

Note that when using a write barrier-based GC, read barriers detecting memory access errors are not strictly necessary because read operations do not change the color of the object, therefore they do not affect the GC coherence [196]. The restriction on critical tasks can be reduced to write barriers checks since reads does not interfere with the GC graph and objects within the heap are not reallocated. Note that we use a non-moving collector (i.e., no-copying-based and no-having compaction), read barriers can be avoided. For copying-based or compacting GC, we must use a handle to objects allocated within the heap in order to facilitate relocation and to maintain its transparency for the use of write barriers instead of read barriers. For moving GC needing relocation which does not use handles (i.e., the HotSpot JVM [396]), read barriers are required.

We apply the same optimization as for the incremental GC which is to use write barriers instead of read barriers (see Fig. 5.4). In this case, the `MemoryAccessError()` exception which is raised when a critical task attempts to access an object `v` within the heap, is changed by the `IllegalAssignment()` exception which is raised when a critical task attempts to create a reference to an object Y which belongs to the heap. Whereas for non-critical tasks, a reference from a red object to an object within the heap (i.e., non-red) causes the addition of the `v` object to the root-set of the collector; for critical tasks, a reference to a non-red object (i.e., white, black, or grey) causes a `MemoryAccessError()` exception.

This modification is not compliant with the RTSJ specification, but introduces an important improvement: about 6% of the introduced overhead (i.e., whereas the 11% of executed bytecodes performs a load operation, the 5% performs a store operation).

Taken into account the distributed real-time Java applications 2, the penalty of GC in the end-to-end path must be avoided, which makes the use of scoped regions in this context crucial. Problems associated with scoped regions such as

dimensioning (e.g., the size of the region) or nesting (e.g., the single parent rule), as well as problems associated to object accessing (e.g., the assignment rules) are still present in remote objects. In this context, a extension of the strong semantics of these rules has been achieved via portals in [40].

5.3.5 Analyzing the Behavior of the Single Parent Rule

Note, cycles on the stack must be avoided. For example, if both scoped regions `S` and `T` appear on the following order: `A`, `B`, `A`, then reference types: from `S` to `T`, and from `T` to `S` are allowed. That means that the `S` scoped MR is inner to the `S` scoped MR, and vice-versa. Since the assignment rules and the stack-based algorithm by themselves does not enforce safety pointers, the RTSJ defines the single parent rule, which goal is to avoid scoped MR cycles on the stack. This single parent rule establishes a nested order for scoped regions and guarantees that a parent scope will have a lifetime that is at least that of its child scopes. An in depth study of the behaviour of this rule reveals some problems [198]:

- It introduces race conditions whether a region cycle involves two or more threads, which result in a non-deterministic behavior. Consider two real-time threads τ_1 and τ_2: the real-time thread τ_1 enters regions in the following order: `S` and `T`, whereas τ_2 enters regions as follows: `T` and `S`. We found four different behaviors when executing this program depending on the exact order of execution.
  - If τ_1 enters `S` and `T` before τ_2 enters `T`, τ_2 violates the single parent rule raising the `ScopedCycleException()` exception.
  - But, if τ_2 enters `T` and `S` before τ_1 enters `S`, when τ_1 tries to enter `S`, it violates the single parent rule and raises raising the `ScopedCycleException()` exception.

  Let us suppose that τ_1 and τ_2 have entered respectively the `S` and `T` regions and both stay there for a while. In this situation, the application has two different behaviours:

  - When τ_1 tries to enter the `T` scoped region violates the single parent rule.
  - When τ_2 tries to enter the `S` scoped region violates the single parent rule.

  Then the `ScopedCycleException()` exception thrown by four different conditions. As consequence the programmer must deal with four executions errors, which makes this code hard to debug it. More over, there are also four execution cases where the single parent rule is not violated.
- It limits the degree of concurrency within an application. Consider a real-time thread τ_1 entering a scope region `S`, whether its parent is the *primordial scope* (i.e., the heap or an immortal region) the entry is allowed. Otherwise, τ_1 must have entered every ancestor of `S`, or must wait until the region has exited for all real-time threads how previously entered it (i.e., then the parent of `S` is the

primordial scope again) which introduces some sequentiality in the τ_1 execution (i.e., introduces mutual exclusion). Considering the former example, the single parent rule is not violated and the application gives the correct result in the following cases:

- τ_1 enters S and T, and exits both regions before τ_2 enters T and S.
- τ_2 enters T and S, and exits both regions before τ_1 enters S and T.
- τ_1 enters S and T, τ_1 exits T before τ_2 enters it, and τ_1 exits S before τ_2 tries to enter it.
- τ_2 enters T and S, τ_2 exits S before τ_1 enters it, and τ_2 exits T before τ_1 tries to enter it.

- It is not natural; there are orphans regions and the parent of a region can change along its life, which results in an unfamiliar programming model. Considering the former example, the execution cases alternate two parentage relations: *A* is parent of T while the τ_1 execution, and T is parent of S while the τ_2 execution. Then, assignments from objects allocated within T to objects within S are allowed when the executing real-time thread is τ_1, and are illegal when the executing real-time thread is τ_2. And assignments from objects allocated within S to objects within T are allowed when the executing real-time thread is τ_2, and are illegal when the executing real-time thread is τ_1. This situation, which must be taken into account by the RTSJ programmer, makes difficult and tedious the programming task.

In [200], we try to avoid these problems by proposing two alternative solutions: one of them redefines the single parent while the other avoids the single parent rule. Another solution attempts to circumvent the assignment rules (i.e., illegal references). The former solution redefines the parentage relation among MRs as follows:

> *The parent of a scoped memory area is the memory area in which the object representing the scoped memory area is allocated.*

Note that the parent of a scoped region is assigned when creating the region and does not change along the live of the region, making the scoped ancestor tree static. Hence, an important improvement is the use of name-based checking for illegal references, which is efficient and time-predictable.

The second approach avoids the single parent rule by allowing cycles references among scoped regions. Since we maintain the RTSJ assignment rules, there are not cycles references among objects within scoped regions and objects within the heap or an immortal region. Region cycles including the heap increase considerably the complexity of the collector within the heap. Also a region in a cycle that includes the immortal memory region effectively becomes immortal itself. This approach requires to introduce some modifications in the reference-counter collector for scoped regions. (e.g., a new data structure containing information about all scoped regions that must be collected before to collect each scoped region).

Finally, the third approach, which avoids both the single parent rule and assignment rules, is the ideal because it liberates the programmer from having to

think about the assignment rules making the code more expressive and powerful and easier to write. However this solution requires a carefully study for it support and to redesign the reference-counter collector of scoped memory regions.

The RTSJ version 1.1 (i.e., JSR-282) [129, 130, 133] considers 21 enhancement requests. Where four of them are related to memory management. This version add *weak scoped references*, which are the equivalent of Java's weak references for scoped memory area. Also consider whether to allow explicit references between objects that would violate the RTSJ assignment rules (e.g., introducing bi-directional reference rules), an example is to melt scoped regions that are adjacent to each other on the scope stack to be into a single scope allowing references between objects within they.

5.4 Minimizing the Write Barrier Impact

The execution of the collector write barriers (i.e., to maintain both the tri-color invariant and the external) and the memory region write barriers (i.e., references into a scoped region from objects within a non inner region), and the critical tasks write barrier (i.e, references to objects within the heap) may lead to a quite substantial time overhead.

The most common approach to implementing write barriers is by adding in-line code, consisting in generating the instructions executing write barrier events with every store operation. This approach has been presented in [53]. This solution requires compiler cooperation (e.g., JIT), and presents a serious drawback because it will double the application's size. Regarding systems with limited memory such as PDAs, this code expansion overhead is considered prohibitive. Also, given the dynamic nature of the Java language, and that there is not a special representation in the Java bytecode, compile-time static analysis is not enough.

Alternatively, the solution proposed in [202] instruments the bytecode interpreter, avoiding space problems, but this still requires a complementary solution to handle native code. Note that all the objects created in Java are allocated in the heap, only primitive types are allocated in the runtime stack.

The success of RTSJ depends on the possibility to offer an efficient implementation of the assignments restrictions. The use of hardware support for write barriers has been studied in [203], where an existing microprocessor architecture has been used in order to improve the performance of checks for illegal references. This solution minimizes the time overhead introduced by the write barriers by improving their performance by using a specialized hardware support, such as the picoJava-II microprocessor. This hardware support allows performing write barrier checks in parallel with the store operation, making time-cost free the detection of whether an action is required (e.g., to grey an object, to include an external root, or to raise an exception). After presenting this solution, we give characterize the write barrier overhead.

```
genotify
  if (region(Y) ≠ red)
    if (τ = critical) memoryAccessError();          // a critical task must create references into the heap
    else if (color(X) = red) updateRootSet(X, Y)    // a legal external reference is a external root
      else if (color(Y) = white) greyObject(Y);     // concurrent execution of the GC and the application
```

Fig. 5.5 Handling the `gc_notify` exception for the picoJava-II reference-based write barriers

5.4.1 Hardware-Based Support for Memory Regions

Upon each instruction execution, the picoJava-II core checks for conditions that cause a trap. From the standpoint of hardware support for garbage collection, the core of this microprocessor checks for the occurrence of write barriers, and notifies them using the `gc_notify` trap. This trap is triggered under certain conditions when a reference field of some object is assigned some new reference.[5] The conditions under which this trap is generated are governed by the values of the `PSR` and the `GC_CONFIG` registers [394].

The reference-based write barriers of the picoJava microprocessor can be used to implement incremental collectors. The `GC_TAG` field for white objects is set to %00, %01 for grey, %10 for red, and %11 for black objects. An incremental collector traps when a white object is written into a black object (i.e., combination `1100`). We use also this mechanism to detect when a red object references an non-red one (i.e., combinations: `1000`, `1001`, or `1011`). The code executed by the `gc_notify` exception handler (see Fig. 5.5) is the same as the one introduced in the interpreter in the former solution (Sect. 5.3.4).

Since another mechanism must be combined to know if a given assignment statement is an illegal reference (i.e., if the referenced object is within the heap or within a non-outer region), we can adapt our solution to be supported by the page-based barrier mechanism of picoJava-II [201]. The page-based barrier mechanism of picoJava-II was designed specifically to assist generational collectors based on the train-algorithm [211, 366] , such us the HotSpot JVM [396], which provides constant pause times by dividing the collection of the old-space into many tiny steps (i.e., typically less than 10 ms)[6]. In this case, the `gc_notify` trap is thrown when the bits indicated by the `GC_REGISTER.CAR_MASK` field in the address of both objects are different, and the `GCE` bit of the `PSR` register is set. However, we can use this mechanism to detect references across different

[5] when executing `putfield`, `putstatic`, `astore`, or `aastore` bytecodes.

[6] The pauses introduced by this collector provides constant pause times, which makes it possible to run this collector with multimedia applications. But it is not adequate for hard real-time applications because the guaranteed upper limit on pause times is too large.

```
gcnotify
  if (region(Y) = scoped)                    // a reference to the heap or immortal is always allowed
     if (region(X) = scoped) nestedRegions(X,Y)   // a reference to an outer scope is always allowed
     else illegalAssignment();               // an inter-region reference to a non-outer scope is illegal
  priv_ret_from_trap                         // exception return
```

Fig. 5.6 Handling the `gc_notify` exception for the picoJava-II page-based write barriers

```
gc_notify_0:                                 // trap for reference-based write barriers
   if ( τ = critical) greyObject(Y) illegalAssignment(); // illegal references for a critical task
   greyObject(Y);                            // tri-color invariant for a no critical task
   priv_ret_from_trap

gc_notify_1:                                 // trap for both page-based write barriers
   if (region(Y) = scoped) nestedRegions(X,Y);   // nested scoped regions checking
   priv_ret_from_trap

gc_notify_1_0:                               // trap for both reference-based and page-based write barriers
   if ( τ = critical) illegalAssignment();   // illegal reference for a critical tasks
   updateRootSet(X,Y);                       // external root for a no critical task
   priv_ret_from_trap
```

Fig. 5.7 Write barrier exception handlers for both reference-based and page-based write barriers

regions [196]. In this case, the code executed by the `gc_notify` exception handler (see Fig. 5.6) is the one introduced in the interpreter in the former solution (Sect. 5.3.3).

This solution modifies the hardware support of picoJava-II to have three different traps [201] (see Fig. 5.7). In this solution, reference-based write barriers cause the execution of: the `gc_notify_0` exception when a white object is assigned to a black one, and the `gc_notify_1_0` exception when a non-red object is assigned to a red one. Depending on the tasks, these exceptions have a different treatment: for critical tasks it is a memory access error, while for non-critical tasks it can be an incremental barrier or an external root. While page-based write barriers cause the execution of: the `gc_notify_1` exception when any object is assigned to another one allocated in a different MR (i.e., when an inter-region reference), and the `gc_notify_1_0` exception when a non-red object is assigned to a red one (i.e., when the region reference is to an object within the heap). Inter-region references to the heap or an immortal regions are allowed for non-critical tasks, while to a scoped region it is illegal whether the the region to which the referenced object belongs is not outer to the *from-region* reference. The design of a hardware support for the `nestedRegions(X,Y)` function has been also presented in [196]. This solution is based on an associative memory and implements the stack-based algorithm presented in (Sect. 5.3.2).

5.4.2 Write Barrier Overhead

The cost of maintaining the three-color invariant of the GC is considered as a fraction of the total program execution time (without including other garbage collection costs). To estimate the time overhead of different implementations, two measures are combined: (1) the number of relevant events that occur during the execution of a program, and (2) the measured cost of the event. The write barrier overhead is given by dividing the application execution time with the number of relevant events and the cost per event.

In most applications of the SPECjvm98 benchmark suite, less than half of the references are to the heap memory (i.e., 45%), the other half is to either the Java or the C stack, and about 35% of total executed instructions are memory references [235], where typically 70% are load operations and 30% store operations. Furthermore, 0.05% of instructions executed by a Java application is a write into the heap or another region. Thus, we estimate the write barrier overhead of the collector as $0.05 \times writeBarrierCollectorCost$. Where *writeBarrierCollectorCost* is the percentage cost to maintain both the three-color invariant and the external roots per assignment.

The cost of maintaining inter-region references is considered as a fraction of the total program execution time (without including the garbage collection costs). We calculate the write barrier cost introduced by the memory regions as $0.05 \times writeBarrierRegionCost$. The *writeBarrierRegionCost* parameter is the percentage cost to detect illegal inter-region references.

The barrier overhead on different architectures can be estimated by measuring the event cost on each architecture.

5.5 Automated Synthesis of Scoped Memory Regions

RTSJ `ScopedMemory` offers means to gain control over the usage of a critical resource by providing a dynamic-memory organization and management scheme which exhibits both time- and space-predictable behavior. However, these benefits do not come for free, but at the expense of making the software architecture and programming more complicated [297].

Indeed, by deviating from the standard Java memory model, resorting to a program-level mechanism based on lexical scopes rather than on a runtime one which removes unreferenced objects from the heap, regardless of the program structure, RTSJ `ScopedMemory` entails a lifetime-centric programming approach to comply with assignment rules that forbid a long-lived object in an outer scope to point to a short-lived object in an inner scope (Sect. 5.2.2) In addition, the RTSJ `ScopedMemory` requires quantifying the number of objects that have the same scope in order to fix the size of a memory region (Sect. 5.2.1). Clearly, without relevant automated support at compile-time, the use of RTSJ `ScopedMemory` is likely to end up provoking unexpected errors at runtime.

An approach to ease the use of RTSJ `ScopedMemory` consists of automatically transforming a standard Java program into one which complies with RTSJ restrictions. This requires somehow synthesizing scoped regions by identifying the lifetime of dynamically allocated objects through some kind of program analysis. Though certainly appealing, this is far from being easy as predicting object lifetime and dynamic memory occupation are undecidable problems. Hence, only approximate solutions can only be envisaged.

One way of proceeding is by program profiling [126]. Dynamic analyses help discovering likely scopes, but they are unsound by nature, as they do not explore all possible runs of the program. Therefore, runtime checks cannot be safely disabled. An alternative technique relies on static analysis at compile time [77, 101, 103, 164, 339, 340]. Because of undecidability, static analyses must conservatively approximate object lifetime and region size in order to be sound, yielding overly long-lived and large scopes. Nevertheless, they also lead to performance gains by enabling switching off runtime checks.

The rest of this section discusses a full-fledged solution for RTSJ `ScopedMemory` synthesis based on static analysis.

5.5.1 Region Inference

The starting point is to compute a conservative approximation of the heap graph, that is, a rough picture of the way objects are related to each other. Since objects are runtime entities, a static analysis must resort to syntactic program elements to represent them. This is the first approximation: objects are bound to their allocation site, that is, the `new` statement that creates them. The second approximation consists in looking at the heap as an undirected graph, rather than a directed one, therefore establishing that two (classes of) objects are connected regardless of which one points to the other. These approximations are not unreasonable, since it has been empirically observed that all such objects are likely to have similar lifetimes [205].

The analysis algorithm represents the program as a set of *methods*. Each one has a control flow graph of *statements* and a set $\mathtt{V}$ of *local variables* some of which are its *formal parameters* $\mathtt{P} \subseteq \mathtt{V}$. For the purpose of the analysis, only memory-relevant statements are considered: `v=new C` (object allocation), `v=u` (reference copy), `v= u.f` (load), `v.f=u` (store), or $\mathtt{v.m(v}_1\mathtt{,...,v}_n\mathtt{)}$ (method call). To handle call statements, the algorithm relies on a *call graph* which maps each call statement with a set of possible target methods.

The analysis is context-insensitive, that is, a method is analyzed without knowing its callers, but the algorithm computes a result which is a conservative approximation for all calling contexts. The first pass is an intra-procedural analysis of each method m which computes an equivalence relation $\sim_m$ where $\mathtt{v} \sim_m \mathtt{u}$ for all variables `v` and `u` such that `v=u` `v=u.f` or `v.f=u` are statements of m. The second pass is an inter-procedural analysis that works as follows: wherever a method m may call a method m', e.g., there is an arc from m to m' in the call graph, with formal

```
State state;
Aircraft[] crafts;
Vector3d pos;
int hash;

...

public void nearest() {
  int N = crafts.length;
  Container ct = new Container(N);

  for(int i=0; i<N; i++) {
    int h = crafts[i].hashCode();
    Integer e = new Integer(h);
    ct.add(e);
  }

  int min = MAX_VALUE;
  for(int i=0; i<N; i++) {
    Integer g = ct.get(i);
    Vector3d p = state.pos(g);
    float d = pos.dist(p);
    if (d < min) {
      min = d;
      hash = g.intValue();
    }
   }

}
```

```
public class Aircraft {
 final private byte[] val;
 public Aircraft(...) {...}
 public int hashCode() {...}
}

public class Container {
 Object[] data;
 int size;
 int cap;

 public Container(int k)
 {
   size = 0;
   cap = k;
   tmp = new Object[cap];
   data = tmp;
 }

 public void add(Object o)
{
   if (size < cap)
     data[size++] = o;
 }

 public Object get(int i) {
   return data[i];
 }
}
```

Fig. 5.8 A simple example

parameters, say p and q bound to variables, say v and u the algorithm computes $v \sim_m u$ if $p \sim_{m'} q$. That is, if a method connects two of its formal parameters, the effect of this connection is impacted on the appropriate variables of the caller methods.

Figure 5.8 shows a simple Java program to illustrate how the analysis works. The example is inspired in the CDx benchmark suite[7]. CDx [233] implements an aircraft collision detection algorithm based on simulated radar frames. Each aircraft is identified by its call sign, which is indeed a vector of bytes. The system state is a set of aircrafts, together with their current position, a three-dimensional vector.

Given a vector of aircrafts and a position, the `nearest` method computes the hash code of the nearest aircraft to the position. For this, it first computes the

[7] http://adam.lille.inria.fr/soleil/rcd/.

hash codes of the aircrafts, it stores them in a temporal container object of class `Container`, and then uses this container to lookup the nearest one.

For class `Container`, the intra-procedural phase gives `this` $\sim_{Container}$ `tmp`, because of the assignment `data = tmp` in the constructor method `Container`, and `this` $\sim_{add}$ `o`, as a consequence of `data[size++] = o` in method `add`. For the `nearest` method, the analysis results in the singletons {`ct`} and {`e`}. The inter-procedural phase yields `ct` $\sim_{nearest}$ `e`, because `add` is called from `nearest` with `this`$\mapsto$`ct` and `o`$\mapsto$`e`.

For every new statement, we have to determine in which memory area it has to be allocated. For this, we use the parameter to which the variable is related to. If it is not bound to any parameter, then it is allocated in the current scope. In our example, since `this` $\sim_{Container}$ `tmp`, the array of objects referenced by `tmp` is allocated in the memory area of `this`. Since `ct` $\sim_{nearest}$ `e`, objects referenced by `ct` and `e`, are local to `nearest`, that is, they are both allocated in the memory area in which `nearest_run` is executed. In other words, method `nearest` could be run in a region which would contain the objects allocated in `new` statements: `Container ct = new Container(N);` and `Integer e = new Integer(h);` in `nearest`, and `tmp = new Object[cap];` in the constructor method of class `Container` when called from the latter.

5.5.1.1 Static Region Size Analysis

The size of RTSJ memory areas needs to be fixed when they are created. Trying to allocate an object in an already full region will cause a runtime error. Therefore, in order to safely allocate objects in regions it is very important to appropriately bound their size. For this, we provide a mechanism to automatically compute region sizes.

The basic approach consists in counting the number of objects that will live in a given region [77]. For this, we rely on an algorithm like the above for inferring scopes. Then, knowing which `new` statements conform a region, the goal is to quantify how many of them will be executed. Of course, this number may vary from run to run and may depend on some variables. Therefore, the computed value has to be a conservative approximation for all runs, and a parametric expression on relevant program variables. Such an expression can be characterized as the number of solutions of the program invariant defining the iteration space where the `new` statements reside. For linear invariants, the number of (integer) solutions can be expressed as an Ehrhart polynomial [106].

In our example, there are three allocation sites that contribute to the region of method `nearest`:

```
1 : Container ct = new Container(N);
2 : e = new Integer(h);
3 : tmp = new Object[cap];
```

The relevant program variable in this case is the number of aircrafts, given by `crafts.length`. The method allocates one object of type `Container` in a statement which is not within a loop, then:

$$C_1(\texttt{crafts.length}) = 1.$$

The iteration space at `e = new Integer(h);` is characterized by the linear invariant

$$\phi_2 \equiv \{1 \leq \texttt{i} \leq \texttt{N} \wedge \texttt{N} = \texttt{crafts.length}\}.$$

Therefore, the number of integer points in this space is:

$$C_2(\texttt{crafts.length}) = \#\phi_1(\texttt{crafts.length}) = \texttt{crafts.length}.$$

This expression gives the number of instances of type `Integer` created by method `nearest`. The constructor of `Container` allocates an array of `cap` objects that is assigned to a field. Note that the value of `cap` is indeed `crafts.length`. This relation is captured by the linear invariant:

$$\phi_3 \equiv \{\texttt{N} = \texttt{crafts.length} \wedge \texttt{k} = \texttt{N} \wedge \texttt{cap} = \texttt{k} \wedge 1 \leq \textit{arrIns} \leq \texttt{cap}\}.$$

The auxiliary mathematical variable *arrInss* is introduced to model the effect of allocating an array of type `Object[]` of length `cap`, which is similar to allocating `cap` objects of type `Object`. Then, the number of objects allocated by the constructor method of `Container` is:

$$C_3(\texttt{crafts.length}) = \#\phi_3(\texttt{crafts.length}) = \texttt{crafts.length}.$$

Summarizing the amount of objects allocated by `nearest` is:

$$\begin{aligned} C_{\texttt{Container}} &= C_1 = 1 \\ C_{\texttt{Integer}} &= C_2 = \texttt{crafts.length} \\ C_{\texttt{Object[]}} &= C_3 = \texttt{crafts.length} \end{aligned}$$

The computed size $\mathscr{R}_{\texttt{nearest}}$ of the region of method `nearest` is:

$$\begin{aligned} \mathscr{R}_{\texttt{nearest}}(\texttt{crafts.length}) = {} & \mathscr{T}(1, \texttt{Container}) + \\ & \mathscr{T}(\texttt{crafts.length}, \texttt{Integer}) + \\ & \mathscr{A}(\texttt{crafts.length}, \texttt{Object}) \end{aligned}$$

where $\mathscr{T}(n, T)$ and $\mathscr{A}(n, T)$ give the space in bytes occupied by n objects of type T and by an array of n objects of type T, respectively. These functions are specific for each virtual machine.

In addition to compute region sizes for RTSJ, the counting method serves as a basis for predicting overall memory requirements. For instance, [76] proposes a technique which yields a polynomial that bounds from above the amount of memory necessary to *safely* execute a given method without running out of memory. The idea is to leverage on the region-size estimation algorithm to characterize the *maximum* region size of each possible scope entered by a method.

5.5.2 RTSJ Code Generation

The tool presented in [165] shows an approach which integrates region and region-size inference to produce sound and predictable RTSJ scoped-memory managed code from conventional Java code. The tool-suite supports the following approach to assist programmers generate regions. First, it infers memory scopes with the aid of escape analysis. Second, it allows fine-tunning the memory layout by resorting to a region editing tool [161] and explicit program annotations. Third, it computes region sizes and total memory consumption [76, 77]. If the overall solution is not satisfactory, the developer is given detailed information which could be used to refactor the code and repeat the procedure.

To generate RTSJ from plain Java the general idea is as follows. For each method `m` we construct a class, say `Run_m` which implements `Runnable`. The purpose of `Run_m` is to encapsulate the logic of `m` into a `run` method. We also construct an associated method, say `m_scoped` where a `ScopedMemory` of appropriate size is allocated, such that `m` is executed inside. The code of `m` is replaced by a call to `m_scoped`. Then, each new statement is left as is, if the corresponding object is to be allocated in the current scope, or substituted by an allocation in the appropriate region. The RTSJ code for our example is shown in Fig. 5.9.

To know the size of the `MemoryArea` at runtime, the generated method `m_size()` returns a conservative estimation of the amount of memory occupied by the objects to be allocated in the region, based on the expression computed by the region-size inference algorithm, as a function of the parameters of `m`. In RTSJ, this can be done through the `SizeEstimator` class. Figure 5.10 shows a possible implementation of this method for our example.

```
public void nearest() {
  nearest_scoped();
}

class Runnable Run_nearest
      implements Runnable {
  public void run() {
    int N = crafts.length;
    Container ct = new Container(N);

    for(int i=0; i<N; i++) {
      int h = crafts[i].hashCode();
      Integer e   new Integer(h);
      ct.add(e);
    }

    int min = MAX_VALUE;
    for(int i=0; i<N; i++) {
      Integer g = ct.get(i);
      Vector3d p = state.pos(g);
      float d = pos.dist(p);
      if (d < min) {
        min = d;
        hash = g.intValue();
      }
    }
  }
}
%\end{verbatim}
```

```
public void nearest_scoped() {
  int N = crafts.length;
     ScopedMemory r =
     new LTMemory(nearest_size(N));
  Runnable nearest_run =
     new Run_nearest();
  r.enter(nearest_run);
}

public class Container {
 Object[] data;
 int size;
 int cap;

 public Container(int k)
 {
   size = 0;
   cap = k;
       MemoryArea r =
    MemoryArea.getMemoryArea(this);
   tmp = r.newArray(
    Class.forName("Object"), cap);
   data = tmp;
 }

 ...

}

%\end{verbatim}
```

Fig. 5.9 The aircraft example in RTSJ

Fig. 5.10 Region size estimation in RTSJ

```
long nearest_size(int N) {
   SizeEstimator e = new SizeEstimator();
   e.reserve(Class.forName("Container"), 1);
   e.reserve(Class.forName("Integer"), N);
   e.reserveArray(N, Class.forName("Object"));
   return e.getEstimate();
}
%\end{verbatim}
```

5.6 Conclusions

This chapter has presented the RTSJ memory model, its definition and suggested implementation. It has discussed dynamic and static techniques to detect illegal assignments at runtime and to ensure correct use of scopes at compile-time, respectively. It has also described a hardware-based solution to improve the performance.

Region-based memory management is an active area of research. Current work includes improvements of the static analysis approach for synthesizing scopes, in order to determine more precise lifetimes and region sizes. Moreover, there is intensive work to develop industry-standard RTSJ platforms for critical application areas such as avionics [362].

The RTSJ has introduces an unfamiliar programming model that is difficult and error prone to use. The difficulty of programming with RTSJ scoped memory is remarkable and error-prone, due to scoped memory regions usage. An alternative to this complex model RTSJ is a real-time GC. However, real-time GC is not suited for all real-time applications (e.g., safety-critical).

In this chapter, we review two different solutions to simplify the complexity of RTSJ memory regions by redefining or eliminating the rules introducing complexity (i.e., the single parent rule). From our point of view the single parent rule must be avoided or redefined, which is imperative regarding critical applications. More over, within a safety-critical environment (i.e., JSR-320), recent research [304] simplifies the RTSJ scoped memory model by avoiding the scoped cactus stacks, which implicitly redefines the single parent rule (i.e., a scoped memory area can have only one parent along its life). It could be also possible, to eliminate the assignment rules by redesigning a more complex collector for scoped memory regions guarantee the safety execution of the application and it deterministic behavior. We are currently exploring this last and interesting solution.

Acknowledgements This research was supported by the Research Program S2009/TIC-1468, and by Ministerio de Educación y Ciencia, through the research grant TIN2009-.07146.

Chapter 6
Programming Embedded Systems: Interacting with the Embedded Platform

Peter Dibble, James J. Hunt, and Andy J. Wellings

Abstract Interacting with the underlying platform and its environment is one of the key activities for embedded programs. This chapter focuses on how real-time Java programs can access the platform's physical memory so that it can maximize performance, and how it can interface with external devices. The revised *physical* and *raw* memory frameworks of Version 1.1. of the Real-Time Specification for Java are introduced along with the revised *happening* model. The latter now includes facilities to program first-level interrupt handlers. An extended example illustrates how many of the facilities can be used.

6.1 Introduction

There is a wide range of topics associated with the programming of embedded systems: from the high level abstraction models, through issues of concurrency to low-level control of resources and access to the external environment. This chapter focuses on the lower-level issues. In particular, it addresses how the real-time Java programmer can interact with the embedded platform on which its execution environment is hosted. This chapter assume that this execution environment supports the *Real-time Specification for Java* (RTSJ).

P. Dibble
TimeSys, Pittsburg, USA
e-mail: peter.dibble@timesys.com

J.J. Hunt
aicas GmbH, Karlsruhe, Germany
e-mail: jjh@aicas.com

A.J. Wellings (✉)
Department of Computer Science, University of York, York, UK
e-mail: andy@cs.york.ac.uk

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_6,

Even within this much narrower range of topics, there are many issues to be addressed, including how to

1. Control the placement of objects in physical memory in order to use efficiently the variety of memory types found in modern embedded system;
2. Interact with the environment in which the system is embedded, including access to input and output devices.

Interacting with the embedded environment has always presented Java and the RTSJ with severe technical challenges. While it is easy to define standard classes to provide specific support, often these are implemented using the *Jave Native Interface* (JNI). The RTSJ (as of version 1.0.2) attempts to provide more direct Java-level facilities addressing these issues. However, most RTSJ vendors have shied away from supporting this area of the specification, as some of the presented APIs lack clarity and their underlying models are often ill defined.

Under the auspices of the Java Community Process, the *Java Specification Request* (JSR) 282 expert group is tasked with reassessing its support in this area. The goal of this chapter is to review its progress, and to summarise the issues and problems that it has faced. Section 6.2 considers the RTSJ physical memory manager and its support for physical memory areas. Section 6.3 considers the role of asynchronous events, happenings, and raw memory access in supporting interactions with the external environment.

An extended example is presented in Sect. 6.4 in order to illustrate many of the mechanisms discussed in this chapter. Finally, Sect. 6.5 summarises the strengths and limitations of the RTSJ's approach, and outlines the support that is likely to be provided in Version 1.1 of the specification.

6.2 Physical Memory Types

Embedded systems may have many different types of directly addressable memory available to them. Each type has its own characteristics [29] that determine whether it is

- Volatile – whether it maintains its state when the power is turned off;
- Writable – whether it can be written at all, written once or written many times and whether writing is under program control,
- Synchronous or asynchronous – whether the memory is synchronized with the system bus,
- Erasable at the byte level – if the memory can be overwritten whether this is done at the byte level or whether whole sectors of the memory need to be erased,
- Fast to access – both for reading and writing.

Examples include the following [29]:

- *Dynamic Random Access Memory* (DRAM) and *Static Random Access Memory* (SRAM) – these are volatile memory types that are usually writable at the

byte level. There are no limits on the number of times the memory contents can be written. From the embedded systems designer's view point, the main differences between the two are their access times and their costs per byte. SRAM has faster access times and is more expensive. Both DRAM and SRAM are example of asynchronous memory, SDRAM and SSRAM are their synchronized counterparts. Another important difference is that DRAM requires periodic refresh operations, which may interfere with execution time determinism.

- Read-Only Memory (for example, *Erasable Programmable Read-Only Memory* (EPROM)) – these are nonvolatile memory types that once initialized with data can not be overwritten by the program (without recourse to some external effect, usually ultraviolet light as in EPROM). They are fast to access and cost less per byte than DRAM.
- Hybrid Memory (for example, *Electrically Erasable Programmable Read-Only Memory* (EEPROM), and Flash) – these have some properties of both random access and read-only memory.
  - EEPROM – this is nonvolatile memory that is writable at the byte level. However, there are typically limits on how many time the same location can be overwritten. EEPROMs are expensive to manufacture, fast to read but slow to write.
  - FLASH memory – this is nonvolatile that is writable at the sector level. Like EEPROM there are limits on how many times the same location can be overwritten and they are fast to read but slow to write. Flash memory is cheaper to manufacture than EEPROM.

Some embedded systems may have multiple types of random-access memory, and multiple ways of accessing memory. For instance, there may be a small amount of very fast RAM on the processor chip, memory that is on the same board as the processor, memory that may be added and removed from the system dynamically, memory that is accessed across a bus, access to memory that is mediated by a cache, access where the cache is partially disabled so all stores are "write through", memory that is demand paged, and other types of memory and memory-access attributes only limited by physics and the imagination of electrical engineers. Some of these memory types will have no impact on the programmer, others will.

Individual computers are often targeted at a particular application domain. This domain will often dictate the cost and performance requirements, and therefore, the memory type used. Some embedded systems are highly optimized and need to explore different options in memory to meet their performance requirements. Here are five example scenarios.

- Ninety percent of performance-critical memory access is to a set of objects that could fit in a half the total memory.
- The system enables the locking of a small amount of data in the cache, and a small number of pages in the translation lookaside buffer (TLB). A few very frequently accessed objects are to be locked in the cache and a larger number of objects that have jitter requirements can be TLB-locked to avoid TLB faults.

- The boards accept added memory on daughter boards, but that memory is not accessible to DMA from the disk and network controllers, and it cannot be used for video buffers. Better performance is obtained if we ensure that all data that might interact with disk, network, or video is not stored on the daughter board.
- Improved video performance can be obtained by using an array as a video buffer. This will only be effective if a physically contiguous, non-pageable, DMA-accessible block of RAM is used for the buffer and all stores forced to write through the cache. Of course, such an approach is dependent on the way the JVM lays out arrays in memory, and it breaks the JVM abstraction by depending on that layout.
- The system has banks of SRAM and saves power by automatically putting them to "sleep" whenever they stay unused for 100 ms or so. To exploit this, the objects used by each phase of the program can be collected in a separate bank of this special memory.

To be clear, few embedded systems are this aggressive in their hardware optimization. The majority of embedded systems have only ROM, RAM, and maybe flash memory. Configuration-controlled memory attributes (such as page locking, and TLB behavior) are more common.

As well as having different types of memory, many computers map input and output devices so that their registers can be accessed as if they were resident within the computer memory. Hence, some parts of the processor's address space map to real memory and other parts map to device registers. Logically, even a device's memory can be considered part of the memory hierarchy, even where the device's interface is accessed through special assembly instructions. Multiprocessor systems add a further dimension to the problem of memory access. Memory may be local to a CPU, tightly shared between CPUs, or remotely accessible from the CPU (but with a delay).

Traditionally, Java programmers are not concerned with these low-level issues; they program at a higher level of abstraction and assume the JVM makes judicious use of the underlying resources provided by the execution platform.[1] Embedded systems programmers cannot afford this luxury. Consequently, any Java environment that wishes to facilitate the programming of embedded systems must enable the programmer to exercise more control over memory.

In Chaps. 4 and 5 the general memory model of the RTSJ is discussed. Chapter 4 considers heap memory and the issue of real-time garbage collection. The notions of scoped and immortal memory areas are introduced in Chap. 5. These are memory areas that allow objects to be created and stored outside of the Java heap. The actual memory associated with a particular memory area is called its *backing store*. The physical memory framework of the RTSJ extends the model to

[1]This is reflected by the OS support provided. For example, most POSIX systems only offer programs a choice of demand paged or page-locked memory.

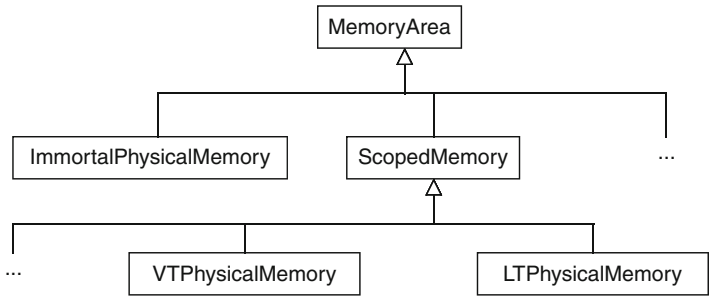


Fig. 6.1 The abridged `MemoryArea` class hierarchy

1. Allow the programmer to create immortal and scoped memory areas with particular physical and virtual memory characteristics, and
2. Provide some memory management infrastructure that facilitates the execution of a JVM between similar platforms; where the main difference between the platforms is the memory types provided.

The raw memory framework of the RTSJ can be used to access device registers. It is not concerned with storage for objects. Discussion of this is deferred until Sect. 6.3.2.

6.2.1 Problems with the Current RTSJ 1.02 Physical Memory Framework

The RTSJ supports three ways to allocate objects that can be placed in particular types of memory:

- `ImmortalPhysicalMemory` allocates immortal objects in memory with specified characteristics.
- `LTPhysicalMemory` allocates scoped memory objects in a memory with specified characteristics using a linear time memory allocation algorithm.
- `VTPhysicalMemory` allocates scoped memory objects in memory with specified characteristics using an algorithm that may be worse than linear time but could offer extra services (such as extensibility).

The only difference between the physical memory classes and the corresponding non-physical classes is that the ordinary memory classes give access to normal system RAM and the physical memory classes offer access to particular types of memory. Figure 6.1 shows the abridged class hierarchy.

The RTSJ supports access to physical memory via a memory manager and one or more memory filters. The goal of the memory manager is to provide a single interface with which the programmer can interact in order to access memory with

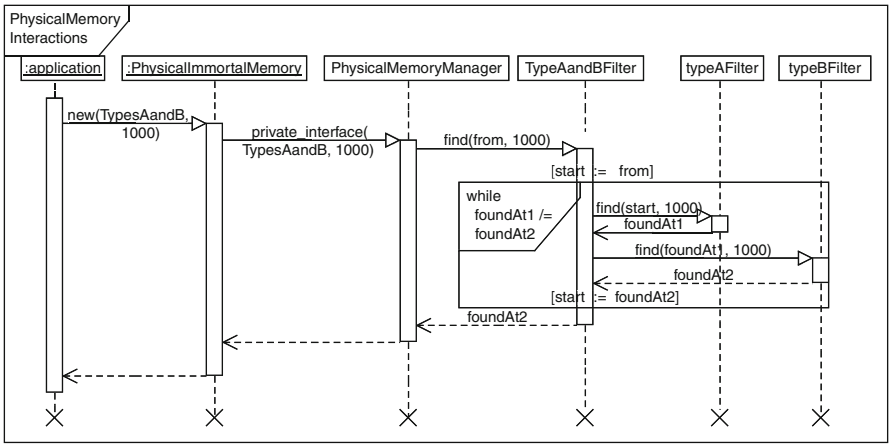


Fig. 6.2 Finding physical memory

a particular characteristic. A memory filter enables access to a particular type of physical memory. Memory filters may be dynamically added and removed from the system, and there can only be a single filter for each memory type. The memory manager is unaware of the physical addresses of each type of memory. This is encapsulated by the filters. The filters also know the virtual memory characteristics that have been allocated to their memory type. For example, whether the memory is readable or writable.

In theory, any developer can create a new physical memory filter and register it with the PMM. However, the programming of filters is difficult for the following reasons.

- Physical memory type filters include a memory allocation function that must respond to allocation requests with whether a requested range of physical memory is free and if it is not, the physical address of the next free physical memory of the requested type. This is complex because requests for compound types of physical memory must find a free segment that satisfies all attributes of the compound type. Figure 6.2 shows a possible sequence of actions to create a 1,000 byte physical immortal memory in physical memory of type A and B characteristics.
- The Java runtime must continue to behave correctly under the Java memory model when it is using physical memory. This is not a problem when a memory type behaves like the system's normal RAM with respect to the properties addressed by the memory model, or is more restricted than normal RAM (as, for instance, write-through cache is more restricted than copy-back cache). If a new memory type does not obey the memory model using the same instruction sequences as normal RAM, the memory filter must cooperate with the interpreter, the JIT, and any ahead-of-time compilation to modify those instruction sequences

when accessing the new type of memory. That task is difficult for someone who can easily modify the Java runtime and nearly impossible for anyone else.

Hence, the utility of the physical memory filter framework at Version 1.02 is questionable, and needs to be replaced in Version 1.1 with an easier to use framework.

6.2.2 *Proposal for the RTSJ Version 1.1 Physical Memory Framework*

The main problem with the current framework is that it places too greater a burden on the JVM implementer. Even for embedded systems, the JVM implementer requires the VM to be portable between systems within the same processor family. It, therefore, cannot have detailed knowledge of the underlying memory architecture. It is only concerned with the standard RAM provided to it by the host operating system.

The proposal for the Version 1.1 model delegates detailed knowledge of the memory architecture to the programmer of the specific embedded system to be implemented. There is less requirement for portability here, as embedded systems are usually optimized for their host environment. The proposal assumes that the programmer is aware of the memory map, either through some native operating system interface[2] or from some property file read at program initialization time.

When accessing physical memory, there are two main considerations:

1. The characteristics of the required physical memory, and
2. How that memory is to be mapped into the virtual memory of the application.

The RTSJ Version 1.1 proposal requires the program to identify (and inform the RTSJ's physical memory manager of) the physical memory characteristics and the range of physical addresses those characteristic apply to. For example, that there is SRAM between physical address range 0x100000000 and 0xA0000000. The proposal assumes that all memory supports the Java memory model in the same manner as the main RAM for the host machine. Hence, no extra compiler or JVM interactions are required.

The physical memory manager supports a range of options for mapping physical memory into the virtual memory of the application. Examples of such options are whether the range is to be permanently resident in memory (the default is that it may be subject to paging/swapping), and whether data is written to the cache and the main memory simultaneously (i.e., a write through cache).

---

[2]For example, the *Advanced Configuration and Power Interface* (ACPI) specification is an open standard for device configuration and power management by the operating system. The ACPI defines platform-independent interfaces for hardware discovery, configuration, power management and monitoring. See http://www.acpi.info/.

Given the required physical and virtual memory characteristics, the programmer requests that the PMM creates a memory filter for accessing this memory. This filter can then be used with new constructors on the physical memory classes. For example,

```
public ImmortalPhysicalMemory(PhysicalMemoryFilter filter, long size)
```

Use of this constructor enables the programmer to specify the allocation of the backing store in a particular type of memory with a particular virtual memory characteristic. The filter is used to locate an area in physical memory with the required physical memory characteristics and to direct its mapping into the virtual address space. Other constructors allow multiple filters to be passed.

Hence, once the filters have been created, physical memory areas can be created and objects can be allocated within those memory areas using the usual RTSJ mechanisms for changing the allocation context of the `new` operator.

6.2.2.1 The New Physical Memory Manager Infrastructure

The new physical memory manager and its infrastructure consists of the following six main components.

1. A class, `PhysicalMemoryModule`, that allows a range of physical memory addresses to be specified.

```
package javax.realtime;
final class PhysicalMemoryModule {
  PhysicalMemoryModule(long base, long length) {
    // base is a physical address
    // length is size of contiguous memory from that address
  }

  // Getter methods only
  public long getBase();
  public long getLength();
}
```

2. A tagging interface used to identify physical memory characteristics.

```
package javax.realtime;
public interface PhysicalMemoryCharacteristic {};
```

   So, for example, the declaration

```
final PhysicalMemoryCharacteristic STATIC_RAM = new ...;
```

   could be used by the programmer to identify SRAM memory.
3. A similar tagging interface used to define virtual memory characteristics, and a set of predefined reference variables (held within the `PhysicalMemoryManager` class).

```
package javax.realtime;
public interface VirtualMemoryCharacteristic {};
```

```
public class PhysicalMemoryManager {
  public final static
         VirtualMemoryCharacteristic PERMANENTLY_RESIDENT;
  public final static
         VirtualMemoryCharacteristic LOCKED_IN_CACHE;
  public final static
         VirtualMemoryCharacteristic WRITE_THROUGH_CACHE;
  ...
}
```

4. An interface to the physical memory filters – filters are created by a factory in the `PhysicalMemoryManager` class.

```
package javax.realtime;
public interface PhysicalMemoryFilter {
  public void setVMCharacteristics(
              VirtualMemoryCharacteristics required []);
}
```

5. The *Physical Memory Manager* (PMM) itself, which coordinates access to the memory.

```
package java.realtime;
public class NewPhysicalMemoryManager {
   // definition of supported VMCharacteristics
   ...
   // The following methods allow the programmer to associate
   // programmer-defined names with ranges of physical
   // memory addresses
   static public void associate(PhysicalMemoryCharacteristic name,
                PhysicalMemoryModule module)};

   static public void associate(
               PhysicalMemoryCharacteristic names [],
               PhysicalMemoryModule module);

   static public void associate(PhysicalMemoryCharacteristic name,
               PhysicalMemoryModule module []);

   /*
    * Each physical memory module can have more than one
    * physical memory characteristic (PMC):
    *   characteristics(PhysicalMemoryModule) ->
        |F PhysicalMemoryCharacteristic
    * A physical memory characteristic can map to many modules:
    *   modules(PhysicalMemoryCharacteristic ) ->
    *           |F PhysicalMemoryModule
    * The PMM determines the physical addresses from
    * the modules and keeps a relation between
    *    PhysicalMemoryModule <-> Physical Memory Addresses
    * The range of physical addresses of modules
    *    should not overlap
    * To find a memory range that supports PMC A and PMC B
    * uses set intersection
    *    modules(A) ∩ modules(B)
    */
```

```
    static public PhysicalMemoryFilter createFilter(
        PhysicalMemoryCharacteristic PMcharacteristics [],
        VirtualMemoryCharacteristic VMcharacteristics []);
}
```

6. The `createFilter` method now has the responsibility of determining the appropriate place in memory that meets the requirements and how this memory will be mapped into the virtual address space. The setting aside of the memory is performed by the constructors of the physical memory classes themselves. E.g.

```
package javax.realtime;
public class ImmortalPhysicalMemory extends MemoryArea {

  public ImmortalPhysicalMemory(PhysicalMemoryFilter filter,
                                long size);
    /*
     * The constructor talks to the PMM, who
     * finds the appropriate physical memory area and
     * uses the POSIX mmap and /dev/mem
     * to map into the virtual address space. Read/Write access
     * is granted. The flag is set to MAP_SHARED.
     * Any virtual memory characteristics are set at this time.
     * The JVM then uses the mapped memory as backing store.
     * Exceptions are thrown if the request cannot be met.
     */
  ...
}
```

6.2.3 An Example

Consider an example of a system that has a SRAM physical memory module configured at a physical base address of `0x10000000` and of length `0x20000000`. Another model (base address of `0xA0000000` and of length `0x10000000`) also supports SRAM, but this module has been configured so that it saves power by sleeping when not in use. The following subsections illustrate how the embedded programmer informs the PMM about the structure during the program's initialization phase, and how the memory may be subsequently used after this. The example assumes that the PMM supports the virtual memory characteristics defined above.

6.2.3.1 Program Initialization

For simplicity, the example requires that the address of the memory modules are known, rather than being read from a property file. The program needs to have a class that implements the `PhysicalMemoryCharacteristic`. In this simple example, this is empty.

```
public class MyName implements PhysicalMemoryCharacteristic {}
```

The initialization method must now create instances of the `PhysicalMemoryModule` class to represent the physical memory module memory modules to represent

```
PhysicalMemoryModule staticRam = new PhysicalMemoryModule(
                                 0x10000000L, 0x100000000L);
PhysicalMemoryModule staticSleepableRam = new PhysicalMemoryModule(
                                 0xA0000000L, 0x100000000L);
```

It then creates names for the characteristics that the program wants to associate with each memory module.

```
PhysicalMemoryCharacteristic STATIC_RAM = new MyName();
PhysicalMemoryCharacteristic AUTO_SLEEPABLE = new MyName();
```

It then informs the PMM of the appropriate associations:

```
NewPhysicalMemoryManager.associate(STATIC_RAM, staticRam);
NewPhysicalMemoryManager.associate(STATIC_RAM,
                                   staticSleepableRam);
NewPhysicalMemoryManager.associate(AUTO_SLEEPABLE,
                                   staticSleepableRam);
```

Once this is done, the program can now create a filter with the required properties. In this case it is for some SRAM that must be auto sleepable.

```
PhysicalMemoryCharacteristic [] PMC =
        new PhysicalMemoryCharacteristic[2];
PMC[0] = STATIC_RAM;
PMC[1] = AUTO_SLEEPABLE;
VirtualMemoryCharacteristic [] VMC =
        new VirtualMemoryCharacteristic[1];
VMC[0] = NewPhysicalMemoryManager.PermanentlyResident;
PhysicalMemoryFilter filter = NewPhysicalMemoryManager.
                              createFilter(PMC,VMC);
```

If the program had just asked for SRAM then either of the memory modules could satisfy the request.

The initialization is now complete, and the programmer can use the memory for storing objects, as shown below.

6.2.3.2 Using Physical Memory

Once the programmer has configured the JVM so that it is aware of the physical memory modules, and the programmer names for characteristics of those memory modules, using the physical memory is straight forward. Here is an example.

```
ImmortalPhysicalMemory IM = new ImmortalPhysicalMemory(filter,
                                                       0x1000);
IM.enter(new Runnable() {
    public void run() {
      // The code executing here is running with its allocation
      // context set to a physical immortal memory area that is
```

```
      // mapped to RAM which is auto sleepable.
      // Any objects created will be placed in that
      // part of physical memory.
    }
  });
```

It is the appropriate constructor of the physical memory classes that now interfaces with the PMM. The physical memory manager keeps track of previously allocated memory and is able to determine if memory is available with the appropriate characteristics. Of course, the PMM has no knowledge of what these names mean; it is merely providing a look-up service.

6.3 Interacting with the Environment

Interacting with the external environment in a timely manner is an important requirement for real-time embedded systems. From an embedded systems' perspective, all interactions are performed by input and output devices. Hence, the problem is one of controlling and monitoring of devices. This has always been an area not considered by Java. The RTSJ provides additional features to support such activites.

There are at least three execution (runtime) environments for the RTSJ:

1. On top of a real-time operating system where the Java application runs in user mode;
2. As part of an embedded device where the Java application runs stand-alone on a hardware/software virtual machine; and
3. As a "kernel module" incorporated into a real-time kernel where both kernel and application run in supervisor mode.

In execution environment (1), interaction with the embedded environment will usually be via operating system calls using Java's connection-oriented APIs. The Java program will typically have no direct access to the I/O devices. Although some limited access to physical memory may be provided, it is unlikely that interrupts can be directly handled. However, asynchronous interaction with the environment is still possible, for example, via POSIX signals. In execution environments (2) and (3), the Java program may be able to directly access devices and handle interrupts.

A device can be considered to be a processor performing a fixed task. A computer system can, therefore, be considered to be a collection of parallel threads. There are several models by which the device 'thread' can communicate and synchronize with the tasks executing inside the main processor. The interface to a device is usually through a set of device registers. Depending on the I/O architecture of the processor, the programmer can either access these registers via predetermined memory location (called *memory mapped I/O*) or via special assembler instructions (called *port-mapped I/O*).

All high-level models of device programming must provide [92]:

1. A suitable representation of interrupts (if interrupts are to be handled), and
2. Facilities for representing, addressing and manipulating device registers.

Version 1.0 of the RTSJ went some way towards supporting this model through the notion of *happenings* and the *physical and raw memory* access facilities. Unfortunately, happenings were under defined, and the mechanisms for physical and raw memory were overly complex with no clear delineation of the separations of concerns between application developers and JVM implementers.

Version 1.1 has significantly enhanced the support for happenings, and has provided a clearer separation between physical and raw memory.

6.3.1 Interrupts and Other External Events

Regehr [320] defines the terms used for the core components of interrupts and their handlers as follows.

- *Interrupt* – a hardware supported asynchronous transfer of control mechanism initiated by an event external to the processor. Control of the processor is transferred through an interrupt vector.
- *Interrupt vector* – a dedicated (or configurable) location that specifies the location of an interrupt handler.
- *Interrupt handler* – code that is reachable from the interrupt vector.
- *An interrupt controller* – a peripheral device that manages interrupts for the processor.

He further identifies the following problems with programming interrupt-driven software on single processors:

- Stack overflow – the difficulty determining how much call-chain stack is required to handle an interrupt. The problem is compounded if the stack is borrowed from the currently executing thread or process.
- Interrupt overload – the problem of ensuring that non-interrupt driven processing is not swamped by unexpected or misbehaving interrupts.
- Real-time analysis – the need to have appropriate schedulability analysis models to bound the impact of interrupt handlers.

The problems above are accentuated in multiprocessor systems where interrupts can be handled globally. Fortunately, many multiprocessor systems allow interrupts to be bound to particular processors. For example, the ARM Cortex A9-MPCore supports the Arm Generic Interrupt Controller.[3] This enables a target list of CPUs to be specified for each hardware interrupt. Software generated interrupts can also be sent to the list or set up to be delivered to all but the requesting CPU or only the requesting CPU.

[3] See http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0375a/Cegbfjhf.html.

Regehr's problems are all generic and can be solved irrespective of the language used to implement the handlers. In general they can be addressed by a combination of techniques.

- Stack overflow – static analysis techniques can usually be used to determine the worst-case stack usage of all interrupt handlers. If stack is borrowed from the executing thread then this amount must be added to the worst-case stack usage of all threads.
- Interrupt overload – this is typically managed by aperiodic server technology in combination with interrupt masking (see Sect. 13.6 of [92]).
- Real-time analysis – again this can be catered for in modern schedulability analysis techniques, such as response-time analysis (see Sect. 14.6 of [92]).

From a RTSJ perspective, the following distinctions are useful

- The *first-level interrupt handlers* are the codes that the platform execute in response to the hardware interrupts (or traps). A first-level interrupt is assumed to be executed at an execution eligibility (priority) and by a processor dictated by the underlying platform (which may be controllable at the platform level). On some RTSJ implementations it will not be possible to write Java code for these handlers. Implementations that do enable Java-level handlers may restrict the code that can be written. For example, the handler code should not suspend itself or throw unhandled exceptions. The RTSJ Version 1.1 optional `InterruptServiceRoutine` class supports first-level interrupt handling.
- The *external event handler* is the code that the JVM executes as a result of being notified that an external event (be it an operating system signal, an ISR or some other program) is targeted at the RTSJ application. The programmer should be able to specify the processor affinity and execution eligibility of this code. In RTSJ 1.1, all external events are represented by instances of the `Happening` interface. Every happening has an associated dispatcher which is responsible for the initial response to an occurrence of the event.
- A happening dispatcher is able to find one or more associated RTSJ asynchronous events and fire them. This then releases the associated asynchronous event handlers. *A happening dispatcher is not required to fire events. If appropriate, the dispatcher can provide an in-line handler itself.*

6.3.1.1 Interrupt Service Routine

Handling interrupts is a necessary part of many embedded systems. Interrupt handlers have traditionally been implemented in assembler code or C. With the growing popularity of high-level concurrent languages, there has been interest in better integration between the interrupt handling code and the application. Ada, for example, allows a "protected" procedure to be called directly from an interrupt [91].

In Java-based systems, JNI is typically used to transfer control between the assembler/C *interrupt service routine* (ISR) and the program. Version 1.1 of the

RTSJ supports the possibility of the ISR containing Java code. This is clearly an area where it is difficult to maintain the portability goal of Java. Furthermore, not all RTSJ deployments can support `InterruptServiceRoutine`. A JVM that runs in user space does not generally have access to interrupts.

The JVM must either be standalone, running in a kernel module, or running in a special I/O partition on a partitioning OS where interrupts are passed through using some virtualization technique. Hence, JVM support for ISR is not required for RTSJ compliance.

Interrupt handling is necessarily machine dependent. However, the RTSJ tries to provide an abstract model that can be implemented on top of all architectures. The model assumes that

- The processor has a (logical) interrupt controller chip that monitors a number of *interrupt lines*;
- The interrupt controller may associate each interrupt line with a particular interrupt priority;
- Associated with the interrupt lines is a (logical) interrupt vector that contains the address of the ISRs;
- The processor has instructions that allow interrupts from a particular line to be disabled/masked irrespective of whether (or the type of) device attached;
- Disabling interrupts from a specific line may disables the interrupts from lines of lower priority;
- A device can be connected to an arbitrary interrupt line;
- When an interrupt is signalled on an interrupt line by a device, the processor uses the identity of the interrupt line to index into the interrupt vector and jumps to the address of the ISR; the hardware automatically disables further interrupts (either of the same priority or, possibly, all interrupts);
- On return from the ISR, interrupts are automatically re-enabled.

For each of the interrupt, the RTSJ has an associated hardware priority that can be used to set the ceiling of an ISR object. The RTSJ virtual machine uses this to disable the interrupts from the associated interrupt line, and lower priority interrupts, when it is executing a synchronized method of the interrupt-handling object. For the handler routine (`handle` method), this is done automatically by the JVM if the `handle` method is flagged as synchronized.

Support for interrupt handling is encapsulated in the `InterruptServiceRoutine` abstract class that has two main methods. The first is the final `register` method that will register an instance of the class with the system so that the appropriate interrupt vector can be initialised. The second is the abstract `handle` method that provides the code to be executed in response to the interrupt occurring. An individual real-time JVM may place restrictions of the code that can be written in this method. The process is illustrated in Fig. 6.3, and is described below.

1. The ISR is created by some application real-time thread.
2. The created ISR is registered with the JVM, the interrupt id is passed as a parameter.

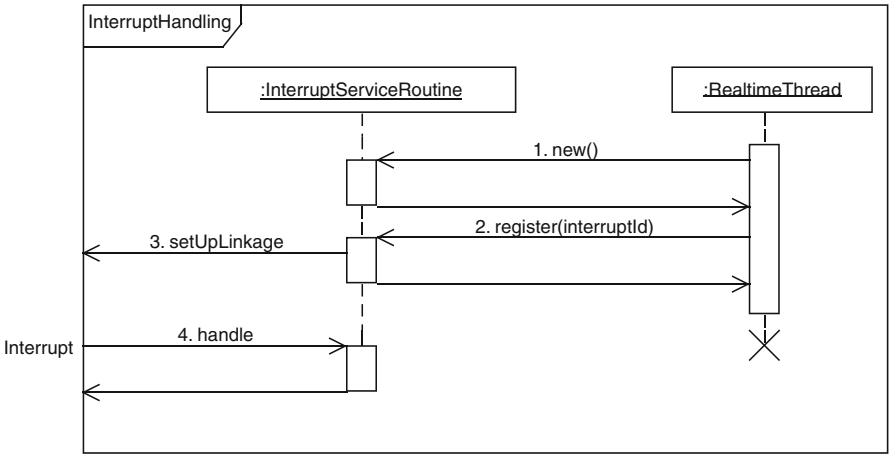


Fig. 6.3 Interrupt servicing

3. As part of the registration process, some internal interface is used to set up the code that will set the underlying interrupt vectors to some C/assembler code that will provide the necessary linkage to allow the callback to the Java handler.
4. When the interrupt occurs, the handler is called.

In order to integrate further the interrupt handling with the Java application, the handle method may trigger a happening or fire an event.

Typically an implementation of the RTSJ that supports first-level interrupt handling will document the following items:

1. For each interrupt, its identifying integer value, the priority at which the interrupt occurs and whether it can be inhibited or not, and the effects of registering ISRs to non inhibitable interrupts (if this is possible).
2. Which run-time stack the `handle` method uses when it executes.
3. Any implementation- or hardware-specific activity that happens before the `handle` method is invoked (e.g., reading device registers, acknowledging devices).
4. The state (inhibited/uninhibited) of the nonreserved interrupts when the program starts; if some interrupts are uninhibited, what is the mechanism a program can use to protect itself before it can register the corresponding ISR.
5. The treatment of interrupt occurrences that are generated while the interrupt is inhibited; i.e., whether one or more occurrences are held for later delivery, or all are lost.
6. Whether predefined or implementation-defined exceptions are raised as a result of the occurrence of any interrupt (for example, a hardware trap resulting from a segmentation error), and the mapping between the interrupt and the predefined exceptions.
7. On a multi-processor, the rules governing the delivery of an interrupt occurrence to a particular processor.

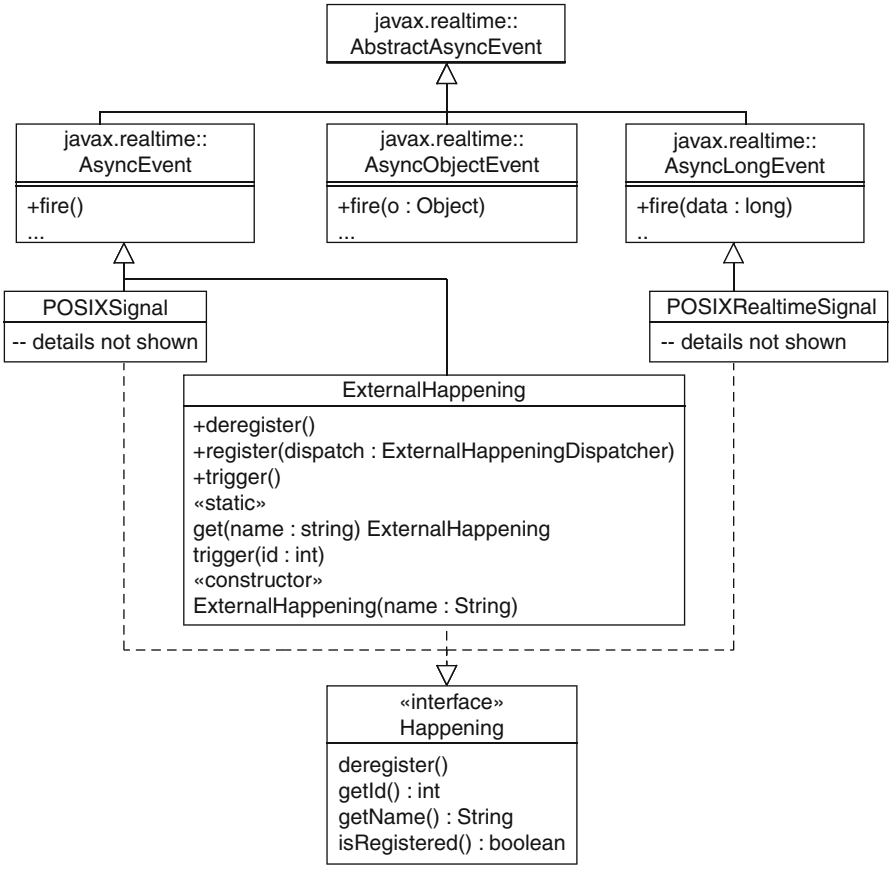


Fig. 6.4 Abridged `AsyncEvent` class hierarchy

6.3.1.2 Happenings and Their Dispatchers

A *happening* is an asynchronous event that is triggered by the environment external to the RTSJ VM. Different types are supported, including: a POSIX signal, a POSIX real-time signal, or a general external happening (such as might be triggered from an ISR or another program running on the same platform). Figure 6.4 illustrates the revised class hierarchy for Version 1.1.

The two main components of the RTSJ's Version 1.1 happenings model are the following:.

- The `Happening` interface – all happenings must implement this interface. When a happening is triggered, by default, the RT JVM calls the `fire` method. This activity is called *dispatching*.

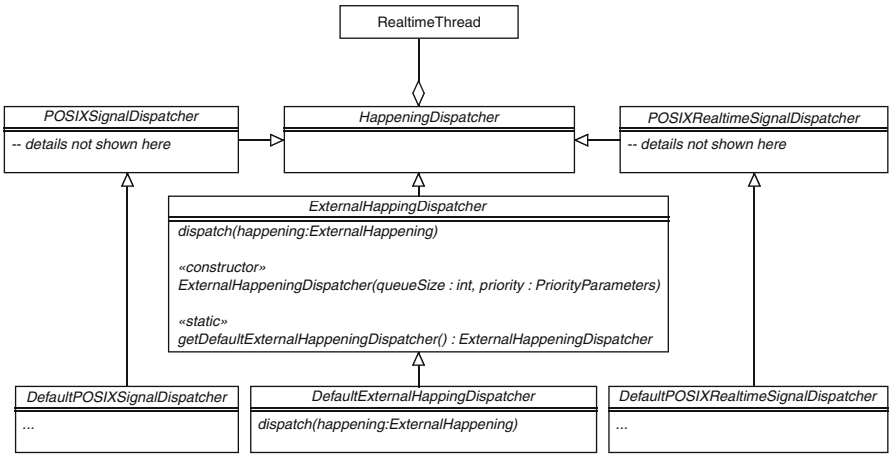


Fig. 6.5 Abridged `HappeningDispatcher` class hierarchy

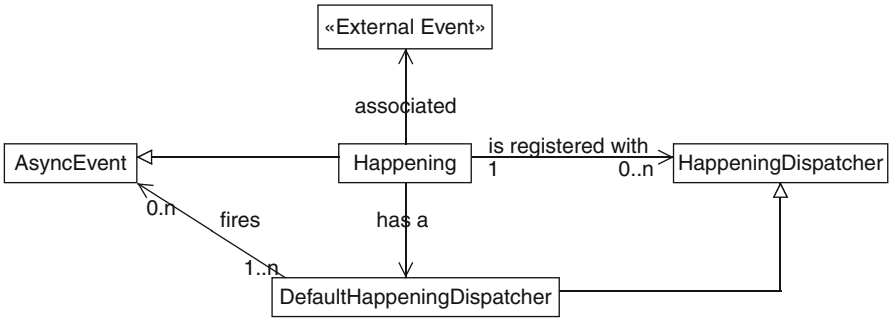


Fig. 6.6 Happening model

- `HappeningDispatcher` – the default dispatching behaviour can be changed by the programmer by *registering* instances of the `HappeningDispatcher` class with an instance of the `Happening` interface. Figure 6.5 shows an abridged version of this class hierarchy.

The model is illustrated in Fig. 6.6.

The occurrence of an external event results in notification being delivered to the real-time JVM. The default behaviour of this routine is to trigger an associated happening. This, in turn, triggers a related happening dispatcher whose default behaviour is to fire any asynchronous events that have been attached to the happening. This default behaviour is run at the highest priority irrespective of the required response latency. If there are many events to be fired, this can result in significant priority inversion. The happening dispatcher gives the programmer more control over the process. This is illustrated if Fig. 6.7.

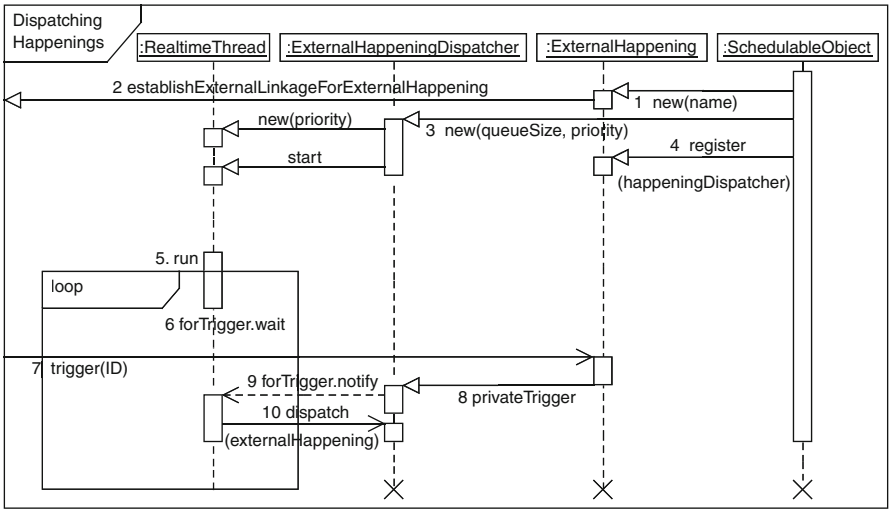


Fig. 6.7 Happening dispatching

1. To interact with an external event, it is necessary to create an `External-Happening`.
2. The JVM provides the linkage necessary to connect to the external event – so that when the external events occur, the appropriate happening is triggered.
3. To change the default dispatching behaviour for a happening, it is necessary to create an external happening dispatcher. As the dispatcher may have to queue up the triggered happenings, the size of the queue can be specified at construction time. The priority of the encapsulated real-time thread can be set. The thread is created and started.
4. The created dispatcher must be registered with the created happening by application code.
5. The encapsulated thread is started.
6. The `run` method loops waiting to be informed of the triggering of a happening.
7. An occurrence of the associated external event will results in the static `ExternalHappening.trigger` method being called with the id of the external happening.
8. The `trigger` method determines the appropriate dispatcher and calls a private method.
9. The private trigger method notifies the waiting thread that the external event has occurred.
10. Once notified, the `dispatch` method is called to perform the dispatching.

The following points should be noted.

- The above initialization sequence of events has been shown in Fig. 6.7 as being performed by a single schedulable object, but this need not be the case.

- The sequence shows the dispatcher's real-time thread waiting for the trigger. This may not be the case, the trigger might arrive first. Outstanding triggering events are queued.
- The default dispatching code can be changed by overriding the `dispatch` method in the happening dispatcher.
- Inside the loop, waiting for any of the associated happening to be triggered has been shown with a monitor-like wait and notify, although in practice this is likely to be a lower-level semaphore-like operation inside the RTSJ infrastructure.

Although the above discussion has been within the context of an external happening, the same approach can be provided for signal handling.

6.3.2 Accessing Device Registers and Other Memory Locations

In real-time and embedded Java systems, it is usually necessary to allow memory to be accessed outside the object model (but under control of the virtual machine). This is necessary for two main reasons [425].

- The memory is being used by a memory-mapped I/O device.
- The memory is being read/written by another application program external to the Java application.

The use of native methods (usually written in C) are the traditional Java work-around. However, using C codes makes formal analysis nearly impossible. Also, none of the Java security and protections mechanisms function in native methods [134]. The RTSJ has introduced the notion of *raw memory* to help remove the need to resort to native methods when accessing specific memory locations.

Clearly, accessing raw memory is very dangerous, and it is the job of the real-time virtual machine to ensure that the integrity of the Java system is not undermined. For example, the application should be prohibited from reading and writing raw memory that has been allocated to objects, say, via `ImmortalPhysicalMemory`. This is because the memory may contain references to Java objects that, if written to, would corrupt the system. Furthermore, again for safety reasons, only primitive data types (`bytes`, `ints`, `longs`, `floats` and `doubles`) can be read and written.

The final related requirement in this area is to provide a mechanism whereby access to device registers, using special machine instructions, can be facilitated.

6.3.2.1 Raw Memory

Raw memory in the RTSJ refers to any memory in which only objects of primitive types can be stored; *Java objects or their references cannot be stored in raw memory*. Version 1.1 of specification distinguishes between three categories:

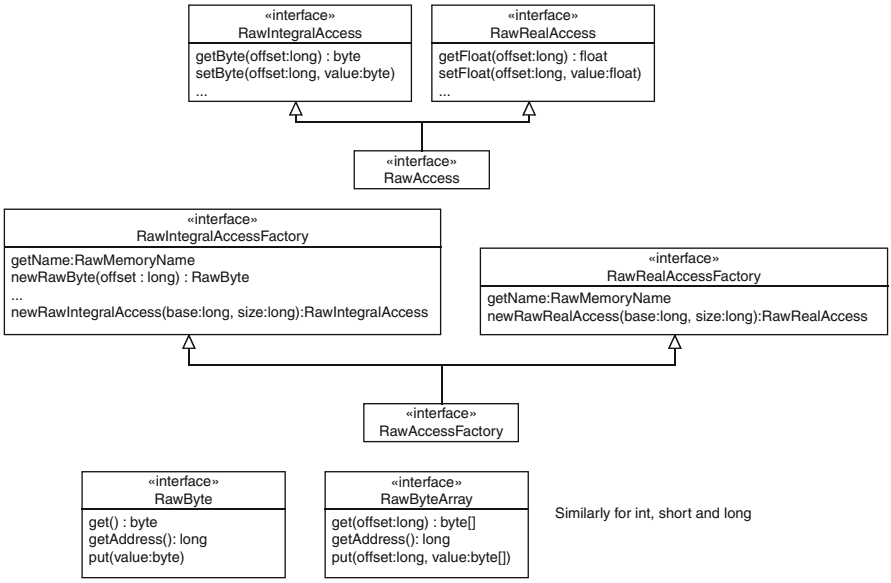


Fig. 6.8 Raw memory interfaces

- Memory that is used to access memory-mapped device registers,
- Logical memory that can be used to access port-based device registers,
- Memory that falls outside that used to accessing devices registers but may be used for other purposes, such as during DMA operations.

Each of these categories of memory is represented by a tagging interface called `RawMemoryName`.

Controlled accesses to the above categories of memory are through implementation-defined classes supporting specification-defined interfaces. A subset of these interfaces is shown in Fig. 6.8.

Java's primitive types are partitioned into two groups: integral (short, int, long, byte) and real (float, double) types, including arrays of each type. For integral types, individual interfaces are also defined to facilitate greater type security during access. Objects that support these interfaces are created by factory methods, which again have predefined interfaces. Such objects are called *accessor* objects as they encapsulates the access protocol to the raw memory.

Control over all these objects is managed by the `RawMemory` class that provides a set of static methods, as shown in Fig. 6.9. There are two groups of methods, those that

- Enable a factory to be registered, and
- Request the creation of *accessor* object for a particular memory type at a physical address in memory.

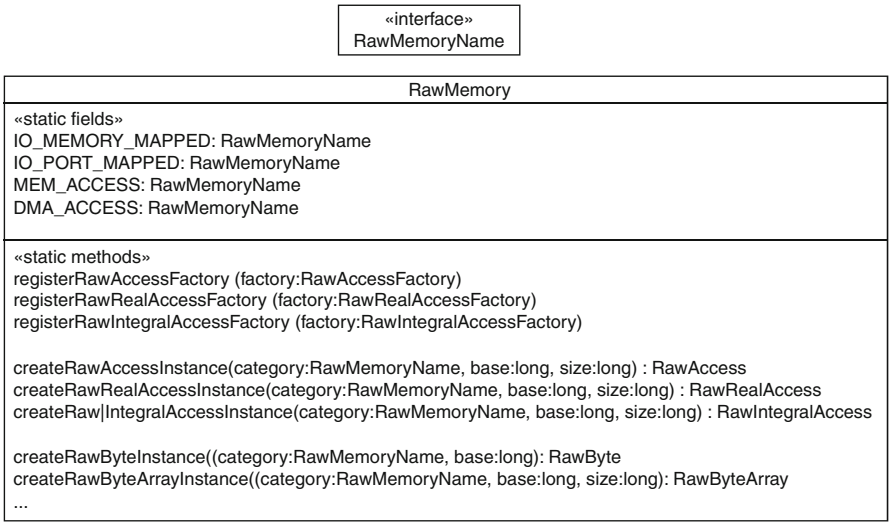


Fig. 6.9 The abridged `RawMemory` class

The latter consists of methods to create

- General accessor objects, (`createRawAccessInstance`, `createRawIntegralAccessInstance`, and `createRawRealAccessInstance`) which provide full access to the memory and will deal with all issues of memory alignment when reading data of primitive types; and
- Java-primitive-type accessor objets, which will throw exceptions if the appropriate addresses are not on correct boundaries to enable the underlying machine instructions to be used without causing hardware exceptions (e.g., `createRawByteInstance`).

As with interrupt handling, some real-time JVMs may not be able to support all of the memory categories. However, the expectation is that for all supported categories, they will also provide and register the associated factories for object creation.

For the case of `IO_PORT_MAPPED` raw memory, the accessor objects will need to arrange to execute the appropriate assemble instructions to access the device registers.

Consider, the simple case where a device has a two device registers: a control/status register that is a 32 bits integer, and a data register that is a 64 bits long. The registers have been memory mapped to locations: 0x20 and 0x24 respectively. Assuming the real-time JVM has registered a factory for the `IO_MEM_MAPPED` raw memory name, then the following code will create the objects that facilitate the memory access

```
RawInt controlReg = RawMemory.createRawIntAccessInstance(
              RawMemory.IO_MEM_MAPPED, 0x20);
RawLong dataReg = RawMemory.createRawLongAccessInstance(
            RawMemory.IO_MEM_MAPPED, 0x24);
```

The above definitions reflect the structure of the actual registers. The JVM will check that the memory locations are on the correct boundaries and that they can be accessed without any hardware exceptions being generated. If they cannot, the create methods will throw an appropriate exceptions. If successfully created, all future access to the `controlReg` and `dataReg` will be exception free. The registers can be manipulated by calling the appropriate methods, as in the following example.

```
dataReg.put(l);
    // where l is of type long and  is data to be sent to the device
controlReg.put(i);
    // where i is of type int and is the command to the device
```

In the general case, programmers themselves may create their own memory categories and provide associated factories (that may use the implementation-defined factories). These factories are written in Java and are, therefore, constrained by what the language allows them to do. Typically, they will use the JVM-supplied raw memory types to facilitate access to a device's external memory. In addition to the above facilities, the RTSJ also supports the notion of removable memory. When this memory is inserted or removed, an asynchronous event can be set up to fire, thereby alerting the application that the device has become active. Of course, any removable memory has to be treated with extreme caution by the real-time JVM. Hence, the RTSJ allows it only to be accessed as a raw memory device. An example of these latter facilities will be given in Sect. 6.4.

6.3.3 Direct Memory Access

DMA is often crucial for performance in embedded systems; however, it does cause problems both from a real-time analysis perspective and from a JVM-implementation perspective. The latter is the primary concern here.

The following points should be noted about the RTSJ philosophy in this area.

- The RTSJ does not address issues of persistent objects; so the input and output of Java objects to devices (other than by using the Java serialization mechanism) is not supported.
- The RTSJ requires that RTSJ programs can be compiled by regular Java compilers. Different bytecode compilers (and their supporting JVM) use different representation for objects. Java arrays (even of primitive types) are objects, and the data they contain may not be stored in contiguous memory.

For these reasons, without explicit knowledge of the compiler/JVM, allowing any DMA into any RTSJ physical memory area is a very dangerous action. DMA to/from raw memory is controlled by the real-time JVM and, consequently, can be performed safely.

6.4 An Illustrative Example

Consider an embedded system that has a simple flash memory device that supports a single type of removable flash memory stick, as illustrated in Fig. 6.10.

When the memory stick is inserted or removed, an interrupt is generated. This interrupt is known to the real-time JVM. The interrupt is also generated when operations requested on the device are completed. For simplicity, it is assumed that real-time JVM has mapped this interrupt to an external happening called `FlashHappening` with a default happening dispatcher.

The example illustrates how

1. A programmer can use the RTSJ facilities to write a device handler,
2. A factory class can be constructed and how the accessor objects police the access,
3. Removable memory is handled.

The flash memory device is accessed via several associated registers, which are shown in Table 6.1. These have all been memory mapped to the indicated locations.

6.4.1 Software Architecture

There are many ways in which the software architecture for the example could be constructed. Here, for simplicity of representation, an architecture is chosen with a minimal number of classes. It is illustrated in Fig. 6.11. There are three key components.

- `FlashHappening` – This is the external happening that is associated with the flash device's interrupt. The RTSJ will provide a default dispatcher, which will fire the asynchronous event when the interrupt occurs.

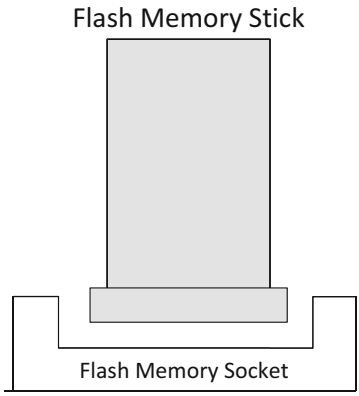


Fig. 6.10 Flash memory device

Table 6.1 Device registers

Register	Location	Bit positions	Values
Command	0x20	0	0 = Disable device, 1 = Enable device
		4	0 = Disable interrupts, 1 = Enable interrupts
		5–8	1 = Read byte, 2 = Write byte
			3 = Read short, 4 = Write short
			5 = Read int, 6 = Write int
			7 = Read long, 8 = Write long
		9	0 = DMA Read, 1 = DMA
		31–63	Offset into flash memory
Data	0x28	0–63	Simple data or memory address if DMA
Length	0x30	0–31	Length of data transfer
Status	0x38	0	1 = Device enabled
		3	1 = 4Interrupts enabled
		4	1 = Device in error
		5	1 = Transfer complete
		6	1 = Memory stick present
			0 = Memory stick absent
		7	1 = Memory stick inserted
		8	0 = Memory stick removed

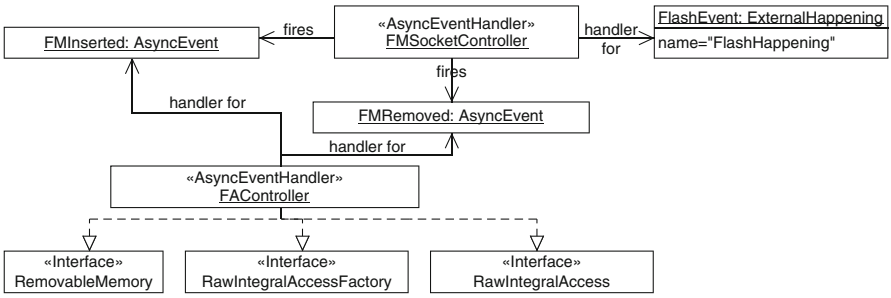


Fig. 6.11 Flash memory classes

- `FMSocketController` – This is the object that encapsulates the access to the flash memory device. In essence, it is the device driver; it is also the handler for the `FlashHappening` and is responsibly for firing the `FMInserted` and `FMRemoved` asynchronous events.
- `FAController` – This is the object that controls access to the flash memory, it
  - Acts as the factory for the creating objects that will facilitate access to the flash memory itself (using the mechanisms provided by the `FMSocket-Controller`),

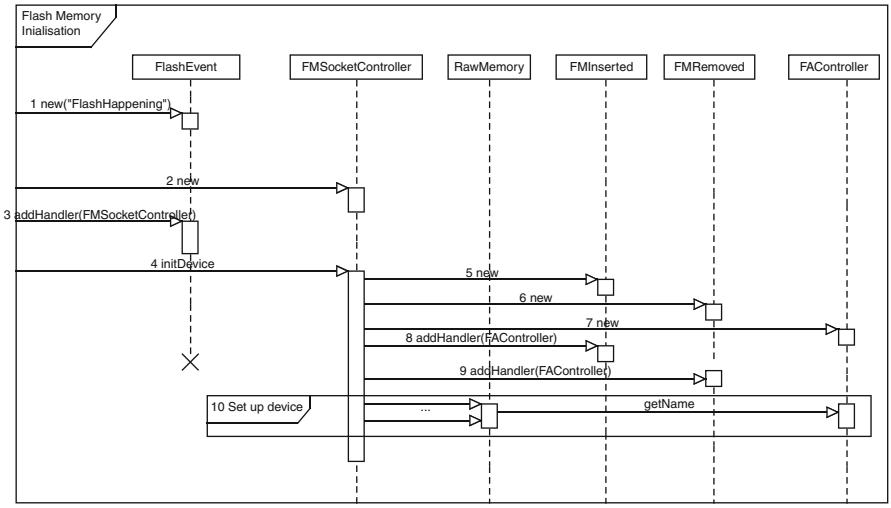


Fig. 6.12 Sequence diagram showing initialization operations

– Is the asynchronous event handler that responds to the firing of the `FMInserted` and `FMRemoved` asynchronous events, and
– Also acts as the accessor object for the memory.

6.4.2 Device Initialization

Figure 6.12 shows the sequence of operations that the program must perform to initialize the flash memory device. The main steps are as follows.

1 The external happening (`FlashEvent`) associated with the flash happening must be created.
2–3 The (`FMSocketController`) object is created and added as a handler for `FlashEvent`.
4 An initialization method is called (`initDevice`) to perform all the operations necessary to configure the infrastructure and initialize the hardware device.
5–6 Two new asynchronous events are created to represent insertion and removal of the flash memory stick.
7–9 The `FAController` class is created. It is added as the handler for the two events created in steps 5 and 6.
10 Setting up the device and registering the factory is shown in detail in Fig. 6.13. It involves: registering the `FAController` object via the static methods in the `RawMemory` class, and creating and using the JVM-supplied factory to access the memory-mapped I/O registers.

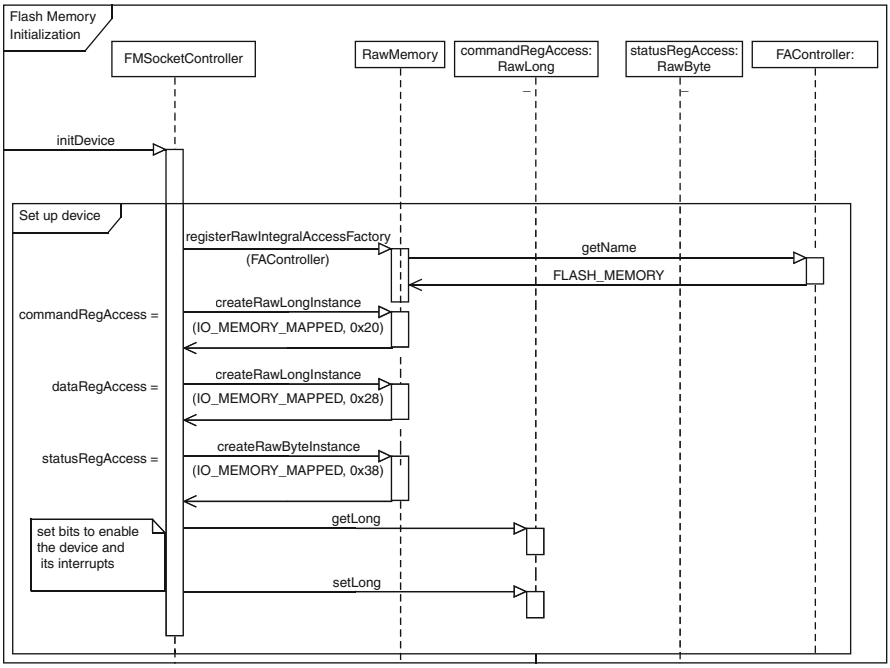


Fig. 6.13 Sequence diagrams showing operations to initialize the hardware device

6.4.3 *Responding to External Happenings*

In the example, interrupts are handled by the JVM, which turns them into an external happening. The application code that indirectly responds to the happening is provided in the `handleAsyncEvent` method in the `FMSocketController` object. Figure 6.14 illustrates the approach. In this example, the actions in response to the *memory stick inserted* and *memory stick removed* flash events is simply shown as the execution of the `FMInserted` and `FMRemoved` handlers. These will inform the application. The memory accessor classes themselves will ensure that the stick is present when performing the required application accesses.

6.4.4 *Access to the Flash Controller's Device Registers*

Figure 6.15 shows the sequence of events that the application follows. First it must register a handler with the `FMInserted` asynchronous event. Here, the application itself is an asynchronous event handler. When this is released, the memory has been inserted.

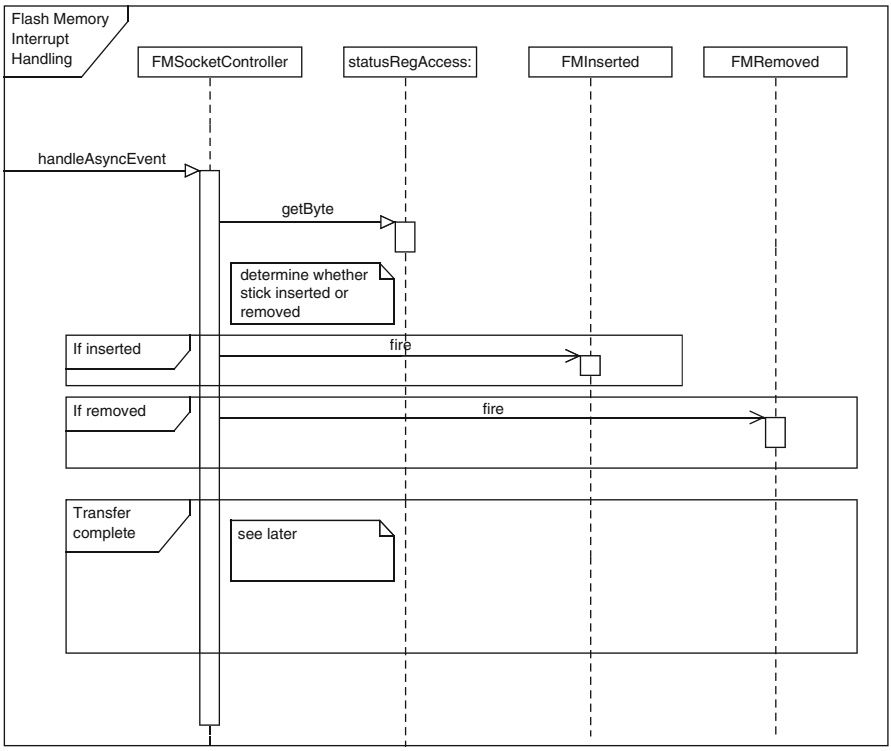


Fig. 6.14 The `FMSocketController.handleAsync` method

In this simple example, the application simply reads a byte from an offset within the memory stick. It, therefore, creates an accessor to access the data. When this has been returned (it is the `FAController` itself), the application can now call the `get` method (called *FA* `get`, in the following, for clarity). This method must implement the sequence of raw memory access on the device's registers to perform the operation. In Fig. 6.15, they are as follows.

1. *FA* `get` calls the `get` method of the status register's accessor object. This can check to make sure that the flash memory is present (bit 6, as shown in Table 6.1). If it is not, an exception can be thrown.
2. Assuming the memory is present, it then sets the control register with the offset required (bits 31–63, as shown in Table 6.1) and sets the read byte request bit (bits 5–8, as shown in Table 6.1).
3. The *FA* `get` method must then wait for indication that the requested operation has been completed by the device. This is detected by the `handleAsyncEvent` method of the `FMController`, which performs the necessary notify.

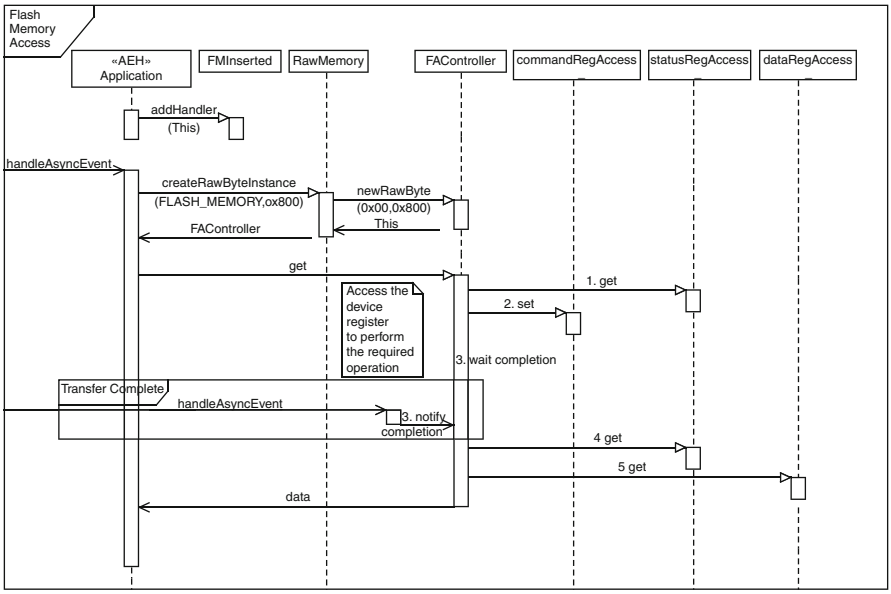


Fig. 6.15 Application usage

4. Once notified of completion, the *FA* `get` method, again reads the status register to make sure there were no errors on the device (bit 4 in Table 6.1) and that the memory is still present
5. The *FA* `get` then reads the data register to get the requested data, which it returns.

6.5 Conclusions

Interacting with the underlying platform and its environment is one of the key activities for embedded programs. With Java software, it has traditionally been the case that the program must execute JNI code to achieve this. It has always been a goal of the RTSJ to provide direct support for as many of these activities as is technically feasible. The core support has been through happenings, and physical and raw memory access.

Version 1 of the RTSJ viewed happenings as a JVM feature. JVM vendors were encouraged to add support for external events that are characteristic of the target environment: signals, hardware interrupts, asynchronous messages, alerts from the memory subsystem, communication with a DSP that shares the die etc. Unfortunately, no RTSJ implementation went beyond offering happening-mediated access to signals.

The RTSJ's Version 1 physical memory facility is little used. There are a number of possible reasons for its unpopularity. The most likely is the scarcity of the

bus-based embedded systems that were the usual place to find the kind of heterogeneous memory that physical memory is designed to manage. Another possibility is Version 1 does a poor job of explaining the use of physical memory. Heterogeneous memory may reenter the embedded systems world in the near future if NUMA multi-processor systems become popular.

The RTSJ's Version 1 raw memory subsystem has been used, but not as heavily as one would expect for a facility of such clear utility to embedded developers. Its worst problem is probably a combination of two factors:

- Raw memory support, beyond facilities for simple access to memory addresses, is left as an option for implementors of the RTSJ. It is probably impossible for an application developer to add support for another type of memory access without modifying the native code of the JVM.
- Most users of the RTSJ use the X86 architecture where all simple I/O access is through special I/O instructions. I/O access is memory mapped, but in a special address space reserved for those instructions. The RTSJ's raw memory subsystem could not reach that address space without a special optional extension by the JVM vendor.

This chapter has discussed how Version 1.1 of the RTSJ is attempting to ameliorate most of the problems identified above. The approach has been to consider the issues from embedded applications perspective. By doing this, the motivation for the RTSJ support, and the underlying models it implements, become much clearer. In terms of the core facilities, Version 1.1 will support

- Happenings that can easily be implemented by an application programmer with no special access to JVM internals,
- Raw memory access that can be easily and efficiently extended by application programmers,
- A much simpler physical memory manager that allows application programmers to specify their platform's memory characteristics.

Acknowledgements The authors acknowledge the contributions of the other members of the JSR 282 Expert Group, in particularly David Holmes, Kelvin Nilsen and Ben Brosgol to the ideas expressed in this chapter. Also, Leandro Soares Indrusiak helped with some of the detailed embedded system memory issues. Martin Schoeberl has also given us valuable feedback.

Chapter 7
Hardware Support for Embedded Java

Martin Schoeberl

Abstract The general Java runtime environment is resource hungry and unfriendly for real-time systems. To reduce the resource consumption of Java in embedded systems, direct hardware support of the language is a valuable option. Furthermore, an implementation of the Java virtual machine in hardware enables worst-case execution time analysis of Java programs. This chapter gives an overview of current approaches to hardware support for embedded and real-time Java.

7.1 Introduction

Embedded systems are usually resource constraint systems and often have to perform computations under time constraints. Although the first version of Java has been designed for an embedded system, current implementations of Java are quite resource hungry and the execution time of a Java application is hard to predict statically. One approach to reduce resource consumption for Java and enable worst-case execution time (WCET) analysis is to provide hardware support for embedded Java. In this chapter hardware implementations of the Java virtual machine (JVM), Java processors, and hardware support for Java specific idioms (e.g., garbage collection) are described.

Standard Java is not the best fit for embedded systems. Sun introduced the Java micro edition and the Connected Limited Device Configuration (CLDC) [397] for embedded systems. CLDC is a subset of Java resulting in a lower memory footprint. Based on CLDC, the real-time specification for Java (RTSJ) [65] defines a new memory model and strengthens the scheduling guarantees to enable Java for

M. Schoeberl (✉)
Department of Informatics and Mathematical Modeling, Technical University of Denmark, Copenhagen, Denmark
e-mail: masca@imm.dtu.dk

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_7,

real-time systems. Safety-critical Java (SCJ) [254] defines a subset of RTSJ for safety-critical systems. Most Java processors base the Java library on CLDC or a subset of it. Although the RTSJ would be a natural choice for embedded Java processors, none of the available processors support the RTSJ.

Embedded Java can be supported by a hardware implementation of the JVM – a Java processor, or by extending a RISC processor with Java specific support hardware. In the following section we provide an overview of current and most influential Java processors for embedded systems. Hardware support for low-level I/O in Java, object oriented caches, and garbage collection is described in Sect. 7.3. Most of the techniques for Java hardware support have been introduced in the context of Java processors. However, they can also be used to enhance a standard RISC processor, which executes compiled Java. The chapter is concluded in Sect. 7.4.

7.2 Java Processors

In the late 1990s, when Java became a popular programming language, several companies developed Java processors to speedup the execution of Java. Besides commercial processors, research on Java processor architectures was also very active at this time. With the introduction of advanced JIT compilers, the speed advantage of Java processors diminished, and many products for general purpose computing were cancelled. In the embedded domain, Java processors and coprocessors are still in use and actively developed. In this section an overview of products and research projects that survived or are still relevant are described. A description of some of the disappeared Java processors can be found in [349].

Table 7.1 lists the relevant Java processors available to date. The entries are listed in the order the processors became available. The last column presents the year where the first paper on the processor has been published. The references in the first column are to the first published paper and to the most relevant paper for the processor.

The resource usage is given either in ASIC gates for an implementation in silicon or in logic cells (LC) when implemented in a field-programmable gate array (FPGA). The memory column gives the size of on-chip memory usage for caches and microcode store. Cache sizes are usually configurable. The column lists the default configuration. When an entry is missing, there is no information available. One design, the BlueJEP, uses Xilinx LCs for distributed RAM and therefore the LC count is high, but no dedicated on-chip memory is used. The listed resource consumptions of the processors are based on latest publications (most research projects grew in size over time). However, most FPGA based projects are configurable and the resource consumption depends on the concrete configuration.

Note, that the clock frequency does not give a direct indication of the performance of the processors, it is more an indication of the pipeline organization. E.g., the implementation of picoJava in an Altera FPGA [310] clocked at 40 MHz performs better than JOP in the same FPGA at 100 MHz.

Table 7.1 Relevant Java processors for embedded Java

	Target technology	Size		Speed (MHz)	Year (pub.)
		Logic	Memory		
picoJava [283, 389]	ASIC, Altera FPGA	27500 LC	38 kB	40	1997
Komodo [240, 241]	Xilinx FPGA	2600 LC		33	1999
aJile aJ-100 [5, 185]	ASIC 0.25 µ	25K gates	48 kB	100	2000
Cjip [182, 217]	ASIC 0.35 µ	70K gates	55 kB	80	2000
jHISC [259, 406]	Xilinx FPGA	15600 LC	14 kB	30	2002
FemtoJava [52]	Xilinx FPGA	2700 LC	0.5 kB	56	2003
JOP [346, 352]	Altera, Xilinx FPGA	3000 LC	4 kB	100	2003
jamuth [416]	Altera FPGA			33	2007
BlueJEP [178]	Xilinx FPGA	6900 LC	0 kB	85	2007
SHAP [445, 446]	Altera, Xilinx FPGA	5600 LC	22 kB	80	2007
aJile aJ-102/200 [6]	ASIC 0.18 µ		80 kB	180	2009

7.2.1 picoJava

Sun introduced the first version of picoJava [283] in 1997. The processor was targeted at the embedded systems market as a pure Java processor with restricted support of C. picoJava-I contains four pipeline stages. A redesign followed in 1999, known as picoJava-II that is now freely available with a rich set of documentation [389, 390].

Sun's picoJava is the Java processor most often cited in research papers. It is used as a reference for new Java processors and as the basis for research into improving various aspects of a Java processor. Ironically, this processor was never released as a product by Sun.

The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions [283] and is the most complex Java processor available. The processor can be implemented in about 440K gates [127]. An implementation of picoJava in an Altera FPGA was performed by Puffitsch and Schoeberl [310]. As seen in Table 7.1, picoJava is the biggest Java processor implemented in an FPGA. Nevertheless, it provides a baseline for comparison with other research processors.

Simple Java bytecodes are directly implemented in hardware, most of them execute in one to three cycles. Other performance critical instructions, for instance invoking a method, are implemented in microcode. picoJava traps on the remaining complex instructions, such as creation of an object, and emulates this instruction. To access memory, internal registers and for cache management, picoJava implements 115 extended instructions with 2-byte opcodes. These instructions are necessary to write system-level code to support the JVM.

Traps are generated on interrupts, exceptions, and for instruction emulation. A trap is rather expensive and has a minimum overhead of 16 clock cycles. This minimum value can only be achieved if the trap table entry is in the data cache and the first instruction of the trap routine is in the instruction cache. The

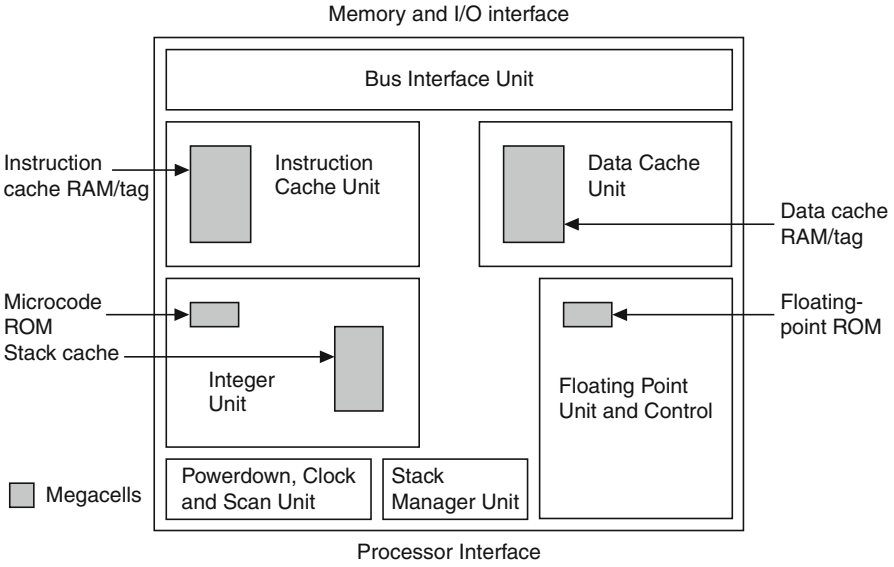


Fig. 7.1 Block diagram of picoJava-II (from [389])

worst-case interrupt latency is 926 clock cycles [390]. This great variation in execution times for a trap hampers tight WCET estimates.

Figure 7.1 shows the major function units of picoJava. The integer unit decodes and executes picoJava instructions. The instruction cache is direct-mapped, while the data cache is two-way set-associative, both with a line size of 16 bytes. The caches can be configured between 0 and 16 kB. An instruction buffer decouples the instruction cache from the decode unit. The floating-point unit (FPU) is organized as a microcode engine with a 32-bit datapath, supporting single- and double-precision operations. Most single-precision operations require four cycles. Double-precision operations require four times the number of cycles as single-precision operations. For low-cost designs, the FPU can be removed and the core traps on floating-point instructions to a software routine to emulate these instructions. picoJava provides a 64-entry stack cache as a register file. The core manages this register file as a circular buffer, with a pointer to the top of stack. The stack management unit automatically performs spill to and fill from the data cache to avoid overflow and underflow of the stack buffer. To provide this functionality the register file contains five memory ports. Computation needs two read ports and one write port, the concurrent spill and fill operations need additional read and write ports. The processor core consists of following six pipeline stages:

Fetch: Fetch 8 bytes from the instruction cache or 4 bytes from the bus interface to the 16-byte-deep prefetch buffer.

Decode: Group and precode instructions (up to 7 bytes) from the prefetch buffer. Instruction folding is performed on up to four bytecodes.

Register: Read up to two operands from the register file (stack cache).

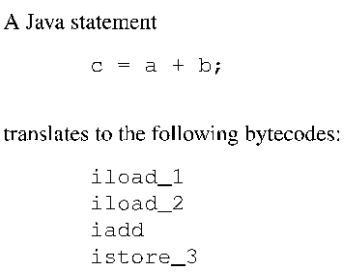


Fig. 7.2 A common folding pattern that is executed in a single cycle with instruction folding

Execute: Execute simple instructions in one cycle or microcode for multi-cycle instructions.
Cache: Access the data cache.
Writeback: Write the result back into the register file.

The integer unit together with the stack unit provides a mechanism, called instruction folding, to speed up common code patterns found in stack architectures, as shown in Fig. 7.2. Instruction folding is a technique to merge several stack oriented bytecode instructions into fewer RISC type instructions dynamically. When all entries are contained in the stack cache, the picoJava core can fold these four instructions to one RISC-style single cycle operation.

Instruction folding was implemented in picoJava and proposed in several research papers. However, none of the current Java processors implements instruction folding. The theoretical papers on instruciton folding ignored the complexity of the folding pattern detection and the influence on the maximum clock frequency (besides the larger chip space). Gruian and Westmijze evaluated the instruction folding by implementing the proposed algorithms in an FPGA [179]. Their result shows that the reduced cycle count due to folding is more than offset by the decreased clock frequency.

picoJava contains a simple mechanism to speed-up the common case for monitor enter and exit. The two low order bits of an object reference are used to indicate the lock holding or a request to a lock held by another thread. These bits are examined by `monitorenter` and `monitorexit`. For all other operations on the reference, these 2 bits are masked out by the hardware. Hardware registers cache up to two locks held by a single thread.

To efficiently implement a generational or an incremental garbage collector picoJava offers hardware support for write barriers through memory segments. The hardware checks all stores of an object reference if this reference points to a different segment (compared to the store address). In this case, a trap is generated and the garbage collector can take the appropriate action. Additional two reserved bits in the object reference can be used for a write barrier trap to support incremental collection. The hardware support for write barriers can also be used for assignment checks of RTSJ based memory areas [197]. A combination of GC support and RTSJ assignment checks with the picoJava hardware support is presented in [199].

The distribution of picoJava does not contain a complete JVM and no Java libraries. It is expected that picoJava is just the base platform for different variants of an embedded JVM.

7.2.2 aJile's JEMCore

aJile's JEMCore is a Java processor that is available as both an IP core and a stand alone processor [5, 185]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. JEM2 is an enhanced version of JEM1, created in 1997 by the Rockwell-Collins Advanced Architecture Microprocessor group. Rockwell-Collins originally developed JEM for avionics applications by adapting an existing design for a stack-based embedded processor. Rockwell-Collins decided not to sell the chip on the open market. Instead, it licensed the design exclusively to aJile Systems Inc., which was founded in 1999 by engineers from Rockwell-Collins, Centaur Technologies, Sun Microsystems, and IDT.

The core contains twenty-four 32-bit wide registers. Six of them are used to cache the top elements of the stack. The datapath consists of a 32-bit ALU, a 32-bit barrel shifter, and support for floating point operations (disassembly/assembly, overflow and NaN detection). The control store is a 4K by 56 ROM to hold the microcode that implements the Java bytecode. An additional RAM control store can be used for custom instructions. This feature is used to implement the basic synchronization and thread scheduling routines in microcode. This results in low execution overheads with thread-to-thread yield of less than 1 μs (at 100 MHz). An optional Multiple JVM Manager (MJM) supports two independent, memory protected JVMs. The two JVMs execute time-sliced on the processor. According to aJile, the processor can be implemented in 25K gates (without the microcode ROM). The MJM needs additional 10K gates.

The first two silicon versions of JEM where the aJ-80 and the aJ-100. Both versions comprise a JEM2 core, the MJM, 48 kB zero wait state RAM and peripheral components, such as timer and UART. 16 kB of the RAM is used for the writable control store. The remaining 32 kB is used for storage of the processor stack. The aJ-100 provides a generic 8-bit, 16-bit or 32-bit external bus interface, while the aJ-80 only provides an 8-bit interface. The aJ-100 can be clocked up to 100 MHz and the aJ-80 up to 66 MHz. The power consumption is about 1 mW per MHz.

The third generation of aJile's Java processor (JEMCore-II) is enhanced with a fixed-point MAC unit and a 32 kB, 2-way set-associative, unified instruction and data cache. The latest versions of the aJile processor (aJ-102 [6] and aj-200 [7]), based on the JEMCore-II, are system-on-chips, including advanced I/O devices, such as an Ethernet controller, a LCD panel interface, an USB controller, and a hardware accelerator for encryption/decryption. Both processors are implemented in 0.18 μ and can be clocked up to 180 MHz. The aJ-102 is intended as a network processor, whereas the aJ-200, with hardware support for image capturing and a media codec, targets the real-time multimedia market.

The aJile processor is intended for real-time systems with an on-chip real-time thread manager. The RTOS and all device drivers are written entirely in Java. Furthermore, aJile Systems was part of the original expert group for the RTSJ. However, the aJile runtime system does not support the RTSJ, but implements their own version of real-time threads. The aJile processor could be a reasonable platform for WCET analysis, but no information about the bytecode execution times is disclosed.

7.2.3 Komodo and Jamuth

Komodo [240] is a multithreaded Java processor with a four-stage pipeline. It is intended as a basis for research on real-time scheduling on a multithreaded microcontroller. The unique feature of Komodo is the instruction fetch unit with four independent program counters and status flags for four threads. A priority manager is responsible for hardware real-time scheduling and can select a new thread after each bytecode instruction.

Komodo's multi-threading is similar to hyper-threading in modern processors that are trying to hide latencies due to cache misses and branch misspredictions. However, this feature leads to very pessimistic WCET values if all threads are considered. For a real-time setting one thread can be given top priority in the hardware scheduler. The other threads can use the stall cycles (e.g., due to a memory access) of the real-time thread. Therefore, a single real-time thread can provide timing guarantees and the other hardware threads can be used for soft real-time tasks or for interrupt service threads. Multiple real-time threads are supported by a software based real-time scheduler.

The Java processor jamuth is the follow-up project to Komodo [416], and is targeted for commercial embedded applications with Altera FPGAs. jamuth is well integrated in the Alter's System-on-a-Programmable-Chip builder. The memory and peripheral devices are connected via the Avalon bus. The standard configuration of jamuth uses a scratchpad memory for trap routines and the garbage collector. An additional instruction cache is shared between all hardware threads.

7.2.4 Java Optimized Processor (JOP)

JOP [352] is an implementation of the JVM in hardware, especially designed for real-time systems. To support hard timing constraints, the main focus of the development of JOP has been on time-predictable bytecode execution. All function units, and especially the interactions between them, are carefully designed to avoid any time dependencies between bytecodes. This feature simplifies the low-level part of WCET analysis, a mandatory analysis for hard real-time systems.

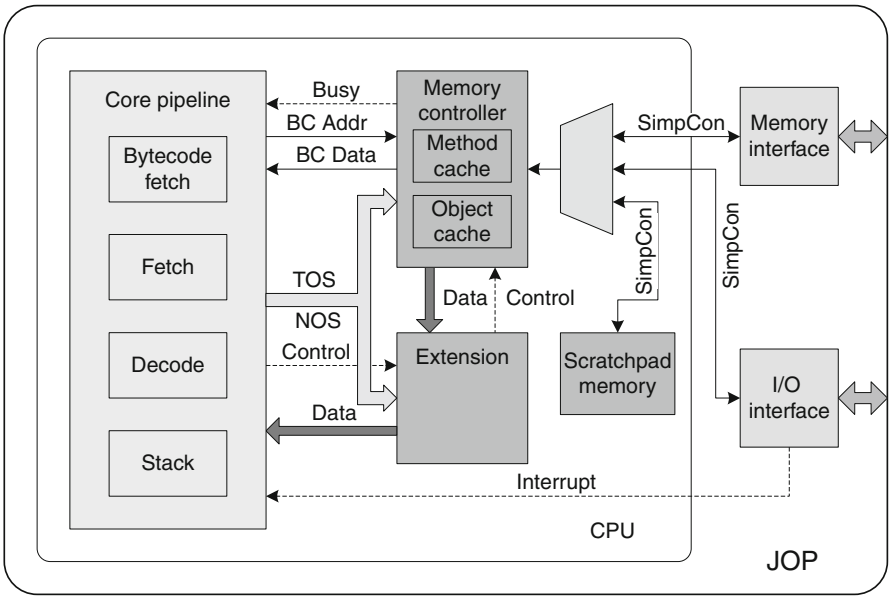


Fig. 7.3 Block diagram of JOP (from [353])

Figure 7.3 shows the block diagram of JOP. The main components are: the 4-stage pipeline, the memory controller with the method cache [348] and object cache [355], the extension module for hardware accelerators, and a scratchpad memory. Main memory and I/O devices are connected via the SimpCon interface to the CPU core.

JOP dynamically translates the CISC Java bytecodes to a RISC, stack based instruction set (the microcode) that can be executed in a 3-stage pipeline. The translation takes exactly one cycle per bytecode and is therefore pipelined (adding a fourth pipeline stage). Compared to other forms of dynamic code translation, the translation in JOP does not add any variable latency to the execution time and is therefore time-predictable. Interrupts are inserted in the translation stage as special instructions and are transparent to the microcode pipeline. All microcode instructions have a constant execution time of one cycle. No stalls are possible in the microcode pipeline. Loads and stores of object fields are handled explicitly. The absence of time dependencies between bytecodes results in a simple processor model for the low-level WCET analysis [361].

JOP contains a simple execution stage with the two topmost stack elements as discrete registers. No write back stage or data forwarding logic is needed. The short pipeline (4 stages) results in short conditional branch delays and therefore helps to avoid any hard-to-analyze branch prediction logic or branch target buffer.

JOP introduced a special instruction cache, the method cache [348], which caches whole methods. With a method cache, only invoke and return bytecodes can result in

a cache miss. All other bytecodes are guaranteed cache hits. The idea to cache whole methods is based on the assumption that WCET analysis at the call graph level is more practical than performing cache analysis for each bytecode. Furthermore, loading whole methods also leads to better average case execution times for a memory with long latency but high bandwidth. The load time depends on the size of the method. However, on JOP, the cache loading is done in parallel with microcode execution in the core pipeline. Therefore, small methods do not add any additional latency to the invoke or return bytecodes. The method cache is also integrated in the embedded Java processor SHAP [306] and considered for jamuth [416] as a time-predictable caching solution[1].

JOP has it's own notion of real-time threads that are scheduled with a preemptive, priority based scheduler. In the multi-processor configuration of JOP [296], the scheduling is partitioned. A thread is pinned to a core and each core has it's own scheduler based on a core local timer.

JOP is available in open-source under the GNU GPL. The open-source approach with JOP enabled several research projects to build upon JOP: a hardware GC implementation [176]; the BlueJEP processor architecture [178] and the first version of SHAP [307][2] are based on JOP; and JOP has been combined with the LEON processor for a real-time Agent computing platform targeting distributed satellite systems [80]. JOP is also in use in several industrial applications [273, 351]. Furthermore, the WCET friendly architecture of JOP enabled development of several WCET analysis tools for Java [62, 186, 209]. The WCET analysis tool WCA is part of the JOP source distribution [361].

7.2.5 BlueJEP

BlueJEP is a Java processor specified in Bluespec System Verilog (BSV) to evaluate BSV for system design [178]. BlueJEP's design starting point was the architecture of JOP [352]. With respect to the Java build process and microcode, BlueJEP is JOP compatible. However, the pipeline has a different structure. Six stages are connected via searchable FIFOs and are as follows: fetch bytecode, fetch microcode, decode and fetch register, fetch stack, execute, and write back. The last stages contain a forwarding network. Quite unconventional is the bypass option where the execution stage can be skipped by individual instructions. BlueJEP can be configured to use a memory management unit that performs mark-compact based garbage collection in hardware [177].

[1] Personal communication with Sascha Uhrig.

[2] The similarity can be found when comparing the microcode instructions of JOP and SHAP. The microcode instructions of SHAP are described in the appendix of the technical report [307].

7.2.6 Java Accelerators

Another approach to speedup Java programs in an embedded system is to enhance a RISC core with a Java accelerator. The main idea is to support (legacy) C code *and* Java code in a single chip. The accelerator can work as translation unit or as a Java coprocessor. The translation unit substitutes the switch statement of an interpreting JVM (bytecode decoding) through hardware and/or translates simple bytecodes to a sequence of RISC instructions on the fly. A coprocessor is placed in the instruction fetch path of the main processor and translates Java bytecodes to sequences of instructions for the host CPU or directly executes basic Java bytecodes. The complex instructions are emulated by the main processor.

Nozomi's JA108 [277], previously known as JSTAR, Java accelerator sits between the native processor and the memory subsystem. JA108 fetches Java bytecodes from memory and translates them into native microprocessor instructions. JA108 acts as a pass-through when the core processor's native instructions are being executed. The JA108 is targeted for use in mobile phones to increase performance of Java multimedia applications. The coprocessor is available as standalone package or with included memory and can be operated up to 104 MHz. The resource usage for the JSTAR is known to be about 30K gates plus 45 kB for the microcode.

Jazelle [17] is an extension of the ARM 32-bit RISC processor, similar to the Thumb state (a 16-bit mode for reduced memory consumption). A new ARM instruction puts the processor into Java state. Bytecodes are fetched and decoded in two stages, compared to a single stage in ARM state. Four registers of the ARM core are used to cache the top stack elements. Stack spill and fill is handled automatically by the hardware. Additional registers are reused for the Java stack pointer, the variable pointer, the constant pool pointer, and locale variable 0 (the `this` pointer in methods). Keeping the complete state of the Java mode in ARM registers simplifies its integration into existing operating systems. The Jazelle coprocessor is integrated into the same chip as the ARM processor. The hardware bytecode decoder logic is implemented in less than 12K gates. It accelerates, according to ARM, some 95% of the executed bytecodes. Hundred and forty bytecodes are executed directly in hardware, while the remaining 94 are emulated by sequences of ARM instructions. This solution also uses code modification with `quick` instructions to substitute certain object-related instructions after link resolution. All Java bytecodes, including the emulated sequences, are re-startable to enable a fast interrupt response time.

The Cjip processor [182, 217] supports multiple instruction sets, allowing Java, C, C++ and assembler to coexist. Internally, the Cjip uses 72 bit wide microcode instructions, to support the different instruction sets. At its core, Cjip is a 16-bit CISC architecture with on-chip 36 kB ROM and 18 kB RAM for fixed and loadable microcode. Another 1 kB RAM is used for eight independent register banks, string buffer and two stack caches. Cjip is implemented in 0.35-micron technology and can be clocked up to 80 MHz. The JVM of Cjip is implemented largely in microcode (about 88% of the Java bytecodes). Java thread scheduling and garbage collection

are implemented as processes in microcode. Microcode instructions execute in two or three cycles. A JVM bytecode requires several microcode instructions. The Cjip Java instruction set and the extensions are described in detail in [216]. For example: a bytecode `nop` executes in 6 cycles while an `iadd` takes 12 cycles. Conditional bytecode branches are executed in 33–36 cycles. Object oriented instructions such `getfield`, `putfield` or `invokevirtual` are not part of the instruction set.

7.2.7 Further Java Processor Projects

Several past and current research projects address execution of Java applications in hardware. This section briefly introduces those projects.

FemtoJava [52] is a research project to build an application specific Java processor. The bytecode usage of the embedded application is analyzed and a customized version of FemtoJava is generated in order to minimize the resource usage.

The jHISC project proposes a high-level instruction set architecture for Java [406]. The processor consumes 15600 LCs and the maximum frequency in a Xilinx Virtex FPGA is 30 MHz. The prototype can only run simple programs and the performance is estimated with a simulation.

The SHAP Java processor [445] contains a memory management unit for hardware assisted garbage collection. An additional RISC processor, the open-source processor ZPU, is integrated with SHAP and performs the GC work [305].

The research project JPOR [98] aims for a Java processor design that is optimized for the execution of RTSJ programs. It is a typical implementation of a Java processor combining direct support for simple bytecodes and microcode instructions for more complex bytecodes.

A Java processor (probably JOP) has been extended to support dual issue of bytecodes [237]. To simplify the runtime on the Java processor, the system is extended with a RISC processor [388]. The interface between the RISC processor and the Java processor is based on cross-core interrupts. A few micro benchmarks provide a comparison of the Java processor with an interpreting JVM (Sun's CVM).

7.2.8 Chip-Multiprocessors

Several of the active research projects on Java processors explore chip-multiprocessor (CMP) configurations. The main focus of the CMP version of JOP [295, 296] is to keep the CMP time-predictable, even with access to shared main memory. Pitter implemented a TDMA based memory arbiter and integrated the timing model of the arbiter into the WCET analysis tool for JOP. Therefore, the JOP CMP is the first time-predictable CMP that includes a WCET analysis tool.

The multi-threaded jamuth processor uses the Avalon switch fabric to build a CMP system of a small number of cores [415]. As the jamuth core is already multi-threaded, the effective supported concurrent threads if four times the number of cores. The evaluation shows that increasing the number of cores provides a better performance gain than increasing the number of threads in one core. For the jamuth system, the optimum number of threads in one core is two. The second thread can utilize the pipeline when the first thread stalls in a memory access. Adding a third thread results only in a minor performance gain.

The CMP version of SHAP contains a pipelined memory arbiter and controller to optimize average case performance [446]. By using a pipelined, synchronous SRAM as main memory, the memory delivers one word per clock cycle. On a CMP system the bottleneck is the memory bandwidth and not the memory latency. Therefore, this pipelined memory delivers the needed memory bandwidth for the SHAP CMP system.

With true concurrence on a CMP the pressure on efficient synchronization primitives increases. A promising approach to simplify synchronization at the language level and provide more true execution concurrency is transactional memory (TM) [194]. A real-time TM (RTTM) has been implemented in the CMP version of JOP [357]. The Java annotation `@atomic` on a method is used to mark the transaction. The code within the transaction is executed concurrent to other possible transactions. All writes to the memory are kept in a core local transaction buffer during the transaction. At the end of the transaction that buffer is written atomically to main memory. When a conflict between transactions occurs, a transaction is aborted and the atomic section is retried. For real-time systems this retry count must be bounded [356]. To enable this bound, RTTM is designed to avoid any false conflict detections that can occur on a cache-based hardware TM.

7.2.9 Discussion

The basic architecture of all Java processors is quite similar: simple bytecodes are supported in hardware and more complex bytecodes are implemented in microcode or even in Java. JVM bytecode is stack oriented and almost all instructions operate on the stack. Therefore, all Java processor provide a dedicated cache for stack content. The main difference between the architectures is the amount of bytecodes that are implemented in hardware and the caching system for instructions and heap allocated data. As the market for embedded Java processors is still small, only the JEMCore is available in silicon. picoJava is discontinued by Sun/Oracle and the other processor projects target FPGAs.

Standard Java is too big for embedded systems. The subset defined in the CLDC is a starting point for the Java library support of embedded Java. JEMCore, jamuth, JOP, and SHAP support the CLDC. As the CLDC is missing an API for low-level I/O, all vendors provide their own Java classes or mechanism for low-level I/O (see

next section). CLDC is based on Java 1.1 and is missing important features, such as collection classes. Therefore, most projects add some classes from standard Java to the CLDC base. The results in libraries somewhere between CLDC and standard Java, without being compatible. A new library specification for (classic) embedded systems, based on the current version of Java, would improve this situation.

Most processors (JEMCore, jamuth, and JOP) target real-time systems. They provide a real-time scheduler with tighter guarantees than a standard JVM thread scheduler. As this real-time scheduling does not fit well with standard Java threads, special classes for (periodic) real-time threads are introduced. None of the runtime systems supports the RTSJ API. We assume that the RTSJ is too complex for such resource constraint devices. It might also be the case that the effort to implement a RTSJ JVM is too high for the small development teams behind the processor projects. However, with the simpler SCJ specification there is hope that the embedded Java processors will adapt their runtime to a common API.

For real-time systems the WCET needs to be known as input for schedulability analysis. WCET is usually derived by static program analysis. As the presented processor architectures are relative simple (e.g., no dynamic scheduling in the pipeline), they should be an easy target for WCET analysis. However, only for three processors the execution time of bytecodes is documented (picoJava, Cjip, and JOP). JOP is the only processor that is supported by WCET analysis tools.

7.3 Support Technology for Java

Several techniques to support the execution of Java programs have been proposed and implemented. Most techniques have been introduced in the context of a Java processor. However, the basic concepts can also be applied to a RISC processor to enhance execution performance or time predictability of Java applications. In this section we review support for low-level IO and interrupts, special cache organizations for heap allocated data, and support for garbage collection.

7.3.1 Low-Level I/O and Interrupts

Java, as a platform independent language and runtime system, does not support direct access to low-level I/O devices. However, embedded Java systems often consist only of a JVM without an underlying operations system. In that case, the JVM acts as the operating system. The different solutions to interface I/O devices and implement interrupt handlers directly in Java on Java processors and embedded JVMs are described in the following section.

The RTSJ defines an API for direct access to physical memory, including hardware registers. Essentially one uses `RawMemoryAccess` at the level of primitive

data types. Although the solution is efficient, this representation of physical memory is not object oriented. A type-safe layer with support for representing individual registers can be implemented on top of the RTSJ API. The topic of RTSJ support for low-level I/O is discussed in detail in Chap. 6.

The RTSJ specification suggests that asynchronous events are used for interrupt handling. Yet, it neither specifies an API for interrupt control nor semantics of the handlers. Second level interrupt handling can be implemented within the RTSJ with an `AsyncEvent` that is bound to a *happening*. The happening is a string constant that represents an interrupt, but the meaning is implementation dependent.

The aJile Java processor [5] uses native functions to access devices. Interrupts are handled by registering a handler for an interrupt source (e.g., a GPIO pin). Systronix suggests[3] to keep the handler short, as it runs with interrupts disabled, and delegate the real handling to a thread. The thread waits on an object with ceiling priority set to the interrupt priority. The handler just notifies the waiting thread through this monitor. When the thread is unblocked and holds the monitor, effectively all interrupts are disabled.

On top of the multiprocessing pipeline of Komodo [240] the concept of interrupt service threads is implemented. For each interrupt one thread slot is reserved for the interrupt service thread. It is unblocked by the signaling unit when an interrupt occurs. A dedicated thread slot on a fine-grain multithreading processor results in a very short latency for the interrupt service routine. No thread state needs to be saved. However, this comes at the cost to store the complete state for the interrupt service thread in the hardware. In the case of Komodo, the state consists of an instruction window and the on-chip stack memory. Devices are represented by Komodo specific I/O classes.

One option to access I/O registers directly in an embedded JVM is to access them via C functions using the Java native interface. Another option is to use so called hardware objects [358], which represent I/O devices as plain Java objects. The hardware objects are platform specific (as I/O devices are), but the mechanism to represent I/O devices as Java objects can be implemented in any JVM. Hardware objects have been implemented so far in six different JVMs: CACAO, OVM, SimpleRTJ, Kaffe, HVM, and JOP. Three of them are running on a standard PC, two on a microcontroller, and one is a Java processor.

In summary, access to device registers is handled in both aJile and Komodo by abstracting them into library classes with access methods. This leaves the implementation to the particular JVM and does not give the option of programming them at the Java level. Exposing hardware devices as Java objects, as implemented with the hardware objects, allows safe access to device registers directly from Java. Interrupt handling in aJile is essentially first level, but with the twist that it may be interpreted as RTSJ event handling, although the firing mechanism is atypical. Komodo has a solution with first level handling through a full context shift.

[3] A template can be found at http://practicalembeddedjava.com/tutorials/aJileISR.html.

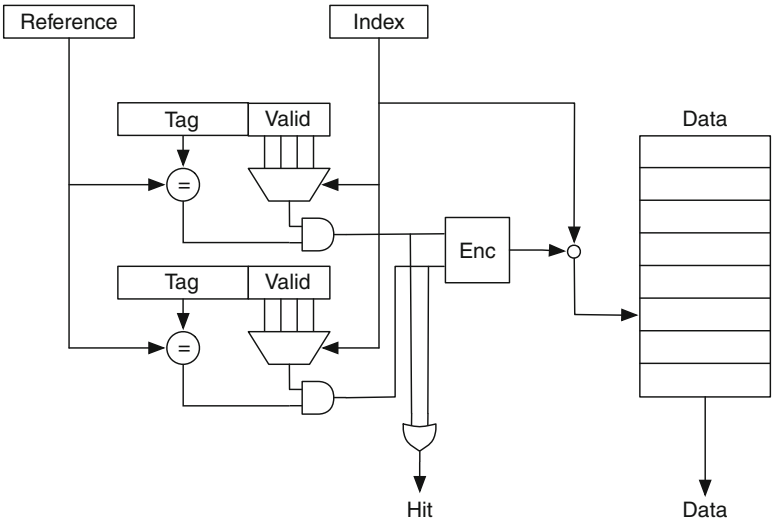


Fig. 7.4 Object cache with associativity of two and four fields per object (from [348])

7.3.2 Cache Organizations for Java

Java releys heavily on objects allocated on the heap and the automatic memory management with a garbage collector. The resulting data usage patterns are different from e.g., C programs. Therefore, a system optimized for the execution of Java programs can benefit from a specialized cache, an object cache.

Figure 7.4 shows one possible organization of an object cache. The cache is accessed via the object reference and the field index. In this example figure the associativity is two and each cache line is four fields long. All tag memories are compared in parallel with the object reference. Parallel to the tag comparison, the valid bits for the individual fields are checked. The field index performs the selection of the valid bit multiplexer. The output of the tag comparisons and valid bit selection is fed into the encoder, which delivers the selected cache line. The line index and the field index are concatenated and build the address of the data cache.

One of the first proposals of an object cache [436] appeared within the Mushroom project [437]. The Mushroom project investigated hardware support for Smalltalk-like object oriented systems. The cache is indexed by a combination of the object identifier (the handle in the Java world) and the field offset. Different combinations, including xoring of the two fields, are explored to optimize the hit rate. The most effective generation of the hash function for the cache index was the xor of the upper offset bits (the lower bits are used to select the word in the cache line) with the lower object identifier bits. Considering only the hit rate, caches with a block size of 32

and 64 bytes perform best. However, under the assumption of realistic miss penalties caches with 16 and 32 bytes lines size result in lower average access times per field access.

With an indirection based access to an object two data structures need to be cached: the actual object and the indirection to that object. In [436], a common cache for both data structures and a split cache are investigated. As the handle indirection cache is only accessed when the object cache results in a miss, a medium hit rate on the handle indirection cache is sufficient. Therefore, the best configuration is a large object cache and a small handle indirection cache.

Object oriented architecture support for a Java processor is proposed in [420], including an object cache, extended folding, and a virtual dispatch cache. The object cache is indexed by (part of) the object reference and the field offset. The virtual method cache is motivated by avoiding a virtual dispatch table (a very common implementation approach in OO languages) to save memory for embedded devices. This cache assumes monomorphic call sites. The cache is indexed by some lower bits of the PC at the call site. As polymorphic call sites trash the cache, an extension as a hybrid polymorphic cache is proposed. The whole proposal is based on a very high level simulation – running some Java applications in a modified, interpreting JVM (Sun JDK 1.0.2). No estimates on the hardware complexity are given.

A dedicated cache for heap allocated data is proposed in [419]. The object layout is handle based. The object reference with the field index is used to address the cache – it is called virtual address object cache. Cache configurations are evaluated with a simulation in a Java interpreter and the assumption of 10 ns cycle time of the Java processor and a memory latency of 70 ns. For different cache configurations (up to 32 kB) average case field access times between 1.5 and 5 cycles are reported. For most benchmarks, the optimal block size was found to be 64 bytes, which is quite high for the medium latency (7 cycles) of the memory system. The proposed object cache is also used to cache arrays. Therefore, the array accesses favor a larger block size to benefit from spatial locality.

Wright et al. propose a cache that can be used as object cache and as conventional data cache [440]. To support the object cache mode the instruction set is extended with a few object-oriented instructions such as load and store of object fields. The object layout is handle based and the cache line is addressed with a combination of the object reference (called object id) and part of the offset within the object. The main motivation of the object cache mode is in-cache garbage collection [439]. The youngest generation of objects in a generational GC is allocated in the object cache and can be collected without main memory access.

The possible distinction between different data areas of the JVM (e.g., constant pool, heap, method area) enables unique cache configurations for a Java processor. It is argued that a split-cache design can simplify WCET analysis of data cachesn [354]. For heap allocated objects a time-predictable object cache has been implemented in JOP [355]. The object cache is organized to cache single objects in a cache line. The cache is highly associative to track object field accesses in the

WCET analysis via the symbolic references instead of the actual addresses. WCET analysis based evaluation of the object cache shows that even a small cache with a high associativity provides good hit rate analyzability [210].

7.3.3 Garbage Collection

A real-time garbage collector has to fulfill two basic properties: ensure that programs with bounded allocation rates do not run out of memory and provide short blocking times. Even for incremental garbage collectors, heap compaction can be source of considerable blocking time. Chapter 4 discusses real-time garbage collection in detailed. Here the focus is on hardware support for garbage collection.

Nielsen and Schmidt [281] propose hardware support, the object-space manager (OSM), for real-time garbage collector on a standard RISC processor. The concurrent garbage collector is based on [24], but the concurrency is of finer grain than the original Baker algorithm as it allows the mutator to continue during the object copy. The OSM redirects field access to the correct location for an object that is currently being copied. [345] extends the OSM to a GC memory module where a local microprocessor performs the GC work. In the paper the performance of standard C++ dynamic memory management is compared against garbage collected C++. The authors conclude that C++ with the hardware supported garbage collection performs comparable with traditional C++.

One argument against hardware support for GC might be that standard processors will never include GC specific instructions. However, Azul Systems has included a read barrier in their RISC based chip-multiprocessor system [107]. The read barrier looks like a standard load instruction, but tests the TLB if a page is a GC-protected page. GC-protected pages contain objects that are already moved. The read barrier instruction is executed after a reference load. If the reference points into a GC-protected page a user-mode trap handler corrects the stale reference to the forwarded reference.

Meyer proposes in [269, 270] a new RISC processor architecture for exact pointers. The processor contains a set of pointer registers and a set of data registers. The instruction set guarantees correct handling of pointers. Furthermore, the object layout and the stack are both split to pointer containing and data containing regions (similar to the split stack). A microprogrammed GC unit is attached to the main processor [271]. Close interaction between the RISC pipeline and the GC coprocessor allow the redirection for field access in the correct semi-space with a concurrent object copy. The hardware cost of this feature is given as an additional word for the back-link in every pointer register and every attribute cache line. It is not explicitly described in the paper when the GC coprocessor performs the object copy. We assume that the memory copy is performed in parallel with the execution of the RISC pipeline. In that case, the GC unit *steals* memory bandwidth from the application thread. The GC hardware uses an implementation of Baker's read-barrier

[24] for the incremental copying algorithm. The cost of the read-barrier is between 5 and 50 clock cycles. The resulting minimum mutator utilization for a time quantum of 1 ms was measured to be 55%. For a real-time task with a period of 1 kHz the resulting overhead is about a factor of 2.

The Java processor SHAP [445] contains a memory management unit with a hardware garbage collector. That unit redirects field and array access during a copy operation of the GC unit.

During heap compaction, objects are copied. Copying is usually performed atomically to avoid interference with application threads, which could render the state of an object inconsistent. Copying of large objects and especially large arrays introduces long blocking times that are unacceptable for real-time systems. In [360] an interruptible copy unit is presented that implements non-blocking object copy. The unit can be interrupted after a single word move. The evaluation showed that it is possible to run high priority hard real-time tasks at 10 kHz parallel to the garbage collection task on a 100 MHz Java processor.

7.4 Conclusions

To enable Java in resource constraint embedded systems, several projects implement the Java virtual machine (JVM) in hardware. These Java processors are faster than an interpreting JVM, but use fewer resources than a JIT compiler. Furthermore, the direct implementation of the JVM enables WCET analysis at bytecode level.

Java also triggered research on hardware support for object-oriented languages. Special forms of caches, sometimes called object cache, are optimized for the data access pattern of the JVM. Garbage collection is an important feature of Java. As the overhead of GC, especially for an incremental real-time GC, can be quite high on standard processors, several mechanisms for hardware support of GC have been proposed.

So far, Java processors failed in the standard and server computing domain. However, with the Azul system, hardware support for Java within a RISC processor has now entered the server domain. It will be interesting to see whether hardware support for Java and other managed languages will be included in mainstream processor architectures.

With the introduction of the safety-critical Java specification the interest in Java processors might increase. Java processors execute bytecode, which itself is easier to analyze. Certification of safety-critical applications will benefit from the direct execution of Java bytecode as the additional translation steps to C and machine code are avoided.

Chapter 8
Interfacing Java to Hardware Coprocessors and FPGAs

Jack Whitham and Neil Audsley

Abstract The previous chapter discussed how the Java Virtual Machine could benefit from some of its components being implemented in hardware. This chapter continues this hardware-support theme by considering how Java applications can interface to more general hardware coprocessors.

8.1 Introduction

Hardware coprocessors dedicated to a specific function have significant advantages over software implementations of that function – improved speed and reduced energy consumption are typical [95, 421, 442, 448], alongside increased timing predictability [422, 423, 433]. The speed-up of coprocessors over software implementations is often greater than one order of magnitude (Fig. 8.1). Typically, up to three orders of magnitude improvement can be achieved for certain applications [188, 189], e.g., image processing, encryption, simulation and other naturally data-parallel forms of computation. However, the available improvements are highly dependent on the nature of the application.

A simple example of a coprocessor is the *Floating-Point Unit* (FPU) within a CPU. Early 80x86 CPUs, such as the 80386 [221] did not contain an FPU. Instead, floating-point operations were emulated using integer instructions. The introduction of dedicated FPUs (i.e., 80387 [221]) provided an order of magnitude performance improvement compared with software emulation via integer instructions.

The performance speed-up of coprocessors is due to the application-specific nature of the coprocessor. Often the coprocessor performs a single dedicated function, and hence can be highly optimised to the specific characteristics of that

J. Whitham (✉) • N. Audsley
University of York, York, UK
e-mail: jack@cs.york.ac.uk; neil@cs.york.ac.uk

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_8,

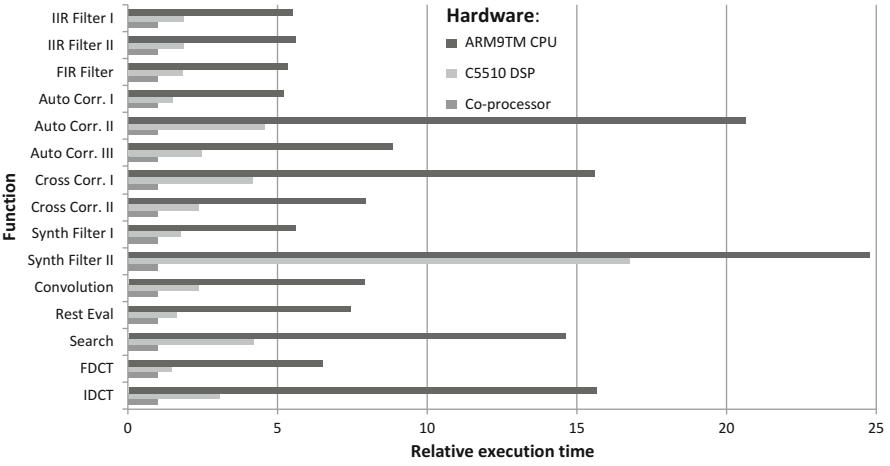


Fig. 8.1 The performance difference between coprocessor and software implementations of various multimedia processing functions, based on data published in [95]. The software implementations of each function were executed using a conventional ARM9TM CPU and a C5510 *Digital Signal Processor* (DSP); the coprocessor implementation was realised on an FPGA-like device. The coprocessor implementation is typically 1.5–2.5 times faster than software on the DSP, and five to ten times faster than software on the CPU

function. Coprocessors are able to use application-specific data paths, pipelined logic, specialised logic functions and fine-grained parallelism. This specialisation carries the disadvantage that the coprocessor is only useful for one sort of function. It contrasts with software, where generic logic and reusable data paths are used within a CPU to execute an effectively unlimited range of functions.

The use of coprocessors within a system raises two key issues:

- What functions should be implemented in coprocessors?
- How can coprocessors be used from software?

Returning to the example of floating-point implementation, clearly common floating-point operations (multiply, divide) should be included within the coprocessor. Interfacing from software is via appropriate native types within the programming language, enabling the compiler to generate appropriate FPU instructions.

Coprocessor implementation is not suitable for all functions: some are more effectively carried out by software (Fig. 8.2). The trade-off between software and coprocessor implementation is complex, and has given rise to a large research community studying the hardware–software co-design of systems [123, 343]. The fundamental issue is how much hardware area is needed to implement a particular function with a particular speed. Unfortunately, the complexity of software functions can increase almost arbitrarily due to the random access capabilities of memory. In software, functions do not need to be physically close to each other in order to communicate, and all memory space is reusable. Coprocessors do not share these properties. Silicon space is valuable, non-reusable, and efficiency

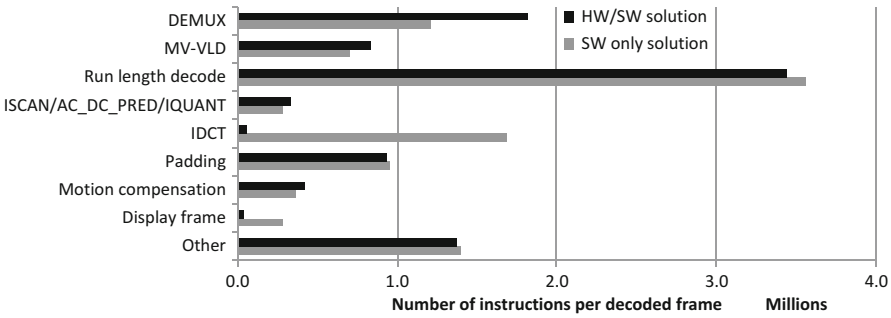


Fig. 8.2 The performance difference between pure software and coprocessor-assisted implementations of parts of an MPEG-4 video decoder, based on data published in [448]. The coprocessor implementation is significantly better than software for the IDCT and "Display frame" functions, but worse for DEMUX and MV-VLD

requires close proximity between logical functions that need to communicate. This disadvantage means that hardware coprocessors are only suitable for certain *hotspots*: the parts of an application that dominate the execution time and hence are most in need of optimisation [268]. The infrequently-used bulk of an application should remain in software.

Accessing coprocessors from software can be difficult. Floating point operations have been incorporated within programming languages, with compiler support for generating appropriate FPU instructions. However, in general the coprocessor implements application specific functions, with no standard method for accessing the coprocessor from software. The device driver paradigm solves part of the problem: the *Operating System* (OS) hosts the low-level parts of the interface, providing access to hardware via system calls. But further questions remain: what protocol should be used to communicate between software and the coprocessor? How can the interface be kept consistent as parts are upgraded? How can the overhead of the OS accesses be minimised? On top of this, engineers are forced to maintain three separate but interacting components, namely the coprocessor, the application software and the device driver. It is a costly arrangement that does not fit neatly into any conventional software engineering process.

Integration with Java software adds a further practical difficulty. Being part of the hardware platform, a coprocessor breaks the assumption of platform independence inherent within Java. By design, Java provides no standard way to communicate directly with the OS, except for simplistic file and network operations which are not useful in this case. Thus, a fourth component must be maintained: a JVM extension that can link Java code to the OS.

This chapter explores the various ways in which a Java system can interact with coprocessor hardware by describing solutions and workarounds for these problems. It begins with a taxonomy of coprocessor technologies (Sect. 8.2), followed by descriptions of various hardware–software interface paradigms (Sect. 8.3). An example implementation is described in Sect. 8.4. Section 8.5 concludes the chapter.

8.2 Classifying Coprocessors

Coprocessors cannot be understood without some reference to the environment in which they exist. This section begins by examining the sorts of computer architecture that may include coprocessors (Sect. 8.2.1), then looks at the sorts of hardware technology that are used to implement coprocessors (Sect. 8.2.2) and the languages used to describe coprocessor hardware (Sect. 8.2.3).

8.2.1 Coprocessor Architectures

Figure 8.3a illustrates a simplified computer architecture surrounding a coprocessor. This coprocessor exists as a *memory-mapped device* within the computer's memory space. It occupies a small range of memory addresses. Software communicates with the coprocessor by accessing the *registers* within this memory range. This simple coprocessor arrangement is common within simple embedded systems with only one memory space.

Another popular configuration links the CPU and coprocessor via a dedicated coprocessor bus (Fig. 8.3b). This type of coprocessor is accessed by special coprocessor instructions embedded within the program. On the ARM and 80x86 platforms, the coprocessor bus is conventionally used for floating-point acceleration [221], but some platforms allow several coprocessors to be attached in this way, implementing almost any application-specific function. For example, the Microblaze *Fast Simplex Link* (FSL) bus provides eight channels for coprocessors, accessed using instructions named GET and PUT [442], while ARM's CPI instruction can reference up to 16 coprocessors [364].

Dedicated buses enable high-speed communication between software and a coprocessor while maintaining some isolation from the CPU, so that either can be upgraded or replaced separately. The *bus latency* for coprocessor access is greatly reduced in relation to configurations where accesses must cross a memory bus. There is no possibility of bus contention, so the bus protocol can be very simple.

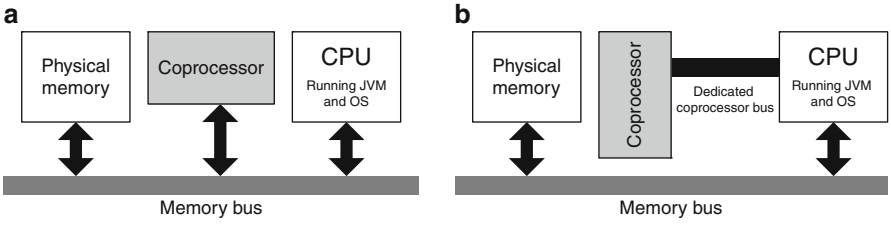


Fig. 8.3 Simplified architectures of computer systems that include coprocessors. (**a**) Includes a coprocessor configured as a memory-mapped device while (**b**) shows a coprocessor linked directly to the CPU via a dedicated bus

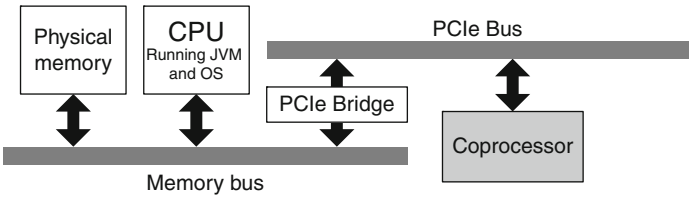


Fig. 8.4 Simplified PC or server architecture; the coprocessor is found on an expansion bus such as PCI Express

An *Application Specific Instruction-set Processor* (ASIP) integrates coprocessor functionality directly into the CPU data path so that it can be accessed using custom instructions [102]. In principle, this is even faster than a dedicated bus, but the presence of customisable units on the CPU's data path lowers the maximum frequency, as timing-critical CPU components cannot be optimally placed. Furthermore, the compiler must be modified to understand the new instructions. These issues have limited the uptake of ASIP technology.

The simple models of Fig. 8.3 are not suitable for use within a complex computer system such as a desktop PC or server. Multiple memory spaces and memory buses exist, and there is no practical way to connect coprocessors directly to the CPU core. The only suitable place for a coprocessor is on an expansion bus, such as PCI Express (PCIe), USB or HyperTransport [245]. This leads to a configuration similar to Fig. 8.4.

8.2.2 Coprocessor Hardware

Independently of its position within the computer architecture, a coprocessor may be implemented using a variety of different hardware technologies. At one time, coprocessors were implemented using discrete logic, just like early CPUs. As integrated circuits became more affordable, computer manufacturers began to use these for coprocessors. The result is usually known as an *Application Specific Integrated Circuit* (ASIC); a custom silicon chip for a particular application [380].

The cost of fabricating ASICs is high as a custom silicon design is necessary. An ASIC process is only considered worthwhile for systems that will be sold in high volumes [443]. Consequently, the industry attempted to reduce this cost by reusing common components within their ASIC designs. The *standard cell* is an innovation [441] enabling ASICs to be created from standardised logic gate designs in place of the previous *fully custom* designs, where individual transistors are placed on silicon. For a small reduction in performance, standard cells bring a large reduction in the cost of designing and testing [380]; additionally, it is possible to use fully custom techniques for the time-critical parts of the design.

The *structured ASIC* is a further development in which a two-dimensional matrix of standard cells are already in place, needing only a custom interconnection layer to define the functionality of the ASIC. This results in further reduction in design costs and a further reduction in performance.

Coprocessors can also be implemented by reprogrammable hardware. This minimises design costs because mass-produced off-the-shelf hardware can be used and because changes can be accommodated at any time during the design process. Various forms of programmable device can be used to host coprocessors, but devices such as *Programmable Logic Arrays* (PLAs) are typically too simplistic to be useful [434]. However, *Field Programmable Gate Arrays* (FPGAs) are a popular platform for coprocessors. These devices can be characterised as reconfigurable structured ASICs: they include a large matrix of standard cells with an interconnect that is controlled by reprogrammable memory. Although FPGAs do not compare well to ASICs in terms of energy consumption, density and speed, coprocessors built on FPGA are still significantly faster than software implementations [442].

FPGAs now offer sufficient density to host complete embedded systems, including a soft CPU core that may be customised with dedicated coprocessor buses and specialised instructions.[1]

The most recent implementation technology for coprocessors is the *Graphics Processing Unit* (GPU). Originally intended only for 3D graphics output, these devices have subsequently become general-purpose engines for parallel computing. Although GPUs are not as versatile as FPGAs, they benefit from the low cost of mass-produced off-the-shelf hardware and a high-speed interface to an expansion bus, similar to Fig. 8.4.

8.2.3 Coprocessor Design Languages

Largely independently of both hardware and the computer architecture, the languages and tools used for design entry can also form part of the classification of coprocessors. These are *Hardware Description Languages* (HDLs). They are superficially similar to software programming languages, but the semantics are very different. Importantly, everything declared within the HDL modules implicitly occurs in parallel. In general, HDLs are not suitable for writing software, and software languages are not suitable for designing hardware.

The same HDLs are used for implementing designs on FPGAs and ASICs. The most popular are VHDL (*Very-high-speed integrated circuit Hardware Description Language*) and Verilog [19]. For the purposes of describing hardware, the two are roughly equivalent, but have quite different syntax. These HDLs are not specific to particular technologies or manufacturers – they are widely-supported open standards – but they are low-level languages which describe only the *implementation*,

[1] Soft CPU cores are implemented as FPGA configurations – they use only the logic gates and flip flops provided by the standard cells within the FPGA.

i.e., the logic. All of the first-order language features map directly onto simple logical functions, wires and registers. JHDL is another example of a low-level HDL despite its Java-like syntax [81]. In these languages, features such as buses, channels and state machines must be implemented by the designer.

Higher-level HDLs have been designed, but uptake has been limited. When these languages are used, the designer specifies the *function* to be implemented, rather than the logic that will implement it. Handel-C is one example of this sort of HDL [74], based on both C and Occam. In Handel-C, channels and state machines are first-order language features, simplifying design entry. However, this abstraction restricts the low-level control available. Handel-C is a proprietary language supported by a single tool set. The Handel-C compiler generates lower-level code such as VHDL which is then processed by an FPGA or ASIC design tool.

For GPUs, software-like languages such as *Open Computing Language* (OpenCL) and C for *Compute Unified Device Architecture* (CUDA) are used to describe coprocessor functionality. VHDL and Verilog are not suitable for GPU platforms, as these are parallel computing engines rather than generic programmable hardware. Nevertheless, many useful types of coprocessor can be implemented [82].

8.3 Coprocessor Interfaces

In order to be useful, a coprocessor must be accessible from software. Access is provided via an *interface* between software and hardware, which allows data, control and status information to be relayed in both directions. Conventionally, the software acts as master, giving commands to the coprocessor.

The interface design is independent of the hardware platform and the design language, but it is not entirely independent of the architecture. The latency advantages of ASIPs and dedicated coprocessor buses are lost unless software can use them directly via specialised machine instructions, i.e., without going through any intermediate driver layer.

Within this section, interface designs are classified according to their integration with software languages in general and Java in particular. Section 8.3.1 considers low-level (minimally integrated) interfaces, Sect. 8.3.2 considers higher-level integrated interfaces, and Sect. 8.3.3 describes a midpoint between the two.

8.3.1 Low-level Interfaces

A low-level interface to hardware will involve an absolutely minimal level of integration with a software language. While coprocessor communication can be encapsulated by method calls, the programmer will have to write these. They will not be generated automatically. There will be no automatic mapping of language features onto hardware features. Typically, a programmer would implement a class to encapsulate all of the features relating to a particular coprocessor.

A coprocessor control class might make use of memory-mapped registers (represented as variables) or inline assembly code. Within the Real-Time Specification for Java, the `javax.realtime.RawMemoryAccess` class can be used to represent a range of physical memory as a fixed sequence of bytes. This can be the physical memory space assigned to a memory-mapped device. A `RawMemoryAccess` object is created with a physical memory address (the `base`) and a `size`:

```
RawMemoryAccess rawMemory =
    new RawMemoryAccess(PhysicalMemoryManager.IO_PAGE, base, size);
```

Having done this, it is possible to communicate with the device by reading and writing the addresses assigned to its registers:

```
rawMemory.setInt(0, 100);
x = rawMemory.getInt(1);
```

The `RawMemoryAccess` object might be created within the constructor of the coprocessor control class. In principle, this is all that is necessary to communicate with a memory-mapped coprocessor, but in reality, the resulting code is not easy to read or maintain. The source code contains nothing that could be used to automatically check the coprocessor accesses for semantic correctness. The programmer must ensure that the coprocessor control class is written correctly, by testing and manual inspection.

A minor concession to readability is possible by assigning names to registers and flags via constants. These would be defined within the coprocessor control class:

```
private final static int STATUS_REGISTER = 0;
private final static int DATA_REGISTER = 1;
```

However, these constants still do not encode any information about the structure of registers within the device. A further improvement is possible using the *Hardware Objects for Java* API [358,359], currently implemented by the JOP CPU and various JVMs including CACAO, SimpleRTJ and OVM. A hardware object is effectively a declaration of the register map for a memory-mapped device:

```
public final class SerialPort
             extends com.jopdesign.io.HwObject {
    public volatile int status;
    public volatile int data;
}
```

The class declaration can also encapsulate flags (as constants) and methods. The methods provide access to the device at a higher level; for example, a `SerialPort` class might provide a `read` method:

```
public int read() { return data; }
```

A hardware object is instantiated by a factory function, `makeHWObject`, in which the object is associated with the physical memory address of the hardware device:

```
SerialPort sp = (SerialPort) makeHWObject(new SerialPort(),
                                Const.IO_UART1_BASE, 0);
```

A hardware object can abstract some functionality of the hardware through methods and constants, providing some level of integration with Java language features. However, it cannot contain any *state* that is independent of the hardware. For example, if a coprocessor includes a control register, it is common to store the current contents of that register in a so-called *shadow copy* within memory. This makes it possible to update *part* of the control register (e.g., flip a single bit) with only one access: a valuable feature for a device driver, particularly if the control register is *write-only*. Hardware objects cannot support this, because all read-write fields are mapped to hardware registers. This also prevents a hardware object enforcing semantics for coprocessor access, so the programmer may need to add a secondary, higher-level coprocessor control class to wrap around the hardware object.

Coprocessor control classes can also make use of *native methods* via the *Java Native Interface* (JNI). These make it possible to use inline assembly code to access coprocessors, which is useful whenever coprocessor access requires special instructions that cannot be generated by a compiler, such as the `outb` 80x86 instruction which writes one byte to an I/O port, or the FSL instructions on the Microblaze platform.

JNI allows the declaration of *native methods* which are implemented by functions within *Dynamic Link Libraries* (DLLs) or *shared objects*. These may include code that accesses coprocessors. For example, here is a class containing a native method, which is implemented by a library named `serialport`:

```
public class SerialPort {
    public native int read();
    static {
        System.loadLibrary("serialport");
    }
}
```

JNI is also useful if a coprocessor can only be accessed via non-standard OS system calls. The OS interfaces built in to Java are all platform-independent by design; there is no generic device control operation similar to the Unix `ioctl` system call. If a coprocessor interface requires the use of `ioctl` or similar platform-specific system calls, then Java needs to be extended via JNI.

8.3.2 Integrated Interfaces

The low-level interfaces described above share the problem of poor integration with Java. The programmer must write an interface layer for the coprocessor by hand, and there will be no automatic way to check the correctness of this layer. An *integrated* interface attempts to avoid this manual process by mapping coprocessor features onto Java language features.

The coprocessor may be represented as an object, but this object may have capabilities beyond a simple interface to hardware. For example, it may be possible

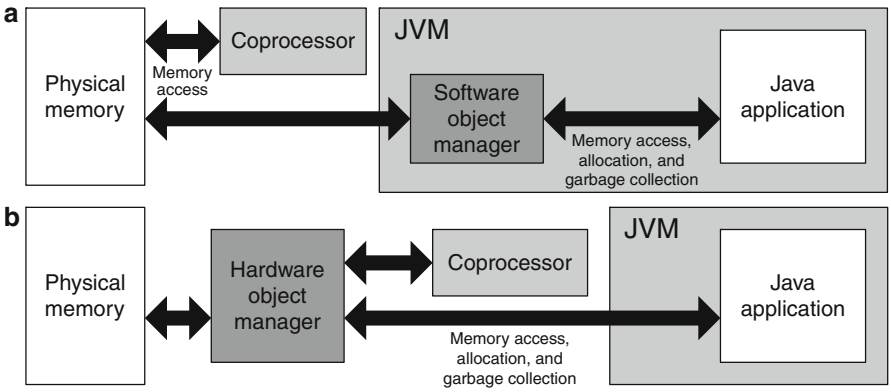


Fig. 8.5 Relationship between a JVM, coprocessor and physical memory for (**a**) a conventional system, and (**b**) a Javamen system [70]

to pass other objects to the coprocessor. The coprocessor may also be able to call methods implemented in software.

Two notable research projects have attempted this. The *Javamen* [70] project moves Java's object memory management system from the JVM into a dedicated coprocessor called the *Object Manager*. This component is responsible for garbage collection and dynamic memory allocation. It also maintains the mapping between Java references and physical memory locations; this mapping may be changed at any time by garbage collection.

Normally, these operations are carried out within the JVM, which means that they are not available to hardware and software outside of the JVM, including coprocessors (see Fig. 8.5a). These components are not able to safely access Java objects in memory, because they cannot resolve the physical memory address for a reference. The hardware Object Manager solves this problem by placing the required functionality *outside* the software JVM, where it can be shared by coprocessors and software applications (see Fig. 8.5b). In this arrangement, coprocessors can use Java references directly. Effectively, Java objects can be passed between hardware and software. This is useful whenever a coprocessor acts on more than one piece of data at the same time; all the required data can be stored within multiple fields within a single object. Coprocessor functionality is activated by method calls.

However, the Object Manager approach requires major changes within the JVM. The Java standard allows the JVM to define any object management system that works correctly for Java code, so there is no standard for this component. The Javamen project's Object Manager is highly specific to the JVM implemented by the JOP CPU, and any upgrade to this JVM might force a redesign of the Object Manager.

Java Hardware Threads [377] use an alternative approach to the same problem. Coprocessors are represented as Java-like threads, controlled by calls to methods

Table 8.1 Functions and opcodes used for communication between a coprocessor (hardware thread) and a proxy thread, from [377]

Opcode (coprocessor to proxy thread)	Description
GETDATA	Read memory location
PUTDATA	Write memory location
CALL	Call Java function
Functions (proxy thread to coprocessor)	
START	Start coprocessor operation
RESET	Reset coprocessor
CONTINUE	Resume operation after WAIT
Functions (coprocessor to proxy thread)	
WAIT_FOR_NEXT_PERIOD	Call RTSJ `waitForNextPeriod` method
WAIT_FOR_EVENT	Call RTSJ `waitForEvent` method
EXIT	Coprocessor has finished

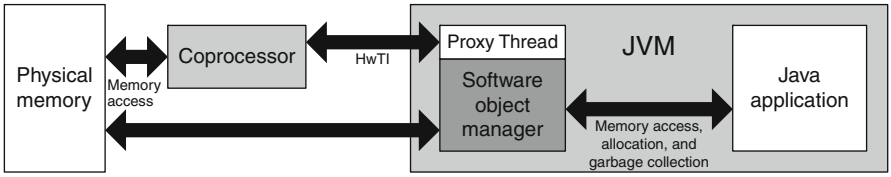


Fig. 8.6 Relationship between a JVM, coprocessor and physical memory for Hardware Threads [377]

such as `start`. Each coprocessor is linked by a *Hardware Thread Interface* (HwTI) to a *proxy thread*, implemented by software within the JVM. HwTI is a bidirectional physical bus connection between the two components, capable of relaying opcodes and data. The available opcodes are listed in Table 8.1. Figure 8.6 illustrates how the Hardware Threads arrangement differs from the Javamen arrangement.

The coprocessor can call Java methods by sending the CALL opcode via HwTI. These methods are actually executed by the proxy thread. The proxy thread initiates and resets coprocessor operations when called by other parts of the Java application. The proxy thread will also access memory on behalf of the coprocessor (GETDATA and PUTDATA); because these accesses are actually performed within Java, they use the software object manager that is part of the JVM. The solution involves no changes to the JVM, and is therefore portable.

Memory accesses via the proxy thread will be quite slow, because they must be serviced by software. An *interrupt* will be generated to indicate that messages are waiting on the HwTI bus; this will become a *signal* within the OS, and lastly an *event* within Java. Finally, a method within the proxy thread is called, and the request is processed. This overhead can be avoided by using GETDATA only to *dereference* memory locations, and then accessing data directly. However, the procedure for doing this is specific to a JVM, and garbage collectors may relocate objects at any time. Therefore, this approach is not suitable for all JVMs.

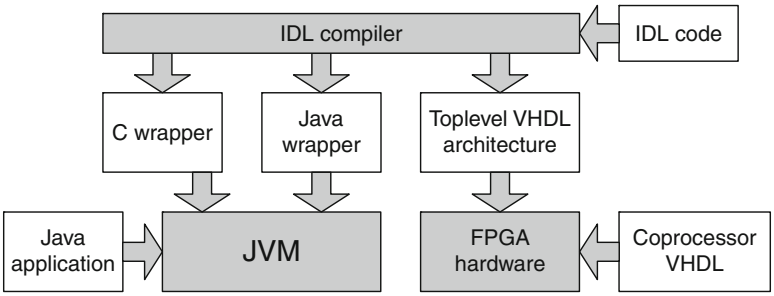


Fig. 8.7 All of the components of a generated interface are automatically produced from an *Interface Description Language* (IDL)

8.3.3 Generated Interfaces

A third class of coprocessor interface are partially automatically generated. Like integrated interfaces, the generation process hides low-level details of the hardware platform from the coprocessor and software designer.

Java language integration is not necessary if a secondary language is introduced to describe the interface between a coprocessor and an application. Such an *Interface Description Language* (IDL) is an input for a code generator which generates all of the code needed to implement the interface (Fig. 8.7). An IDL naturally restricts the interface functionality that is available, and the code generator produces interface code that is correct by construction.

Although the ability to implement any Java functionality as a coprocessor may be useful in some cases, the typical industrial use case for a coprocessor is very simple. Coprocessors are used as *stream processors*: they receive a number of data streams, usually expressed in the form of fixed-sized packets stored within byte or integer arrays, and produce output streams in a similar form. Coprocessors may also send and receive parameters in the form of scalar values, but direct access to Java objects and methods is not usually important. An IDL can easily accommodate the required functionality, and generate the required hardware and software interface components, similar to software-only interface generators such as SWIG [51].

8.4 Example Implementation

This section describes an example of a generated coprocessor interface. *Hardware Methods* was designed to meet typical industrial requirements [227]. It links Java code to hardware on two hardware platforms (JOP and PC), supports three JVMs (JOP, OpenJDK, and Jamaica), and two OSes (Linux and PikeOS).

Each element of coprocessor functionality is represented as a Java method, which may take primitive Java types (e.g., `int`, `char`) and arrays of primitives (e.g., `int[]`) as parameters. A primitive type may also be returned.

The operation of Hardware Methods is best illustrated by an example, which is based on the following Java class:

```
public class example {
    public int xorme(int [] work, int key)
    {
        int i;
        for (i = 0; i < work.length; i++) {
            work[i] = work[i] ^ key;
        }
        return work[i-1];
    }
}
```

`xorme` applies an *exclusive OR* (XOR) operation to each element within the array named `work`. The following sections describe the process of moving `xorme` to a coprocessor implementation. Coprocessors usually carry out far more complex operations: `xorme` is deliberately a very simple method to aid understanding.

8.4.1 Hardware Interface

The Hardware Methods interface generator performs most of the required work based upon an interface description:

```
coprocessor example {
    method xorme {
        parameter work {
            stream int inout;
            infifo 4;
        }
        parameter key {
            primitive int;
        }
    }
}
```

This specifies the name of the class, method and parameters. The array parameter is declared as an `inout stream` of `int`s. This bidirectional stream will carry the contents of the `work` parameter (1) *to* the coprocessor on startup, and (2) *from* the coprocessor on completion. A FIFO buffer (`infifo`) is used for the incoming data so that the coprocessor does not need to buffer it internally. Based on this description, the code generator produces a *top-level interface wrapper* to surround a VHDL component with the following interface:

```
entity example is
port (
    xorme_work_idata    :in std_logic_vector(0 to 31);
```

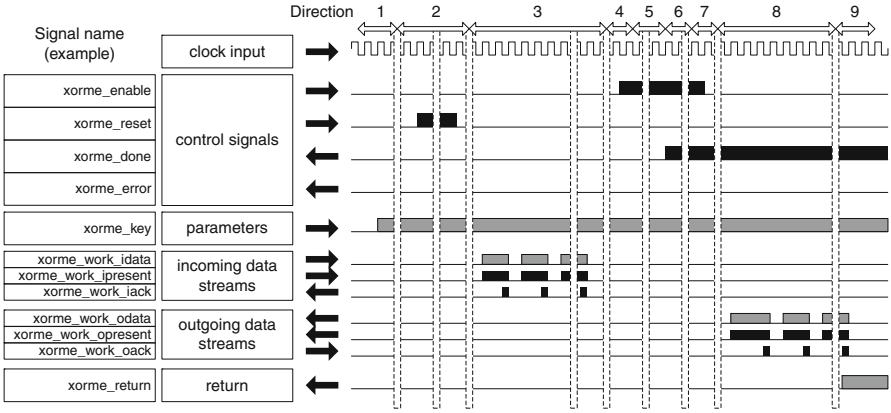


Fig. 8.8 The sequence of operations carried out by a Hardware Method, divided into nine phases. Signals that are shaded black should be asserted high; signals that are shaded grey carry data

```
    xorme_work_ipresent :in std_logic;
    xorme_work_iack     :out std_logic;
    xorme_work_odata    :out std_logic_vector(0 to 31);
    xorme_work_opresent :out std_logic;
    xorme_work_oack     :in std_logic;
    xorme_key           :in std_logic_vector(0 to 31);
    xorme_return        :out std_logic_vector(0 to 31);
    xorme_enable        :in std_logic;
    xorme_done          :out std_logic;
    xorme_error         :out std_logic;
    xorme_reset         :in std_logic;
    clk                 :in std_logic);
end entity example;
```

The contents of this component must implement `xorme`. A detailed explanation of VHDL code is outside of the scope of this chapter, but briefly, the parameters of the component (listed above) are the parameters of the `xorme` method, plus control signals. The `work` array is represented as two 32-bit synchronous channels for input and output. The `key` parameter is available as a 32-bit input; the `return` value is a 32-bit output. Processing should begin when the generated wrapper asserts the `xorme_enable` control line, as this indicates that all of the data has been received; the data size is an implicit property of the request, and is not explicitly passed to the coprocessor. The coprocessor asserts `xorme_done` when it is ready to produce output data; again, the data size is not explicitly stated.

Figure 8.8 shows the sequence of operations. The nine phases of operation are:

1. Primitive parameters are copied from the Java application to the Hardware Method. For example, the signal named `xorme_key` is given the value of `key` that was passed to the `xorme` method on invocation. It retains this value throughout the operation of the Hardware Method.

2. The `reset` control signal is set high then low.
3. Data is streamed from Java's memory to each of the incoming data streams defined by the interface, in parameter order. For `xorme`, there is only one incoming data stream (`xorme_work`).
4. The `enable` control signal is set high to signify that all of the input has now been received.
5. The Hardware Method carries out its operation. Some or all parts of the operation may actually be carried out while data is streamed.
6. The Hardware Method sets the `done` output to high, or the `error` output to high to signal an error condition. Once these signals are asserted, they must remain high until `reset`. The `done` or `error` output can follow `enable` after an arbitrary delay – it may occur in the same clock cycle, after several seconds or even longer.
   A timeout may be specified in the IDL code for coprocessors that carry out lengthy processing tasks.
7. The `enable` control signal is set low.
8. Data is taken from the outgoing data streams and stored in the PC's memory.
9. A return value, if any, is copied from hardware and returned by the Java function.

This operation sequence is implemented by a platform-specific combination of hardware, device drivers and the Java software itself. All of the required code is generated from the interface description, so that the user-defined parts of the coprocessor and the Java code do not need to change. The following sections describe two implementations of this platform-specific hardware.

8.4.2 Interface Components for PC

On PC platforms, Hardware Methods coprocessors are implemented using an FPGA on a PCIe expansion card. The OS prevents tasks accessing the PCIe bus directly, so a device driver is required. A PCIe bus interface component is also required on the FPGA. Figure 8.9 shows all of the components that are needed within this environment.

Some of these components are generated: the top-level interface wrapper for the coprocessor (bottom of Fig. 8.9), and the Java and C wrappers (top left). The device driver and PCIe interface stay the same for all interface descriptions. The device driver provides a device node, which is `/dev/hwmethods` on Linux. This allows any application to access Hardware Methods if granted permission by the OS.

For the `xorme` example on the PC platform, the generated Java code is:

```
public class example {
    final static Object Monitor_xorme = new Object();
    native int Native_xorme (int [] work, int key);
```

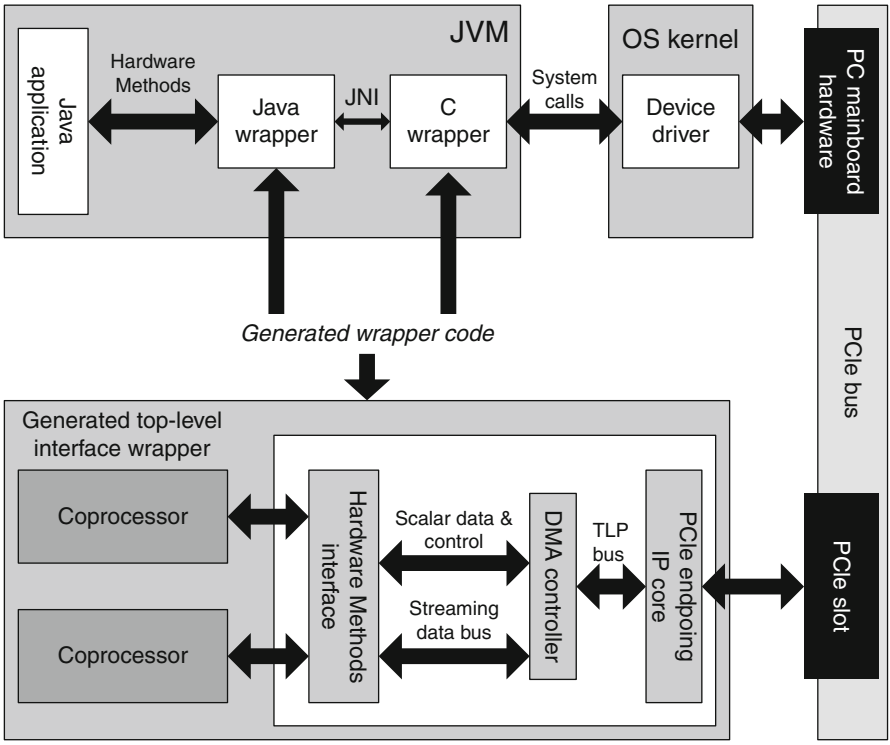


Fig. 8.9 The components that are used to link Java to a coprocessor within a PC platform

```
    public int xorme (int [] data, int key)
    {
        synchronized(Monitor_xorme) {
            return Native_xorme (data, key);
        }
    }
}
```

The `Native_xorme` method is generated along with the Java code. It is C code, and it is responsible for communicating with the device driver on behalf of Java. The generated Java code cannot do this without JNI, because OS-specific system calls such as `ioctl` are used. `Native_xorme` implements the sequence shown in Fig. 8.8, and contains:

```
JNIEXPORT void JNICALL Java_example_Native_1xorme (
    JNIEnv *env, jobject o, jintArray work, jint key)
{
    jsize work_size = sizeof(jint) *
            (*env)->GetArrayLength(env, work);
    jint * work_c =
            (*env)->GetIntArrayElements(env, work, 0);
    struct hwm_reg r;
```

```
    r.reg = 1;
    r.value = (__u32) key;
    _HWM_Ioctl(0, HWM_SET_REG, &r);
    _HWM_Ioctl(0, HWM_RESET_METHOD, (void *) 0);
    _HWM_Write(0, 1, work, work_size);
    _HWM_Ioctl(0, HWM_RUN_METHOD, (void *) 0);
    _HWM_Read(0, 1, work, work_size);
    r.reg = 1;
    _HWM_Ioctl(0, HWM_GET_REG, &r);

    (*env)->ReleaseIntArrayElements(env,
                            work, work_c, 0);
    return (int) r.value;
}
```

The Java array `work` is locked so that it can be accessed from C. This is important to prevent the relocation of the array by garbage collection. Some JVMs may also store the array in non-contiguous physical memory: the `GetIntArrayElements` function ensures that the array occupies contiguous space to enable access from C.

The `_HWM_Ioctl`, `_HWM_Read` and `_HWM_Write` functions send commands to the coprocessor via the device driver. Commands are sent using the `ioctl` system call:

- HWM_SET_REG command sets the parameter input `key`. This parameter is represented by a memory-mapped register.
- HWM_RESET_METHOD resets the coprocessor by toggling `xorme_reset`. This is a flag within a memory-mapped register.
- HWM_RUN_METHOD starts the coprocessor. A start command is sent to the top-level interface wrapper, which holds `xorme_enable` high until `xorme_done` is asserted.

The `_HWM_Write` function sends the contents of the `work` array to the coprocessor using the `write` system call. `_HWM_Read` obtains the modified contents of `work` from the coprocessor using the `read` system call. Finally, a `HWM_GET_REG` operation via `ioctl` obtains the return value.

The "magic" numbers used by these function calls (such as the first parameter, 0) match the top-level interface wrapper and indicate which Hardware Method feature is being accessed. To detect a mismatch between the software interface and the hardware interface, a checksum value is generated from the interface description and shared between the two.

8.4.3 Interface Components for JOP

The JOP platform has no OS and no JNI, so the implementation of `xorme` for JOP must access the coprocessor directly. The communication path is shown in Fig. 8.10. The low-level `rdMem` and `wrMem` functions implemented by JOP (in `com.jopdesign.sys.Native`) are used to get access to the registers within the Hardware Methods interface.

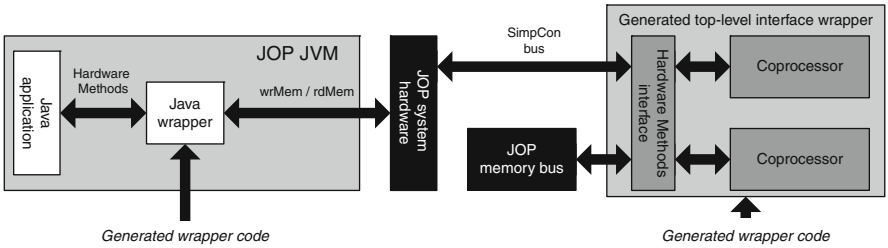


Fig. 8.10 The components that are used to link Java to a coprocessor within the JOP platform

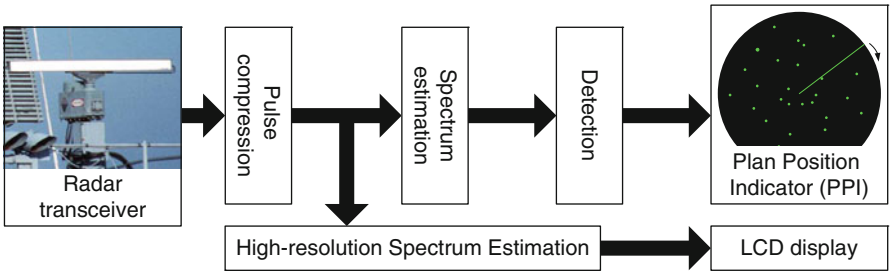


Fig. 8.11 The Radar Signal Processor (RSP) involves four distinct processing tasks which form a natural pipeline

On the JOP platform, the coprocessor has access to the memory bus, as illustrated in Fig. 8.3a. Therefore, the input and output data streams are passed by reference; the generated Java code determines the physical location of the arrays that contain the data, and then passes this location to the coprocessor which accesses the memory directly. The required memory access functionality can be generated as a part of the wrapper.

JOP enables *Worst Case Execution Time* (WCET) analysis of Java code. This is also possible for Hardware Methods provided that the execution time of the coprocessor can be estimated [433].

8.4.4 Industrial Use Case

The PC implementation of Hardware Methods was used by engineers at EADS to create a *Radar Signal Processor* (RSP) based on Java. The RSP obtains raw data from a radar transceiver and produces a radar display to represent detected signals (Fig. 8.11). RSPs once required expensive application-specific hardware, but general purpose computers are now sufficiently powerful to allow most of the required functionality to be implemented in Java. Where application-specific hardware is

Table 8.2 Bandwidth for a simple Hardware Method implemented on an FPGA with a PCIe X1 connection to a PC

Array size (words)	Average execution time of `sum` (μs)	Average bandwidth (megabytes/s)	Average overhead of generated Java code (μs)
1	45.7	0.083	0.310
100	73.6	5.19	0.311
200	84.4	9.03	0.308
400	88.5	17.2	0.306
1000	100	38.1	0.306
2000	124	61.3	0.306
4000	145	105	0.306
10000	316	120	0.306
20000	550	139	0.306
40000	1051	145	0.306
100000	2590	147	0.306

required, it can be introduced using Hardware Methods. The requirements of EADS were a useful guide for the development of the Hardware Methods feature set, and the work was carried out as part of the JEOPARD project [227].

The RSP is organised as a pipeline of distinct operations (Fig. 8.11). The steps can occur in parallel, and it is also possible to carry out subparts of some of the steps in parallel. A *burst* of raw data from the transceiver contains eight *sweeps*, and the data from each sweep can be processed separately. Thus, pulse compression can be carried out by eight independent Java threads.

Spectrum estimation can also be parallelized. For this step, EADS experimented with both software and hardware implementations, making use of Hardware Methods to connect their Java code to coprocessors. They found that an coprocessor solution would be useful if sufficient FPGA space was available. In their experiments, it was parallelized using two Xilinx ML505 FPGA boards [228].

High resolution spectrum estimation (HRSE) is an additional feature of the RSP which allows the user to focus on a specific position within the radar display. EADS also experimented with coprocessor implementations of this component. Hardware Methods made it possible to switch easily between Java and coprocessor implementations.

8.4.5 Bandwidth

The maximum theoretical bandwidth of a PCIe X1 lane is 250 megabytes/s [293]. An implementation of Hardware Methods based on one Xilinx ML505 FPGA board cannot reach this because of the overhead of relaying commands between Java and the coprocessor hardware. However, it can approach it.

Table 8.2 shows some properties for a Hardware Method named `sum`, which implements the following function:

```
public class example {
    public int sum(int [] work)
    {
        int total = 0;
        for (i = 0; i < work.length; i++) {
            total += work[i];
        }
        return total;
    }
}
```

This method requires very little processing: the addition operation can be performed in parallel with the data transfer. It is therefore a useful way to reveal the overheads of Hardware Methods. These are shown in Table 8.2, which was calculated by a Java test harness for the `sum` method. The test harness called `sum` with array sizes between 1 element and 100,000 elements. Each `sum` operation was repeated until 20 s of real time had elapsed. The total time taken was measured using a second Hardware Method which provides the number of FPGA clock cycles that elapsed, which is converted into seconds and then divided by the number of operations completed to obtain the average execution time of each `sum`. The average PCIe bandwidth was also calculated by dividing the number of bytes transferred by the total time. Additionally, the overhead of executing Java code and calling the JNI functions was calculated by removing all code within the JNI function for `sum`, then re-executing the test program. The code was compiled using Jamaica Builder 6.0beta2 and executed on a quad-core Intel Q9450 PC with Linux 2.6.26.

On this platform, for small arrays (e.g., one element), there is an overhead of nearly 50 µs for each Hardware Method invocation. The overhead becomes unimportant as the array size increases. The bandwidth increases sharply, up to a maximum of 147 megabytes/s. PCIe is most effective when carrying large *burst* transactions and Hardware Methods take advantage of this.

8.4.6 Latency

For a real-time application, the worst-case latency of a Hardware Method is very important. Figure 8.12 shows the execution time of the C implementation of the `sum` Hardware Method, as sampled one million times on an 80x86 platform using the `rdtsc` instruction. This was done using the Linux OS in single-user mode (to avoid interference from other tasks) and using a single CPU core. `rdtsc` reads a 64-bit clock cycle counter within the CPU core, providing highly accurate data about instruction execution.

The mean latency is 37.5 µs; close to that shown in Table 8.2, where additional overheads from the JVM apply. However, the measured worst-case latency is 72.9 µs. As the shape of Fig. 8.12 illustrates, latencies under 45 µs are overwhelmingly more likely than outliers such as 72.9 µs, occurring 99.7% of the time. For a soft real-time application with some tolerance for deadline misses, this is

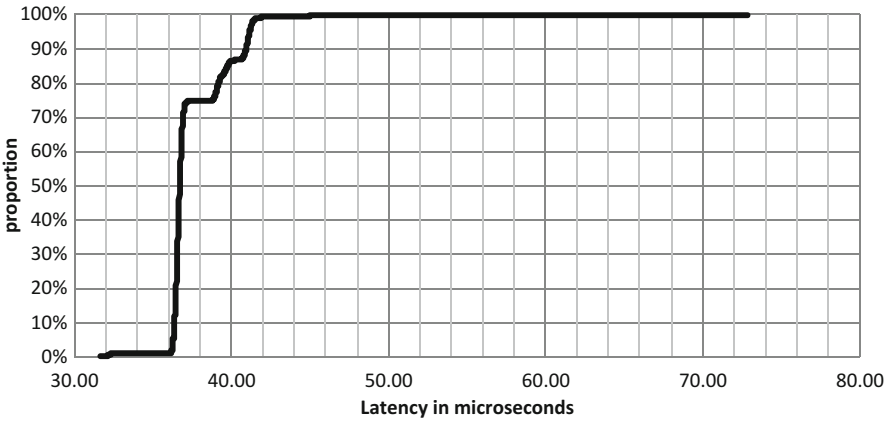


Fig. 8.12 Latencies of the `sum` Hardware Method. The line indicates the proportion of samples that were *less than or equal to* a specific latency in microseconds

quite acceptable. The RSP implemented by EADS is a good example of a real-time application that can tolerate occasional increases in latency.

A hard real-time application with no tolerance for deadline misses should not be implemented on a PC platform. A time-predictable platform such as JOP would be more suitable. On JOP, it is possible to apply WCET analysis to determine the absolute worst-case latency of Hardware Method execution [433].

8.5 Conclusion

Some parts of data-intensive applications will benefit from coprocessor implementation. This chapter has summarised how coprocessors can be implemented using technologies such as FPGAs and ASICs, and how they can be closely linked to software languages such as Java. It has also described an example implementation named Hardware Methods which has been applied in an industrial application based on real-time Java. Coprocessors are not a replacement for all parts of any embedded application, and some applications will not benefit at all, but for suitable applications, significant improvements in performance and energy consumption are possible.

Acknowledgements The authors are grateful to the staff at EADS Ulm for their help in developing and evaluating the Hardware Methods for PC platform, and to Martin Schoeberl for his assistance with Hardware Methods for JOP.

Chapter 9
Safety-Critical Java: The Mission Approach

James J. Hunt and Kelvin Nilsen

Abstract Safety-Critical Java is designed to simplify the runtime environment and code generation model for safety-critical applications compared with conventional Java, as well as improve the reuse and modular composability of independently developed software components and their corresponding certification artifacts. While the conventional object-oriented abstractions provided by the Java language already enable strong separation of concerns for passive components such as subroutines and data structures, safety-critical Java's Mission abstraction generalizes these encapsulation benefits to the domain of active software components.

Example active components that can be built as a Mission include device drivers, network communication stacks, plotting of RADAR and SONAR data, track management of RADAR and SONAR plots, and implementation of graphic user interfaces. These various active software components will likely be reused, reconfigured, and combined in different ways for different applications. Each safety-critical mission is comprised of one or more logical threads of execution which are generally structured as a set of periodic event handlers, asynchronous event handlers, and no-heap realtime threads.

The Safety-Critical Java specification enforces strong separation of concerns between independently developed missions. And outer-nested missions are never allowed to access the data contained within inner-nested missions. A safety-critical Java application consists of one or more missions running in sequence or concurrently. This chapter introduces the concept of Safety-Critical Java Missions and motivates their use as a tool for software engineering abstraction.

J.J. Hunt (✉)
aicas GmbH, Karlsruhe, Germany
e-mail: jjh@aicas.com

K. Nilsen
Atego, North America
e-mail: kelvin.nilsen@atego.com

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_9,

9.1 Introduction

Though programming correctness is desirable for all programming domains, it is particularly important for safety-critical applications. Whereas a programming error in a desktop or server system may be annoying and even quite costly, a failure in a safety-critical system can result in significant injury or loss of life. For this reason, safety-critical developers employ a more rigorous development process to help ensure program correctness. *Safety-Critical Java* (SCJ) and its mission concept where developed with this process in mind.

Ada has been the premier language for safety-critical programming, but its popularity has been declining. Java offers many of the advantages of Ada, such as a well defined syntax and semantics as well as built-in multithreading plus a simpler object model than that of Ada 95 and C++. Java also avoids certain known vulnerabilities in popular alternative languages, including the line-oriented macro preprocessor and pointer manipulation of the C and C++ languages.

However, Java is inherently a heap-based language, designed for use with dynamic memory allocation and garbage collection. Standards such as DO-178b are geared towards less dynamic environments, making safety certification of conventional Java program exceedingly difficult. Today's safety-critical practitioners generally advise against all dynamic heap memory management in safety-critical programs. An objective of the SCJ design is to enable the use of Java without the need for tracing garbage collection.

SCJ derives from the *Real-Time Specification for Java* (RTSJ) [65,132], borrowing both its execution model and its use of scoped memory. In both cases, the SCJ specification restricts the use of these features. The mission concept of the SCJ specification provides the structure for these restrictions. All execution (usually in the form of asynchronous event handlers) in a Safety-Critical Java application are associated with a particular mission; event handlers are all started during mission initialization; and they all halt prior to mission termination. Further, each mission has an associated memory scope which is known as its mission memory. Only objects allocated within a mission memory scope may be shared between multiple threads. Each thread belonging to a mission may also have one or more private memory scopes, which nest within the shared mission memory scope.

The scoped memory capabilities defined by the RTSJ provide a means of managing memory without a heap, but it is a complex mechanism requiring runtime checks to ensure the absence of dangling pointers and scope cycles. The simplified form of scoped memory supported by SCJ enables the use of a static analyzer to enforce consistency between programmer-supplied annotations and Java source code. This static analysis system is able to prove the absence of assignments that would violate the rules of scoped memory, resulting in exceptions at run time.

The mission concept of SCJ was first developed in a European Union research and development project [3]. The key idea was that a typical safety-critical program has two main phases, an initialization phase and an active phase in which the system performs its intended tasks. Two different types of restricted memory scopes are

used to hold mission data and the task release data respectively. Data structures used to share information between tasks or to represent information that must persist throughout the mission's active phase are allocated within the common mission memory area during the mission initialization phase. Data created in the active phase is generally limited to the context of a single release of a task.

A mission may embody the life cycle of a complete application, the functionality associated with a single phase of an application, or one of multiple concurrent modules that comprise an application. During the development of JSR-302, the mission concept was extended to support a cyclic executive execution mode for single threaded systems and a nested mission execution mode for more complex systems. All of these modes have a common mission model consisting of an initialization phase, an active phase, and a cleanup phase. In addition, a set of missions can be wrapped in a mission sequencer which can restart the mission or transition to a new mission. This is a much more constrained programming model than most programming languages offer, but it simplifies the runtime environment by limiting interaction between software components.

9.2 Current Practice

Safety-critical software serves critical roles in the control of aircraft, air traffic, passenger and freight rail systems, railway switching and segment access, automobile and truck electronics, street traffic signals, medical equipment, nuclear power generation, safety aspects of manufacturing and chemical processes, deep sea oil drilling rigs, and a diverse variety of defense applications. Each application domain is governed by different safety guidelines and each software development project navigates its own path through the safety certification processes. The result is a diversity of approaches that reflects this diversity of industries and regulatory bodies.

9.2.1 Resource Management in Safety Critical Systems

Not only do developers of safety-critical systems limit the languages they use, but also how resources are managed, particularly memory and processor. An error in their use can result in complete system failure. In order to guard against such failure, some simplifications are usually made.

Manual dynamic memory management is notoriously error prone, particularly when using languages such as C and C++, where pointers can be manipulated. For this reason, safety-critical system designers usually avoid it. Instead, all objects needed for a program are either allocated at the beginning of the program or on the stack. Pointers to local stack-allocated objects may be shared with invoked procedures. Such sharing is unsafe in C and C++, because there is no way to

enforce that these pointers do not outlive the objects they refer to. With Ada, such sharing can be done safely, with the help of its stronger type system and nested procedures.

Of the languages currently used for safety-critical systems, only Ada supports multithreading in the language. Other languages must rely on libraries to provide threading capabilities. Typically, safety-critical systems are composed of threads using priority preemptive scheduling; however, many safety-critical systems do not even use threads. Instead many tasks are multiplexed in a single thread of execution referred to as a cyclic executive.

9.2.2 Languages

Language use in safety-critical systems lags behind the desktop and server markets. Though precise market data is elusive, there is general agreement among industry insiders that the large majority (more than 70%) of current safety-critical development is done in the C and C++ languages. Ada, though declining, is still well represented in military and aerospace projects. Even assembly still plays an important role, particularly for applications running on 8 and 16 bit processors.

Software engineering managers responsible for selecting technologies often cite the popularity of C and C++ as the primary reason for selecting these languages. Popularity yields improved economies of scale, lower cost, more capable development tools, easier recruiting of skilled software developers, and improved compatibility with legacy software capabilities. Since neither C nor C++ was designed for safety-critical development, safety-critical development efforts based on these languages generally adopt guidelines that restrict access to particular language capabilities, prohibiting language features that are less reliable or may be more difficult to prove safe. For example, the large majority of software incorporated into the Joint Strike Fighter was implemented in a restricted subset of C++ [255].

In contrast to the approach taken by Lockheed Martin with the Joint Strike Fighter project, most safety-critical development efforts adopt off-the-shelf industry standards for safety-critical development rather than defining their own. Several such standards are reviewed in brief below: MISRA C, Ravenscar Ada, and SPARK Ada. The MISRA C++ standard is not discussed, since the concepts are largely compatible with MISRA C.

9.2.2.1 MISRA C

MISRA C [275] was initially intended to "provide assistance to the automotive industry in the application and creation within vehicle systems of safe and reliable software." Subsequent to the original publication, these guidelines have been adopted by many other industries and are in fact cited by the Joint Strike Fighter coding guidelines mentioned above. The MISRA-C guidelines can be best characterized as "words to the wise."

The guidelines are organized as 122 mandatory rules and 20 advisory rules. The rules recommend avoidance of C language features and libraries that are large, complex, difficult to understand, or likely to be implemented in different (non-portable) ways with subtle behavioral differences on different platforms. Many of the restrictions, such as the prohibitions on heap memory allocation (rule 20.4), use of the `offsetof` macro (rule 20.6), use of `setjmp` macro and `longjmp` function (rule 20.7), use of signal handling facilities (rule 20.8), and use of input/output libraries (rule 20.9), can easily be enforced by lint-like tools and several vendors of embedded system C compilers have incorporated enforcement of these MISRA-C guidelines into their standard tool chains.

Other MISRA-C guidelines are more difficult to enforce with automatic software development tools. For example, Rule 17.6 states "The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist." While adherence to this rule is required for reliable operation of all C programs, even those that are not safety critical, this rule cannot be reliably enforced by pattern-based static analysis tools. Tools may identify code that potentially lead to violation of this rule, but a human understanding of the program's behavior is generally required to assess whether the rule is in fact violated. Other more safety-conscious languages enable automatic enforcement of this sort of restriction by introducing type system rules or runtime checks to guarantee this problem does not arise.

9.2.2.2 Ada

The Ada programming language [417] was designed in response to a competition sponsored by the U.S. Department of Defense because of a growing awareness of the critical and expanding role of software in modern defense systems, and a general sense that the defense industry was experiencing a "software crisis." Unlike C and C++, Ada is block structured. This means procedure declarations can be nested. One way that Ada guarantees safe sharing of references to objects with automatic storage (stack allocated objects) is by enforcing special type safety rules. If the address of a stack-allocated object is taken, it must be represented by a variable of a locally defined named type. According to the scoping rules of Ada, this named type is visible to inner-nested procedures but nowhere else. Thus, there is no way to leak a pointer to the stack-allocated object to outer scopes without violating the type system that is enforced by the Ada compiler. This is one of many reasons why software engineers familiar with the Ada language claim the language is a better technical fit for the needs of safety-critical development than C or C++.

9.2.2.3 Ravenscar Ada

Whereas the Ada language is designed for programming in the large and has an assortment of complex features designed to support the full generality required in this domain, Ravenscar Ada [89] is a set of restrictions that simplify the Ada runtime environment and reduce the nature of interactions that may occur between Ada tasks. Together, these restrictions make it possible to analyze the schedulability of a collection of Ada tasks in order to prove that all tasks satisfy their realtime constraints. The Ravenscar Ada profile is best known for enabling the use of rate monotonic analysis in safety-critical Ada programs. Some of the restrictions imposed on full Ada are listed below.

1. Tasks cannot be dynamically created or terminated. All tasks must persist for the lifetime of the application. Task priorities cannot be changed dynamically. Interrupt handlers cannot dynamically attach or detach.
2. Complicated intertask interactions such as Ada rendezvous, Ada select statements, requeue operations, and abort statements are prohibited. All intertask coordination is implemented with protected objects (similar to Java monitors). Tasks are not allowed to self suspend while executing within a protected object (similar to executing a Java synchronized statement). At most one task can wait on each protected object at a time. All protected objects must reside at the library (outer-most scope) level and must be statically allocated.
3. Standard Ada libraries are not allowed to perform implicit heap memory allocation. Application code may allocate from the "standard storage pool" or from "user-defined storage pools."
4. Applications are not allowed to perform asynchronous transfer of control.
5. All delays must be specified as absolute rather than relative times in order to avoid race conditions in establishing the start of each relative time span.
6. Ravenscar Ada uses a special elaboration control that suspends execution of tasks and interrupt handlers until the "environment task" has completed elaboration. Ada elaboration is similar in concept to Java's execution of static initialization code for each loaded class.

Many of the Ravenscar Ada limitations are enforced with runtime checks rather than by static analysis of application code. For example, if a thread's priority exceeds the ceiling priority of a protected object that it attempts to access, a runtime error condition will be raised. Similarly, if multiple tasks queue for entry to a protected object, or a task terminates, or a task attempts to perform a potentially blocking operation from within a protected object, a runtime error indication will be raised. Ravenscar Ada provides a foundation on which Ada developers can, through careful software engineering, build safety-critical applications. It offers little beyond the built-in type enforcement of the Ada language itself to assure that software is properly constructed.

9.2.2.4 SPARK Ada

SPARK Ada [28] combines the strengths of data flow analysis with the strong type checking already built into the Ada language. A SPARK program is written in a subset of the Ada language, with supplemental SPARK-specific information represented in stylized Ada comments. The information represented within these comments include preconditions and postconditions for each Ada procedure and function, descriptions of intended dependencies between input and output parameters, and identification of all nonlocal variables that may be accessed by the subroutine. SPARK is implemented by four tools which run in combination with the Ada compiler. These are the *Examiner*, the *Simplifier*, the *Proof Checker*, and the *Proof Obligation Summarizer* (POGS).

The Examiner parses the original Ada source program and enforces certain restrictions on the use of full Ada. SPARK prohibits anonymous types (all types must be named) and requires that all constraints, including array length constraints, be static. SPARK, for example, prohibits class-wide types and dynamic dispatching (virtual method calls). It also prohibits the use of access types (an Ada access is like a Java reference or a C pointer). SPARK prohibits `goto` statements and labels, and it forbids the raising of exceptions and exception handlers. Further, SPARK enforces that an actual parameter cannot be aliased by both an input and output formal parameter. This prevents Ada code from exhibiting implementation-specific behavior if one Ada compiler passes the argument by reference but a different compiler passes the argument by copy. Additionally, SPARK prohibits dynamic heap storage allocation other than on the execution stack, the declaration of arrays with dynamic bounds, and recursion. It prohibits type coercions between values of unrelated types. It also restricts initialization expressions for elaboration to depend only on statically available information. Thus, unlike traditional Ada, the results of elaboration have no dependency on the orders in which packages are elaborated.

The Examiner supports three different levels of analysis, depending on the level of rigor requested of the tool. Intraprocedural data flow analysis represents the most basic of these analysis levels. This analysis proves that every variable is initialized before its first use, assures that all `in` parameters are consulted within the body of the subprogram, and assures that all `out` parameters are set on every path through the body of the subprogram. The second analysis level is characterized as information flow analysis. SPARK subprogram declarations may be augmented with `derives` annotations to clarify the programmer's intent with respect to information flow between `in` and `out` parameters. For example, the annotation

```
--# derives x from y;
```

indicates that the value assigned to `x` within the body of this procedure must depend on the incoming value of variable `y`. Information flow analysis assures that subprogram implementations are consistent with the SPARK `derives` annotations.

The third and highest level of analysis is characterized as verification. This involves proving certain characteristics of the program, including proofs that the SPARK postcondition annotations logically follow from the SPARK precondition

annotations and all possible executions of the program. Based on examination of the source code, the Examiner outputs various conjectures that must be proven in order to satisfy the constraints specified in the SPARK-specific comments and to assure that assignments to integer subranges are always within range. If these conjectures can be proven, then the program is known to execute without runtime errors that would raise exceptions in traditional full Ada.

The Simplifier examines the conjectures output from the Examiner and endeavors to prove the conjectures by performing routine rule-based transformations on the conjectures. Some conjectures are easily reduced to `true` by these routine transformations. For other conjectures, the proof requires additional effort. The Simplifier's search for conjecture proofs is restricted in order to guarantee that it produces its results with a predictable and bounded amount of effort. The Simplifier intentionally avoids proof techniques that involve traversing very large and sometimes unboundedly deep search spaces.

The output of the Simplifier, which includes simplified conjectures that are not yet proven, is passed to the Proof Checker. The Proof Checker is an interactive tool that relies on humans to guide the proof strategies. The human user is presented with a conjecture that needs to be proven. In response, the human suggests proof strategies to be applied by the proof checker in order to establish the desired proof. Among strategies that the human user can request are

1. Pattern-guided transformations (as performed by the Simplifier),
2. Deduction based on truth tables,
3. Proof by independent analysis of individual cases,
4. Proof by contradiction, and
5. Proof by induction.

In some cases, difficulty with construction of a proof may motivate restructuring of the original source code and annotations. The ultimate objective of running SPARK verification is to establish a proof that the safety-critical application runs correctly.

9.3 Safety Standards for Software

SCJ has been designed for use in systems where certification is required. The emphasis has been to reduce Java to a set of functionality that would be relatively easy to pass certification requirements. This is much easier said than done. The difficulty is that most standards do not address implementation language features directly.

In general, safety-critical software controls mechanical and electronic systems that have the potential of causing human casualties in the case that the software behaves erroneously. These systems tend to be realtime systems as well, i.e., systems that must always react to some event within a bounded amount of time.

Each industry has its own standards for guiding the development of safety-critical software. Most of these standards establish recommended development practices

and identify software development artifacts that must be gathered and audited before relevant government regulatory agencies approve deployment of the safety-critical system. The goal of these standards is to ensure that the resulting software properly fulfills its requirements and does nothing else.

9.3.1 Avionics Software

Since the avionics domain has the longest history of safety certification, avionics standards have the most impact on other standards. Three standard documents were considered during development of the SCJ specification:

- DO-178B [335] (ED-12B [156]) Software Considerations in Airborne Systems and Equipment Certification,
- DO-248B [333] (ED-94B [159]) Final Annual Report for Clarification of DO-178B (ED-12B), and
- DO-278 [332] (ED-109 [158]) Software Standard for Non-Airborne Systems.

These documents set out a requirements driven process for developing avionics software. The process covers the entire software life cycle from plan to deployment. At each step, a well defined set of acceptance criteria must be fulfilled by the applicant. In general, this includes a full set of high-level requirements, low-level requirements and software architecture, source code, and object code. In addition, tests with test coverage analysis must be provided to demonstrate that the software implements all its requirements and no extraneous functionality. Each test and the code it tests must be traceable back to some set of low-level requirements. These low-level requirements and the architecture must in turn be traceable back to the high-level requirements and ultimately back to the system requirements that are implemented in software.

The stringency of the standard depends on the level of criticality of the application. For airborne systems, criticality is classified according to the following levels.

Level A Software whose anomalous behavior could cause or contribute to a failure of system function resulting in a catastrophic failure condition for the aircraft. A catastrophic failure is one which would prevent continued safe flight and landing.

Level B Software whose anomalous behavior could cause or contribute to a failure of system function resulting in a hazardous/severe major failure condition for the aircraft. A hazardous/severe-major failure is one which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a large reduction in safety margins or potentially fatal injuries to a small number of the aircrafts' occupants.

Level C Software whose anomalous behavior could cause or contribute to a failure of system function resulting in a major failure condition for the aircraft. A major

failure is one which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or discomfort to occupants, possibly including injuries.

Level D Software whose anomalous behavior could cause or contribute to a failure of system function resulting in a minor failure condition for the aircraft. A minor failure is one which would not significantly reduce aircraft safety or functionality.

Level E Software whose anomalous behavior could cause or contribute to a failure of system function with no effect on aircraft operational capability.

Level A applications require extensive testing and code coverage. Not only is full statement and decision coverage expected, but also multiple condition decision coverage (MCDC) is required as well. This means that each variable used in a condition for a branch must be tested separately to ensure that it independently affects the outcome of the condition.

Though Java has been successfully used for Level D systems, the requirements of higher levels and current accepted practice make it difficult to use Java at higher certification levels. In particular, though it is not expressly forbidden, only very restricted dynamic memory allocation is allowed in level A and B systems. This was the main motivation for removing garbage collection from SCJ.

ED-217 [155][1] supplement to ED-12C [157, 331] explicitly addresses dynamic memory allocation including garbage collection. Still, it will probably take some time before an implementation of Java with garbage collection is certified. Even then, a specially design Java runtime environment will be needed, and special techniques will need to be developed to certify that application code interacts in a reliable way with the certified implementation of automatic garbage collection.

9.3.2 Railway System Software

Rail system certification is most advanced in Europe, due to the heavy use of rail for passenger traffic in dense population zones. The main standard in Europe is EN 50128 [138]. Internationally, IEC 61508 [222], "Functional safety of Electrical/-Electronic/Programmable Electronic safety-related Systems" is the main reference standard. This standard is also used in other areas such as industrial automation as well.

As with avionics standard, applications are categorized according to their criticality, i.e., the damage their failure could cause. Additionally, the probability of failure is considered, with the maximum allowed probability of a catastrophic failure (for Level 4) specified to range between 10^{-5} and 10^{-4} per year per system.

[1]The RTCA has not yet assigned the DO designation.

The type of verification techniques that must be used to show that a software component meets its specification depends on the *Safety Integrity Level* (SIL) that has been assigned to that component. For example, Level 4 software might be constrained so it can be subjected to various static analysis techniques (such as control flow analysis).

Evidence is also required for structural coverage analysis, an analysis of the execution flows for the software that determines that all paths through the software have been tested or analyzed, and that there is an absence of unintended function within the code. Additionally, decisions affecting control flow may also need to be examined and evidence produced to show that all decisions, and that conditions within those decisions, have been independently exercised though testing or analysis.

9.3.3 Medical Device Software

Software standards for medical devices are mainly regulated by national bodies. In the USA, the Food and Drug Administration (FDA) is responsible both for promulgating regulatory guidelines and for product approval. In the EU, guidelines are devised by the European Council (EC) but products are certified by organizations known as Notified Bodies, which are independent organization authorized by EU individual Member States.

ISO 13485:2003, "Medical Device Quality Management Systems", is recognized by many regulators as a basis for certifying medical devices when incorporated into device manufacturers' management systems. The current document supersedes its 1996 incarnation as well as EN 46001, EN 46002 and ISO 13488. ISO 13485 is based on ISO 9001, but it removes emphasis on continual improvement and customer satisfaction. Instead, ISO 13485 emphasizes meeting regulatory as well as customer requirements, risk management, and maintaining effective processes, in particular processes specific to the safe design, manufacture, and distribution of medical devices. ISO 13485 is in part designed to produce a management system that facilitates compliance to the requirements of customers and global regulators.

While being certified to ISO 13485 does not fulfill the requirements of either the FDA or foreign regulators, the certification aligns an organization's management system to the requirements of the FDA's *Quality System Regulation* (QSR) requirements as well as many other regulatory requirements around the world. Therefore, ISO 13485 certification requires a management system that can be thought of as a framework on which to build compliance to regulatory requirements.

Medical device companies who market within the USA must ensure that they comply with medical device regulations as governed by the FDA. The medical device companies must be able to produce sufficient evidence to support compliance in this area. The FDA *Center for Devices and Radiological Health* (CDRH) provides guidance for industry and medical device staff which include

risk-avoidance activities to be performed when incorporating off-the-shelf software into medical devices, recommended software testing and validation practices, and materials to be submitted to regulatory agencies in order to obtain permission to market medical devices. Although the CDRH guidance documents provide information on which software activities should be performed, including risk avoidance activities, they do not enforce any specific method for performing these activities.

9.3.4 Automotive Software

Software for automotive applications is not yet subject to safety regulations. The MISRA standards where set up by the industry in Europe to help guide automotive software development, but are not mandated. A new standard for automotive is being developed: ISO 26262 Road vehicles—Functional safety. As with other safety-critical standards, it concentrates on the software development process with a requirements driven development and testing strategy. Its classification of application according to their criticality is similar to IEC 61508, based on *Automotive Safety Integrity Level (ASIL)* instead of SIL. It remains to be seen whether or not governments will start to mandate ISO 26262 or some other software standard. Until then, the industry will remain self regulated with regards to software development practices.

9.3.5 Applicability of Java

Most safety-critical standards address general process rather than specifying specific techniques. In fact, the standards strive to be as technology independent as possible. This leaves each development organization to independently determine which technologies are best suited to each safety-critical process, resulting in a safety critical practitioner culture that is resistant to any new technology that differs from whatever technology was successfully deployed on previous projects.

One problem with using Java technology for critical systems is that though the process models need not change, the definitions used for requirements traceability are predicated on procedural programming models. Typical object-oriented features such as dynamic dispatch of polymorphic methods and dynamic allocation of memory for temporary objects are not part of the existing safety-critical standardization vocabulary. Only DO-178C (ED-12C) has begun to address this difficulty with its *Object-Oriented Technology Supplement*.

The overview description of safety certification above indicates that the effort required to achieve certification far outweighs the time required to actually write the software. This means that for a programming language to offer a substantial cost savings, more than just coding efficiency must be addressed. Rather, reducing

the scope of errors, improving analysis, and improving reuse of not just code but certification artifacts as well must be achieved.

9.4 A Different Kind of Java

Today's developers of safety-critical systems use combinations of the approaches described above, along with other techniques that are less well known. The intent in developing a SCJ standard has been to build upon the successes of existing approaches while improving upon certain aspects of the current state of practice. Compared with C, Java is a much more rigorous language. The SCJ standard, for example, avoids the problems that C has with dangling pointers. In comparison with Ada, Java offers similar safety with improved portability and scalability. Benefits such as those embodied in SPARK are available in the form of various static analysis and formal methods tools for Java. These existing tools can be directly applied to SCJ since it is a subset of full Java. An advantage of Java over Ada is that Java is a very popular programming language that is taught in almost all undergraduate education programs. The popularity of Java results in improved economies of scale, offering higher quality development tools at lower costs, an abundance of off-the-shelf reusable software components, and easier recruiting of competent off-the-street software developers.

There are significant differences between SCJ and the conventional Java used on desktop and server machines. These differences reflect the difference between realtime control systems and general purpose applications in a time-sharing environment. SCJ is designed for consistency of response instead of flexibility and throughput.

Conventional Java's runtime model supports incremental loading of applications. When a running program first encounters a dependency on a particular Java class, it dynamically loads and initializes the class. In fact, the conventional Java specification requires that each class be initialized immediately before its first use. With SCJ, all of the classes are preloaded and all class initialization happens before the safety-critical program begins to execute. JIT compilers are not appropriate for a SCJ environment since they make timing analysis unnecessarily complex and complicate the traceability analysis that is typically required during safety certification to demonstrate equivalence of source and binary code. Instead, class loading and compiling of byte code to native machine code will typically be done ahead of time as in more traditional language implementations.

Conventional Java relies on a garbage collector to reclaim memory for objects that are no longer in use. SCJ does not rely on garbage collection. Instead, memory for temporary objects is provided by a stack of scopes associated with each thread.

When a thread leaves a scope, all of the memory associated with objects allocated within that scope is immediately reclaimed. Only a very few permanent objects are allocated in immortal memory (a heap without garbage collection).

Conventional Java profiles have a single threading abstraction represented by `java.lang.Thread`. The RTSJ complements this basic threading mechanism by adding support for realtime scheduling through several additional types of schedulable entities, known as `NoHeapRealtimeThread`, `RealtimeThread`, and `AsyncEventHandler`. In the RTSJ, all of these classes implement the `Schedulable` interface. Informally, the realtime programmer refers to all of these types as *schedulables*. SCJ is a subset of the capabilities of the RTSJ, supporting only specialized subclasses of `AsyncEventHandler` and `NoHeapRealtimeThread`. SCJ refers to these specialized subclasses collectively as *managed schedulables*, and to the specialized `NoHeapRealtimeThread` subclass as a *managed thread*.

Scope-based allocation in SCJ is a specialization of the scope-based memory allocation provided by the RTSJ. The scoped memory model of the SCJ specification is a simplification of the RTSJ in that SCJ does not allow arbitrary sharing of scopes between threads and it does not execute object finalizers when scopes are exited. SCJ supports only two specialized forms of scoped memory: `MissionMemory`, which holds the objects that are shared between the various tasks of a safety-critical mission, and `PrivateMemory`, which holds the temporary objects that are allocated by a particular task for its specific needs. Private memory areas may not be shared between threads. Multiple private memory areas may be associated with each task according to a LIFO (stack) organization. As with the RTSJ, objects allocated within inner-nested private memory areas may refer (point) to objects residing in outer-nested private memory areas and objects belonging to the surrounding mission memory area.

9.5 The Mission Concept

The mission concept is at the heart of SCJ. It is composed of two parts: a *mission sequencer* and a set of *mission* objects. The mission sequencer is responsible for managing the memory area used for starting a mission and deciding the sequence in which missions are executed. In some sense, the sequencer is a batch scheduler of applications with each application being represented by an independent mission.

A mission has three phases: an *initialization phase*, an *active phase*, and a *cleanup phase*. This corresponds to a common design pattern for safety-critical systems, in which shared data structures are allocated before the system becomes active. The introduction of mission sequencing enables warm restart of and switching between missions. By using a memory scope that stays active from the beginning of the first phase to the end of the last phase, but no longer, the memory used by a mission can be reclaimed safely at the end of a mission cycle. Figure 9.1 illustrates this three-phase model.

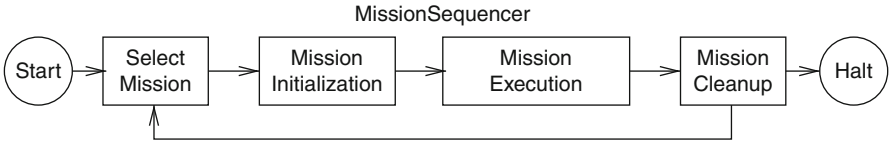


Fig. 9.1 Mission concept

A SCJ developer needs to structure a safety-critical application as one or more missions. Missions are run sequentially by a `MissionSequencer`. The SCJ library includes implementations of mission sequencers for certain common design patterns, such as repeated or one-time execution of a preallocated sequence of missions, but more sophisticated interactions generally require programmers to implement a custom mission sequencer.

9.5.1 Mission Sequencing

A mission sequencer provides a means of controlling restart and mission adaptation. For example, if a mission terminates abnormally, the sequencer may start a simpler mission to recover. In other words, it decides which mission to run at application commencement and each time a mission completes. It also provides an interface for terminating a sequence of missions.

A safety-critical application is started via a `Safelet`: an interface similar to an `Applet` for web browsers or an `Xlet` in multimedia applications. The `Safelet` interface defines the `getSequencer` method. The infrastructure invokes this method to obtain a reference to the application's main mission sequencer.

`MissionSequencer` is an abstract class. The developer must extend this class to implement the `getNextMission` method. The infrastructure invokes this method to obtain the next mission to run. If this method returns `null`, the sequencer considers its job done and the sequencer terminates.

9.5.1.1 Deciding What Comes Next

How missions are sequenced is application dependent. There are several criteria that might be used to decide what mission to run next. For example, if a mission returns because a particular activity failed, the sequencer might replace the failed mission with a simpler mission that has degraded capabilities. In other situations, sequencing of missions may be the natural result of physical phase changes. For example, a mission that runs during aircraft takeoff may be replaced by a different mission that runs during normal flight, and that mission may be replaced by another during the aircraft's landing activities. The mission sequencer object may contain

state variables that are visible to executing missions, and missions may modify these variables in order to influence the sequencer's selection of which mission to execute next.

9.5.1.2 Ending a Sequence

A sequencer ends in one of two ways. The sequencer may terminate itself by returning `null` from the `getNextMission` method. Or application code may force termination by invoking the `requestSequenceTermination` on the `MissionSequencer` object or on the `Mission` object that is currently running under the direction of the `MissionSequencer`. The `requestSequenceTermination` method arranges for the current mission to terminate and for the sequencer to refrain from starting another mission.

9.5.2 Mission Life Cycle

As illustrated in Fig. 9.2, a mission sequencer controls the execution of a sequence of missions, but the sequencer's thread of control is idle while one of its controlled mission is active. Only initialization and cleanup phases of a mission are executed in the sequencer's thread. During the mission initialization phase, the sequencer's thread invokes the mission's `initialize` method and during the mission termination phase, the sequencer's thread invokes the mission's `cleanUp` method.

9.5.2.1 Mission Initialization

A mission sequencer invokes its own `getNextMission` to determine the next mission to be run, and enters the mission's initialization phase by invoking the mission's `initialize` method. This method must be provided by the mission implementer and is responsible for setting up all tasks that are to run during the execution phase and the associated data structures. At the end of the initialization phase, after the `initialize` method completes, all tasks are started and the mission transitions to the active phase. The sequencer's thread now waits until the mission ends.

9.5.2.2 Mission Execution

The active or execution phase is where the mission actually performs its function. The tasks of the mission are structured as event handlers or threads. An event handler runs when fired, either by application code or, in the case of periodic event handlers, by the SCJ infrastructure. Application developers implement the `handleAsyncEvent`

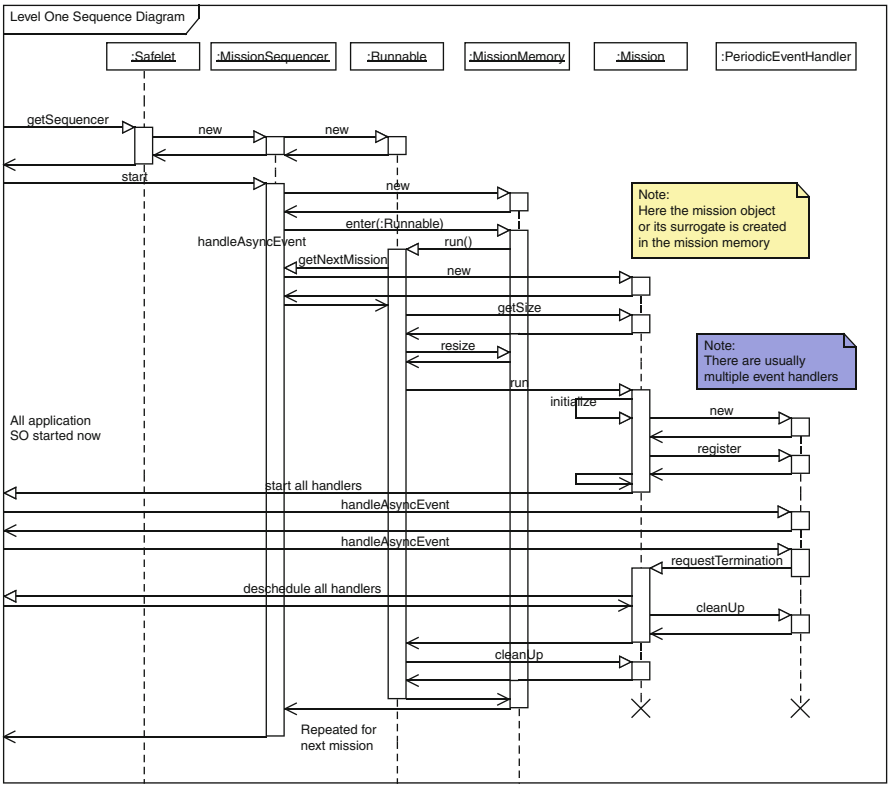


Fig. 9.2 Mission life cycle

method to provide the code that executes every time an event handler is fired. Each task has its own memory area for limited allocation, and this memory is cleared at the end of each release. Therefore, `handleAsyncEvent` starts with an empty memory area each time it executes. There is no explicit limit on the duration of the mission phase. Rather, one of the tasks or, in the case of nested mission, an external task, needs to explicitly end the mission by invoking its `requestTermination` method. If this does not occur, then the mission phase continues until the hardware is powered off.

9.5.2.3 Mission Cleanup

The mission cleanup phase is responsible for releasing external resources and leaving the system in a state that is appropriate for execution of the next mission. Invocation of `Mission.requestTermination` method wakes up the enclosing sequencer's control thread so that it can send termination requests to the various threads that are bound to the event handlers that comprise the active mission. Each

of these event handler threads will terminate execution following return from any currently executing invocations of `handleAsyncEvent` method and the infrastructure will refrain from further firings of all event handlers. The SCJ infrastructure does not force termination of any currently executing `handleAsyncEvent` methods and does not force termination of any running managed threads that are associated with the current mission. Rather, application programmer implementations of event handlers and managed threads are expected to voluntarily relinquish control when mission termination is requested. Application programmers implement their own protocols for communicating the need for a voluntary shutdown by overriding the mission's `requestTermination` method. When overriding this method, it is generally recommended that the overriding method invoke the superclass method of the same name.

After the sequencer's thread confirms that all threads associated with this mission have terminated, it invokes the mission's `cleanUp` method. Upon return from `cleanUp`, the sequencer thread invokes its own `getNextMission` and, assuming a non-`null` result is returned, the sequencer proceeds with initialization and execution of a new replacement mission.

9.5.3 Memory and Thread Management

Garbage collection is not part of the SCJ specification. Instead, RTSJ scoped memory is used to manage memory. There is a strong correlation between the mission classes and memory areas. Both missions and tasks have their own memory areas. They are coordinated to eliminate memory fragmentation and simplify the use of scoped memory. This is aided by assigning each mission sequencer and mission task its own memory budget.

Each schedulable object has its own memory budget from which its scoped memories are allocated. These budgets correspond to areas of memory referred to as backing store. SCJ allocates these areas in a stack-like manner.

Figure 9.3 illustrates the layout of memory for an SCJ program whose top level sequencer is running a mission with two tasks and a subsequencer running another mission with a single subtask. During the initialization of the top level mission, task 1 was created, then the subsequencer, and finally task 2. Here, one can see that each of the missions retained a bit more memory than was needed for its mission memory. For instance, task 2s budget does not include all the free area above its private memory. This layout assumes that backing store for mission memories are allocated in one direction and for private memories in the other, but other arrangements are possible.

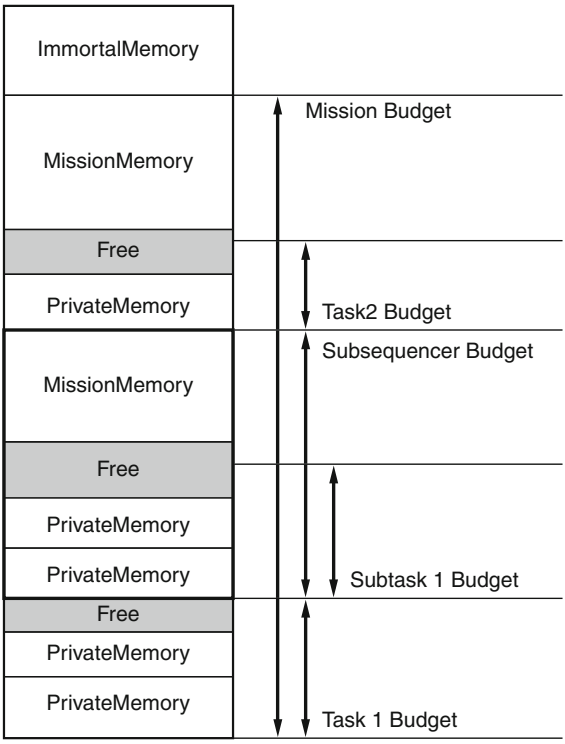


Fig. 9.3 Managing storage

9.5.3.1 Missions and Their Sequencer

Each sequencer manages a `MissionMemory` area on behalf of the missions that run under its control. This is a specialization of RTSJ scoped memory. All missions of a mission manager use the same mission memory, which is always emptied and may be resized to suit the needs of each mission each time a new mission is started. The backing store for the `MissionMemory` is taken from the backing store reservation associated with the mission sequencer. Whatever backing store memory is not associated with the running mission's `MissionMemory` is divided between the backing store reservations for all of the threads associated with the running mission. When a mission terminates, all of the scopes for its associated threads are discarded and their backing stores are coalesced once again into one large contiguous region of memory which can be used to satisfy the backing store needs of the replacement mission. Thus, the mission sequencer's backing store reservation must be larger than the maximum of the backing store requirements associated with any of the missions that run under the control of this mission sequencer. This discipline enables the system to avoid fragmentation of the underlying program memory. Figure 9.3 shows an example layout.

9.5.3.2 Tasks

A task is implemented as a `ManagedSchedulable` object. In levels zero and one, this is an event handler. Level two provides threads as well. Each task has its own private scoped memory that is not shared with any other thread. For event handlers, this memory provides volatile storage for each release of the handler. The memory is cleared at the end of each release. A thread is not bound to a release so its memory is not cleared until the thread terminates. Within a thread, deallocation occurs when control exits private memory areas. For all kinds of tasks, nested private scopes can be used to manage memory more finely.

9.5.3.3 Sharing Information

Sharing data structures between threads requires the use of surrounding contexts, either an enclosing `MissionMemory` area or the `ImmortalMemory` area. Whenever objects are allocated within outer-nested memory areas, care must be given to assure that the allocations do not exhaust the available storage within those outer areas. This is particularly true for allocations from `ImmortalMemory`, since this area is never be cleared, and for `MissionMemory`, which is only cleared when its corresponding mission is terminated.

9.6 Additional SCJ Architectural Features

Though the mission concept is the central feature of SCJ, it is not the only significant differentiating attribute. SCJ was designed to support several different deployment scenarios, both in terms of its threading model as well as interaction with the outside world. Three deployment levels are provided to help tailor the system to the application at hand. This flexibility of deployment can support simple programs with minimal overhead while still providing support for complex systems.

9.6.1 Single vs. Multitasking

The first differentiator between the levels is the tasking model. Is there just a single execution context or many? Level zero provides a cyclic executive with just a single thread of control. Levels one and two provide separate threads of control for each task.

In accordance with the cyclic executive model supported by SCJ, a cyclic schedule consists of multiple frames, each of which has a sequence of event handlers which are executed sequentially within the frame, and a fixed duration which represents the amount of time allotted for execution of the entire sequence of event handlers. Each frame may have a different duration.

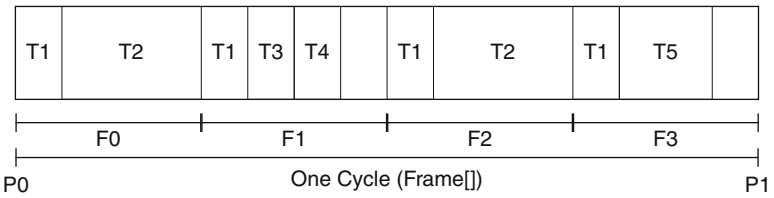


Fig. 9.4 One cycle of a cyclic executive

Figure 9.4 illustrates a cyclic schedule comprised of four frames `F0`–`F3`. Here, five tasks are scheduled periodically. Task `T1` runs four times in each cycle, Task `T2` runs twice, and all other tasks run just once. This is done by scheduling `T1` and `T2` in frame `F0`, `T1`, `T3`, and `T4` in frame `F1`, `T1` and `T2` again in frame `F2`, and finally `T1` and `T5` in frame `F2`. In this illustration, each frame has identical duration, though cyclic schedules can be constructed in which each frame has a different duration.

At level one, instead of assigning tasks to frames of a cyclic schedule, each task runs as a separate event handler. The same effect as in Fig. 9.4 can be attained by giving each handler the appropriate priority and scheduling parameters. If P is the period of the cyclic schedule, then the following would produce the same schedule: run `T1` at the highest priority with a period of $P/4$; run `T2` at the second highest priority and a period of $P/2$; run `T3` at the third highest priority with a period of P and a time offset of $P/4$; run `T4` at the lowest priority with a period of P and a time offset of $P/4$; and finally run `T5` also at the lowest priority with a period of P but a time offset of $3/4P$.

In order to ease the transition from level zero to level one, the tasks are described using the same SCJ data types: in both cases, the event handling code is structured as a `PeriodicEventHandler`. Though many programs will run the same whether executed with a level-zero or a level-one infrastructure, the two runtime environments are not equivalent. Programs running with level-one or level-two infrastructures must consider the possibility that high priority event handlers will preempt lower priority event handlers. This will never occur in a level-zero environment. In general, this requires that code intended to one at levels one or two must introduce synchronization primitives into their code, e.g., **`synchronized`** methods, whereas these are not necessary in a level-zero application.

9.6.2 Sequential vs. Nested Missions

At levels zero and one, only one mission may run at a time, whereas level two allows multiple missions to run concurrently. A level-two mission is allowed to include one or more mission sequencers among its associated threads. Each of these inner-nested mission sequencers may run its own sequence of missions. Thus, level two offers increased flexibility and generality, at the cost of greater complexity. Mission nesting aids the developer in organizing complex missions.

Allowing only one mission at a time simplifies timing analysis. When the entire application is comprised of a single mission, it may not be necessary to analyze the CPU time requirements of the mission's initialization and cleanup phases, since the initialization and cleanup code does not run during the application's active phase.

Allowing nested mission execution enables more flexibility but requires more subtle analysis of thread schedulability. In particular, rate monotonic analysis must study the combined workload of all concurrently running missions. Furthermore, rate monotonic analysis must consider phases during which certain missions are undergoing initialization while others are in the cleanup or active phases.

9.6.3 Event Handlers vs. Threads

The base model for executing tasks in SCJ is asynchronous event handlers. Two variants are supported: the `PeriodicEventHandler` class and the `AperiodicEventHandler` class. Periodic handlers provide clock based scheduling of tasks, whereas aperiodic handlers can be used to react to asynchronous events.

Level zero provides just periodic handlers, since there is no asynchrony at level zero. The static cyclic schedule is constructed prior to run time, based on the known periods of all the `PeriodicEventHandlers` that comprise the level-zero mission.

Level two provides threads in addition to event handlers. This is intended to be used for ongoing tasks that are not easily structured as event handlers. Level-two threads may coordinate with event handlers and other threads using traditional Java's `wait` and `notify` services. There are no API restrictions to how threads can be used. Though having threads gives some additional flexibility, one cost is that each thread must check for pending termination to ensure proper shutdown behavior. Note that the developer of a level-two mission will typically override the `requestTermiantion` method in order to interrupt any threads that might be blocked in a `wait` request.

9.6.4 Input and Output

A major application domain of safety-critical systems is system control, but this requires data input and output. Text handling is much less important than raw device interaction. As with other capabilities, SCJ provides increasing I/O facilities at each level. This ranges from simple I/O polling in level 0 to full interrupt handling in level 1.

Since text manipulation is less important in SCJ than in conventional Java, a much simpler UTF-8 only text interface is provided. This provides basic text manipulation and full language flexibility without the need for supporting multiple character sets. The facility is based on the connection model of Java 2 Micro Edition.

Beyond text interaction with a user, a safety-critical system must be able to interact with its environment. The simplest mechanism is via `RawMemory`. SCJ uses the `RawMemory` interface from the RTSJ. At level zero, polling `RawMemory` objects is the main mechanism for data acquisition, while writing to `RawMemory` objects representing actuator registers provides the main mechanism for control.

Levels one and two add the ability to handle asynchronous events. This is accomplished by supporting the RTSJ `Happening` model. Happenings provide a means of connecting external events to asynchronous event handlers. There is both a first-level event model based on the `InterruptServiceRoutine` abstraction and a second-level model based on the `AsyncEvent` abstraction.

At all levels, the system can use the J2ME connection interface to interact with the outside world. This can be used to provide a simplified interface to devices, networks, and internet protocols such as HTTP. These interfaces may in turn be implemented with the above- mentioned low-level facilities.

9.6.5 Annotations

SCJ defines an annotation system that is enforced by a special static analysis tool to prove that SCJ programs adhere to the rules for safe manipulation of scope-allocated objects. This gives safety-critical developers an assurance that certain scoped memory exceptions will not be thrown during the execution of their SCJ programs. This also enables the compiler to remove certain checks.

The annotation checker tool enforces that methods declared with the `@SCJRestricted(INITIALIZATION)` or the `SCJRestricted(CLEANUP)` annotations are only invoked by other methods that are similarly declared. A special exception to this rule allows SCJ infrastructure to directly invoke methods that are annotated as both `SCJAllowed(SUPPORT)` and `SCJRestricted(INITIALIZATION)` or `SCJRestricted(CLEANUP)`. The SCJ infrastructure invokes initialization methods only during initialization of a mission or of a `Safelet`, and only invokes cleanup methods during termination of a mission or of a `Safelet`.

The annotation checker also enforces the constraints of `@SCJAllowed` and `@Scope` annotations. A method declared to be `@SCJAllowed` at level *N* is not allowed to invoke any other method or access any fields that are labeled as `@SCJAllowed` at some level that is greater than *N*. The checker also assures that objects annotated with `@Scope` annotations to reside in a particular scope are only allocated within that named scope. And the annotation checker enforces that, unless annotated to the contrary, every instance method of a class is invoked from the same scope that holds the target object of the invocation. Another way to describe this requirement is to state that by default, every instance method carries the `@RunsIn(THIS)` annotation. This annotation denotes that the method must be invoked from the same scope that holds the **`this`** object which is the target of the invocation.

9.6.6 Libraries

The set of standard libraries available to SCJ programmers is a small subset of the standard libraries. Besides simplifying the runtime environment, a secondary goal of the SCJ specification was to reduce the volume of library code that needs to be certified in a SCJ application. Conventional Java libraries that were judged to be less relevant to safety-critical applications and libraries that would not operate reliably in the absence of tracing garbage collection have been excluded from the SCJ standard.

9.7 Example Application

A small railway example illustrates how SCJ works. Collision avoidance in rail systems is a representative safety-critical application. A common approach to the challenge of avoiding train system collisions divides all tracks into independently governed segments. A central rail traffic control system takes responsibility for authorizing particular trains to occupy particular rail segments at a given time. Each train is individually responsible for honoring the train segment authorizations that are granted to it. Note that rail segment control addresses multiple competing concerns. On the one hand, there is a desire to optimize utilization of railway resources. This argues for high speeds and unencumbered access. On the other hand, there is a need to assure the safety of rail transport. This motivates lower speeds, larger safety buffers between traveling trains, and more conservative sharing of rail segments.

Below is a possible implementation of the on-board safety-critical software that is responsible for controlling the train under the direction of a central rail traffic control authority. The sample implementation code is illustrative in nature, addressing many of the issues that are relevant to real world safety-critical systems. However, the sample code is not for actual deployment in a safety-critical railway management system as it has not been subjected to the rigorous processes required for development of safety-critical code.

The central traffic authority's primary objective is to assure safety of rail transport. Once safety is assured, the secondary objective is to optimize utilization of the rail transport network. The on-board software has several important responsibilities, including the following activities.

Maintain reliable and secure communication with the central rail traffic control authority. If communication is unreliable, the train will not receive track segment authorizations which are required to make forward progress or the central authority will not receive notification from the train that it no longer occupies particular rail segments, preventing other trains from making forward progress. If security is compromised by a malicious intruder, forging communications to or from the central authority, multiple trains may be authorized to occupy the same rail segment, significantly increasing the probability of a collision.

Monitor progress of the train along its assigned route. In anticipation of entering new rail segments, the train must request authorization to enter those segments. As the train approaches track switches, it must confirm with the central authority that the switches are set appropriately for the train's intended route. As the train's last car exits a particular rail segment, the train must advise the central authority that it has departed that segment so other trains can be authorized to proceed.

Control the train's speed in accordance with scheduled station stops, rail segment authorizations, local speed limit considerations, and fuel efficiency objectives. Note that a train with hundreds of cars may be more than a mile long. Part of the train may be climbing a hill while another part is on a decline. The tremendous mass of a train requires that braking and acceleration commands be given multiple minutes prior to the intended results. Authorizations to access particular rail segments may be time constrained such that if a train is delayed in reaching a particular track segment, the authorization may be automatically revoked. Therefore, it is important to meet track schedule constraints.

9.7.1 Description of Implementation

For illustrative purposes, this application is divided into multiple independent missions. There are good software engineering reasons to recommend this architecture. When code is partitioned into multiple missions, there is strong separation of concerns between the independent activities. This facilitates independent development and evolution of each component without impacting the behavior or the safety certification evidence associated with other components.

The sample application described in this section is experimental and pedagogical. The source code is not complete. Only those portions required to demonstrate the use of scope annotations in clarifying interactions between the outermost mission and inner-nested missions are shown. Due to space limitations, only excerpts of the completed code are discussed here. The complete source code in its most currently available form can be downloaded from http://scj-jsr302.googlecode.com/hg/oSCJ/tools/checker/examples/railsegment.

Consider, for example, that the communication protocols evolve over time. Perhaps, the initial communication software relies on a 3G cell-phone network as one of multiple redundant data communication channels. As the cell phone network evolves, the communication component may evolve to exploit the improved services of 4G cell-phone networks. Independently, track segment authorization protocols may evolve in order to improve track utilization. Similarly, more sophisticated cipher algorithms may be incorporated to improve security. And new algorithms may be developed to achieve improved fuel efficiency with new motors and new regenerative braking technologies.

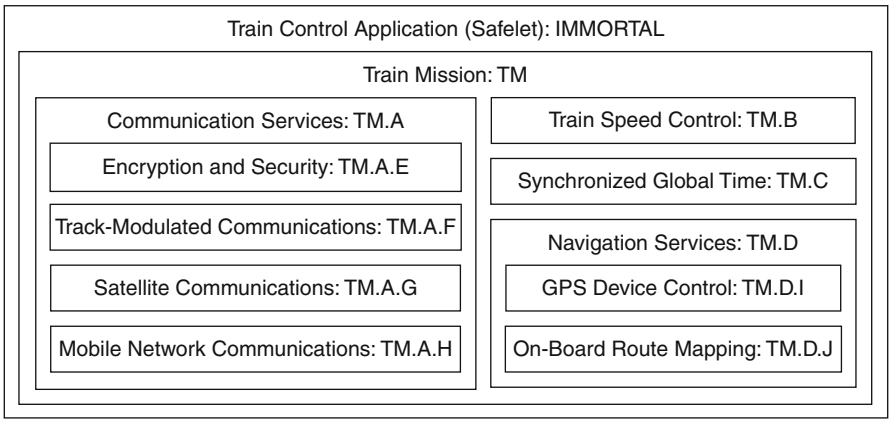


Fig. 9.5 Hierarchical organization of on-board rail segment train access control

The hierarchy of nested missions that implement this application is illustrated in Fig. 9.5. Following the name of each mission is the name of the scope within which the mission object resides. In this particular example, all missions are allocated within their respective mission memories.

The train control application, also illustrated in this figure, is not a mission. The train control application, which implements the `Safelet` interface, resides in immortal memory. Its implementation is shown below.

```
package railsegment;

import static javax.safetycritical.annotate.Level.LEVEL_2;
import static javax.safetycritical.annotate.Level.SUPPORT;
import static javax.safetycritical.annotate.Phase.INITIALIZATION;
import static javax.safetycritical.annotate.Phase.CLEANUP;
import static javax.safetycritical.annotate.Scope.IMMORTAL;

import static javax.safetycritical.annotate.Level.LEVEL_2;

import javax.safetycritical.MissionSequencer;
import javax.safetycritical.Safelet;
import javax.safetycritical.annotate.SCJAllowed;
import javax.safetycritical.annotate.SCJRestricted;
import javax.safetycritical.annotate.Scope;

@Scope(IMMORTAL)
@SCJAllowed(value=LEVEL_2, members=true)
public class TrainControlApp implements Safelet
{
  @SCJAllowed(SUPPORT)
  @SCJRestricted(INITIALIZATION)
  public TrainControlApp() { init(); }

  private void init() {}
```

```
  @SCJAllowed(SUPPORT)
  @SCJRestricted(INITIALIZATION)
  public MissionSequencer getSequencer()
  {
    return new TrainMissionSequencer();
  }
}
```

The `Safelet` interface defines the `getSequencer`, method. The method is declared with the `@SCJAllowed(SUPPORT)` annotation to indicate that the method is not to be called by user code. The method is only called by the vendor-supplied SCJ infrastructure libraries. Note that the constructor for `TrainControlApp` and the `getSequencer` method are both annotated `@SCJRestricted(INITIALIZATION)`. When combined with the `SCJAllowed(SUPPORT)` annotation, this indicates that the infrastructure code only invokes these methods during mission initialization.

The `@Scope(IMMORTAL)` annotation on the `TrainControlApp` class definition denotes that all instances of `TrainControlApp` reside in `ImmortalMemory`. This is enforced by the annotation checker.

The `TrainMissionSequencer` object instantiated by the `getSequencer` method oversees creation of a `TrainMainMission` object within an inner-nested `MissionMemory` area. The code for `TrainMissionSequencer` is provided below.

```
package railsegment;

import static javax.safetycritical.annotate.Level.SUPPORT;
import static javax.safetycritical.annotate.Level.LEVEL_2;
import static javax.safetycritical.annotate.Phase.INITIALIZATION;
import static javax.safetycritical.annotate.Scope.IMMORTAL;

import javax.realtime.PriorityParameters;
import javax.safetycritical.Mission;
import javax.safetycritical.MissionSequencer;
import javax.safetycritical.StorageParameters;
import javax.safetycritical.annotate.DefineScope;
import javax.safetycritical.annotate.RunsIn;
import javax.safetycritical.annotate.SCJAllowed;
import javax.safetycritical.annotate.SCJRestricted;
import javax.safetycritical.annotate.Scope;

@DefineScope(name="TM", parent=IMMORTAL)
@SCJAllowed(value=LEVEL_2, members=true)
@Scope(IMMORTAL)
public class TrainMissionSequencer extends MissionSequencer
{
  private static final int SequencerPriority = 20;

  private boolean returned_mission;

  @SCJRestricted(INITIALIZATION)
  public TrainMissionSequencer()
```

```
  {
    super(new PriorityParameters(SequencerPriority),
          new StorageParameters(TrainMission.BackingStoreRequirements,
                                storageArgs(), 0, 0));
    returned_mission = false;
  }

  private static long[] storageArgs()
  {
    long[] storage_args = {TrainMission.StackRequirements};
    return storage_args;
  }

  @Override
  @RunsIn("TM")
  @SCJAllowed(SUPPORT)
  public Mission getNextMission()
  {
    if (returned_mission) { return null; }
    else
    {
      returned_mission = true;
      return new TrainMission();
    }
  }
}
```

The constructor for `TrainMissionSequencer` invokes its superclass constructor, passing as arguments `PriorityParameters` and `StorageParameters` objects. A `MissionSequencer` is a managed schedulable object. Its corresponding thread will run at the specified priority, using the memory resources described by the `StorageParameters` argument. A schedulability analysis of the complete system workload must account for the work performed by this thread at the specified priority. The activities of this thread include the following:

1. Invoke the `getNextMission` method of the `TrainMissionSequencer` instance.
2. Invoke the `initialize` method of the `Mission` object returned from `getNextMission`.
3. Block itself waiting for a signal indicating that the spawned `Mission` has terminated. This signal will arrive if some other schedulable invokes the `Mission`'s `requestTermination` or `requestSequenceTermination` methods, or this `MissionSequencer`'s `requestSequenceTermination` method, or if all the `Mission`'s schedulables terminate.
4. In the case that this `MissionSequencer`'s `requestSequenceTermination` method was invoked, or the running `Mission`'s `requestTermination` or `requestSequenceTermination` methods were invoked, this thread may perform a small amount of implementation-defined work in order to initiate the `Mission`'s termination.
5. After all of the managed schedulables associated with the running `Mission` terminate their execution, this thread invokes the `Mission`'s `cleanUp` method.

6. Then control returns to step 1.

The second argument to the `StorageParameters` constructor is an array of long integers. How this array is used by a compliant SCJ product is implementation defined. This particular example assumes an implementation that expects a one-element array, with that single element representing the total amount of memory required to represent the runtime stack of the associated thread. This code assumes that the underlying virtual machine uses a single runtime stack which provides memory for each method's so-called "Java stack", for the backing stores of all `PrivateMemory` areas instantiated by this thread, and for the inner-nested stack memories corresponding to all managed schedulable objects that are associated with any inner-nested mission objects. If a particular managed schedulable is a mission sequencer, its stack memory will be subdivided to represent the stack memories for each of the managed schedulable objects that comprise the mission objects associated with the mission sequencer. Thus, the stack memory reservation for a mission sequencer must be large enough to represent both the needs of the mission sequencer thread and the combined needs of all managed schedulable threads spawned by the mission sequencer's spawned missions.

This particular sequencer only returns one instance of `TrainMission`. The storage requirements for the `TrainMission` are provided in the form of symbolic constants associated with the `TrainMission` class. It is beyond the scope of the discussion of this example to explain how the values for these symbolic constants are calculated. These values, and the mechanisms to reliably compute them, are generally implementation defined.

Note that the code for `TrainMissionSequencer` includes a `DefineScope` annotation which introduces the scope named "`TM`" to be nested within `IMMORTAL`. The `getNextMission` method is annotated to run in the "`TM`" scope. The SCJ infrastructure arranges for the `getNextMission` method to run in a new scope. The annotation checker enforces that the name of this scope is consistently known by the "`TM`" scope name. Thus, the `TrainMission` object allocated within `getNextMission` resides within the scope named "`TM`". Note that the declaration of the `TrainMission` class below is annotated to require that all instances of `TrainMission` reside in the "`TM`" scope.

The `TrainMission`'s `initialize` method is responsible for allocating all objects that are to be shared between the various internally nested missions and starting up each of the these inner-nested missions. In particular, shared objects enable the `TrainControl` mission to reliably exchange data with a centralized (train traffic control) dispatcher by communicating with a network communications layer; to obtain the current globally synchronized time, accurate to within a known drift tolerance; and to obtain navigation information such as rail segment boundaries, track switch locations, and track speed limit restrictions. The implementation of `TrainMission` is shown below.

```
package railsegment;

import static javax.safetycritical.annotate.Level.SUPPORT;
import static javax.safetycritical.annotate.Level.LEVEL_2;
import static javax.safetycritical.annotate.Phase.INITIALIZATION;
import static javax.safetycritical.annotate.Scope.CALLER;
import static javax.safetycritical.annotate.Scope.IMMORTAL;

import javax.safetycritical.Mission;
import javax.safetycritical.annotate.DefineScope;
import javax.safetycritical.annotate.RunsIn;
import javax.safetycritical.annotate.SCJAllowed;
import javax.safetycritical.annotate.SCJRestricted;
import javax.safetycritical.annotate.Scope;

import railsegment.clock.SynchronizedTime;
import railsegment.clock.TrainClock;


@Scope("TM")
@SCJAllowed(value=LEVEL_2, members=true)
public class TrainMission extends Mission
 {
  public final static long BackingStoreRequirements = 1000;
  public final static long StackRequirements = 16000;

  private final static long MissionMemorySize = 10000;

  public final int CONTROL_PRIORITY = 8;

  public final int TIMES_TICK_PRIORITY = 22;
  public final int TIMES_COORDINATION_PRIORITY = 18;

  // The maximum of TIMES_TICK_PRIORITY and
  // TIMES_COORDINATION_PRIORITY
  public final int TIMES_CEILING = TIMES_TICK_PRIORITY;

  public final int NAVS_GPS_PRIORITY = 20;
  public final int NAVS_SERVER_PRIORITY = CONTROL_PRIORITY;

  // The maximum of NAVS_GPS_PRIORITY and NAVS_SERVER_PRIORITY
  public final int NAVS_CEILING = NAVS_SERVER_PRIORITY;

  public final int COMMS_DRIVER_PRIORITY = 32;
  public final int COMMS_CONTROL_SERVER_PRIORITY = CONTROL_PRIORITY;
  public final int COMMS_TIMES_SERVER_PRIORITY =
                        TIMES_COORDINATION_PRIORITY;

  public final int COMMS_CONTROL_CEILING =
                        COMMS_CONTROL_SERVER_PRIORITY;
  public final int COMMS_TIMES_CEILING = COMMS_TIMES_SERVER_PRIORITY;

  private CommunicationsQueue comms_control_data;
```

```
  private CommunicationsQueue comms_times_data;
  private NavigationInfo navs_data;

  private SynchronizedTime times_data;

  private CommunicationServiceSequencer commsq;
  private TimeServiceSequencer timesq;
  private NavigationServiceSequencer navsq;
  private TrainControlSequencer controlsq;

  @SCJRestricted(INITIALIZATION)
  public TrainMission() { /* nothing much happens here */ }

  @Override
  @SCJAllowed
  public final long missionMemorySize()
  {
    // must be large enough to represent the four Schedulables
    // instantiated by the initialize() method
    return MissionMemorySize;
  }

  @Override
  @SCJRestricted(INITIALIZATION)
  @SCJAllowed(SUPPORT)
  public void initialize()
  {
    // It all happens here instead
    final int NUM_BUFFERS = 8;
    final int BUFFER_LENGTH = 1024;

    // CommunicationsQueue provides buffers for communication between
    // the COMMS_CONTROL_SERVER thread and the CONTROL thread
    comms_control_data = new CommunicationsQueue(NUM_BUFFERS,
                                                 BUFFER_LENGTH,
                                                COMMS_CONTROL_CEILING);
    comms_control_data.initialize();

    // A separate buffer provides communication between the
    // COMMS_TIME_SERVER thread and the implementation of globally
    // synchronized time services.
    comms_times_data = new CommunicationsQueue(NUM_BUFFERS,
                                                BUFFER_LENGTH,
                                                COMMS_TIMES_CEILING);
    comms_control_data.initialize();

    times_data = new SynchronizedTime(TIMES_CEILING);
    TrainClock train_clock = times_data.initialize();

    navs_data = new NavigationInfo(train_clock, NAVS_CEILING);
    navs_data.initialize();

    commsq = new CommunicationServiceSequencer(
```

```
    comms_control_data, comms_times_data,
                COMMS_DRIVER_PRIORITY,
                COMMS_CONTROL_SERVER_PRIORITY,
                COMMS_TIMES_SERVER_PRIORITY);

    navsq = new NavigationServiceSequencer(train_clock, navs_data,
                                           NAVS_GPS_PRIORITY,
                                           NAVS_SERVER_PRIORITY);

    timesq = new TimeServiceSequencer(comms_times_data,
                                      times_data,
                                      train_clock,
                                      TIMES_TICK_PRIORITY,
                                      TIMES_COORDINATION_PRIORITY);

    controlsq = new TrainControlSequencer(comms_control_data,
                                          navs_data,
                                          train_clock,
                                          CONTROL_PRIORITY);

    commsq.register();
    timesq.register();
    navsq.register();
    controlsq.register();
  }

  // no need to override the default implementation of cleanup,
  // which does nothing.

  @Override
  @RunsIn(CALLER)
  @SCJAllowed(LEVEL_2)
  public void requestTermination()
  {
    commsq.requestSequenceTermination();
    timesq.requestSequenceTermination();
    navsq.requestSequenceTermination();
    controlsq.requestSequenceTermination();

    // Unblock server threads that may be waiting for communication
    // with client threads and mark shared buffers so they do not
    // accept additional end-user requests..
    comms_control_data.shutdown();
    comms_times_data.shutdown();
    times_data.shutdown();
    navs_data.shutdown();

    super.requestTermination();
  }
}
```

In order to manage complexity and separate concerns between independent functional components, this `TrainMission` is comprised of multiple inner-nested missions. In the implementation of the `TrainMission` class shown above, each

mission is represented by the mission sequencer that controls it. By dedicating a distinct mission sequencer to each object, each mission can be allocated within its own `MissionMemory` area. The list below discusses important constraints on the missions that comprise this application. Note that the `TrainMission` needs to consider these global issues because it needs to resolve resource contention and enable communication and coordination between missions by providing shared buffers, setting aside appropriate backing store reservations for each sub-mission, and establishing appropriate priorities for the managed schedulable activities carried out within each of its inner-nested missions.

1. The `TrainControlSequencer` object spawns the `TrainControl` mission, which is responsible for controlling the speed of the train under normal operating conditions.

   Because of the huge mass of a typical train, adjustments to speed are very gradual. The `TrainControl` mission recomputes desired speed and issues appropriate engine and braking controls once per second. This activity runs at priority 8.

   The `TrainControl` mission depends on the `NavigationService` mission to obtain current position, rail map information including the boundaries of each independent rail segment, speed limits along the rail route, locations of upcoming track switches; depends on the `CommunicationService` mission to report current location to central dispatch and to obtain authorizations to access particular rail segments at particular times; and on the also depends on the user-defined clock which is implemented by the `clock.TimeServices` mission. This clock represents a globally synchronized time which is shared with all other trains and with central and local dispatching and track switching operations.

   Communication with the `NavigationService` mission is provided by the `navs_data` object. Communication with the `CommunicationService` mission is provided by the `comms_control_data` object.

   The `TrainControl` mission does not interact directly with the `clock.TimeServices` mission. Rather, it makes use of the `train_clock` object, which provides an abstract representation of globally synchronized time as implemented by the `clock.TimeServices` mission.
2. The `Time-Service-Sequencer` mission sequencer spawns the `clock.Time-Services` mission. The implementation of globally synchronized time requires reliable and timely network communications with other nodes in the network. The algorithms used to implement global time synchronization are assumed to run with a 2 ms period. This activity, running at priority 18, is known as time coordination. The tick for globally synchronized time advances every 250 ms under the control of a timer tick thread, running at priority 22.

   The coordination activity of the `clock.TimeServices` mission uses services provided by the `CommunicationService` mission to exchange information with other nodes in the distributed network. The `comms_times_data` object passed as an argument to the `TimeServiceSequencer` constructor provides a buffer

for passing information to and from the `CommunicationService` mission. The `times_data` object enables communication between the `clock.TimeServices` mission and the implementation of the `train_clock` object.

3. The `CommunicationService` mission runs under the control of the `CommunicationServiceSequencer` mission sequencer. Activities within the `CommunicationService` mission run at three distinct priorities. Device drivers for the various communications devices run as interrupt handlers, with expected response-time deadlines of 50 ms. These run at interrupt priority level 32.

   Additionally, the `CommunicationService` mission performs communication services on behalf of the `TrainControl` and `clock.TimeServices` missions. Since the threads running in these companion missions are not able to access the data structures of this mission, this mission provides server threads to perform the relevant actions on behalf of the companion missions. Since requests from the `TrainControl` mission have a lower priority than requests from the `clock.TimeServices` mission, the `CommunicationService` mission provides a different server thread for each. The `TrainControl` server thread runs at priority 8. The `clock.TimeServices` server thread runs at priority 18. The priorities of these server threads are identical to the priorities of the threads that they serve.

   When the `TrainControl` mission needs the `CommunicationService` mission to perform a service on its behalf, its control thread invokes a synchronized method of the `comms_control_data` object. This encodes the service request within the object's state information, notifies the thread that is responsible for servicing this request within the `CommunicationService` mission, and blocks the control thread until the requested service has been performed. The service thread stores an encoding of the operation's results into the `comms_control_data` object, notifies the blocked control thread, and then blocks itself to await another service request upon completing the assigned activity. In the rate monotonic analysis of the complete system workload, it is possible to treat the pairing between the `TrainControl` mission's control thread and the `CommunicationService` mission's server thread as a single thread running at priority 8 because only one of these two threads is running at any given time. Treating the pairing of threads as a single thread simplifies the rate monotonic analysis. This is a common pattern in dividing efforts between sibling missions that collaborate with one another, each maintaining data that cannot be seen by the other.

4. The `NavigationService` maintains map information and interfaces directly with a global positioning system to monitor the train's current location and its speed. It calculates the train's speed by comparing recent position readings, with each position reading timestamped using the globally synchronized time clock. The `NavigationService` mission provides services to the `TrainControl` mission's control thread, including the track segment that corresponds to the train's current position, the speed limit for the current track segment, and the number of feet to the beginning of the next track segment.

9.8 Conclusion

The SCJ's notion of a mission is an important tool for separating concerns between independent software development activities. Missions impose structure on the spawning of threads, the termination of spawned threads, and the sharing of information between multiple threads. This structure simplifies the runtime environment and enables formal reasoning about resource sharing. An important benefit of hierarchical decomposition into multiple missions is the ability to eliminate fragmentation of backing store memory.

SCJ is more restrictive and its runtime environment much simpler than the full RTSJ. Besides simplifying analysis of application software, the simpler runtime environment offers the potential of improved performance. Further experimentation is required with commercial implementations of the SCJ specification in order to investigate whether it is sufficiently expressive to support common safety-critical programming requirements and to compare its memory consumption and CPU efficiency with full RTSJ implementations.

Though the latest certification standards for avionics provide guidance for the use of garbage collection in such systems, SCJ will provide an important stepping stone for Java technology in safety-critical systems.

Chapter 10
Memory Safety for Safety Critical Java

Daniel Tang, Ales Plsek, and Jan Vitek

Abstract Memory is a key resource in computer systems. Safety-critical systems often must operate for long periods of time with limited available memory. Programmers must therefore take great care to use memory sparingly and avoid programming errors. This chapter introduces the memory management API of the Safety Critical Java specification and presents a static technique for ensuring memory safety.

10.1 Introduction

Memory is a key resource in any embedded system. Programmers must carefully apportion storage space to the different tasks that require it and, when necessary, reallocate memory locations that are currently not in use. This is traditionally done by a combination of static allocation at program startup and manual management of object pools. While static allocation is possible in Java, as shown by the Java Card standard [99], it is at odds with object-oriented programming principles and best practices. For this reason, the Real-time Specification for Java (RTSJ) [132] adopted a memory management API that allows dynamic allocation of objects during program execution but gives control to programmers over the time and cost of memory management. In Java, the memory management API is exceedingly simple. The `new` keyword allocates data in the heap and, before running out of memory, a *garbage collection* algorithm is invoked to reclaim objects that are unreferenced. Before reclaiming data, `finalize()` methods are called to clean up external state. This API together with other features in the Java programming language, such as array bounds checking, provides a property referred to as *memory safety*. This property ensures that a Java program will never access a memory location that

D. Tang (✉) • A. Plsek • J. Vitek
Purdue University, West Lafayette, IN, USA
e-mail: dan@alumni.purdue.edu; aplsek@purdue.edu

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_10,

it has not been given access to and will never access a memory location after that location has been freed. Thus, common software faults such as dangling pointers and buffers overflow are guaranteed to never occur in Java programs. The RTSJ eschews garbage collection in favor of a more complex API based on the notion of scoped memory areas, regions of memory which are not subject to garbage collection and which provide a pool of memory that can be used to allocate objects. A scoped memory area is active while one or more threads are evaluating calls to the memory area's `enter()` method. Objects allocated with `new` by those threads remain alive until all threads return from the `enter()` call. At this point in time, `finalize()` methods can be called and memory used by the data can be reclaimed. A hierarchy of scopes can be created at run-time allowing programmers to finely tune the lifetime of their data. This form of memory management is reminiscent of stack allocation in languages like C or C++. But, whereas stack allocation is inherently unsafe, the RTSJ mandates dynamic checks to ensure memory safety. This means that every assignment statement to a reference variable will be checked for safety, and if the assignment could possibly lead to the program following a dangling pointer, an exception will be thrown.

Safety-critical applications must undergo a rigorous validation and certification process. The proposed Safety Critical Java (SCJ) specification[1] is being designed to facilitate the certification of real-time Java applications under safety-critical standards such DO-178B in the US [334]. To ease the task of certification, SCJ reduces the complexity of the RTSJ memory management interface by removing some methods and classes and forbidding some usage patterns that may lead to errors. Furthermore, SCJ provides an optional set of Java metadata annotations which can be used to check memory safety at compile-time. SCJ program must perform the same dynamic checks as with the RTSJ, but a fully annotated program is guaranteed to never throw a memory access exception, thus making it possible to optimize those checks away. It is important to observe that memory safety does not imply the absence of all memory errors such as, for instance, that the program will not run out of memory. This is a separate and orthogonal issue. We expect that other, independently developed, tools will provide static memory usage checks [76].

The RTSJ memory management API has proven rather controversial. Over the years, most vendors have offered real-time garbage collectors as a way to return to the simple Java memory API while bounding pause times. Unfortunately, real-time garbage collection slows down program execution and requires additional resources that are not always available in small devices. The main drawback of RTSJ-style memory management is that any reference assignment must be treated with suspicion as it may lead to an `IllegalAssignmentError` being thrown. Researchers have proposed disciplined uses of the API to reduce the likelihood of error [1,54,63,297]. However, users have continued to view scoped memory as hard to use correctly. SCJ annotations are intended as a way to help programmers structure their program to simplify reasoning about allocation contexts. Memory safety

[1] JCP JSR-302, http://www.jcp.org/en/jsr/detail?id=302.

annotations are added to the Java source code to provide additional information to both developers and the Java Virtual Machine. The SCJ specification also defines metadata annotations that indicate behavioral characteristics of SCJ applications. For example, it is possible to specify that a method performs no allocation or self-suspension. We refer the interested reader to the specification for more details.

This chapter introduces the SCJ memory management API and describes a set of Java metadata annotations that has been proposed as a solution for ensuring static memory safety of real-time Java programs written against the SCJ memory API. We give examples of simple SCJ programming patterns and discuss how to validate them statically.

10.2 Scoped Memory and Safety Critical Java

The memory management API exposed by SCJ is a simplification of that of the RTSJ. A proper introduction to the RTSJ is outside of the scope of this chapter; we refer interested readers to [132, 297]. Explaining the memory management API requires some understanding of the main concepts of SCJ[2]; therefore we start with an overview.

10.2.1 SCJ Overview

An SCJ compliant application consists of one or more *missions*, executed concurrently or in sequence. Every application is represented by an implementation of the `Safelet` class which arranges for running the sequence of `Missions` that comprise the application. A mission consists of a bounded set of event handlers and possibly some RTSJ threads, known collectively as *schedulable objects*. For each mission, a dedicated block of memory is identified as the *mission memory*. Objects created in mission memory persist until the mission is terminated. All classes are loaded into a block of immortal memory when the system starts up. Conforming implementations are not required to support dynamic class loading, so all of the code of a mission is expected to be loaded and initialized at startup. Each mission starts in an initialization mode during which objects may be allocated in mission memory. When a mission's initialization has completed, execution mode is entered. During execution mode, mutable objects residing in mission or immortal memory may be modified as needed. All application processing for a mission occurs in one or more schedulable objects. When a schedulable object is started, its initial memory area is a private memory area that is entered when the schedulable object is released and exited (and cleared) at the end of the release. By default, objects are allocated in this private memory which is not shared with other schedulable objects. A mission can

[2]This chapter is based on SCJ v0.71 from August 2010.

```
@SCJAllowed public interface Safelet {

  @SCJAllowed(SUPPORT) @SCJRestricted(phase=INITIALIZATION)
  MissionSequencer getSequencer()

  @SCJAllowed(SUPPORT) @SCJRestricted(phase=INITIALIZATION)
  void setUp()

  @SCJAllowed(SUPPORT) @SCJRestricted(phase=CLEANUP)
  void tearDown()
}
```

Fig. 10.1 Interface javax.safetycritical.Safelet

be terminated. Once termination is requested, all schedulable objects in the mission are notified to cease operating. Once they have all stopped, the mission can cleanup before it terminates. This provides a clean restart or transition to a new mission.

The complexity of safety-critical software varies. At one end of the spectrum, some applications support a single function with only simple timing constraints. At the other end, there are complex, multi-modal systems. The cost of certification of both the application and the infrastructure is sensitive to their complexity, so enabling the construction of simpler systems is highly desirable. SCJ defines three compliance levels to which both implementations and applications may conform. SCJ refers to them as *Level 0, Level 1*, and *Level 2*. Level 0 refers to the simplest applications and Level 2 refers to the more complex applications. A Level 0 application's programming model is a familiar model often described as a timeline model, a frame-based model, or a cyclic executive model. In this model, the mission can be thought of as a set of computations, each of which is executed periodically in a precise, clock-driven timeline, and processed repetitively throughout the mission. A Level 0 application's schedulable objects consist only of a set of periodic event handlers (PEH). Each event handler has a period, priority, and start time relative to the beginning of a major cycle. A schedule of all PEHs is constructed by the application designer. A Level 1 application uses a programming model consisting of a single mission with a set of concurrent computations, each with a priority, running under control of a fixed-priority preemptive scheduler. The computation is performed in a mix of periodic and aperiodic event handlers. A Level 1 application shares objects in mission memory among its schedulable objects, using synchronized methods to maintain the integrity of its shared objects. A Level 2 application starts with a single mission, but may create and execute additional missions concurrently with the initial mission. Computation in a Level 2 mission is performed in a combination of event handlers and RTSJ threads. Each child mission has its own mission sequencer, its own mission memory, and may also create and execute other child missions.

The `Safelet` interface is shown in Fig. 10.1. The `@SCJAllowed` annotation indicates that this interface is available at all compliance levels. `Safelet` methods are invoked by the infrastructure (i.e. the virtual machine) and can not be called from

Fig. 10.2 Class javax.safetycritical.Mission

```
@SCJAllowed
public abstract class Mission {

  @SCJAllowed(SUPPORT)
  protected abstract void initialize()

  @SCJAllowed
  public static Mission getCurrentMission()

  @SCJAllowed
  public final void requestTermination()

  @SCJAllowed(SUPPORT)
  protected void cleanUp()
}
```

user-defined code; such methods are annotated with `@SCJAllowed(SUPPORT)`. The infrastructure invokes in sequence `setUp()` to perform any required initialization actions, followed by `getSequencer()` to return an object which will create the missions that comprise the application. The missions are run in an independent thread while the safelet waits for that thread to terminate its execution. Upon termination, the infrastructure invokes `tearDown()`. The `Mission` class, shown in Fig. 10.2, is `abstract`; it is up to subclasses to implement the `initialize()` method, which performs all initialization operations for the mission. The `initialize()` method is called by the infrastructure after memory has been allocated for the mission. The main responsibility of `initialize()` is to create and register the schedulable objects that will implement the behavior of the mission. The `cleanUp()` method is called by the infrastructure after all schedulable objects associated with this mission have terminated, but before the mission memory is reclaimed. The infrastructure arranges to begin executing the registered schedulable objects associated with a particular `Mission` upon return from the `initialize()` method. The `CyclicExecutive` class, shown in Fig. 10.3, is used at Level 0 to combine the functionality of `Safelet` and `Mission`. Finally, we have the abstract class `PeriodicEventHandler` in Fig. 10.4, which extends `ManagedEventHandler`, which implements the `Schedulable` interface. Instances of this class are created in the `initialize()` method of `Mission`; the method `register()` is called to add them to the current mission. The infrastructure will call the `handleAsyncEvent()` method periodically following the `PeriodicParameters` passed in as argument. Users must subclass `PeriodicEventHandler` to provide an implementation for `handleAsyncEvent()`.

```
@SCJAllowed
public class CyclicExecutive
       extends Mission implements Safelet {

  @SCJAllowed
  public CyclicExecutive(PriorityParameters p, StorageParameters s)
}
```

Fig. 10.3 Class javax.safetycritical.CyclicExecutive

```
@SCJAllowed
public abstract class PeriodicEventHandler
       extends ManagedEventHandler {

  @SCJAllowed @SCJRestricted(phase=INITIALIZATION)
  public PeriodicEventHandler(PriorityParameters pp,
      PeriodicParameters r, StorageParameters sp)

  @SCJAllowed @SCJRestricted(phase=INITIALIZATION)
  public final void register()

  @SCJAllowed(SUPPORT)
  public abstract void handleAsyncEvent()

  @SCJAllowed(SUPPORT) @SCJRestricted(phase=CLEANUP)
  public void cleanUp()
}
```

Fig. 10.4 Abstract class javax.safetycritical.PeriodicEventHandler

10.2.2 Memory Management Interface

SCJ segments memory into a number of scoped memory areas, or *scopes*. Each scope is represented by an object, an instance of one of the subclasses of `MemoryArea` and a *backing store*, a contiguous block of memory that can be used for allocation. Scopes are related by a *parenting* relation that reflects their order of creation and the lifetime of the objects allocated within them. Child scopes always have a strictly shorter lifetime than their parent scope. A distinguished scope, called *immortal memory*, is the parent of all scopes, represented by a singleton instance of the `ImmortalMemory` class. Each mission has a scope called the *mission memory*, which stores objects that are needed for the whole mission; this is represented by an instance of `MissionMemory` allocated in its own backing store. Each schedulable object has its own *private memory* represented by an instance of `PrivateMemory`. SCJ supports nested missions as well as nested private memories. Private memories are the default allocation context for the logic of the schedulable object to which they belong. As the name suggests, private memories are inaccessible to anything outside of the schedulable object that owns them. This is unlike scopes in the RTSJ, which may be accessed by multiple threads simultaneously.

```
@SCJAllowed
public abstract class MemoryArea
       implements AllocationContext {

  @SCJAllowed
  public void executeInArea(Runnable logic)
         throws InaccessibleAreaException

  @SCJAllowed
  public static MemoryArea getMemoryArea(Object object)

  @SCJAllowed
  public Object newInstance(Class type)
    throws  InstantiationException, InaccessibleAreaException

  @SCJAllowed
  public Object newArray(Class type, int size)

  @SCJAllowed
  public static Object newArrayInArea(Object object,
                                      Class type, int size)

  @SCJAllowed
  public abstract long memoryConsumed()

  @SCJAllowed
  public abstract long size()
}
```

Fig. 10.5 Abstract Class javax.realtime.MemoryArea

```
@SCJAllowed
public final class ImmortalMemory extends MemoryArea {

  @SCJAllowed
  public static ImmortalMemory instance()
}
```

Fig. 10.6 Class javax.realtime.ImmortalMemory

Figure 10.5 shows the subset of the RTSJ `MemoryArea` class which is allowed in SCJ. This class is the root of the memory area class hierarchy and it defines much of the memory management interface. The `executeInArea()` method accepts a `Runnable` and executes it in a memory area represented by the receiver object. The `getMemoryArea()` method returns the scope in which the argument was allocated. The `newInstance()` and `newArray()` methods allocate objects and arrays in the memory area represented by the receiver object. The `newArray-InArea()` allocate arrays in the memory area referenced by the object passed as the first argument. `memoryConsumed()` and `size()` return the number of bytes currently allocated in the backing store and the total size of the backing store of the receiver. Figure 10.6 presents the `ImmortalMemory` class which has a single

```
@SCJAllowed public abstract class ManagedMemory extends LTMemory {

  @SCJAllowed
  public static ManagedMemory getCurrentManagedMemory()

  @SCJAllowed
  public void enterPrivateMemory(long size, Runnable logic)

  @SCJAllowed
  public static boolean allocatedInParent(Object c, Object p)

  @SCJAllowed
  public static boolean allocatedInSame(Object c, Object p)
}
```

Fig. 10.7 Abstract Class javax.safetycritical.ManagedMemory

```
@SCJAllowed public class MissionMemory extends ManagedMemory {

  @SCJAllowed
  public synchronized Object getPortal()

  @SCJAllowed
  public synchronized void setPortal(Object value)
}
```

Fig. 10.8 Class javax.safetycritical.MissionMemory

static method `instance()` to return the singleton instance of the class. The size of the immortal memory is given at startup as an argument to the JVM. Figure 10.7 lists the methods of the `ManagedMemory` class which extends `LTMemory` (it is an RTSJ class that extends `MemoryArea` and guarantees linear time allocation; not shown here). The class introduces `getCurrentManagedMemory()` to return the instance of `ManagedMemory` used to allocate objects at the point of call, and `enterPrivateMemory()` to create a nested `PrivateMemory` area and execute the `run()` method of the `Runnable` argument in that memory area. Figure 10.8 lists the methods of `MissionMemory` which extends `ManagedMemory`. The class has two methods, `getPortal()` and `setPortal()`, that provide means of passing information between `Schedulable` objects in a memory area, thus implementing a concept of a shared object. Figure 10.9 shows `PrivateMemory`, the other subclass of `ManagedMemory`, which introduces no new methods. Neither of these classes can be created directly from user-defined code. Figure 10.10 shows the `StorageParameters` interface which is used to reserve memory for the backing stores. It is passed as a parameter to the constructor of mission sequencers and schedulable objects. The class can be used to tune vendor specific features such as the native and Java call stack sizes, and the number of bytes dedicated to message associated with exceptions and backtraces.

```
@SCJAllowed public class PrivateMemory extends ManagedMemory {}
```

Fig. 10.9 Class javax.safetycritical.PrivateMemory

```
@SCJAllowed public class StorageParameters {

  @SCJAllowed
  public StorageParameters(long totalBackingStore, long nativeStackSize,
      long javaStackSize)

  @SCJAllowed
  public StorageParameters(long totalBackingStore, long nativeStackSize,
      long javaStackSize, int messageLength, int stackTraceLength)
}
```

Fig. 10.10 Class javax.safetycritical.StorageParameter

10.2.3 Semantics of Memory Management API

Any statement of a SCJ application has an *allocation context* which can be either one of immortal, mission or private memory. Immortal memory is used for allocation of classes and static variables, and is the allocation context of the `Safelet.setup()` method. Mission memory is the allocation context of the `Mission.initialize()` method. Private memory is the allocation context of the `handleAsyncEvent()` method of the `PeriodicEventHandler` class. By extension, we also refer to the scope in which an object was allocated as the allocation context of this object.

The parenting relation is a transitive, non-reflexive relation between scopes that is defined such that `ImmortalMemory` is the parent of any `MissionMemory` of a top-level `Mission` (nested missions are parented to their enclosing `MissionMemory`). `MissionMemory` is the parent of any `PrivateMemory` associated to a schedulable object registered to the corresponding `Mission`. A nested `PrivateMemory` is parented to its directly enclosing `PrivateMemory`. In SCJ, a reference assignment `x.f = y` is valid if `ax` is the scope of `x` (obtained by `MemoryArea.getMemoryArea(x)`) and `ay` is the scope of `y`, and either `ax == ay` or `ay` is a parent of `ax`. If a SCJ program attempts to perform an assignment that is not valid, an `IllegalAssignmentError` will be thrown. Assignments to local variables and parameters, as well as assignments to primitive types, are always valid. There are no checks for reference reads.

The SCJ memory API is designed to ensure that a `PrivateMemory` is accessible to only one schedulable object. This is not the case for `MissionMemory` and `ImmortalMemory`, both of which can be accessed by multiple threads. Some of the complexity of the RTSJ memory management goes away because creation of scopes is not under programmer control and the RTSJ's `enter()` can not be used by user code. In particular, the RTSJ has `ScopeCycleException` to cover the case where multiple threads enter the same scope from different allocation contexts. This can not occur by construction of the SCJ API.

Objects allocated within a `MissionMemory` are reclaimed when the mission is terminated, all registered schedulable objects are inactive and the mission's `cleanUp()` has returned. Objects allocated within a `PrivateMemory` are reclaimed with the `enterPrivateMemory()` returns, or, if this is a top-level `PrivateMemory`, when the `handleAsyncEvent()` method returns.

The method `executeInArea(runnable)` can be used to temporarily change the allocation context to another scope and execute the `run()` method of the `runnable` argument in that context. The target of the call can be an instance of `ImmortalMemory`, `MissionMemory` or `PrivateMemory` and must be a parent. However, after invoking `executeInArea()`, it is not possible to use `enterPrivateMemory()`.

The backing store for a scoped memory is taken from the backing store reservation of its schedulable object. The backing store is managed via reservations for use by schedulable objects, where the initial schedulable object partitions portions of its backing store reservation to pass on to schedulable objects created in its thread of control. Backing store size reservation is managed via `StorageParameters` objects.

The `Mission.initialize()` method call can `enterPrivateMemory()` on the mission memory and use the backing store that is reserved for its schedulable objects for temporary allocation. Once `initialize()` returns, the backing stores for all registered schedulable objects are created. Calling `enterPrivateMemory()` if the receiver is not the current allocation context will result in an `IllegalStateException`. Calling `enterPrivateMemory()` on a `MissionMemory` object after the corresponding mission's `initialize()` method has returned will result in an `IllegalStateException`. Calling `newInstance()`, `newArray()` or `executeInArea()` on a scope that is not on the current schedulable object's call stack is a programming error that results in an exception. Every schedulable object has `ImmortalMemory` and, once the mission is running, `MissionMemory` on its call stack by construction.

Figure 10.11 provides an illustration of the memory representation of a SCJ application. The figure shows a mission consisting of two schedulable objects, one with a nested private memory area. The figure also shows the arrangement of the backing stores and highlights that schedulable objects use the backing store of their mission, and nested private memories use the backing store of their schedulable object. The parenting relation is depicted graphically as well.

10.2.4 Differences with the RTSJ

There are a number of simplifications in the SCJ memory API that lead to a reduction in possible errors. One of the most significant differences is that the RTSJ has a garbage collected heap. The RTSJ further makes the difference between `RealtimeThreads` and `NoHeapRealtimeThreads` (NHRTs). NHRTs are

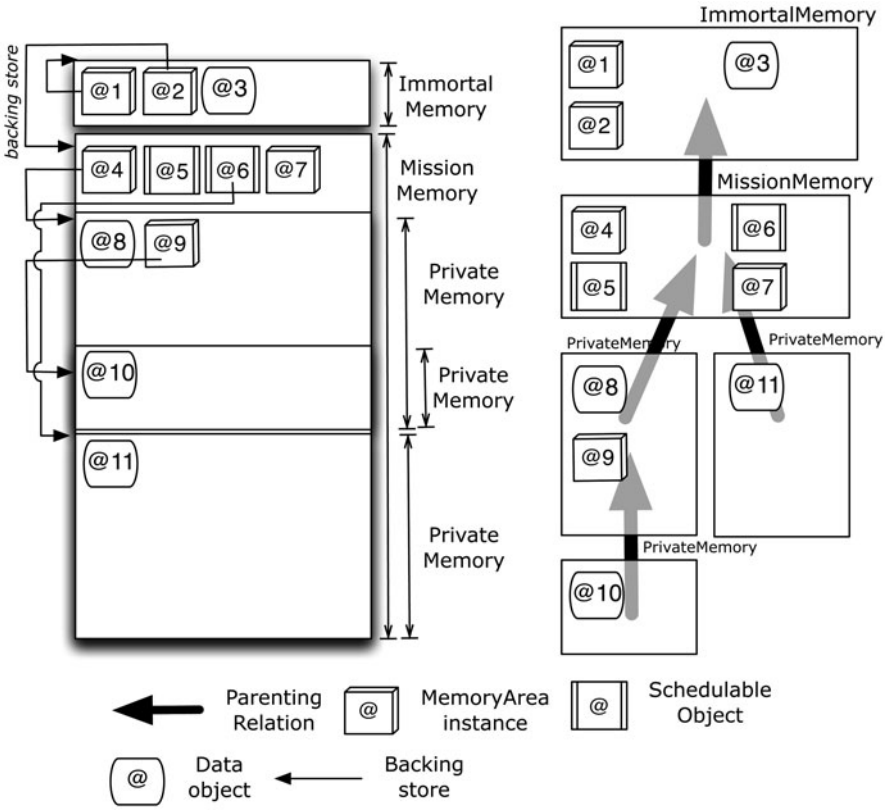


Fig. 10.11 Memory model. The figure on the *left* depicts the layout in memory of the objects and the figure on the *right* is the logical representation of the scopes with their parenting relation

designed to not have to block for the garbage collector. In SCJ, all threads are NHRTs and there is no heap. We list some of the benefits of the design:

- In the RTSJ, any field access `x.f` or method invocation `x.m()` must be checked for validity. An attempt to follow a heap reference from a NHRT will result in an exception. This can not occur in SCJ as there is no heap.
- In the RTSJ, all scopes can be accessed by multiple threads. Finalizers must be run before a scope can be reused. If the thread running the finalizers is a NHRT, it may throw an exception if the finalizers manipulate heap data; if it is not, then finalization may have to wait for the garbage collection to complete, preventing other NHRTs from using the scope. This can not occur in SCJ: finalization is not supported, the last thread to leave a scope is always well defined, and there is no garbage collection.
- In the RTSJ, the parenting relation is established as a result of calling `enter()` on an unparented scope. Multiple threads may call `enter()` on the same scope

from different allocation contexts, which is illegal. To prevent this, an exception is thrown if a scope that already has a parent is entered from a different allocation context. In SCJ, this can not occur for `MissionMemory` instances as they are created and entered by the infrastructure. For `PrivateMemory` instances, a runtime check is needed.

10.3 Programming with SCJ

We now illustrate how the SCJ memory API can be used for common tasks. We start with simple design patterns for pre-allocated memory and then show that per-release memory is easy to use; then we show how to fine tune the lifetime of data by choosing the right allocation context for every object. Finally, we illustrate on concrete examples errors caused by an incorrect use of the SCJ memory API.

10.3.1 Static Allocation

Very conservative safety-critical systems may require pre-allocation of all of the data in advance. In SCJ, this can be achieved by allocating all data in static initializers and setting the backing stores of the mission to the minimal amount of memory (enough to hold the infrastructure-created objects like the `Mission` instance). Assume an application needs to measure release jitter in a periodic task for 1,000 iterations. A first step towards this goal would be to have an event handler record the time of every periodic release. Figure 10.12 shows such a class, `MyPEH`, which has a `handleAsyncEvent()` method that records timestamps in a statically allocated array data structure. All data manipulated by this class is allocated in static variables and no allocation will be performed in the `handleAsyncEvent()` method. To ensure prompt termination in case a developer was to accidentally allocate at run time, the `StorageParameters` request only 50 bytes as the size of the `PrivateMemory` used for `MyPEH`.

Figures 10.12 and 10.13, which add set up code necessary to create and start the event handler, constitute a complete Level 0 application. The class `MyApp` extends the Level 0 `CyclicExecutive` class, which merges the functionality of `Mission` and `Safelet`. Most supporting data is created in immortal memory, but the SCJ API mandates that the event handler be created in the `initialize()` method as the event handler's `register()` method (called during construction) must add the newly created handler to a mission. The `getSchedule()` method must also allocate because its argument holds a reference to the `MyPEH` instance, which is stored in mission memory. In the implementation used for this example, the size of mission memory can be limited to 500 bytes.

A variant of static allocation is to have all the data needed by each mission reside in `MissionMemory`. This has the advantage that once a `Mission` is reclaimed,

```
@SCJAllowed(members=true) class MyPEH extends PeriodicEventHandler {

  static int pos;
  static long[] times = new long[1000];
  static StorageParameters sp = new StorageParameters(
                                50L,1000L,1000L);

  MyPEH() { super(null, null, sp); }

  void handleAsyncEvent() {
    times[pos++] = Clock.getRealtimeClock().
                         getTime().getMilliseconds();
    if (pos == 1000) Mission.getCurrentMission().
    requestTermination();
  }
}
```

Fig. 10.12 Static allocation example

```
@SCJAllowed(members=true) class MyApp extends CyclicExecutive {

  static PriorityParameters p = new PriorityParameters(18);
  static StorageParameters s = new StorageParameters(
                                    500L, 1000L, 1000L);
  static RelativeTime t = new RelativeTime(5,0);

  MyApp() { super(p,s); }

  CyclicSchedule getSchedule(PeriodicEventHandler[] handlers) {
    return  new CyclicSchedule(
        new CyclicSchedule.Frame[]{
            new CyclicSchedule.Frame(t, handlers)});
  }

  void initialize()  { new MyPEH().register(); }
}
```

Fig. 10.13 Static allocation example, setup code

the space used for its objects can be safely reused for the next mission. Figure 10.14 shows that little changes are needed to support static allocation in mission memory. Instead of using static fields, the data manipulated by the `MyPEH2` event handler resides in mission memory. To initialize the data, the allocation context of the constructor of `MyPEH2` is mission memory and therefore the array of long can be created in the constructor. For simplicity, we can reuse the set up code of Fig. 10.13 and extend it with a new `initialize()` method that instantiates the `MyPEH2` event handler.

```
@SCJAllowed(members=true) class MyPEH2 extends PeriodicEventHandler {

  MyPEH2() {
    super(null, null, new StorageParameters(1000L, 1000L, 1000L));
    times = new long[1000];
  }

  int pos;
  long times[];
  void handleAsyncEvent() {
    times[pos++] = Clock.getRealtimeClock().
                          getTime().getMilliseconds();
    if (pos == 1000) Mission.getCurrentMission().
    requestSequenceTermination();
  }
}
```

Fig. 10.14 Mission memory allocation example

Fig. 10.15 Per-release allocation example

```
void handleAsyncEvent() {
   long times[] = new long[1000];
}
```

10.3.2 Per-release Allocation

Applications often have need of temporary storage to compute a result. In SCJ, any data allocated in the `handleAsyncEvent()` method of any event handler is temporary by default, lasting only until `handleAsyncEvent()` returns. Thus, no special programming idiom is required. Figure 10.15 allocates an array for the duration of the release. The size of the scope is specified as an argument when the event handler is created. The application can allocate freely as long as allocated data does not overflow the backing store.

It is possible to have finer control over the lifetime of temporary data. Figure 10.16 continues the previous example by creating a copy of the `times` array. That copy is allocated in a nested scope which is reclaimed when the `run()` method returns.

The SCJ memory API further provides support for flexible allocation patterns that allow users to implement different allocation strategies. The `newInstance()` creates an object in any scope that has been entered by the current thread, while `executeInArea()` allows the user to change the allocation context to a previously entered scope. Figure 10.17 illustrates the two ways for programmers to allocate longer-lived objects. The allocation context of the `handleAsyncEvent()` method is an instance of `PrivateMemory`, but the `PeriodicEventHandler` (referenced by `this` variable) is allocated in `MissionMemory`. The code snippet shows two ways to allocate an object in `MissionMemory`. The first way is to obtain a reference to `MissionMemory` from `this` and to call `newInstance()`.

```
long median;

void handleAsyncEvent() {
  final long times[] = new long[1000];
  Runnable r = new SCJRunnable(){
    void run() {
      long[] copy = new long[1000];
      for(int i=0;i<1000;i++)  copy[i]=times[i];
      Arrays.sort(copy);
      median = copy[500];
    }
  };
  ManagedMemory m = ManagedMemory.getCurrentManagedMemory();
  m.enterPrivateMemory(8000,r);
}
```

Fig. 10.16 Per-release allocation with nested scopes

```
@SCJAllowed(members=true) class MyPEH4 extends PeriodicEventHandler {

  Tick tock;
  void handleAsyncEvent() {

    ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(this);
    Tick time = (Tick) m.newInstance(Tick.class);

    m.executeInArea(new SCJRunnable() { void run() {
        MyPEH4.this.tock = new Tick();}});
  }
}
```

Fig. 10.17 Allocation context change in SCJ

The second way is call `executeInArea()` with a `SCJRunnable` argument. In both cases an object will be allocated in `MissionMemory`. The assignment to `this.tock` is valid only because we have changed allocation context.

10.3.3 Memory Management Errors

This section illustrates the pitfalls of the SCJ memory management API with concrete examples. SCJ has two exceptions specific to memory related errors, `IllegalAssignmentError` and `IllegalStateException`. An assignment error can be thrown by any reference assignment. The challenge for developers is that inspection of the code of a single method is often insufficient to rule out errors. Consider the following code fragment.

```
class Err extends PeriodicEventHandler {
  List a = new List();
  ...
  void handleAsyncEvent() {
    List b = new List();
    setTail(b, a);
    setTail(a, b);
    this.a = b;
  }
}
```

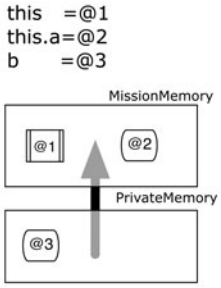


Fig. 10.18 IllegalAssignmentError

```
void setTail(List from, List to) {   from.tail = to;   }
```

As there is no information in the program text as to where the arguments to the method were allocated, it is impossible to rule out the possibility that the assignment will throw an `IllegalAssignmentError`.

Consider the class declaration in Fig. 10.18. The instance of `List` referenced by field `a` is allocated in `MissionMemory`, whereas the instance referenced by variable `b` is allocated in the allocation context of `handleAsyncEvent()`, namely `PrivateMemory`. The first call to `setTail()` is valid as it will set a reference from an object allocated in a child scope (`@3`) to an object allocated in a parent scope (`@2`). The second call will throw an exception as the direction of the reference is reversed. The assignment `this.a=b` is also invalid. The `this` keyword variable refers to the event handler (`@1`) allocated in `MissionMemory`.

An `IllegalStateException` can be thrown if a schedulable object attempts to use a memory area that it has not entered for one of `newInstance()`, `executeInArea()` or `enterPrivateMemory()`. The SCJ specification makes this kind of error unlikely, but still possible. The program of Fig. 10.19 shows the steps needed for the error to occur. Handler `P1` stores a reference to its own `PrivateMemory` (`@1`) into a field of the mission `M1.pri`. Another handler, `P2`, tries to either allocate from `P1`'s memory or enter it. In both cases an exception will be thrown.

10.4 Static Memory Safety with Metadata Annotations

The SCJ specification ensures memory safety through dynamic checks. While all assignments that may lead to a dangling pointer will be caught, they are only caught at runtime and can be hard to detect through testing alone. The problem for verification of SCJ code is that the information needed to check an assignment statement is not explicit in the program text. Consider an assignment statement such as:

```
class M1 extends Mission {
  void initialize() { new P1().register(); new P2().register(); }
  PrivateMemory priv;
}

class P1 extends PeriodicEventHandler {
  void handleAsyncEvent() {
    M1 m1  = (M1) Mission.getCurrentMission();
    m1.priv = (PrivateMemory)
        MemoryArea.getMemoryArea(new int[0]);
  }
}

class P2 extends PeriodicEventHandler {
  void handleAsyncEvent() {
    M1 m1 = (M1) Mission.getCurrentMission();
    m1.priv.newInstance(List.class);
    m1.priv.enterPrivateMemory(500,
        new Runnable(){void run(){}});
  }
}
```

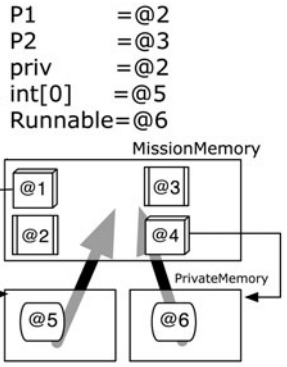
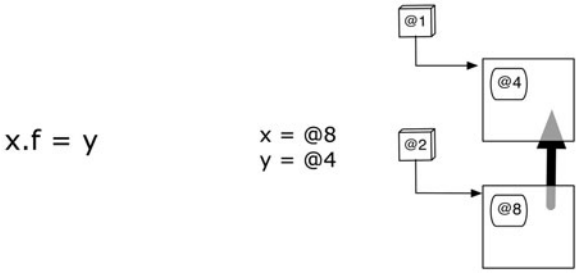
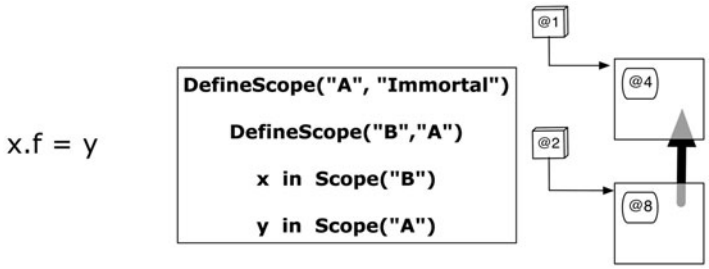


Fig. 10.19 IllegalState exceptions



The steps that the infrastructure must take to ascertain that the assignment is valid are: (1) obtain the scope in which the object referenced by `x` is allocated (the scope `@2` here), (2) obtain the scope in which the object referenced by `y` (scope `@1`) is allocated, and (3) check that `@1` is a parent of `@2`. To do this at compile time, one either needs to perform a whole-program points-to analysis to discover the sets of potential objects referenced by `x` and `y` and their scopes, or ask for more information to be provided by the developer. A checker needs these three pieces of information: the respective scopes of the source and target of each assignment statement and the parenting relation between the scopes. Of course, it would be rather impractical if the developer had to provide memory addresses of scopes, as this would hard-wire the program to exactly that pair of scopes. A better solution is to provide symbolic, compile-time, names for scopes and guarantee through a set of rules that the variables will refer to objects that are allocated in those named scopes. We present an annotation system that lets developer express the relationship

between scopes and specify where the objects referenced by program variables are allocated. Thus, in the above example, the metadata would express the following information:



It would specify that the program has at least two scopes, with symbolic names `A` and `B`, that `B` is a child of `A`, that the object referenced by variable `x` is allocated in scope `B` and that the object referenced by `y` is allocated in `A`. Equipped with this information, a checker can validate the assignment at compile time. The challenge for the design of memory safety annotations is to balance three requirements. The annotations should be *expressive*, letting developers write the code they want naturally. The annotations should be *non-intrusive*, requiring as few annotations as possible. The annotations must be *sound*; an annotated program must not be allowed to experience a memory error. We purposefully chose to emphasize soundness and non-intrusiveness with a very lightweight set of annotations. The core of this system is a set of annotations on classes and methods. These are very lightweight but somewhat restrictive. To recover expressiveness, the system also supports dynamic guards and variable annotations.

The system differentiates between *user* code and *infrastructure* code. User code follows the restrictions outlined in this chapter and is verified by a dedicated checker tool implementing this annotation system. Infrastructure code is verified by the vendor. Infrastructure code includes the `java` and `javax` packages as well as vendor specific libraries. The infrastructure code is assumed to be correct and will not be verified by the checker.

10.4.1 Annotations Overview

The SCJ specification introduces three Java metadata annotations, listed in Table 10.1, that guarantee the absence of memory errors. This guarantee requires that all classes are successfully checked together.

The annotation checker requires the following information. Given a statement, such as `this.x=y.n()`, occurring in the body of some method `m(T y)`, the checker must know in which scope the method will be called (the *current allocation*

Table 10.1 Annotation summary. Default values in bold

Annotation	Where	Arguments	Description
@DefineScope	Any	*Name*	Define a new scope
	Class	*Name*	Instances are in named scope
		CALLER	Can be instantiated anywhere
	Field	*Name*	Object allocated in named scope
		UNKNOWN	Allocated in unknown scope
		THIS	Allocated enclosing class' scope
@Scope	Method	*Name*	Returns object in named scoped
		UNKNOWN	Returns object in unknown scope
		CALLER	Returns object in caller's scope
		THIS	Returns object in receiver's scope
	Variable	*Name*	Object allocated in named scope
		UNKNOWN	Object in an unknown scope
		CALLER	Object in caller's scope
		THIS	Object in receiver's scope
@RunsIn	Method	*Name*	Method runs in named scope
		CALLER	Runs in caller's scope
		THIS	Runs in receiver's scope

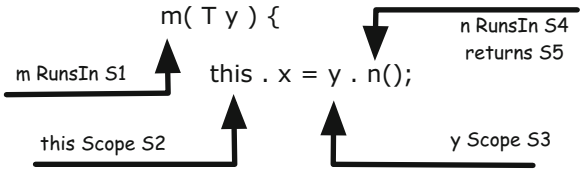


Fig. 10.20 A statement and associated scope information

context), the scope of the receiver (the *allocation context* of the object referenced by the `this` variable), the scope of variable `y`, the scope in which the method `n()` expects to execute, and the scope in which its result will be allocated. The situation is illustrated in Fig. 10.20.

The @DefineScope Annotation. The first step is to define a *static scope tree*. The static scope tree is the compile-time representation of the run-time parenting relation. For this, we introduce the `@DefineScope` annotation with two arguments, the symbolic name of the new scope and of its parent scope. The checker will ensure that the annotations define a well formed tree rooted at `IMMORTAL`, the distinguished parent of all scopes. Scopes are introduced by mission (`MissionMemory`) and schedulable objects (`PrivateMemory`). Thus we require that each declaration of a class that has an associated scope (for instance, subclasses of the `MissionSequencer` class which define missions, and subclasses of the `PeriodicEventHandler` class which hold task logic) must be annotated with a scope definition annotation. Furthermore, nested `PrivateMemory` scopes are created by invocation of the `enterPrivateMemory()` method. As Java does not allow annotation on expressions, we require that the argument of the method, an instance of a subclass of `SCJRunnable` be annotated with a scope definition.

The @Scope Annotation. The key requirement for being able to verify a program is to have a compile-time mapping of every object reference to some node in the static scope tree. With that information, verification is simply a matter of checking that the right hand side of any assignment is mapped to a parent (or same) scope of the target object. This mapping is established by the `@Scope` annotation which takes a scope name as argument. Scope annotations can be attached to class declarations to constrain the scope in which all instances of that class are allocated. Annotating a field, local or argument declaration constrains the object referenced by that field to be in a particular scope. Lastly, annotating a method declaration constrains the value returned by that method.

Scope CALLER, THIS, and UNKNOWN. Since a general form of parametric polymorphism for scopes such as full-fledged Java generics [78] was felt to be too complex by the SCJ expert group, we introduced a limited form of polymorphism that seems to capture many common use cases. Polymorphism is obtained by adding the following scope variables: `CALLER`, `THIS` and `UNKNOWN`. These can be used in `@Scope` annotations to increase code reuse. A reference that is annotated `CALLER` is allocated in the same scope as the "current" or calling allocation context. References annotated `THIS` point to objects allocated in the same scope as the receiver (i.e. the value of `this`) of the current method. Lastly, `UNKNOWN` is used to denote unconstrained references for which no static information is available. Classes may be annotated `CALLER` to denote that instances of the class may be allocated in any scope.

The @RunsIn Annotation. To determine the scope in which an allocation expression `new C()` is executed we need to associate methods with nodes in our static scope tree. The `@RunsIn` annotation does this. It takes as an argument the symbolic name of the scope in which the method will be executed. An argument of `CALLER` indicates that the method is scope polymorphic and that it can be invoked from any scope. In this case, the arguments, local variables, and return value are by default assumed to be `CALLER`. If the method arguments or returned value are of a type that has a scope annotation, then this information is used by the Checker to verify the method. If a variable is labeled `@Scope(UNKNOWN)`, the only methods that may be invoked on it are methods that are labeled `@RunsIn(CALLER)`.

An overriding method must preserve and restate any `@RunsIn` annotation inherited from the parent. `@RunsIn(THIS)` denotes a method which runs in the same scope as the receiver.

Default Annotation Values. To reduce the annotation burden for programmers, annotations that have default values can be omitted from the program source. For class declarations, the default value is `CALLER`. This is also the annotation on `Object`. This means that when annotations are omitted classes can be allocated in any context (and thus are not tied to a particular scope). Local variables and arguments default to `CALLER` as well. For fields, we assume by default that they infer to the same scope as the object that holds them, i.e. their default is `THIS`. Instance methods have a default `@RunsIn(THIS)` annotation.

Static Fields and Methods. This paragraph describes the rules for static fields and methods. We talk about methods/fields in other places but we assume that they are not static by default. The allocation context of static constructors and static fields is IMMORTAL. Thus, static variables follow the same rules as if they were explicitly annotated with IMMORTAL. Static methods are treated as being annotated CALLER.

Strings constants are statically allocated objects and thus should be implicitly IMMORTAL. However, this prevents users from assigning a string literal to a local variable even though the string literal is immutable. Therefore, we chose to treat these strings as CALLER so they may be safely assigned to local variables.

Dynamic Guards. Dynamic guards are our equivalent of dynamic type checks. They are used to recover the static scope information lost when a variable is cast to UNKNOWN, but they are also a way to side step the static annotation checks when these prove too constraining. We have found that having an escape hatch is often crucial in practice. A dynamic guard is a conditional statement that tests the value of one of two pre-defined methods, `allocatedInSame()` or `allocatedInParent()` or, to test the scopes of a pair of references. If the test succeeds, the check assumes that the relationship between the variables holds. The parameters to a dynamic guard are local variables which must be `final` to prevent an assignment violating the assumption. The following example illustrates the use of dynamic guards.

```
void method(@Scope(UNKNOWN) final List unk, final List cur) {
  if (ManagedMemory.allocatedInSame(unk, cur)) {
    cur.tail = unk;
  }
 }
```

The method takes two arguments, one `List` allocated in an unknown scope, and the other allocated in the current scope. Without the guard the assignment statement would not be valid, since the relation between the objects' allocation contexts can not be validated statically. The guard allows the checker to assume that the objects are allocated in the same scope and thus the method is deemed valid.

Arrays. Arrays are another feature that requires special treatment. By default, the allocation context of an array `T[]` is the same as that of its element class, `T`. Primitive arrays are considered to be labeled THIS. The default can be overriden by adding a `@Scope` annotation to an array variable declaration.

10.4.1.1 Scope Inference

The value of polymorphic annotations such as THIS and CALLER can be inferred from the context in certain cases. A concretization function (or scope inference) translates THIS or CALLER to a named scope. For instance a variable annotated THIS takes the scope of the enclosing class (which can be CALLER or a named scope). An object returned from a method annotated CALLER is concretized to the value of the calling method's `@RunsIn` which, if it is THIS, can be concretized to the enclosing class' scope. We say that two scopes are the same if they are identical after concretization.

The concretization is used to determine the allocation context for an expression. In an ideal situation, where every class is annotated with `@Scope`, it is easy to determine the scope of an expression simply by examining the `@Scope` annotation for the type of the expression. However, unannotated types and `@Scope`-annotated variables complicate the situation. For example, suppose a method declares a new `Integer` object:

```
Integer myInt = new Integer();
```

It must be possible to know in which scope `myInt` resides. In the general case, it can be assumed to be `@Scope(THIS)`, since use of the `new` operator is, by definition, in the current allocation context; however, if the method has a `@RunsIn` annotation which names a specific scope `"a"`, then it is more precise to state that `myInt` is `@Scope("a")`.

It is important to infer a scope for every expression to ensure that an assignment is valid. Since a field may only refer to an object in the same scope or a parent scope, statically knowing the scope of every expression that is assigned to the field makes it possible to determine whether or not the assignment is actually legal.

Local variables, unlike fields and parameters, may have no particular scope associated with them when they are declared and are of a type that is unannotated. Scope inference is also used to bind the variable to the scope of the right-hand side expression of the first assignment. In the above example, if the containing method is `@RunsIn(CALLER)`, `myInt` is bound to `@Scope(CALLER)` while the variable itself is still in lexical scope. In other words, it is as if `myInt` had an explicit `@Scope(CALLER)` annotation on its declaration. It would be illegal to have the following assignment later in the method body:

```
myInt = Integer.MAX_INT;
```

It is intuitive to derive the scope of other, more complex expressions as well. Method calls that have a `@Scope` annotation or are of an annotated type take on the specific scope name of the annotation. If there is no `@Scope` annotation, then the method is assumed to return a newly allocated object in the current scope.

The scope of a field access expression may depend on the scope of the object itself. For example, if we have a field access `exp.f`, if the type of `f` and the declaration of `f` have no `@Scope` annotation, then the scope of `exp.f` is the same as the scope of `exp`. Note that `exp` can represent any expression, not just a variable.

10.4.1.2 Memory Safety Rules

We will now review the constraints imposed by the checker.

Overriding Annotations. Subclasses must preserve annotations. A subclass of a class annotated with a named scope must retain the exact same scope name. A subclass of a class annotated `CALLER` may override this with a named scope. Method annotations must be retained in subclasses to avoid upcasting an object to the supertype and executing the method in a different scope.

A method invocation `z=x.m(...,y,...)` is valid (1) if its `@RunsIn` is the same as the current scope or it is annotated `@RunsIn(CALLER)`, (2) if the scope of every argument `y` is the same as the corresponding argument declaration or the argument is `UNKNOWN`, (3) if the scope of the return value is the same as `z`.

An assignment expression `x.f=y` is valid if one of the following holds: (1) `x.f` and `y` have the same scope and are not `UNKNOWN` or `THIS`, (2) `x.f` has scope `THIS` and `x` and `y` has the same, non-`UNKNOWN` scope, or (3) `x.f` is `THIS`, `f` is `UNKNOWN` and the expression is protected by a dynamic guard.

A cast expression `(C) exp` may refine the scope of an expression from an object annotated with `CALLER`, `THIS`, or `UNKNOWN` to a named scope. For example, casting a variable declared `@Scope(UNKNOWN) Object` to `C` entails that the scope of expression will be that of `C`. Casts are restricted so that no scope information is lost.

An allocation expression `new C()` is valid if the current allocation context is the same as that of the class `C`. A variable or field declaration, `C x`, is valid if the current allocation context is the same or a child of the allocation context of `C`. Consequently, classes with no explicit `@Scope` annotation cannot reference classes which are bound to named scopes, since `THIS` may represent a parent scope.

10.4.1.3 Additional Rules and Restrictions of the Annotation System

The SCJ memory safety annotation system further dictates a following set of rules specific to SCJ API methods.

MissionSequencer and Missions. The `MissionSequencer` must be annotated with `@DefineScope`, its `getNextMission()` method has a `@RunsIn` annotation corresponding to this newly defined scope. Every `Mission` associated with a particular `MissionSequencer` is instantiated in this scope and they must have a `@Scope` annotation corresponding to that scope.

Schedulables. Each `Schedulable` must be annotated with a `@DefineScope` and `@Scope` annotation. There can be only one instance of a `Schedulable` class per `Mission`.

MemoryArea Object Annotation. The annotation system requires every object representing a memory area to be annotated with `@DefineScope` and `@Scope` annotations. The annotations allow the checker to statically determine the scope name of the memory area represented by the object. This information is needed whenever the object is used to invoke `MemoryArea` and `ManagedMemory` API methods, such as `newInstance()` or `executeInArea()` and `enterPrivateMemory()`. The example in Fig. 10.21 demonstrates a correct annotation of a `ManagedMemory` object `m`.

The example shows a periodic event handler instantiated in memory `M` that runs in memory `H`. Inside the `handleAsyncEvent()` method we retrieve a `Managed-`

```
@Scope("M") @DefineScope(name="H", parent="M")
class Handler extends PeriodicEventHandler {

    @RunsIn("H") @SCJAllowed(SUPPORT)
    void handleAsyncEvent() {
      @Scope(IMMORTAL)
  @DefineScope(name="M", parent=IMMORTAL)
  ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(this);
  ...
    }
}
```

Fig. 10.21 Annotating `ManagedMemory` object example

`Memory` object representing the scope `M`. As we can see, the variable declaration is annotated with `@Scope` annotation, expressing in which scope the memory area object is allocated – in this case it is the `IMMORTAL` memory. Further, the `@DefineScope` annotation is used to declare which scope is represented by this instance.

executeInArea(). Calls to a scope's `executeInArea()` method can only be made if the scoped memory is a parent of the current allocation context. In addition, the `SCJRunnable` object passed to the method must have a `@RunsIn` annotation that matches the name of the scoped memory. This is a purposeful limitation of what SCJ allows, since the annotation system does not know what the scope stack is at any given point in the program.

enterPrivateMemory(). Calls to a scope memory's `enterPrivateMemory (size, runnable)` method are only valid if the runnable variable definition is annotated with `@DefineScope(name="x",parent="y")` where `x` is the memory area being entered and `y` is a the current allocation context. The `@RunsIn` annotation of the runnable's `run()` method must be the name of the scope being defined by `@DefineScope`.

newInstance(). Certain methods in the `MemoryArea` class encapsulate common allocation idioms. The `newArray()`, `newArrayInArea()`, and `newInstance()` methods may be used to allocate arrays and objects in a different allocation context than the current one. In these cases, invocations of these methods must be treated specially. Calls to a scope's `newInstance()` or `newArray()` methods are only valid if the class or element type of the array are annotated to be allocated in target scope or not annotated at all. Similarly, calls to `newArrayInArea()` are only legal if the element type is annotated to be in the same scope as the first parameter or not annotated at all. The expression

```
ImmortalMemory.instance().newArray(byte.class, 10)
```

Fig. 10.22 A library class `BigInteger` with a method returning objects in two different scopes

```
class BigInteger {

  static BigInteger ZERO = new BigInteger(0);
  BigInteger add(BigInteger o) {
    if (this == ZERO) return o;
    if (o == ZERO) return this;
    return slowAdd(this, o);
  }
}

ZERO.add(ZERO);
ZERO.add(new BigInteger(3));
```

should therefore have the scope `IMMORTAL`. An invocation MemoryArea.newArrayInArea(o, byte.class, 10) is equivalent to calling MemoryArea.getMemoryArea(o).newArray(byte.class, 10). In this case, we derive the scope of the expression from the scope of `o`.

getCurrent*() methods. The `getCurrent*` methods are static methods provided by SCJ API that allow applications to access objects specific to the SCJ infrastructure. The `getCurrent*()` methods are `ManagedMemory.getCurrentManagedMemory()`, `RealtimeThread.getCurrentMemoryArea()`, `MemoryArea.getMemoryArea()`, `Mission.getCurrentMission()`, `MissionManager.getCurrentMissionManager()`, and `Scheduler.getCurrentScheduler()`. Typically, an object returned by such a call is allocated in some upper scope; however, there is no annotation present on the type of the object. To explicitly express that the allocation scope of returned object is unknown, the `getCurrent*()` methods are annotated with `@RunsIn(CALLER)` and the returned type of such a method call is `@Scope(UNKNOWN)`.

10.4.2 Impact on Standard Library Classes

The presented system was designed in part to minimize changes to the standard libraries. However, there are some patterns that the system cannot capture with full precision when using library classes across different scopes. This occurs for example when a method returns objects that live in different scopes, as illustrated in Fig. 10.22.

The `BigInteger` class attempts to prevent unnecessary allocation when adding two objects together. If either operand is zero, the result will be the same as the other operand; since `BigInteger` objects are immutable, it is acceptable to simply return the other operand. However, with respect to SCJ, this means that `add()` can return objects in several different scopes. On the first use of `add()`, an object living in the immortal memory is returned. However, the second addition returns the newly created object representing the value 3, which is allocated in the current allocation context that may or may not be immortal memory.

There are a few solutions to this problem. First, `add()` could be annotated to return an object in the `UNKNOWN` scope. This means that, in order to minimize the number of dynamic checks to call `BigInteger` methods, most of the methods must be labeled `@RunsIn(CALLER)`. This is safe to do because `BigInteger` objects are treated as value types and are therefore never mutated after construction, but litters the standard library class with annotations.

Another solution is to make `add()` always return a fresh object, so that the method can be implied as `@Scope(CALLER)`. This has the advantage of not requiring explicit annotations. However, the lack of `@RunsIn(CALLER)` annotations limits `BigInteger` operands and operators to living in the same scope. This both simplifies and limits how the standard library may be used.

Even though the system requires annotation of standard Java libraries, we believe that this one-time cost paid by JVM vendors is negligible in comparison to the costs of sanitizing those libraries to qualify them for safety certification and then actually gathering all of the required safety certification evidence.

10.5 Collision Detector Example

In this section we present the Collision Detector (CD*x*)[3] [233] example and illustrate the use of the memory safety annotations. The classes are written with a minimum number of annotations, though the figures hides much of the logic which has no annotations at all.

The CD*x* benchmark consists of a periodic task that takes air traffic radar frames as input and predicts potential collisions. The main computation is executed in a private memory area, as the CD*x* algorithm is executed periodically; data is recorded in a mission memory area. However, since the CD*x* algorithm relies on positions in the current and previous frame for each iteration, a dedicated data structure, implemented in the `Table` class, must be used to keep track of the previous positions of each airplane so that the periodic task may reference it. Each aircraft is uniquely represented by its `Sign` and the `Table` maintains a mapping between a `Sign` and a `V3d` object that represents current position of the aircraft. Since the state table is needed during the lifetime of the mission, placing it inside the persistent memory is the ideal solution.

First, a code snippet implementing the Collision Detector mission is presented in Fig. 10.23. The `CDMission` class is allocated in a scope named similarly and implicitly runs in the same scope. A substantial portion of the class' implementation is dedicated to the `initialize()` method, which creates the mission's handler and then shows how the `enterPrivateMemory()` method is used to perform some initialization tasks in a sub-scope using the `MIRun` class. The `ManagedMemory` variable `m` is annotated with `@DefineScope` and `@Scope` to correctly define which scope is represented by this object. Further, notice the use of `@DefineScope` to define a new `MI` scope that will be used as a private memory for the runnable.

[3]The CD*x* open-source distributions is at www.ovmj.net/cdx (Version miniCDj).

```
@DefineScope(name="M", parent=IMMORTAL)
@Scope("M") class CDMission extends Mission {

  void initialize() {
    new Handler().register();
    MIRun run = new MIRun();
    @Scope(IMMORTAL)
    @DefineScope(name="M", parent=IMMORTAL)
    ManagedMemory m   (ManagedMemory) ManagedMemory.getMemoryArea(this);
    m.enterPrivateMemory(2000, run);
  }
}

@Scope("M")
@DefineScope(name="MI", parent="M")
class MIRun implements SCJRunnable {
  @RunsIn("MI") void run() {...}
}
```

Fig. 10.23 CD*x* mission implementation

The `Handler` class, presented in Fig. 10.24, implements functionality that will be periodically executed throughout the mission in the `handleAsyncEvent()` method. The class is allocated in the `M` memory, defined by the `@Scope` annotation. The allocation context of its execution is the `"H"` scope, as the `@RunsIn` annotations upon the `Handler`'s methods suggest.

Consider the `handleAsyncEvent()` method, which implements a communication with the `Table` object allocated in the scope `M`, thus crossing scope boundaries. The `Table` methods are annotated as `@RunsIn(CALLER)` and `@Scope(THIS)` to enable this cross-scope communication. Consequently, the `V3d` object returned from a `@RunsIn(CALLER) get()` method is inferred to reside in `@Scope("M")`. For a newly detected aircraft, the `Sign` object is allocated in the `M` memory and inserted into the `Table`. This is implemented by the `mkSign()` method that retrieves an object representing the scope `M` and uses the `newInstance()` and `newArrayInArea()` methods to instantiate and initialize a new `Sign` object.

The implementation of the `Table` is presented in Fig. 10.25. The figure further shows a graphical representation of memory areas in the system together with objects allocated in each of the areas. The immortal memory contains only an object representing an instance of the `MissionMemory`. The mission memory area contains the two schedulable objects of the application – `Mission` and `Handler`, an instance representing `PrivateMemory`, and objects allocated by the application itself – the `Table`, a hashmap holding `V3d` and `Sign` instances, and runnable objects used to switch allocation context between memory areas. The private memory holds temporary allocated `Sign` objects.

```
@DefineScope(name="H", parent="M")
@Scope("M") class Handler extends PeriodicEventHandler {

  Table st;

  @RunsIn("H") void handleAsyncEvent() {
    Sign s = ... ;
  @Scope("M") V3d old_pos = st.get(s);
        if (old_pos == null) {
          @Scope("M") Sign n_s = mkSign(s);
          st.put(n_s);
        } else ...
  }

  @RunsIn("H") @Scope("M") Sign mkSign(@Scope("M") Sign s) {
    @Scope(IMMORTAL) @DefineScope(name="M",parent="IMMORTAL")
    ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(s);

    @Scope("M") Sign n_s = ManagedMemory.newInstance(Sign.class);
    n_s.b = (byte[]) MemoryArea.newArrayInArea(s, byte.class, s.length);
    for (int i : s.b.length) n_s.b[i] = s.b[i];
    return n_s
  }
}
```

Fig. 10.24 CD*x* handler implementation

The `Table` class, presented in Fig. 10.25 on the left side, implements several `@RunsIn(CALLER)` methods that are called from the `Handler`. The `put()` method was modified to meet the restrictions of the annotation system, the argument is `UNKNOWN` because the method can potentially be called from any subscope. In the method, a dynamic guard is used to guarantee that the `Sign` object being passed as an argument is allocated in the same scope as the `Table`. After passing the dynamic guard, the `Sign` can be stored into a field of the `VectorRunnable` object. This runnable is consequently used to change allocation context by being passed to the `executeInArea()`. Inside the runnable, the `Sign` is then stored into the `map` that is managed by the `Table` class. After calling `executeInArea()`, the execution context is changed to `M` and the object `s` can be stored into the `map`. Finally, a proper `HashMap` implementation annotated with `@RunsIn(CALLER)` annotations is necessary to complement the `Table` implementation.

10.6 Related Work

The Aonix PERC Pico virtual machine introduces stack-allocated scopes, an annotation system, and an integrated static analysis system to verify scope safety and analyze memory requirements. The PERC type system [280] introduces annotations indicating the scope area in which a given object is allocated. A byte-code verifier

```
@Scope("M") class Table {*\vspace{-2mm}*

  final HashMap map;
  V3d vectors [];
  int counter = 0;
  final VRun r = new VRun(); *\vspace{-2mm}*

  @RunsIn(CALLER) @Scope(THIS)
  V3d get(Sign s) {
    return (V3d) map.get(s);
  }*\vspace{-2mm}*

  @RunsIn(CALLER) void put(
      final @Scope(UNKNOWN) Sign s) {
    if (ManagedMemory
        .allocatedInSame(r,s))
      r.s = s;
    @Scope(IMMORTAL)
    @DefineScope(name="M",
        parent=IMMORTAL)
    ManagedMemory m = (ManagedMemory)
        MemoryArea.getMemoryArea(this);
    m.executeInArea(r);
  }
}*\vspace{-2mm}*

@Scope("M") class VRun
  implements SCJRunnable {*\vspace{-2mm}*

  Sign s;

  @RunsIn("M") void run() {
    if (map.get(s) != null) return;
    V3d v = vectors[counter++];
    map.put(s,v);
  }
}
```

```
ImmortalMemory   = @1
MissionMemory    = @2
PrivateMemory    = @3
CDMission        = @4
Handler          = @5
Table t          = @6
HashMap map      = @7
VRun r           = @8
V3d old_pos      = @9
Sign n_s         = @10
Sign s           = @11
```

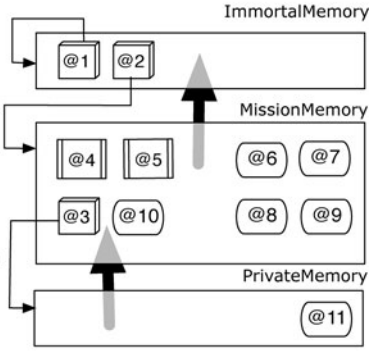


Fig. 10.25 CDx table implementation

interpreting the annotations proves the absence of scoped memory protocol errors. The PERC Pico annotations do not introduce absolute scopes identifiers. Instead, they emphasize scope relationships (e.g. argument `A` resides in a scope that encloses the scope of argument `B`). This allows more generic reuse of classes and methods in many different scopes, rather than requiring duplication of classes for each distinct scope context at the cost of a higher annotation burden. The PERC annotations address sizing requirements which are not considered here.

The authors of [75] proposed a type system for Real-Time Java. Although the work is applied to a more general scenario of RTSJ-based applications, it shows that a type system makes it possible to eliminate runtime checks. In comparison to the approach in this chapter, the proposed type system provides a richer but a more complex solution.

Scoped Types [12, 449] introduce a type system for RTSJ which ensures that no run-time errors due to memory access checks will occur. Furthermore, Scoped Types capture the runtime hierarchy of scopes and subscopes in the program text by the static hierarchy of Java packages and by two dedicated Java annotations. The authors demonstrates that it is possible to statically maintain the invariants that the RTSJ checks dynamically, yet syntactic overhead upon programmers is small. The solution presented by the authors is a direct ancestor of the system described by this chapter.

10.7 Conclusions

This chapter has presented the SCJ memory management API which provides increased safety by simplifying the memory model it inherited from the RTSJ. Furthermore, we have presented a set of metadata annotations which can be used to statically prove that SCJ compliant programs are free of memory errors caused by illegal assignments. This greatly increases the reliability of safety critical applications and reduces the cost of certification.

Acknowledgements The author thanks the JSR-302 expert group (Doug Locke, B. Scott Andersen, Ben Brosgol, Mike Fulton, Thomas Henties, James Hunt, Johan Nielsen, Kelvin Nilsen, Martin Schoeberl, Joyce Tokar, Andy Wellings) for their work on the SCJ specification and their input and comments on the memory safety annotations presented in this chapter.

This work was partially supported by NSF grants CNS-0938256, CCF-0938255, CCF-0916310 and CCF-0916350.

Chapter 11
Component-Oriented Development for Real-Time Java

Ales Plsek, Frederic Loiret, and Michal Malohlava

Abstract The Real-Time Specification for Java (RTSJ) offers developers an attractive platform for development of software that is composed of variously stringent real-time and non real-time tasks. However, the RTSJ introduces a programming model involving several non-intuitive rules and restrictions which make systems modeling and development a complex and error-prone task. In this chapter we introduce an advanced software engineering technology that allows developers to address these challenges – component-oriented development. In the first part, we present the general concepts and identify the key benefits of this technology. We further show one of the possible approaches on how to embrace this technology when mitigating the complexities of RTSJ development – the Hulotte framework. The framework provides a continuum between the component-oriented design and implementation process of RTSJ systems. The RTSJ concepts are treated as first-class entities which enables the modeling and automatized preparation of their implementation by employing code-generation techniques. We conclude this chapter with an extensive case study comparing object-oriented and component-oriented implementation of the *Collision Detector Benchmark*. Our empirical evaluation shows that the performance overhead of the framework is minimal in comparison to manually written object-oriented applications without imposing any restrictions on the programming style.

A. Plsek (✉)
Purdue University, West Lafayette, IN, USA
e-mail: aplsek@purdue.edu

F. Loiret
INRIA Lille Nord-Europe, KTH (Royal Institute of Technology), Stockholm, Sweden
e-mail: floiret@kth.se

M. Malohlava
Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
e-mail: michal.malohlava@d3s.mff.cuni.cz

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_11,

11.1 Introduction

The future of distributed, real-time and embedded systems brings large-scale, heterogeneous, dynamically highly adaptive systems fulfilling tasks of different complexities [193, 239]. This will bring variously stringent *Quality of Service* (QoS) demands presented at the same time for different but interconnected parts of the system. Meeting these challenges represents a task of developing applications composed from hard-, soft- and non-real-time units. The *Real-Time Specification for Java* (RTSJ) [65] brings an attractive solution since it provides a combination of the Java language's expressive power with the programming concepts delivering predictability where needed.

However, using RTSJ is an error-prone process where developers have to be aware of various non-intuitive rules and restrictions. Despite numerous efforts for designing RTSJ compliant patterns mitigating the complexity of RTSJ [1, 54, 297], there has not been found any sufficient solution. A desired solution should mitigate not only RTSJ implementation complexities, but primarily postulate a systematical development approach considering RTSJ limitations from high-level system design to its projection at the implementation level.

One of the answers to these issues is component-oriented frameworks for RTSJ, such as [154,208]. The basic motivation lays in abstracting complexities of the RTSJ development from the developers. A clear separation of RTSJ concerns from the rest of the system would allow developers to obtain applications that are more modular and where RTSJ-related properties are isolated in clearly identified modeling and implementation entities. Such orthogonal separation would ease not only modeling and development of RTSJ-based systems but also would enable application of various model-driven solutions such as code generation or model transformations. Nevertheless, the state-of-the-art solutions still need to fully address adaptability issues of real-time systems, separation of real-time and functional concerns, and the overall design process that would introduce real-time concerns at the system design level.

In this chapter, we present the component-oriented development and apply it to the domain of real-time Java programming. We first introduce the concept of component-oriented development and highlight its key attributes in general scenario and then we present its application to the challenges represented by RTSJ. A complete design process of real-time and embedded applications comprise many complexities [187], especially timing and schedulability analysis, which has to be included in a design procedure. The scope of our proposal is placed directly after these stages, when real-time characteristics of the system are specified but the development process of such a system lies at its very beginning.

In our prior work [302, 303], we argue that an effective development process of real-time Java systems needs to consider the specifics of the RTSJ programming model at early stages of the system model design in order to mitigate the complexities of the implementation phase. Following this philosophy, the challenge is to leverage the concepts of RTSJ by expressing them at the system model

level and to allow users to manipulate transparently the functional and RTSJ-related concepts. To achieve this, we propose a component model tailored to fit the needs of the RTSJ programming model. It extracts the model's concepts and introduces their equivalents in the process of system modeling. Thus, the model allows designers to manipulate RTSJ domain concepts as first-class entities along the whole development lifecycle. We achieve full separation of functional and non-functional concerns, while the designer can reason about the RTSJ related aspects prior to the implementation phase. Expressing the RTSJ domain concepts at the model level further leverages the challenge of reusing the application in variously stringent real-time scenarios. The designed system can be easily adapted to different real-time condition just by modifying the assemblies of RTSJ-related entities in the system model. Furthermore, we provide a continuum between the design and implementation process by implementing the Hulotte framework.[1] Based on a system model, the framework automatically generates an execution infrastructure where RTSJ concerns are managed transparently and separately from the source code implementing the functional concerns.

We conducted an extensive evaluation case study; the *Collision Detector Benchmark* CD*x* [233][2] was reimplemented in our framework to evaluate the benefits of the approach. The case study shows that the implementation of RTSJ concepts can be easily isolated in dedicated component entities without influencing the programming style used for the implementation of the functional concerns. Finally, we present an empirical evaluation showing that the overhead of the framework is minimal in comparison to manually written object-oriented applications.

11.1.1 Structure of the Chapter

The chapter is structured as follows. We present the basic principles of component-oriented development in Sect. 11.2. Based on this background, we introduce our general purpose component model in Sect. 11.3 and consequently enhance it with Real-time Java concepts in Sect. 11.4. This component model is implemented by the Hulotte framewok, presented in Sect. 11.5. To illustrate the application of the framework on non-trivial RTSJ examples, we present a case study in Sect. 11.6. Empirical evaluation of the framework is presented in Sect. 11.7. We compare our approach with other component-oriented frameworks for real-time programming in Sect. 11.8. We conclude in Sect. 11.9.

[1] The Hulotte framework is available at http://adam.lille.inria.fr/soleil/.

[2] Available under open-source license at http://www.ovmj.net/cdx/.

11.2 Component-Oriented Development

Component-based software engineering (CBSE) is a largely adopted technique of software preparation, assembly, deployment and maintenance. The component-based system development process [114] is based on a decomposition of a system into fine-grained building blocks called *components* which have well-defined communication points defined via *interfaces*. A component framework plays a main role in the process. It encapsulates main artifacts for proper design, development, assemblage and deployment of systems. Proper component frameworks covering the whole component-based development process comprises of a (i) component model, (ii) methodology, (iii) runtime environment and (iv) supporting tools simplifying development of software systems [93]. The (i) component model is the cornerstone for each component framework, its extensiveness substantially influences the capabilities of a component framework. The component model defines with help of a selected formalism [50, 86] the structural, interaction and behavioral aspects which have to be satisfied by components and the resulting assembled systems. The whole system development process is driven by the (ii) methodology clarifying the utilization of the component model in each stage. The process is supported by the (iv) tools (e.g., Integrated Development Environments, simulators) mitigating the required effort. A resulting assembled system is executed with help of the (iii) runtime environment.

Based on investigation of several general purpose component models [85, 88, 105, 418] we extract a fundamental characteristic of a state-of-the-art component model: a lightweight hierarchical component model that stresses on modularity and extensibility. It allows the definition, configuration, dynamic reconfiguration, and clear separation of functional and extra-functional (including also *domain-specific*) concerns. The central role is played by re-usable interfaces, which can be either *functional* or *control*. Whereas *functional interfaces* provide external access points to components, *control interfaces* are in charge of extra-functional properties of the component (e.g., life-cycle, binding management or memory allocation strategy). From the system execution perspective, components are divided into *passive* and *active* according to their role in system execution. Whereas passive components generally represent provided services, active components have associated their own thread of execution. Additionally, a concept uniquely provided by the Fractal component model [85] is a *shared component* which defines that a component could have several super-components. This capability allows the specification of the membership in multiple composites representing logical grouping of components with domain-specific semantics (e.g., to specify that components encapsulated at an arbitrary hierarchical levels must be deployed within a single memory area also represented as a component).

At the system runtime-level, component frameworks provide a notion of *container* (also referred to as a *component membrane* in [365]) – a control environment encapsulating every component and responsible for handling various extra-functional properties related to a given component and enabling execution of

component code. This brings better separation of functional and extra-functional concerns, which can be hidden in containers, and thus simplify utilization and reuse of components by the end-users.

Contrary to component frameworks like Enterprise JavaBeans (EJB) [393] where extra-functional concerns are hard-coded in containers, modern frameworks provide the notion of *flexible* containers where arbitrary complex and fine-grained extra-functional services can be provided (e.g., transaction management, timers). A typical modern container is itself implemented as a composite component defining an architectural style and composition rules to specify the link between system-level and container-level components and between the container and its environment.

The container contains control elements of two types – *interceptors* and *controllers*. The interceptors participate in calls on component interfaces by their adaptation, blocking or observation. The controllers constitute the logic of container and they manage communication between component content, interceptors, and other controllers. Controllers can expose their functionality through *control interfaces* to the environment of the container, or to the interceptors, the latter being useful when controlling the component's communication with its environment.

11.3 A General Purpose Component Model

This section introduces a general purpose component model that serves as the cornerstone of the framework. The component model is specially designed to enable a full separation of functional and domain-specific concerns in all steps of the system development. We define the core of the model and enrich it by the concepts of `Domain Component` and `Functional Component`.

We present a graphical illustration of our metamodel in Fig. 11.1. The key element of the metamodel is an abstract entity `Component` including a set of functional interfaces. The `Component` is further extended by `Primitive` and `Composite Component` entities. We thus introduce the notion of hierarchy into the model. Whereas the *primitive* components implement directly some functionality, expressed by the `Content` entity, the composite component encapsulates a set of subcomponents. Furthermore, a notion of *supercomponent* enhances the metamodel with the aforementioned concept of *sharing* – one component can be contained in more than one supercomponent. Finally for the sake of metamodel correctness, we define that each component in the system is either composite or primitive. The other two key entities of the metamodel – `Functional Component` and `Domain Component` – express the orthogonal concept to the composition. They are introduced to enforce separation of functional and domain-specific concerns.

Functional Components. Functional components are basic building units of our model representing functional concerns in the system. The motivation for the introduction of functional components is to precisely separate functional and

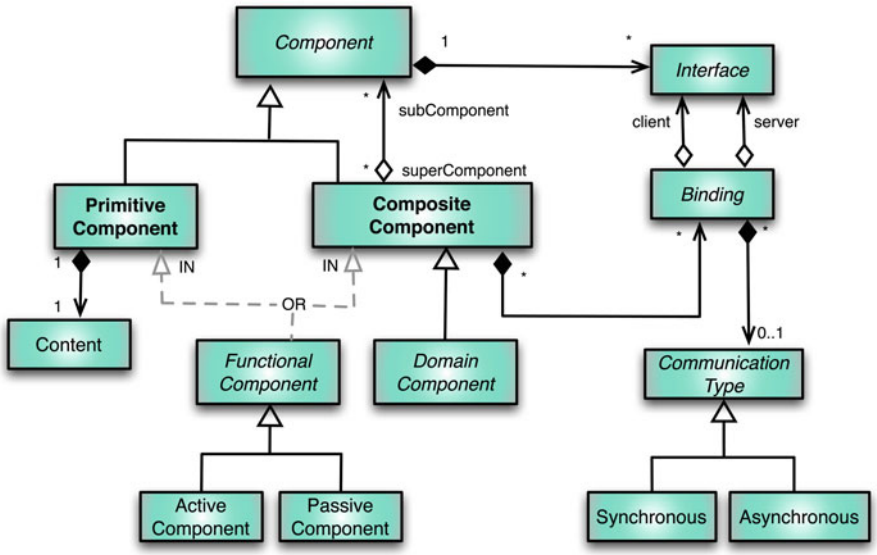


Fig. 11.1 A generic component model

domain-specific concerns of the applications in all steps of the software development lifecycle. Due to the orthogonality of this concept to the primitive/composite component concept, we define `Functional Component` as an extension of either primitive or composite component, in the Fig. 11.1 highlighted by the slashed arrows.

We distinguish two types of functional components – `Active Component` and `Passive Component`. The active component has an associated thread of execution and properties regarding its periodicity, deadline, and priority. We can thus characterize the active component type by setting its attributes, to define, e.g., a periodic/sporadic execution or event-based execution. Using active components, we can define execution modes consistent with the RTSJ semantics (Sect. 11.4). We further distinguish the active components as `periodic` and `sporadic`. Periodic cannot have a provided functional interface (except an implementation of the Java `Runnable` interface specifying the entry point of the thread's periodic execution), whereas sporadic can, but only bound with asynchronous communication.

A passive component is a standard component-oriented unit providing and requiring services. When being called upon, its execution is carried out in the context of the active component demanding its services. If a passive component is used in a concurrent environment, various policies can be applied to guarantee coherence of its internal state, e.g., a mutual exclusion mechanisms. We call such a component a `protected component`. However, implementation of these policies is in the competence of a framework developer, whereas the application developer only specifies the type of the applied policy.

Binding and Composing Functional Components. The metamodel provides the concepts of `Interface` and `Binding`. These well-known concepts introduce notions of *client* and *server* interface. Furthermore, we define different types of Bindings: `Synchronous` and `Asynchronous`. This is motivated by the specific requirements on communication between `Active` and `Passive` components.

Domain Components. A brand new concept that we introduce is `DomainComponent`, inspired by [278]. The main purpose of domain components is to model domain-specific requirements in a unified way. A domain component is a composite component that encapsulates functional components. By deploying functional components into a certain domain component, the developer specifies that these subcomponents support the domain-specific concern represented by the domain component. Moreover, a domain component contains a set of attributes parameterizing its semantics. The sharing paradigm allows developers to fully exploit this concept. A functional component is a part of a functional architecture of the application but at the same time can be contained in different domain components, which thus express domain-specific requirements of this component. We are thus able to express both functional and domain-specific concerns simultaneously in the architecture.

Therefore, a set of super components of a given component directly defines its functional and also its domain-specific roles in the system, the given component thus takes part both in the functional and domain-specific architecture of the system. Moreover, the domain-specific concerns are now represented as first-class entities and can be manipulated at all stages of development process.

The approach of modeling domain-specific aspects as components brings advantages commonly known in the component-based engineering world such as reusability, traceability of selected decisions or documentability of the solution itself. Also, by preserving a notion of a component, it is possible to reuse already invented methods (e.g., model verification) and tools (e.g., graphical modeling tools) which were originally focused on functional components. If we go further and retain domain components at runtime then it is possible to introspect and reconfigure domain-specific properties represented by domain components at runtime.

We illustrate the DomainComponent concept in Fig. 11.2. Components `Writer`, `Readers`, `MailBox`, `Library` and their bindings represent a business logic of the application. The domain component `DC1` encapsulates `MailBox` and `Library`, thus defining a domain-specific service (e.g., logging of every interface method invocation) provided by these two components. At the same time, component `DC2` represents a different service (e.g., runtime reconfiguration) and defines that this service will be supported by components `Writer` and `Readers`.

Overview of the development flow. The Fig. 11.3 presents the development flow associated to the methodology underpinning our component framework, from the end-user perspective. It emphasises on basic development activities performed by the end-user within an iterative process, and on the artifacts manipulated at each step (i.e., ADL files based on an XML syntax, or Java files). The architecture

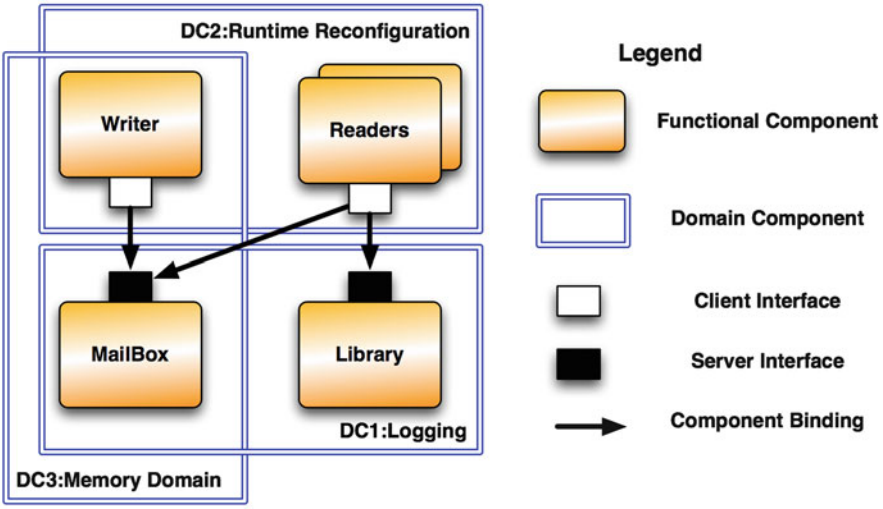


Fig. 11.2 Domain components example

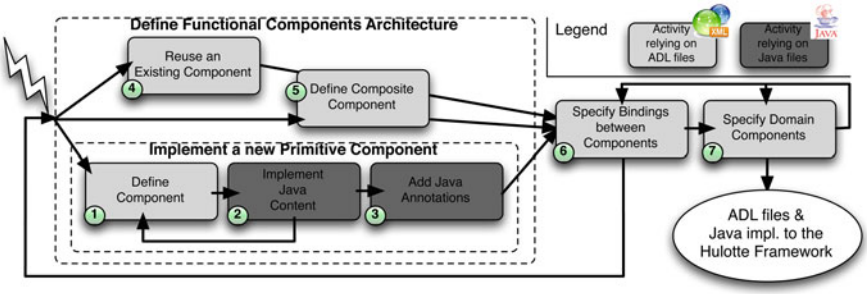


Fig. 11.3 End-user development flow

made of `Functional Components` is defined either by implementing new `Primitive Components` (steps 1–3), by reusing off-the-shelf components (step 4), or by specifying `Composite Components` (step 5). The implementation of new primitives is decomposed into three main activities: the component specifications are defined within an ADL file describing, except other architectural properties presented from Fig. 11.1, the provided and required interfaces of the component and their signatures (step 1); the `Content` of the component is afterwards implemented in Java (step 2), and Java annotations are finally used (step 3) to specify the mapping between architectural specifications and implementation constructs, as further detailed in Sect. 11.5.2. After having defined the `Bindings` between components (step 6), the `Domain Components` are specified (step 7) as already described above. At the end of this workflow, a resulting set of ADL and Java files serves as an input for the Hulotte framework, as detailed further in Sect. 11.5.1.

We will use the domain component concept to represent the RTSJ-related features in the system.

11.4 A Real-Time Java Component Metamodel

When designing a component model for RTSJ, a sufficient level of abstraction from RTSJ complexities has to be provided. This will allow RTSJ concepts to be considered at early stages of the architecture design to achieve an effective development process that mitigates all the complexities. Therefore, while keeping an appropriate level of abstraction, our goal is to define a proper representation of RTSJ concepts in the model. To achieve this, we extend the component model defined in the previous section with the RTSJ concepts represented as domain components.

In Fig. 11.4 we define a set of RTSJ compliant domain components. Their goal is to express RTSJ concerns as components and allow manipulation of these concerns as such. Two basic entities representing RTSJ concerns are defined: `ThreadDomain` and `MemoryArea`. This brings us the advantage of creating the most fitting architecture according to real-time requirements of the system.

ThreadDomain Component. The `ThreadDomain` component represents `RealTimeThread`, `NoHeapRealTimeThread` defined by RTSJ, and Java `Regular Thread`. The model presented in Fig. 11.4 refines each thread type as a corresponding domain component. The goal of the `ThreadDomain` component is to manage threads that have the same properties (thread type, priority, etc.). Since in our model, every execution thread is dedicated to one active component, we deploy every active component as a subcomponent of an instance of the `ThreadDomain`. Consequently, the properties of the `ThreadDomain` are inherited by the active component precisely determining the execution characteristics of its thread of control. The use of this model construct must fulfill some constraints. Indeed, a

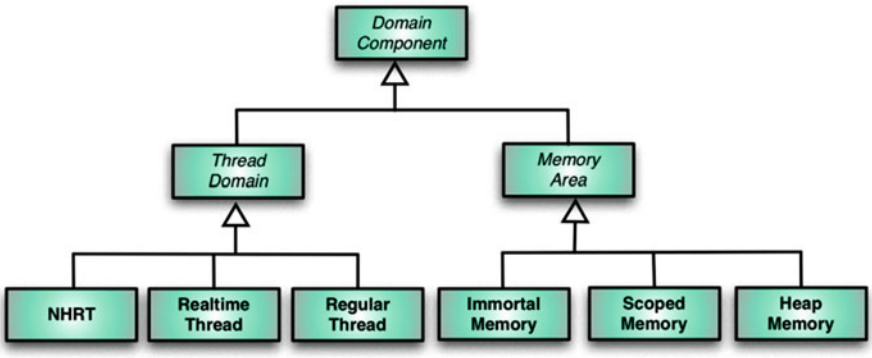


Fig. 11.4 The RTSJ-specific domain components

`ThreadDomain` cannot be arbitrarily nested, no RT-Thread can be defined as a descendant of another RT-Thread, furthermore, an *active component* must always be nested in a unique `ThreadDomain`.

The impact of the thread domain is threefold. First, a centralized management of active threads with the same properties is provided. Second, since communication between different scheduling entities in RTSJ is a crucial issue, we consider beneficial that thread domains allow designers to detect cross-thread communication. Based on this detection, corresponding patterns for its implementation can be applied. Finally, by introducing this entity, we are able to explicitly define those parts of a system that will be executed under real-time conditions. Therefore we precisely know which components have to be RTSJ-aware and we are able to enforce corresponding RTSJ rules. Moreover, communication between the system parts that are executed under different real-time or non-realtime conditions can be expressed at the architectural level. This brings an advantage of creating the most fitting architecture according to real-time demands of the system.

MemoryArea Domain Component. The *MemoryArea* domain components represent the memory areas distinguished by RTSJ: `ImmortalMemory`, `ScopedMemory`, and `HeapMemory`. MemoryArea component thus encapsulates all functional subcomponents which have the same *allocation context* – this is the memory that will be used to allocate data when executing the given component. Such specification of allocation context at the architectural level allows developers to detect communication between different memory areas (i.e., *cross-scope communication*) and apply rules corresponding to RTSJ. Moreover, in combination with the `ThreadDomain` entity we can entirely model communication between different real-time and non-real-time parts of the system.

Apart from the data generated by each component during the runtime, the infrastructure objects – the objects representing the component and the additional runtime support, are persistent throughout the execution of the system and are allocated in the immortal memory.[3]

`MemoryArea` components can be nested, however, RTSJ specification defines several constraints to their application. First, each functional component is deployed in a memory area, thus defining its allocation context. Furthermore, RTSJ defines a hierarchical memory model for memory areas. The immortal memory is defined as a parenting scope of all memory scopes. Additionally, nested memory scopes can be easily modeled as subcomponents. However, dealing with memory scopes, the *single parent rule*, has to be respected. In the context of our hierarchical component model, parenting scope of each memory area can be easily identified, which considerably facilitates the scope management. These constraints are systematically checked within the toolset presented in the next section.

[3]The model can be further extended to optimize the memory usage by enabling allocation of the infrastructure data in scoped memory areas, which would also allow reconfiguration of the system at runtime, these extensions are currently being investigated.

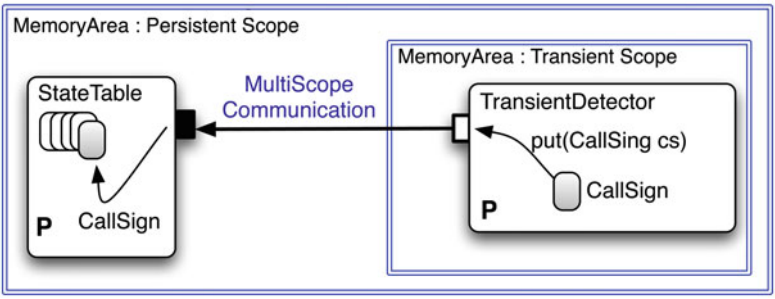


Fig. 11.5 Illustration example : RTSJ domain components and cross-scope communication

Illustration Example. Figure 11.5 illustrates the use of memory domain components, the example presents a part of our case study presented in Sect. 11.6. The `Table` and `TransientDetector` are functional components, the `TransientScope` `Detector` component performs computations in a child scope and stores the intermediate results in the `Table` component that is allocated in the parent scope.

To reflect this scenario in the architecture, we introduce two memory domain components : `PersistentScope` and `TransientScope`. Each domain component encapsulates the functional component, defining the allocation context of that specific component. Furthermore, the `PersistentScope` encapsulates also the `TransientScope` and thus explicitly defines the parent-child relation of these two memory areas.

11.4.1 Binding RTSJ Components

In order to be fully compliant with RTSJ, a set of binding rules needs to be respected during the design. The RTSJ-specific concerns are not only related to functional components but also influence each other. For example, a communication between two threads of different priorities could introduce priority inversion problem [424] and therefore, according to RTSJ, must be properly managed. In our solution, such situation can be easily identified since it corresponds to a binding between active components residing in different thread domains. Cross-thread communications must be asynchronous and therefore the binding between the two active components must be of an asynchronous type. Similarly, the model allows to clearly express cross-scope communication as a binding between two functional components both residing in different memory scopes.

Since adherence to RTSJ rules can be expressed and potentially verified at the architectural level, the designer is able to decide which types of connections between components can be used. This mitigates unnecessary complexities of the implementation phase where only the implementation of chosen binding concepts

is performed. Moreover, once appropriate attributes of component bindings are specified, component frameworks can fully exploit the code generation techniques to implement them without posing any additional burden on the implementation of functional components.

We enable this functionality by revisiting cross-scope patterns introduced in [1, 54, 297] and adapting them for the component-oriented development:

- *Cross-scope pattern*, represents the basic pattern used when implementing communication between two scopes in a parent-child relation. This pattern implements entering a child scope from a parent scope and leaving the scope while deep-copying the returned object from a child scope to a parent scope. The pattern is automatically used for every communication from a parent scope component to a child scope component.
- *Multi-scope pattern* represents a situation when a component sends data from a child scope to a parent scope with the intention to store these computed results and keep them available after the child scope is reclaimed.
- *Handoff pattern* is a more general application of the multi-scope pattern. This pattern is required for many real-time tasks which deal with filtering large amounts of data. Such tasks continuously process the data while retaining a small part of this data in a different scope, the main data area is reclaimed at the end of each computation. In this situation we therefore send data from one scope to another, while these scopes are not in a child-parent relation. Typically, we apply this pattern to a communication between components residing in sibling scopes.

The reasoning about an appropriate communication pattern for each binding is implemented by the framework. The user can further instruct generation of these patterns on specific bindings through a set of corresponding annotations. A detailed description of the annotations and implementation of the cross-scope communication patterns is given in Sect. 11.5.3.

Looking at the illustration example presented in Fig. 11.5, the binding between the `Table` and `TransientDetector` is crossing scope boundaries of a memory area component and therefore a specific cross-scope communication pattern needs to be defined. In this case, a component from a child scope calls a parent scope component to store data in the persistent scope, therefore, the *multi-scope pattern* should be used. A similar type of reasoning is used for any cross-scope binding, the framework, presented in the following section, ensures that appropriate RTSJ code will be implemented by the infrastructure.

11.5 Hulotte Framework

In this section we present the Hulotte Framework [256] – a toolset that implements the RTSJ component model. The framework provides generic principles and extensible tools which can be adapted to various application domains having heterogeneous requirements. The high-level abstractions offered by CBSE

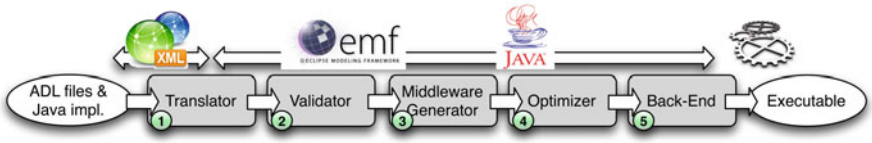


Fig. 11.6 Basic steps of the Hulotte workflow

are then tailored for capturing domain knowledge and are used for generating domain-specific middleware platforms and implementation artifacts. The non-trivial programming model introduced by RTSJ is a typical example of *problem domain* addressed by Hulotte: RTSJ concepts are captured at early stages of the software development and are then used for automatic generation of the middleware platform implementing their semantics. An interested reader can find more examples of domains in which the Hulotte framework can be used in [257, 258].

The remainder of this section gives a general overview of the Hulotte framework and presents the extensions implemented for handling the RTSJ component model.

11.5.1 Hulotte *Workflow*

The goal of the Hulotte framework is to generate an appropriate middleware infrastructure that implements the containers of each component, enable communication between the components, manage the initialization, shutdown of the whole system and many other tasks. To reflect this, the workflow of the Hulotte toolset consists of five basic steps and is sketched out in Fig. 11.6. Each of these steps defines extension points for handling domain-specific features, we give an overview of each of these steps:

1. The *Translator* is in charge of loading the architectural description of the end-user's application stored in an XML-based Architecture Description Language (ADL). It outputs an instance of an EMF[4] model used throughout the Hulotte process. The translation process can be extended for handling domain-specific characteristics of ADL (e.g., *Thread Domains* and *Memory Areas*).
2. The *Validator* performs semantic verifications on the instantiated models. By default, this step checks basic architectural constraints of the component model (such as consistency of hierarchical compositions, type checking between bound interfaces, among other), but domain-specific constraints can be injected by the domain expert to add verifications specific to domain components.
3. The *Middleware Generator* is in charge of generating domain-specific middleware components based on the container and connector models introduced in

[4] *Eclipse Modeling Framework* [86] is one of the most widely accepted metamodeling technology.

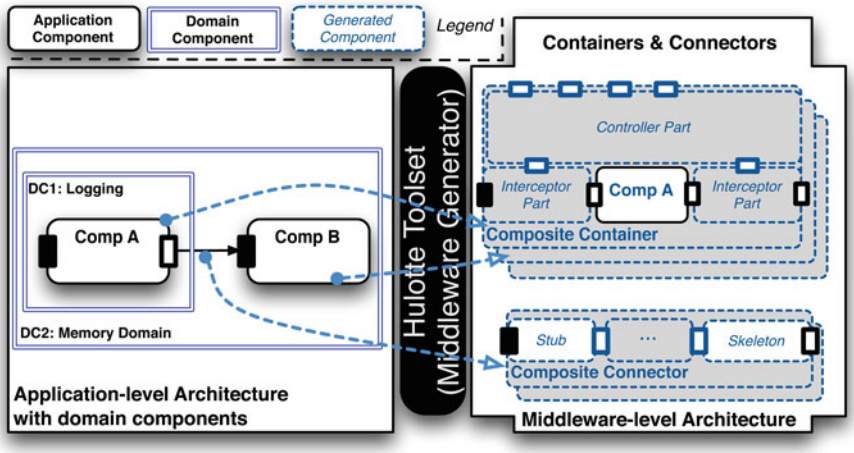


Fig. 11.7 Application-level and middleware-level architectures

Sect. 11.2. In our approach, we promote a homogeneous design methodology where both the application, the middleware platform and the domain-specific services are designed and implemented as components. The process is illustrated in Fig. 11.7. Based on the architecture of the system, a corresponding container is generated around each component, while a connector is generated for each binding. At this step of the process, Hulotte extensions act as domain-specific interpreters whose processing is triggered according to a traversal of domain components, and inject middleware components implementing their operational logic. Middleware components can be imported from an off-the-shelf component library or can be generated on-the-fly – e.g., interceptors implementing cross-scope interactions across boundaries formed by RTSJ memory areas.

4. The *Optimizer* introduces optimization heuristics in order to mitigate the common overhead of component-based applications. The heuristics focus on reducing interceptions in inter-component communication which usually causes performance overhead, and on merging architecture elements in order to decrease memory footprint. The merge algorithm consists in inlining, for instance, several component implementations into a single Java class.
5. The *Back-End* is in charge of producing code-level implementation artifacts, including the generation of bootstrap and substrate code of the architecture intended to be compiled by a classical Java compiler. Typically, this last step generates the *assembly factories* in charge of instantiating the architecture outputted by the Hulotte process, and can be also extended according to domain-specific needs.

Therefore, as an input to the framework, the user must supply the source-code classes implementing the functional components and an ADL file. The ADL file describes the architecture of the system, bindings between the components, and the

a

```
@Component(name =
           "TransientDetector",
  provides = @Interface(
   name = "iTransientDetector",
   signature =
        cdx.ITransientDetector))
class TransientDetector implements
  ITransientDetector {

  @Requires(name="iTable")
  ITable iTable;
  ...
}
```

b

```
<component name=
           "TransientDetector">
  ...
</component>
<DomainComponent>
  <AreaDesc type="Scope"
    name="PersistentScope"
    size="1MB" />
  <DomainComponent>
    <AreaDesc type="Scope"
      name="TransientScope"
      size="240kB" />
     <component
     definition=
       "./TransientDetector"/>
  </DomainComponent>
  ...
</DomainComponent>
```

Fig. 11.8 `TransientDetector` component in Java and its corresponding `Transient Scope` domain component declaration in ADL

domain components together with their attributes. The outcome of the framework is an infrastructure source-code that implements the middleware layer of the system.

11.5.2 Extending Hulotte *with RTSJ*

In this section we show the implementation of the RTSJ-specific component model, presented in Sect. 11.3, in the Hulotte framework.

The architecture of the system is given in an ADL, and Hulotte relies on Fraclet [330], an annotation framework used to specify the mapping between functional components and implementation constructs. Java annotations in Fraclet are then used to mark classes, fields or methods, as illustrated in Fig. 11.8a, where the annotation `@Component` defines a `TransientDetector` component with its provided and required interfaces. The full list of annotations is given in Fig. 11.9. Furthermore, in the Fig. 11.8b we present a definition of a memory area domain component. The constraints related to the use of domain components, such as those presented in Sect. 11.4, are implemented by a *Validator*'s extension, and are systematically checked by the framework prior to the middleware generation phase detailed in the following.

	Annotation	Applies To	Arguments	Description
Architectural Annotations	*@Component*	class	name	Defines a class implementing the component.
	@Provides	class	name signature	Provided interface.
	@Requires	field	*none*	Required interface.
	@Attribute	field	*name*	Component's attribute.
Annotations related to Cross-Scope Communication	*@DeepCopy*	method	args	Deep-copy the defined arguments between the scopes.
	@Direct		none	Direct method call.

Fig. 11.9 Component and RTSJ-specific annotations defined in the Hulotte framework

11.5.3 Implementing RTSJ Domain Components

In Sect. 11.4.1 we described patterns for cross-scope communication. Within the *Middleware Generator* step of Hulotte, extensions are triggered according to allocation contexts of functional components. The relationship among allocation contexts is fully described at the architectural level since memory areas can be hierarchically composed, specifying, e.g., a parent-child relationship between two memory areas encapsulating functional components bound together. These extensions automatically generate interceptor components within containers which implement the required pattern according to the description of allocation contexts. The following list describes the RTSJ-specific aspects handled by these extensions:

- For a binding between a parent scope component to a child scope component, a *cross-scope pattern* is generated. Each method call on the child scope component contains implementation of a default child scope enter by calling `MemoryArea.enter()` method. Furthermore, the framework must copy any object returned by the method call from the child to the parent scope.
- Between a child scope to a parent scope, a *multi-scope pattern* is generated, requiring a `MemoryArea.executeInArea()` method call. If data passed as arguments must be copied within the parent scope, the annotation `@DeepCopy` (see Fig. 11.9) can be specified by the developer and the appropriate code will be generated.
- To implement a sibling scopes communication, the *hand-off pattern* is used causing generation of `MemoryArea.executeInArea()` method calls within interceptors.

We further illustrate the use and implementation of a multi-scope pattern. Consider the interaction between the client component `TransientComponent` and the

```
@Component(name="Table" ...)
class Table
  implements ITable {

  @DeepCopy(args="s")
  void put(Sign s) {
    ...
  }

  void update(Sign s) {
    ...
  }

  V3d get(Sign s) {
    ...
  }
}
```

```
class STInterceptor
    implements ITable {
 ITable iTable;
 ScopedMemory scope;
 R r = new R();

 void put(Sign s) {
   ...
   r.cs = s;
   scope.executeInArea(r);
 }
 class R implements Runnable {
    void run() {
     Sign s;
     iTable.put(s.clone());
    }
}
```

Fig. 11.10 Multi-scope communication pattern implementation

server component `Table`, as illustrated in Fig. 11.5. The left-hand side of the Fig. 11.10 shows the implementation of the component `Table` providing the `ITable` interface. This interface is used to update the list of aircraft `Signs` detected by the system. For the implementation of the `put()` method, we attached the annotation `@DeepCopy`, its argument specifies that the object referenced by the method argument `s` is to be copied into the parent scope. In the case of the `update()` method, no annotation is specified, the `s` object is passed by reference. Since the `update()` modifies the `s` object that is already allocated in the parent scope, passing the value by reference is legal (however, this may result in `MemoryArea` exception in the case that the `s` object is allocated in the child scope.)

In the context of the RTSJ component model, an interceptor implementing the multi-scope communication pattern needs to be generated on each binding between a child scope and a parent scope. An excerpt of the interceptor generated by the framework according to the example above is given in the right-hand side of the Fig. 11.10. It implements the appropriate change of memory area of allocation, using RTSJ's standard `executeInArea()` method call. If required, the deep copy of an object towards a destination area is performed via the method `clone()` which should be implemented by the developer.

The implementation of active components is also performed by Hulotte extensions within the *Middleware Generator* according to their allocations within *Thread Domains*. Controllers and interceptors are then generated and injected in components' containers implementing their execution semantics, i.e., their provided methods are executed periodically or on requests from the environment. These middleware components are then configured for instantiating at initialization time the appropriate kind of thread specified by the domain (see Sect. 11.4).

Finally, the RTSJ domain components also drive dedicated Hulotte extensions in its last processing step, within the *Back-End*. Indeed, RTSJ-specific code is injected at several places within the generation of *assembly factories* invoked at the application's initialization phase, e.g., for launching the execution of periodic components, or for instantiating RTSJ memory areas which in turn are used for component instantiations.

11.5.4 Hulotte *Framework Optimizations*

The last step of the Hulotte workflow performed in the *Back-End* takes as input the complete architecture of the system with functional and middleware components assembled together. It generates by default an infrastructure for which dependency injections between components are handled by dedicated component factories, themselves generated by the back-end. This mechanism relies on proxy objects interposed on component interfaces, and is the basic feature required to provide reflective and reconfigurable component-based systems. However, this can impact the performance in terms of memory footprint and execution time of the deployed executable. This negative influence can become a serious drawback of the approach since the whole middleware layer is implemented by components. In order to mitigate these overheads, we introduced optimization heuristics implemented within the back-end. The heuristics focus on reducing interceptions in inter-component communication and on merging implementations of architectural artifacts. The merge algorithm consists of inlining interceptors, controllers, and applicative component implementations into a single Java class and can be applied on a set of functional components nested in the same RTSJ *memory area*.

A second optimization is considered within the generation process of cross-scope interceptors. Indeed, in several cases, interceptor components implementing the change of memory area of allocation are not required and can be replaced by a simple method call. For example, it is typically the case when a call to a destination area does not perform any object instantiation during its execution. These cases cannot be easily detected by the framework as this would require a complete static analysis of the Java code, but the developer can use a dedicated annotation (`@Direct`) to instruct the framework not to generate interceptor on the concerned bindings.

11.6 Case Study

To evaluate our approach, we implement the Collision Detector (CD*x*) [12,233,449] case study in the Hulotte framework. The CD*x* benchmark suite is an open source family of benchmarks that allows users to target different hard and soft real-time platforms while exhibiting the same algorithmic behavior. A complete description

of the benchmark[5] is given in [233]; here we present only basic information needed to understand the overall nature of the algorithm implemented by the benchmark. This section then describes selected classes of the CD*x* benchmark, identifying the peculiarities of the RTSJ programming model. Consequently, we present an implementation of the CD*x* in the Hulotte framework and highlight the benefits of produced source code. Finally, a detail discussion comparing the object-oriented and component-oriented version of the benchmarks is given.

The CD*x* benchmark consists of a periodic task that takes air traffic radar frames as input and predicts potential collisions. The benchmark itself measures the time between releases of the periodic task, as well as the time taken to compute potential collisions. The periodic task performs mathematically intensive operations to determine potential collisions based on current trajectories of planes. Due to the finer details of the algorithm, differing distances between planes can cause highly varying workloads. Different levels of filtering eliminate pairs of aircraft as potential collisions, with each succeeding filter becoming more expensive.

11.6.1 CDj*-RTSJ Implementation*

In the original CD*x* implementation [233] – denoted here as CDj-RTSJ, the main computation is executed in a transient memory area, as the CD*x* algorithm is executed periodically. Data is recorded in a persistent memory area and is used for generate report after the program terminates. However, since the CD*x* algorithm relies on current and previous positions for each iteration, it is not enough to simply have a periodic realtime thread executing, whose memory is reclaimed after each iteration. A dedicated data structure – `Table`, must be used to keep track of the previous positions, represented by an instance of the `V3d` class, of each airplane, represented by an instance of the `Sign` class, so that the periodic task may reference it. Since the table is relevant during the lifetime of the benchmark, placing it inside the persistent memory is the ideal solution. The implementation of the `Table` class represents a good example of the intricacy of the RTSJ programming model.

Figure 11.11 shows a code-snippet implementing communication between the `TransientDetector` and `Table` classes using a multiscope pattern [297] – an instance of class allocated in the persistent scope but with some of its methods executing in the transient scope. Looking at the `TransientDetector`, the method `createMotions()` is executed in the transient scope and computes motions of aircraft, the new and old position of an aircraft is computed. If the `old` position is `null` then a new aircraft was detected and needs to be added into the list of aircraft. Otherwise, a position of already detected aircraft is updated.

[5] Detail instructions on how to run CD*x* on various platforms together with its open-source distributions are available from http://www.ovmj.net/cdx. The versions used for this book are tagged as "cdj-RTSJ-COM" and "cdj-RTSJ".

```
class Table {
  HashMap map = new HashMap();
  long counter = 0;
  V3d vectors[] = new V3d[1000];
  R r = new R();

  void put(Sign s) {
    r.sign = s;
    MemoryArea.getMemoryArea(this).executeInArea(r);
  }

  class R implements Runnable {
    Sign sign;
     void run() {
      v = vectors[counter++];
      map.put(sign, v);
    }
  }

  V3d get(Sign s) {...}
  void update(Sign s) {...}
}

class TransientDetector {
  Table t;

  List createMotions (RawFrame f) {
    Sign s = ... ;
    V3d old = t.get(mk(s));
    if (old == null) t.put(mk(s));
    else t.set(...);
  }

  Sign mk(byte[] s) {
      r.s = s;
      MemoryArea.getMemoryArea(t).executeInArea(r);
      return r.c;
   }

  class R implements Runnable {
    Sign c;
    byte[] s;
    void run() {
      byte[] b = new byte[s.length];
      for (int i :  b.length) b[i] = s[i];
      c = new Sign(b);
    }
  }
}
```

Fig. 11.11 CDj-RTSJ: `Table` and `TransientDetector` implementation

However, to ensure that these modifications will be permanent, we need to change the allocation context from the transient scope to the persistent scope. This is implemented with the classical RTSJ pattern where a `R` class implementing runnable is used to execute a portion of the code in a selected memory using `executeInArea()`. When obtaining a sign of a newly detected aircraft, the `Sign` object needs to be allocated in the persistent scope so that it is available also in the next iteration. Furthermore, the `Table` implementation must also change the allocation context when storing the position and sign of a new aircraft into the hashmap.

As we can see, the functional code is monolithic and highly entangled with the RTSJ-related code, making the implementation hard to understand, debug, or potentially update. Furthermore, the non-intuitive form of memory management prevents developers from determining exactly in which scope area a given method is being executed. A drawback that is particularly tricky when allocating data that needs to be preserved across the lifespan of memory scopes. This is putting a large burden on developers who must reason about the current allocation context and if necessary, change it. As already shown in [279], manual scoped memory control and changes performed by developers is a highly error prone practice.

11.6.2 CDj*-COM – A Component-Oriented Implementation of* CD*x*

We reimplemented the CD*x* benchmark in Hulotte – denoted as CDj-COM, and present a new architecture of the benchmark in Fig. 11.12. The original object-oriented CD*x* application was very easily re-engineered into a component-oriented one, moreover, the components implement only the functional logic of the application. Our approach allows us to design the application intuitively in correspondence with the logic of the algorithm. The `CollisionDetector` component is periodically starting the computations, which are proceeded in every iteration by the `TransientDetector` component. `TransientDetector` first asks `FrameBuffer`, which simulates aircraft movements, for new aircraft positions, creates motions, and then computes collisions. Moreover, the updates of the current aircraft positions are stored in each iteration into the `Table` component. The designed architecture is very clear and easily understandable, thus the developers are able to reason about the system without being limited by the RTSJ-related issues.

Consequently, domain components are applied to determine the allocation context of each component. The `CollisionDetector` component represents the starting point of the application, by deploying it in the `ThreadDomain` component we precisely define its execution context. Furthermore, `TransientScope` encapsulates functional components responsible for computations performed in

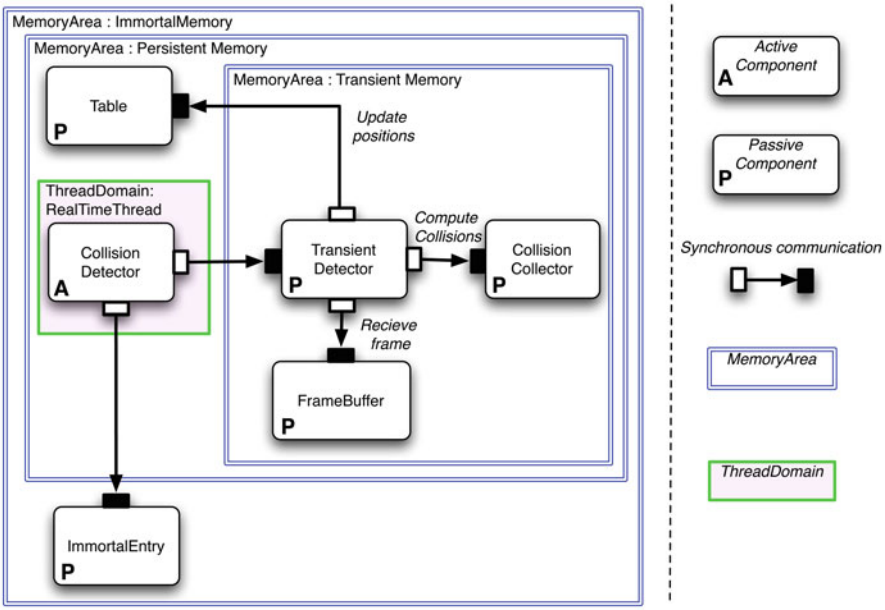


Fig. 11.12 CDj-COM : component architecture of CD*x*

every iteration of the algorithm and thus implements deallocation of temporal data between every iteration.

We illustrate the impact of the component-oriented approach on two components – `TransientScope` and `Table`, respectively in Fig. 11.13. The algorithm starts in the `createMotions` method in the `TransientDetector`, we first retrieve the old position of an aircraft. Afterwards, we determine whether the `old` is `null` and either insert a position of a new aircraft, or update a position of an already known aircraft.

We further show in the right-hand side of the Fig. 11.13 the implementation of `Table` component. It is annotated with according cross-scope annotations that instruct the framework to generate appropriate infrastructure code, as presented in Sect. 11.5.3. Further, the implementation does not contain any RTSJ-concepts, only the `@DeepCopy` annotation suggests that the `s` parameter of the `put()` method will be deep-copied into the persistent scope everytime this method is invoked.

11.6.3 Comparing CDj-RTSJ *and* CDj-COM

The CDj-COM implementation of CD*x* benchmark shows that Hulotte framework enables an intuitive and straightforward approach to implementation of RTSJ systems where RTSJ related concerns are expressed by the domain components

```
@Component(name = "TransientDetector",
  provides = @Interface(
    name = "iTransientDetector",
    signature =
        cdx.ITransientDetector))
class TransientDetector implements
            ITransientDetector {

  @Requires(name="iTable")
  ITable iTable;
  @Requires(name="iIdToIe")
  ITransDetectToImmEntry iIdToIe;

  List createMotions(RawFrame f) {
    Sign s = ...
    V3d old = iTable.get(s);
    if (old == null) {
      Sign n_s = iTable.createS(s);
      iTable.put(n_s,...);
    }
    else iTable.update(s,...);
  }
}
```

```
@Component(name = "Table",
  provides = @Interface(
    name = "iTable",
    signature = cdx.ITable))
class Table
    implements ITable {

  HashMap map =  new HashMap();
   long counter = 0;
   V3d vectors[] =
       new V3d[1000];

  @DeepCopy(args="s")
   void put(Sign s) {
    r.sign = s;
    v = vectors[counter++];
    map.put(sign, v);
  }

  @Direct
   V3d get(Sign s) {...}

  @Direct
  void update(Sign s) {...}
}
```

Fig. 11.13 CDj-COM: `TransientDetector` and `Table` implementation

and their implementation is provided by the underlying infrastructure. Comparing to CDj-RTSJ, the source code is not tangled with the RTSJ concepts and only functional code is present.

Furthermore, the full separation of functional and RTSJ-concerns allows to easily reuse the same source-code in different real-time scenarios.

11.7 Empirical Evaluation

The goal of this empirical evaluation is to show that our framework does not introduce any non-determinism and to measure the performance overhead of the framework. As one of the means of evaluation, we compare differently optimized applications developed in our framework against a manually written object-oriented application.

Our first goal is to show that the framework does not introduce any non-determinism into the developed systems, we therefore evaluate a worst-case execution time. Afterwards, we evaluate the overhead of the framework by performance comparison between an application developed in the framework (impacting the generated code) and an implementation developed manually through object-oriented

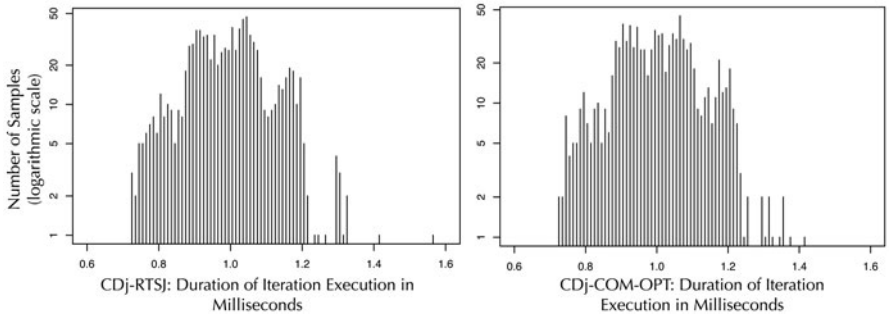


Fig. 11.14 Histograms of iteration execution times for CDj-RTSJ and CDj-COM-OPTCDj-COM-OPT average performance time is by 1% higher than CDj-RTSJ's, however, in the worst-case CDj-COM-OPT performams by 10% faster than CDj-RTSJ

approach. In the results presented bellow, we compare three different implementations of the evaluation scenario. First, denoted as CDj-RTSJ, is the manually developed object-oriented application. Then, denoted as CDj-COM and CDj-COM-OPT, are applications developed in our framework. CDj-COM corresponds to the case where the whole architecture outputed by the Hulotte process (functional and middleware components) is reified at runtime, and embedding the required services to introspect and reconfigure components. CDj-COM-OPT corresponds to the application where the optimization heuristics detailed in Sect. 11.5.1 are applied.

The testing environment consists of a Pentium 4 mono-processor (512 KB Cache) at 2.0 GHz with 1 GB of SDRAM, with the Sun 2.2 Real-Time Java Virtual Machine (a J2SE 5.0 platform compliant with RTSJ), and running the Linux 2.6.24 kernel patched by RT-PREEMPT. The latter converts the kernel into a fully preemptible one with high resolution clock support, which brings hard realtime capabilities[6]. We configured the benchmark to run with an intensive workload, using 60 planes with a 60 ms period for 1,000 iterations. The measurements are based on *steady state observations* – in order to eliminate the transitory effects of cold starts we collect measurements after the system has started and renders a steady execution.

The histogram in Fig. 11.14 shows the frequency of execution times for CDj-RTSJ and CDj-COM-OPT. A snapshot comparing execution times of the CDj-RTSJ, CDj-COM, and CDj-COM-OPT is presented in Fig. 11.15. As the first result, we can see that our approach does not introduce any non-determinism in comparison to the object-oriented one, as the execution time curves of CDj-RTSJ, CDj-COM, and CDj-COM-OPT are correlated. This is caused by the execution platform which ensures that real-time threads are not preempted by GC, and provides a low latency support for full-preemption mechanisms within the kernel.

[6]The Linux RT-PREEMPT patch is available at www.kernel.org/pub/linux/kernel/projects/rt/.

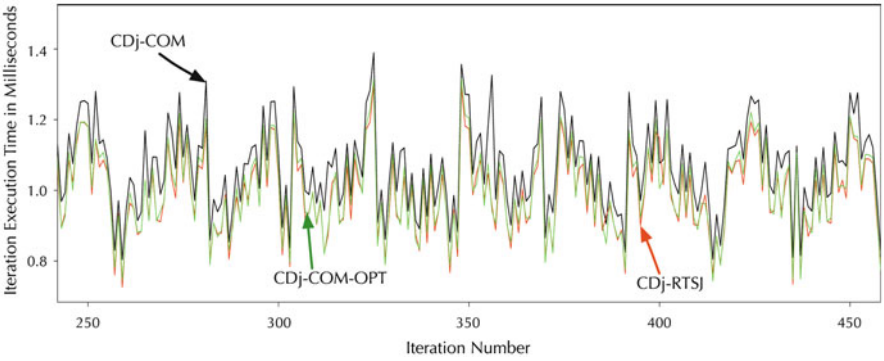


Fig. 11.15 A detailed runtime comparison of CDj-RTSJ, CDj-COM and CDj-COM-OPT for 200 iterations on x86

The data demonstrates that the average case performance time of CDj-COM is 5.7% higher than the average of CDj-RTSJ. Looking at the worst-case performance times, CDj-COM has a 5.6% overhead over CDj-RTSJ's performance. This corresponds to the overhead of component-oriented containers that introduce additional indirections on the method calls implementing the communication between the components, and on the non optimized cross-scope interceptors generated in the context of this scenario.

Considering the CDj-COM-OPT, the optimized version of the CDj-COM, its average performance time is only by 0.8% higher than the CDj-RTSJ's. This small overhead comes from the introduction of "setter method calls" between two components nested in different memory areas in CDj-COM-OPT (and then not merged together by our optimization algorithms by default), while implemented by direct public field accesses in CDj-RTSJ. For the worst-case performance, CDj-COM-OPT performs by 10% faster than CDj-RTSJ. We can observe again that the results are highly correlated.

The bottom line is that our approach does not introduce any non-determinism. Moreover, the overhead of the framework is minimal when considering CDj-COM-OPT optimized version but with the software engineering benefits discussed in Sect. 11.6.3.

11.8 Related Work

Recently, there is significant interest in real-time Java. This is reflected by the intensive research carried out in the area. However, the focus is laid on implementation layer issues, e.g., RTSJ compliant patterns [1, 54, 297], rather than on RTSJ frameworks where only a few projects are involved. Apart from these few

frameworks, other projects are recently emerging with features similar to our work.

Compadres [208], one of the most recent projects, proposes a component framework for distributed real-time embedded systems. A hierarchical component model where each component is allocated either in a scoped or immortal memory is designed. However, the model supports only event-oriented interactions between components. On the contrary to our approach, components can be allocated only in scoped or immortal memories, therefore communication with regular non-real-time parts of applications cannot be expressed. Since the coexistence of real-time and non-real-time elements of an application is often considered as one of the biggest advantages of RTSJ, we believe that it should be addressed also by its component model. Compadres also proposes a design process of real-time applications. However, a solution introducing systematically the real-time concerns into the functional architecture is not proposed, thus the complexities of designing real-time systems are not mitigated.

Work introduced in [154] defines a hierarchical component model for Real-Time Java. Here, components can be either active or passive. Similarly to our work, active components with their own thread of control represent real-time threads. However, the real-time memory management concerns cannot be expressed independently of the functional architecture, systems are thus developed already with real-time concerns which not only lay additional burdens on designers but also hinders later adaptability.

The project Golden Gate [145] introduces real-time components that encapsulate the functional code to support the RTSJ memory management. However, the work is focused only on the memory management aspects of RTSJ, the usage of real-time threads together with their limitations is not addressed.

The work published in [12] presents a new programming model for RTSJ based on aspect-oriented approach. Similarly to our approach, the real-time concerns are completely separated from applications base code. Although, as was shown in [365], aspect- and component-oriented approaches are complementary, but the component-oriented approach offers a more higher-level perspective of system development and brings a more transparent way of managing non-functional properties with only slightly bigger overhead.

The DiSCo project [309] addresses future space missions where key challenges are hard real-time constraints for applications running in embedded environment, partitioning between applications having different levels of criticality, and distributed computing. Therefore, similarly to our goals, the project addresses applications containing units that face variously hard real-time constraints. Here, an interesting convergence of both solutions can be revealed. The DiSCo Space-Oriented Middleware introduces a component model where each component provides a wide set of *component controllers* – a feature extensively supported by our solution.

The work introduced in [68] investigates fitness criteria of RTSJ in model-driven engineering process that includes automated code generation. The authors identify a basic set of requirements on the code generation process. From this point of view,

we can consider our generation tool as an implementation fully compatible to the ideas proposed in this work.

Applying generative methods [116] to propose a general approach to component framework development is not a novel idea. Bures et al. [87] summarize properties and requirements of current component-based frameworks and proposes a generative method for generating runtime platforms and support tools (e.g., deployment tool, editors, monitoring tools) according to specified features reflecting demands of a target platform and a selected component model. Comparing to our approach, the authors provide the similar idea of generation runtime platform, however they merely focus on runtime environment and related tools and neglect a definition of component model requirements by claiming that the proposed approach is generative enough to be tailored to reflect properties of contemporary component models.

Similarly, Coulson et al. [113] argue for the benefits and feasibility of a generic yet tailorable approach to component-based systems-building that offers a uniform programming model that is applicable in a wide range of systems-oriented target domains and deployment environments.

Furthermore, many state-of-the-art domain-specific component frameworks propose a concept of containers with controllers refined as components, e.g., DiSCo framework [309] addressing future space missions where key challenges are hard real-time, embedded constraints, different levels of application criticalities, and distributed computing. Cechticky et al. [97] presented the generative approach to automating the instantiation process of a component-based framework for such onboard systems.

On the other hand, *Aspect-Oriented Programming* (AOP) is a popular choice to non-functional services implementation. Moreno [276] argued that non-functional services should be placed in the container and showed how generative programming technique, using AOP, can be used to generate custom containers by composing differen non-functional features. This corresponds with our approach, however, as shown in [278], aspects can also be represented as domain components – *AspectDomain* component, thus allowing developers to leverage the aspect-techniques to the application design layer, and to represent them as components.

11.9 Conclusions

This chapter presents an application of component-oriented development to the domain of real-time Java programming. To illustrate the positive impact of this technology on the development process, the Hulotte component framework designed for development of real-time and embedded systems with the Real-Time Specification for Java (RTSJ) is proposed. The goal is to alleviate the development process by providing a means to manipulate real-time concerns in a disciplined way during the design and implementation life cycle of the system. Furthermore, we shield the developers from the complexities of the RTSJ-specific code implementation by separation of concerns and automatic generation of the execution infrastructure.

Therefore, we design a component model comprising the RTSJ-related aspects that allow users to clearly define real-time concepts as first-class software entities and to manipulate them through all the steps of the system development. Finally, we alleviate the implementation phase by providing a process generating automatically a middleware layer that manages real-time and non-functional properties of the system. To illustrate the approach and to evaluate its benefits, we implemented Collision Detector benchmark in Hulotte and we compare the result with manually written object-oriented implementation of CD*x*. The study confirms that Hulotte framework achieves full separation of functional and RTSJ-related code while allowing developers to focus on functional concepts. Our empirical evaluation study further shows that the Hulotte framework delivers predictable systems and the overhead of the framework is considerably reduced by the optimization heuristics that deliver competitive performance.

Acknowledgements The CD*x* benchmark was developed by Filip Pizlo and Ben Titzer at Purdue University. We thank Tomas Kalibera and Petr Maj for their help when re-implementing the benchmark. This work was partially supported by the Czech Science Foundation grant 201/09/H057.

Chapter 12
RT-OSGi: Integrating the OSGi Framework with the Real-Time Specification for Java

Thomas Richardson and Andy J. Wellings

Abstract The OSGi Framework,[1] has proven ideal in developing dynamically reconfigurable Java applications based on the principles of Component-Based Software Engineering (CBSE) and Service-Oriented Architecture (SOA). One domain where OSGi has yet to make an impact is real-time systems. This is partly due to the fact that the OSGi Framework is based on standard Java, which is not suitable for developing such systems. Other reasons include the absence of various features such as temporal isolation, admission control, and a reconfigurable garbage collector (along with the associated analysis to determine the pace of garbage collection). Also, a number of features of the OSGi Framework increase the difficulty in determining the worst case execution time (WCET) of threads. This makes real-time analysis more difficult. This chapter discusses the problems associated with extending the OSGi Framework and integrating it with the Real-Time Specification for Java (RTSJ). The focus is on the design of a real-time version of the OSGi Framework (RT-OSGi) which is capable of deploying dynamically reconfigurable real-time Java applications.

[1]Formerly known as the Open Services Gateway initiative, the name has been dropped but the acronym kept.

T. Richardson (✉) • A.J. Wellings
Department of Computer Science, University of York, York, UK
e-mail: tom@cs.york.ac.uk; andy@cs.york.ac.uk

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_12,

12.1 Introduction

A service is an act or performance offered by one party to another, often distinguishable from concepts such as components, objects, modules etc as being dynamically discoverable and dynamically available [183]. Service requesters typically compile to a service interface, finding and binding (i.e., dynamically discovering) an implementation of the service interface at run-time. Service implementations are dynamically available in the sense that at some point during run-time an implementation of a service interface may be available for finding and binding by service requesters, yet at other times, the service implementation may be withdrawn. Such dynamically discoverable and dynamically available services are the basis of service-oriented architecture (SOA) [150], which is an approach to developing software as a collection of service requesters and service providers.

The major benefit of using SOA to develop software is that the dynamic discovery and dynamic availability features make it possible for the software to undergo a limited form of dynamic reconfiguration [385] i.e., the software configuration can change (different service implementation bindings) without having to shut down the application.

The chapter is structured as follows. Section 12.2 introduces the OSGi framework and provides the necessary background material. It identifies the main problems areas that need to be addressed. Each of these are then considered in turn.

Section 12.3 discusses the problem of temporal isolation – how to ensure that execution-time overruns in one component do not affect the timing guarantees given to others. Section 12.4 then looks at the related problem of admission control – how to ensure that dynamically introduced components do not undermine the timing guarantees given to already admitted components. All timing guarantees require accurate worst-case execution time analysis, Sect. 12.5 presents the unique problems that are introduced with dynamically reconfigurable systems. Section 12.6 then turns to the problem of memory management, and in particular how to pace the garbage collector.

Inevitably added extra functionality to OSGi incurs overheads. Section 12.7 examines this issue. Finally, Related Work, Conclusions and Future work is given.

12.2 The OSGi Framework

The OSGi Framework [289] encompasses the service-oriented concepts of dynamic discovery and dynamic availability along with the concepts of Component-Based Software Engineering (CBSE), namely modularity. More specifically, the OSGi Framework is a Java based component framework with an intra-JVM service model. Application developers develop and deploy their application as a number of Java components (known as Bundles in the OSGi Framework). The threads in such components typically register and/or request services using a service registry provided by the OSGi Framework. Since a service is simply a plain old Java object (POJO) within the same JVM as the service requesting thread, once a reference to

the service object is obtained from the service registry, invocations of the methods of the service object are simply synchronous local method calls, the same as innovations of methods of any other Java object in the same JVM as the service requesting thread.

The OSGi Framework offers a high level of dynamic reconfigurability to applications. This is achieved not only through the dynamic features of SOA, but also through the OSGi life cycle operations. These life cycle operations enable components in OSGi to be installed, uninstalled, and updated during run-time. Figure 12.1 shows dynamic reconfiguration of an OSGi application through the invocation of different life cycle operations. The first figure (a) gives an example of an OSGi application. The second figure (b) shows the component install operation, with Component 3 being installed and deployed. It also shows the component update operation, with Component 2 being updated to include a new thread "T3".

The third figure (c) shows the removal of components, in this case, Component 1 being uninstalled. Notice the service dynamism. The threads in Component 2 were using the service in Component 1, after it was uninstalled, they then find and bind to the service in Component 3.

The advantage of dynamic reconfiguration is that it improves system availability. OSGi applications can remain online and available for use, perhaps with some degree of performance degradation, during system maintenance, evolution, and reconfiguration i.e., the application remains available for use whilst components are being added/removed/updated and whilst services are being registered and unregistered. This is unlike most other non-OSGi applications which typically must be taken offline for maintenance/evolution purposes.

The ability to dynamically reconfigure an OSGi application makes the OSGi Framework an attractive option for software development in a number of widely different domains, for example [288], the automotive industry, desktop computing such as Integrated Development Environments, home automation, and enterprise servers etc. One domain where OSGi has not yet been utilised, but where we believe it would be immensely beneficial, is the real-time systems (RTS) domain. In addition to their timing requirements, RTS often have high availability requirements i.e., it is important that the systems is available for use as long as possible e.g., for both safety and financial reasons. Therefore, it is of great benefit if the RTS can be kept operational and thus continue to be available for use even during system maintenance/evolution.

Dynamic reconfiguration is also useful for real-time systems as the application only needs to install the currently active components i.e., components are installed only when necessary (representing the current mode of operation), and removed when no longer required (during mode change). This minimises the use of CPU and memory by not using resources for inactive parts of the application. Such efficient use of resources is important in many real-time systems as they are also embedded systems with resource constraints. Furthermore, in OSGi, dynamic reconfiguration can be controlled from a remote location. This is useful for reconfiguring RTS that are deployed in harsh environments, that is, environments where there is a danger in being physically present. It is also useful for remotely reconfiguring mass produced RTS, such as some consumer electronics, with millions of units being sold.

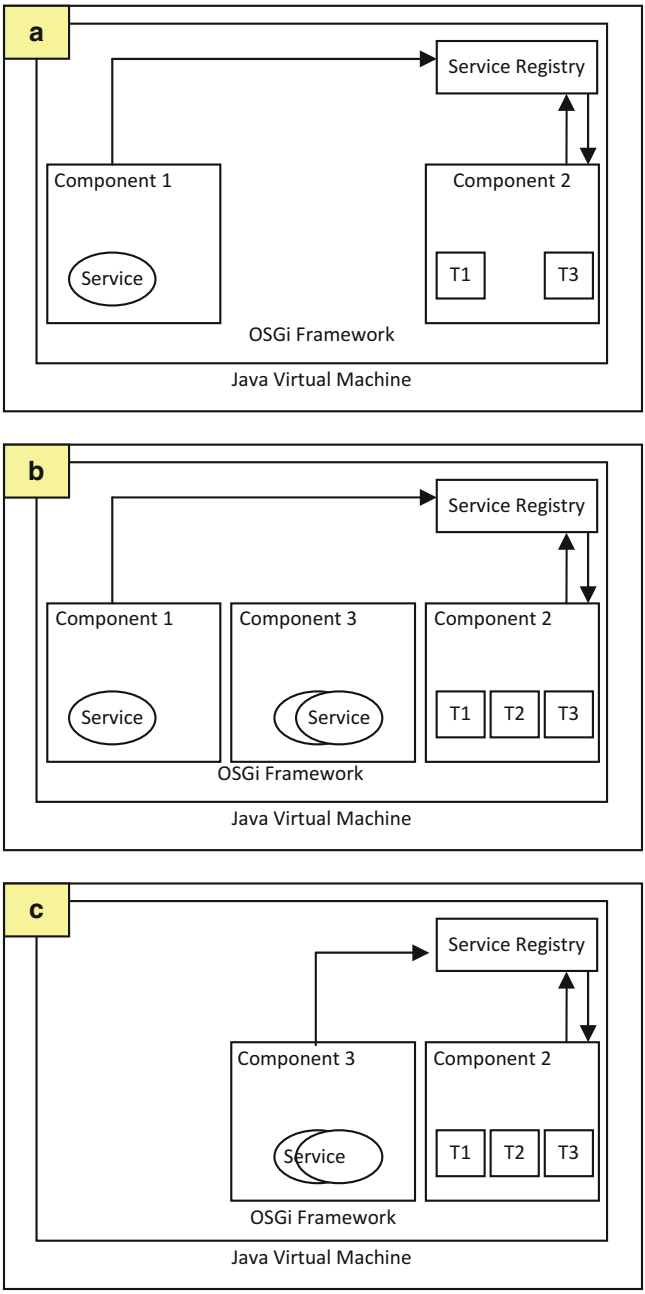


Fig. 12.1 Dynamic reconfiguration in OSGi

Finally, OSGi offers more to RTS than just the benefits of dynamic reconfiguration (high application availability, low resource requirements, remote reconfiguration control), it also provides the benefits traditionally associated with CBSE. The most significant benefit of using CBSE is third party component development. Using third party components typically reduces software development costs and reduces the time to market of an application. Also, the modular design of a component-based application improves the task of program comprehension as components have well defined functionality and well defined dependencies on other components.

Despite these advantages of developing and deploying real-time applications using the OSGi Framework, the OSGi Framework is currently not suitable for developing such systems. The OSGi Framework is written in standard Java, as are target application components and services. Standard Java is not suitable for the development of RTS. This issue can easily be resolved by using the Real-Time Specification for Java (RTSJ) [399] in place of standard Java, allowing component/service developers access to fixed priority pre-emptive scheduling, real-time threads, real-time garbage collection, and an alternative region-based approach to memory management named Scoped Memory (SM).

In addition to using the RTSJ in place of standard Java for component/service development, the OSGi Framework must also be extended to provide features necessary to support real-time systems, for example, OSGi must ensure that application reconfiguration can only take place when the timing constraints of all threads can still be met. It is also important for the OSGi Framework to be able to guarantee timing requirements despite the fact that it provides an "open" environment where components comprising multiple independent applications can be deployed. Clearly, in such an environment, it is imperative to provide isolation between such applications. Similarly, since components are often developed by third parties, it is imperative that faulty/malicious components cannot interfere with the timing constraints of other independent components.

12.3 Temporal Isolation

Temporal isolation [94] essentially means preventing threads in one component affecting the timing constraints of threads in other independent components. Temporal isolation can be broken if a component uses excessive amounts of a resource such as the CPU. An example of this is when the threads within one component uses more CPU time than was specified in schedulability analysis. The result is that the schedulability of the system is compromised, and other threads in the system may be starved of the CPU and may miss their deadlines as a consequence. Another example of breaking temporal isolation can be seen through memory management. Threads may use excessive amounts of memory, starving other threads of memory which may then cause those threads to block on memory exhaustion. Threads may also break temporal isolation indirectly through their impact on garbage collection. The issue of temporal isolation and CPU time overruns is discussed in this section.

In order to prevent CPU related breakage of temporal isolation, it is necessary to reserve each component enough CPU time for its threads to meet their deadlines. Such a reservation means that regardless of the behaviour of other independent threads, each thread is guaranteed to have its CPU requirements met. To provide a CPU reservation for components, two features are required: admission control (discussed in Sect. 12.4), and CPU cost enforcement (discussed in this section). The admission control will bound the number of components (and thus threads) deployed on the OSGi Framework, and as a result, will control the CPU load. For example, if adding a new component would cause threads to miss deadlines; the request for deployment is rejected. The cost enforcement will ensure that those components and threads that pass admission control, and are thus deployed, do not use more than the CPU time specified in admission control (specifically, in schedulability analysis).

Simply controlling the CPU load in terms of the number of components deployed may not be effective since it relies on the CPU-time requirements specified by components being accurate. Without cost enforcement actually bounding a component's CPU usage to that specified, the component has unrestricted use of the CPU and may accidentally (through errors in Worst-Case Execution-Time (WCET) calculation), or deliberately (through a CPU denial-of-service attack) use the CPU more than specified after it has passed admission control and been deployed. As a result, without cost enforcement, deployed component may starve other components of the CPU.

The combination of enforcing a bound on the CPU usage of currently deployed components/threads (through cost enforcement), and preventing new components/threads from being deployed when this would lead to insufficient CPU time for those already currently deployed (through admission control), gives resource reservation guarantees. However, the reservations are not hard in the sense that they are not guaranteed from the point of view of the Operating System (OS). This means that entities outside of the Java Virtual Machine (JVM) may be able to take CPU time from RT-OSGi threads. Therefore, it is assumed that no other application threads are running on the same machine as RT-OSGi. To remove this assumption, some kind of contract between the JVM and OS is required. Such contracts are discussed in [428].

The CPU cost enforcement [427] necessary to support temporal isolation is an optional feature of the RTSJ. It works by monitoring the CPU time used by a thread, and if the thread overruns its CPU budget value, the thread is descheduled immediately. The thread will then be rescheduled when its next release event occurs. Although *cost enforcement* is not implemented by currently available RTSJ implementations, at least one implementation of the RTSJ does support *cost monitoring* [144]. This cost monitoring can be used to support temporal isolation.

The difference between cost monitoring and cost enforcement is that cost monitoring does not deschedule threads upon detecting a CPU budget overrun. Instead, cost monitoring simply fires an asynchronous cost-overrun event. If any asynchronous event handlers have been set to handle the event, these handlers are released. This allows the application to take some customised action against

overrunning threads. Cost enforcement-like behaviour can thus be provided by having RT-OSGi provide the cost enforcement behaviour in a cost overrun handler. For example, the cost overrun handler could lower the priority of the overrunning thread such that it runs in the background until the start of its next period, and this allows other threads to not be affected by the overrun.

12.3.1 Execution-Time Servers

Providing cost enforcement at the thread level has the disadvantage that, in the worst case, every thread could fire a cost-overrun event and would need handling. This is a significant event firing/handling overhead. Another disadvantage of working at the thread level is the overhead of priority assignment in RT-OSGi, which will be discussed in Sect. 12.4. A more suitable approach to use for cost enforcement is execution-time-servers (servers) such as *Sporadic Server* [382], *Deferrable Server* [387], and *Periodic Server* [248].

Servers enable a collection of threads to share a group CPU budget, if the threads executing under the server collectively use more than the group budget, all of the server's threads are descheduled until the group budget is replenished, which depends on the server algorithm chosen. By creating a server for each application component to be deployed in RT-OSGi, cost overrun handling is at the component rather than the thread level, which has less overhead associated with it.

In terms of implementing servers in the RTSJ, this can be achieved by using the RTSJ's `ProcessingGroupParameters` (PGP). PGP allow threads to be grouped together and share a group budget. If the threads collectively use more than their PGP's budget within the PGP's period, the cost monitoring facility will detect this and fire a cost-overrun event, the same behaviour as it would if it had detected an overrun in an individual thread not assigned to a PGP. Once released, the associated cost overrun handler can lower the priority of all of the PGP's threads to a background level.

Unlike servers however, the execution eligibility of threads in a PGP are not automatically returned to their original value upon PGP budget replenishment i.e., in the case of RT-OSGi, the threads' priorities are not automatically raised to their original values on PGP budget replenishment. Budget replenishment can however be easily implemented through the `PeriodicTimer` and `AsyncEventHandler` RTSJ facilities and thread priority manipulation. The general approach is to create a periodic timer for each PGP with a period equal to the PGP budget replenishment period. Rather than having the timer fire the associated budget replenishment event periodically, the event firing should be disabled until a budget overrun occurs. At this point, the budget-replenishment event firing should be enabled so that on the next replenishment period after an overrun, the replenishment event is fired and the associated replenishment event handler is subsequently released. The budget replenishment event handler should raise the PGP's threads' priorities back to the value they had before they were lowered on budget overrun, in addition, it should

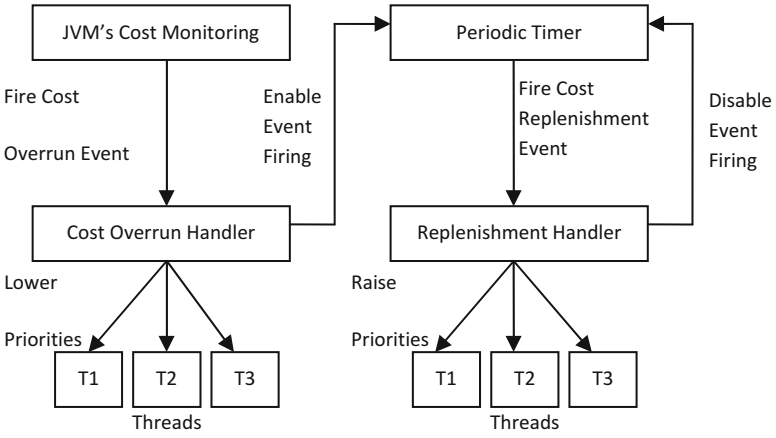


Fig. 12.2 Model of cost-overrun and budget replenishment

also disable the periodic event firing associated with the budget replenishment periodic timer until the next cost overrun. Figure 12.2 Shows the process of lowering a PGP's threads' priorities on PGP cost overrun, and the process of raising the PGP's threads' priorities on the PGP's next budget replenishment period.

As a note, temporal isolation only requires manipulation of thread priorities upon server budget overrun and does not need to periodically raise and lower the thread priorities. This is because RT-OSGi is mostly concerned with periodic and sporadic threads (as these can be given timing guarantees pre-component deployment-time), which, provided they abide by their estimated Worst-Case Execution Time and Minimum Interarrival Time (MIT) constraints, should not overrun their CPU budget e.g., a periodic thread will block for its next period before using the entire group budget. If aperiodic threads are executing under a server, they may always consume the entire server budget and in such a case, the periodic timer for firing replenishment event to release the replenishment handler could well be permanently enabled. However, there appears to be little benefit in keeping the replenishment event firing enabled as opposed to enabling it after an overrun and disabling it after the following budget replenishment.

By introducing servers into a system, it is essential that the scheduler supports the required semantics i.e., that the threads under control of a server execute only when the server is eligible for execution. For this, hierarchical scheduling [119] is typically required. Hierarchical scheduling works by having a global scheduler to schedule servers, and local schedulers (one for each server) to schedule each server's threads. Unfortunately, the RTSJ does not currently support hierarchical scheduling. Until such a time when the RTSJ provides support for hierarchical scheduling, it is necessary to simulate the behaviour of hierarchical scheduling on the RTSJ's single level fixed priority pre-emptive scheduler.

Hierarchical scheduling can be simulated by providing a priority mapping scheme. A priority mapping scheme is necessary because PGPs are not schedulable

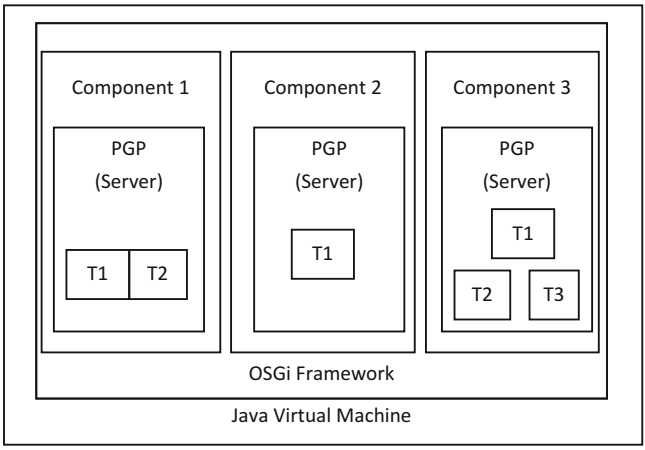


Fig. 12.3 Basic temporal isolation scheme of RT-OSGi

entities in the RTSJ thus it is necessary to assign a logical priority to each PGP which is then used as part of the priority assignment of threads executing under that PGP. For example, if a PGP has a logical priority higher than all other PGPs, then its threads must all be assigned higher priorities than threads in the other lower priority PGPs. This simulates hierarchical scheduling as it appears that the PGP with the highest priority is being scheduled followed by the next highest etc. Of course, in actual fact, it is the threads that are scheduled directly, with the priority mapping being responsible for this simulated PGP scheduling. The priority mapping is discussed in further detail in Sect. 12.4. As a note, by simulating hierarchical scheduling in this way, it is not possible to mix priorities across components i.e., the priorities of threads in one component must all be higher than or lower than the priorities of threads in another component in order to reflect the logical priority of their respective servers. Allowing otherwise would violate the behaviour of hierarchical scheduling as it would means that the threads executing under a lower priority server may execute in preference to threads executing under a higher priority server. As a result, any schedulability analysis would be invalidated because the rate monotonic priority ordering [252] at the server level would not be adhered to.

In summary then, PGP with cost monitoring and an overrun handler, along with a replenishment handler to raise the PGP's threads' priorities to their original values at the start of the next replenishment period after an overrun, will give cost enforcement at the component level in RT-OSGi, preventing the threads in one component (executing under one PGP) from overrunning their CPU budget and affecting the timing constraints of threads executing in other components (running under other PGPs). The simulated hierarchical scheduling enables temporal isolation to be provided at the component level rather than at the thread level, which reduces the associated run-time overheads of CPU cost enforcement. The temporal isolation model for RT-OSGi is shown in Fig. 12.3.

12.4 Admission Control

As discussed in Sect. 12.3, in order to provide components with a CPU resource reservations, it is necessary to enforce computation-time budgets for threads in currently deployed components, and it is also necessary to control the load on the CPU in terms of the number of components to be installed/updated. The latter topic is discussed in this section.

The OSGi Framework offers life cycle operations to component developers enabling components to be installed, updated, uninstalled, started, and stopped during run-time. The life cycle operations can have two sources, those that can be invoked from the code of components, and those that a user can invoke through the use of an interactive user interface to the OSGi Framework. In order to distinguish between these two types of life cycle operations, we name the former component-derived life cycle operations, and the latter, user-derived life cycle operations.

In both component-derived and user-derived life cycle operations, the install and update operations may increase the load on the CPU by essentially permitting more threads to be deployed, furthermore, life cycle operations can occur at any time. This means that the CPU can be overloaded by the repeated use of install/update operations. In order to enable components to have resource reservations, along with the cost enforcement, admission control is provided.

Admission control is a mechanism for controlling the system load by using acceptance tests to filter requests for deployment; only entities passing the acceptance tests may be deployed. The result of applying admission control to the install and update OSGi life cycle operations is that these operations fail when they would result in the system becoming unschedulable. Whilst both component-derived and user-derived life cycle operations require admission control, the admission control protocol will differ between these two types. In this section, the admission control extensions for install and update life cycle operations are discussed further, along with the component removal operation. Although, the latter does not require admission control, it nevertheless needs extending for RT-OSGi.

12.4.1 Component-Derived Life Cycle Operations

Life cycle operations can be invoked programmatically within components. The result of this is that the dynamic reconfiguration of the application, which occurs as a consequence, is known before the component is deployed. This is a key point, as it allows dynamic reconfiguration of an application to be pre-planned. This is discussed further in the upcoming section. A more flexible approach to dynamic reconfiguration is the user-derived life cycle operations which are discussed after the component-derived life cycle operations.

12.4.1.1 Admission Control During Installation

For admission control, we need to know: what the resource requirements of a component are, and, whether the system has sufficient resources to meet these demands. The CPU resource requirements of a component can be gathered by using a *Server Parameter Selection* algorithm [120]. Checking whether there are sufficient CPU resources available can be achieved by using schedulability analysis [253]. Finally, priority range assignment must take place to ensure that the priority ordering assumed in the schedulability analysis is still valid after application reconfiguration. Each of these three features is discussed in detail below.

As a note, because a component may need to install other components as part of its threads' execution, admission control is carried out for the entire group of components rather than in isolation i.e., during the installation of the first component of a group of components; the admission control guarantees resources for all other components in the group. For example, if a thread in Component A needs to install Component B, and one of its threads needs to install Component C, then the installation of Component A will involve performing admission control for all three components. In this way, Component A will only be installed and deployed if resources can be guaranteed for Component B and Component C. Essentially, RT-OSGi applications have a non-real-time initialisation phase during which, admission control for all of the components of an application that are installed/updated in a component-derived manner takes place.

Having a non-real-time initialisation phase for carrying out the admission control for all component-derived life cycle operations (i.e., reserving resources in advance of performing the life component-derived install and update life cycle operations) is necessary because there is little benefit in reserving resources for a component which must function as part of a group. When a component is installed, either, resources should be guaranteed for the whole group, or, installation should fail.

Server Parameter Selection

A component's server (PGP) must be assigned parameters (computation-time (C), period (T), and deadline (D)) such that all of the component's threads executing under the server have sufficient CPU time to meet their deadlines. However, at the same time, it is important that the parameters assigned to the server are not too pessimistic. The issue with over allocating the CPU to each server is that it may result in components failing admission control on schedulability grounds, which will reduce the total number of components that can be deployed. The reason for this is that it appears during admission control that the system is heavily loaded, when in fact, most of the CPU time assigned to components is unnecessary for making their threads schedulable.

In terms of the server parameter selection algorithms, they can be classified as either offline [447] or online [8] algorithms. There is a trade-off between the degree of pessimism of server parameters generated, and the execution-time

of the algorithm. The online algorithms have smaller execution-times than the offline algorithms, but as a result, the server parameters generated are much more pessimistic, over-allocating the CPU to the server. Both approaches are applicable to OSGi, the component developer can generate server parameters offline and include the generated parameters in their component's manifest file. Alternatively, an online server parameter algorithm can be called from the component install/update life cycle operations as part of admission control.

Schedulability Analysis

After server parameter selection, it is essential to check that the server parameters (CPU resource requirement) of a component can be assigned without causing threads in other components to miss their deadlines, i.e., without making the system unschedulable. Since servers are like periodic threads in the sense that no more than "C" units of computation time can be consumed within a period "T" with cost enforcement, the schedulability analysis used for sporadic task systems is also applicable to systems with fixed priority servers[2].

Whilst an exact schedulability test such as *Response Time Analysis* (RTA) [232] is desirable, it is computationally expensive and therefore not well suited for use in online systems such as OSGi. However, it is observed that the execution-time of RTA can be significantly reduced by using different initial values for the algorithm. This approach is taken by Davis et al. [122] and is known as *Boolean Schedulability Analysis*. This variant of RTA is known as *Boolean* because it can only be used to determine the schedulability of the system, i.e., schedulable or non-schedulable. The response times of tasks cannot be used for anything else as they are pessimistic estimates and not the exact response times.

To further reduce the time it takes to determine whether the system is schedulable, a sufficient schedulability test known as *Response Time Upper Bound* (RUB) [56] is used in combination with the *Boolean* test previously mentioned. The general approach is to use the *RUB* test on a task by task basis. Tasks failing the *RUB* test undergo the *Boolean* test. The combination of these tests acts as an acceptance test for CPU admission control. If a task fails the *RUB* test it is not a problem since the system may still be schedulable. However, since the *Boolean* RTA is an exact schedulability test, any task failing this indicates that the system is not schedulable and the component undergoing admission control should be rejected. In this case, it may be desirable to remove one or more currently deployed components in order to free up enough resources to allow the component to pass admission control and be successfully installed. Whether or not this technique is used depends on whether components are considered to have importance levels.

[2]With the exception of Deferrable server, which has to be modeled as a periodic task with release jitter.

Table 12.1 Priority range reassignment

Server	Period/ Deadline	Before S_{New} start priority	Before S_{New} end priority	After S_{New} start priority	After S_{New} end priorityl
S_1	1200	25	26	25	26
S_{new}	1300	NA	NA	21	23
S_2	3100	19	23	15	19
S_3	5000	02	06	02	06

Priority Assignment

The RTSJ only supports fixed priority pre-emptive scheduling. The aforementioned stages of admission control, namely server parameter selection and schedulability analysis, therefore, assume a priority assignment policy that supports such fixed priority systems, for example Rate Monotonic Analysis (RMA) [252]. Furthermore, as discussed in Sect. 12.3, so as to provide temporal isolation (at the component level), hierarchical scheduling is simulated by assigning servers logical priorities and using these logical priorities with RMA priority assignment in order to assign the component a range of priorities which can then be assigned to its threads. This is discussed in more detail below.

As part of admission control, the range of priorities that can be used by a component's threads can be calculated by having RT-OSGi maintain a priority ordered list of servers (according to RMA), and when a component is undergoing install admission control, the server is inserted in the correct place in the server list. The priority range assigned to the server's component can then be assigned such that the range is higher than the next lowest priority server in the list but lower than the next highest priority server in the list. This ensures that the semantics of hierarchical scheduling are respected i.e., that the threads in a higher priority server execute in preference to the threads in a lower priority server.

During priority range calculation, it is entirely possible that priorities of threads of currently deployed components may need to be remapped so that the priorities assigned to the new component and existing components' threads continue to reflect the RMA priority assignment policy. For example, if a newly installed component's server has a logical priority higher than all other servers in the system and the highest priorities are already assigned to the currently deployed server with the highest priority, then priority reassignment will be necessary. Clearly this behaviour is undesirable, but unavoidable since the RTSJ only provides fixed priority scheduling yet the component (and thus thread) set can change during run-tine (e.g., during. install/update). An example of priority reassignment is shown in Table 12.1: S_{New} is the server of a component undergoing admission control. It has a smaller period than S_2 but a larger period than S_1. According to RMA S_{New} therefore requires a priority range higher than S_2, but lower than S_1. However, there are no free priorities available within this range thus priority reassignment must take place, in this example, S_2 has its priority range reassigned.

Even if hierarchical scheduling was supported, reassignments may still be necessary for the reason mentioned above. However, such reassignments would only be at the server level which means that it is less probable (than at the thread level) that priority ranges would overlap because, typically, the number of real-time priorities that an OS provides far exceeds the potential number of components that could be deployed in RT-OSGi.

12.4.1.2 Admission Control During Updates

A component update allows a component's contents to be changed and then deployed again. As a component's thread set may change as part of an update e.g., a component may change the number of threads it creates, or the temporal specification of threads may be changed, one would assume that the component must undergo admission control. However, because component-derived updates are known prior to deploying a component, such updates can be treated as different versions of a component which the application code knows about and wishes to switch between during run-time. As the different versions of a component (update versions of a component) are known pre-deployment time, admission control is unnecessary for the component-derived update life cycle operation. Instead, as part of the component-derived install operation's admission control; resources should be reserved for the worst-case version of a component. Switching between different versions (updating) is then not problematic because all other versions will require resources equal to or less than those that were reserved on installation. Although this approach is necessary, it has the drawback that when the worst-case version of a component is not deployed, other components may fail admission control because, despite sufficient resources being available for immediate use, they are reserved for later use by the worst-case resource using versions of the currently deployed components.

12.4.2 User-Derived Life Cycle Operations

The user may wish to remove components, install new components, and update components i.e., carry out corrective, perfective, or adaptive maintenance on currently deployed components. Such life cycle operations are termed user-derived. In standard OSGi, user-derived updates are issued via an implementation dependent means; for example, one implementation of the OSGi specification may use a *Web Browser* and a *Web Server* hosted in an OSGi component in order to receive user-derived life cycle operations, whilst another implementation may provide a user interface through the command-line. In any case, the life cycle operation invocation will take place in a standard i.e., non-real-time Java thread. In RT-OSGi, such a model would provide poor performance in terms of the response time of user-derived life cycle operation requests, and as a result, the time taken to perform the associated dynamic reconfiguration may be quite long.

For RT-OSGi, the response time of dynamic reconfiguration can be improved by creating a real-time thread for user-derived life cycle operation processing.

12.4.2.1 Admission Control During Installation and Updates

User-derived installation utilises the same admission control as the admission control used in component-derived installation.

The admission control for user-derived updates differs slightly from that used in component-derived updates. In user-derived updates, the update versions of a component are unknown pre-deployment time – as such updated versions of components are typically based on the user's observations of the currently running component-set. For example, the user may notice software errors, or may identify software components which would benefit from optimisations etc. Since the updated version of a component is unknown pre-deployment time, it means that the resource requirements of a component will change in ways unknown at the time of component install, and therefore, unlike with component-derived updates, it is not possible to reserve resources for the worst-case version of a component. As a result, user-derived updates typically require the same admission control as the install life cycle operation.

However, because a user-derived update may change the thread set very little (e.g., by adding some configuration or HTML files to a component), update analysis can be carried out to determine whether it is necessary to perform the more computationally expensive install admission control. As a component is being updated, it must have been previously installed, and therefore must have server parameters generated for it as part of install admission control. Update Analysis [338] checks whether the server's parameters generated on component install are sufficient to make the threads of the updated version of the component schedulable. Only if the updated version of the component isn't schedulable with the previously generated server parameters is it necessary to perform component install admission control.

12.4.3 Removing Components

Although removing components does not require admission control, the OSGi life cycle operation responsible for removing components nevertheless needs extending for use with RT-OSGi. The two ways in which component removal needs extending are discussed below.

12.4.3.1 Controlling the Life-Time of Threads

As the OSGi Framework is Java based, and the standard Java language provides no safe way of terminating threads, OSGi does not attempt to coordinate the life cycle of threads with the life cycle of the component from which they belong. An implication of this is that, unless an application is designed with synchronising

component and thread life cycles in mind, threads may continue to execute after their component has been uninstalled from the OSGi Framework. Such "runaway" threads are a resource leak using up CPU time and memory. Furthermore, threads may cause errors if they continue execution beyond the point when their component is uninstalled. This is because they may attempt to use code and data resources of their component, which is no longer available. Such problems are not tolerable when the OSGi Framework is to be used in the development of real-time systems.

The RTSJ introduces *Asynchronous Transfer of Control* (ATC) [84] which allows a thread to cause another thread to abort its normal processing code and transition to some exception processing code. Such an asynchronous transfer of control is safe because certain methods such as synchronized methods defer the ATC until after they have finished executing. This means that before ATC takes place, any locks being held are released. ATC can, therefore, be used for the asynchronous termination of threads (ATT) when a component is uninstalled from the OSGi Framework. This ATC is enforced by integrating it into the RT-OSGi class hierarchy, as discussed in [324].

As a note, since ATC can be deferred, when a component is stopped, its threads may not terminate immediately. It is, therefore, recommended that component developers avoid using long ATC deferred methods. In the case where this is not possible, the impact of long running ATC deferred methods executing after the component has been stopped can be minimised by lowering the priority of any remaining executing threads to a background priority. They will remain at this priority until they terminate on return from an ATC deferred execution context.

12.4.3.2 Resource Reclamation

After a component is uninstalled, the resources used by the component (such as CPU reservation and priority range) must be made available for future components to use. As discussed, the resource reservation is simply a specification (i.e., C,T,D) used in schedulability analysis along with cost enforcement. Therefore, removing the reservation is easily achieved by removing the component's server from the list of servers considered as part of schedulability analysis, destroying the component's server (i.e., PGP) and replenishment timer, and by making the priority range used by the component available to other components.

12.5 Worst-Case Execution Time (WCET) Analysis

The WCET of a component's threads is required as input to CPU admission control, server parameters selection specifically. In order to calculate the WCET of threads, a measurement-based approach [435] is used.

However, there are a number of features of the OSGi Framework which make WCET calculation an even more difficult task than it already is. The general issues

stem from the fact that threads typically synchronously interact with components and services written by third parties, for which the execution time of such code is unknown to the caller offline. The result is that the calling thread's WCET is affected in an unknown way. Examples of such synchronous calls include (1) service method execution, (2) service factories, (3) component activation and deactivation, and (4) synchronous event handling. In this section, the two most significant factors affecting a thread's WCET (service method execution and synchronous event handling) are discussed further.

12.5.1 Service Execution

OSGi provides an intra-JVM service model i.e., service requesters and providers reside within the same JVM. Service-requesting threads call service methods as synchronous local method calls. As a result of the dynamic discovery aspect of service-orientation, the service implementation that a service requester binds with is unknown offline. All that a service requester compiles to offline is a service interface, which is not executable and cannot be used in a measurement-based approach to WCET calculation. This means that until service bind-time, the WCET of a thread is unknown.

The dynamic availability feature of service-orientation further complicates the WCET process; even if a service requester knows offline which service implementation it will bind to at run-time, the service implementation may at some point during run-time become unavailable and be replaced with a different implementation. An example of the effects of dynamic availability on the WCET of a service-calling thread can be seen with a "sort" service. A service requester performs WCET analysis and includes the cost of calling a Mergesort service implementation. However during run-time, the Mergesort service implementation goes offline and the service requester then binds with a Bubblesort service implementation. Clearly, the WCET of the service-calling thread will increase because Mergesort typically outperforms Bubblesort.

In order to solve the issues associated with synchronous service calls, service execution-time contracts are proposed. With such contracts, the service requester annotates the service interface with an acceptable WCET for each service method in the interface. This essentially gives each service method an execution-time budget which service implementations must abide by in order to make them compatible for use with the service requester.

The service WCET annotations are in the style of Java annotations. This enables either the *Annotation Processing Tool* (APT) or the more recent Java compilers to read the service interface WCET annotations and perform some user-defined actions. When a service interface annotation is read by the APT or compiler, a stub service implementation class should be created. For each service method annotation of the service interface, implementation methods should be generated in the corresponding stub implementation class. The stub service implementation

```
public interface PrinterService
{
  @WorstCaseExecutionTime(100)
  public void print();
}
```

Fig. 12.4 Service execution-time contract

```
public class Printer implements PrinterService
{
  public void print()
  {
    ((OSGiRTT)RealtimeThread.currentRealtimeThread()).addTime(100);
  }
}
```

Fig. 12.5 A stub service implementation class

methods should obtain a reference to a calling thread and modify a field in the thread which is used to keep track of the WCET it would have incurred had it actually executed a service implementation with the WCET specified in the contract. In this way, the stub service implementations generated as part of annotation processing can be used as part of measurement-based WCET analysis to include the WCET of calls to any required services. An example of a service execution-time contract and the corresponding stub service implementation class are shown in Figs. 12.4 and 12.5 respectively.

The use of service contracts and stub service implementations during a thread's WCET analysis still allow the thread flexibility with regard to binding to service implementations at run-time. However, it is imperative that the service implementations that a service requester binds with at run-time abide by the service contract used as part of WCET analysis. The service requester can check this by using the service contract as part of service discovery at run-time. Service providers register WCET information along with their service implementation, and before a service requester obtains a reference to a service implementation, contract matching takes place in order to ensure that the WCET of the service implementation is less than or equal to that used as part of a service requester's WCET analysis.

12.5.2 Synchronous Event Handling

The OSGi Framework is a very dynamic environment, with the component set being changed through the invocation of life cycle operations, and the service set being changed through service registration/unegistration. In order to allow threads to keep track of the current state of the Framework, OSGi offers synchronous event handling through a "publish subscribe" (pub sub) model [57]. In this model, threads subscribe to events by passing a listener object to the Framework. Threads invoking life cycle operations and threads registering/unregistering services publish events.

The event-publishing thread must then synchronously execute the listener objects of any subscribers of the event.

The synchronous event handling model discussed above may drastically affect the WCET of any event-publishing thread. The effect on the publishing thread's WCET will depend both on the total number of subscribers and on the subscribers' listener's event handling method. Clearly, even if the WCET of each subscriber's event handling listener method is small, if the number of subscribers is large, the event- publishing thread's WCET will still be greatly affected.

To solve this problem, the RTSJ's asynchronous event handling model can be applied to the OSGi Framework. The approach is to have an asynchronous event associated with every component and each service interface. Subscribing threads add an asynchronous event handler to the asynchronous event representing the entity of interest e.g., component, or service interface. Threads invoking life cycle operations or registering/removing services fire asynchronous events. Any asynchronous event handlers associated with the event fired will then be released with whatever real-time parameters they were configured with. This asynchronous event model is advantageous as the event firing is decoupled from the event handling thus the WCET of the thread firing the event is unaffected by the event handling. The handlers can be treated as sporadic activities during schedulability analysis.

12.6 Memory Management

In addition to the garbage collected heap of standard Java, the RTSJ provides other memory areas including a region-based approach to memory management called *Scoped Memory* (SM) [297]. A scoped memory area is a region of memory that has an associated reference count, which keeps track of how many real-time entities are currently using the area. When the reference count goes to zero, all of the memory associated with objects allocated in the memory region is reclaimed. One of the benefits of scoped memory is that it avoids the overheads and possible unpredictability of garbage collection (GC). However, the dynamic nature of SOA means that SM is not a general solution to memory management in RT-OSGi, as discussed below.

There are two possible approaches to using SM with RT-OSGi services. Threads can either enter SM before calling services, or the service can take responsibility for creating SM and having calling threads enter it. In the former case, `IllegalAssignmentErrors` will be thrown if a service method breaks the RTSJ memory assignment rules. In the latter case, `ScopedCycleExceptions` may be thrown depending on the scope stack of calling threads. Also, since multiple threads are able to call a service concurrently, it is necessary to synchronize access to the service's SM. Not doing so would mean that there is the potential for the SM's reference count to constantly remain above zero eventually causing memory exhaustion of the SM area. Synchronization, however, introduces a further issue in that calling

threads may experience blocking, and this must be taken into account when doing schedulability analysis.

Due to the issues with SM, along with the fact that RT-OSGi does not require the stronger timing guarantees which SM can provide over *Real-Time Garbage Collection* (RT-GC), RT-GC rather than SM is considered for use with RT-OSGi.

In real-time systems, the RT-GC must recycle memory often enough to prevent memory exhaustion without disrupting the temporal determinism of real-time threads. As the RTSJ does not specify any particular GC algorithm, current major RTSJ implementations use different GC algorithms. A brief survey of the GCs provided by current RTSJ implementations is given below along with the issues of using these GCs with dynamic environments such as RT-OSGi applications.

The aicas JamaiacaVM [373] provides both static and dynamic work-based GC. In the static work-based approach, heap allocating threads perform a fixed number of GC work units per allocation block. The number of blocks is determined based on a static analysis of the worst-case live memory of the application. This cannot be changed at run-time and, consequently, is not appropriate for a dynamically changing environment as the work carried out may be insufficient to prevent memory exhaustion.

In the dynamic approach to work-based GC provided by aicas, the number of GC work units per allocation is varied during run-time based on the current worst-case live memory. As a result, any statically calculated threads' WCET will not be correct, which may cause the system to become unschedulable. In order to prevent this situation, each time the application is reconfigured, it would be necessary to determine what the new worst-case number of GC work units per allocation block would be, then perform WCET analysis for all threads, and finally perform schedulability analysis. This is impractical in a system such as RT-OSGi, where reconfiguration is expected to happen on a relatively frequent basis.

The GC provided by IBM WebSphere [23] RT is time-based. Reconfiguring the pace of GC is easily achieved in time-based collectors by modifying the computation time, period, and deadline of the GC thread. Such an approach is well suited to dynamically reconfigurable RT-OSGi applications. Unfortunately, IBM provides no facilities for changing the parameters of their time-based GC and so the IBM GC will not be discussed further.

Sun Java RTS provide a Henriksson-style GC [191] that can be dynamically reconfigured. The default behaviour of this JVM is to have the GC thread run at a background priority. If the free memory available to an application drops below a user-defined threshold, the priority of the GC is increased from a background level to a "boosted" user-configurable level. In this way, the rate of GC is temporarily increased until the GC completes one or more cycles and increases the free memory above the safe level threshold, it then returns to executing at a background priority level. This model is depicted in Fig. 12.6 [400].

From Fig. 12.6, it can be seen that the threads that have a priority above the "RTGC Normal Priority" level but below the RTGC Boosted Priority level (termed non-critical in Java RTS) may be subjected to an unbounded amount of interference from the RT-GC thread whilst it executes at its boosted priority level for as long as

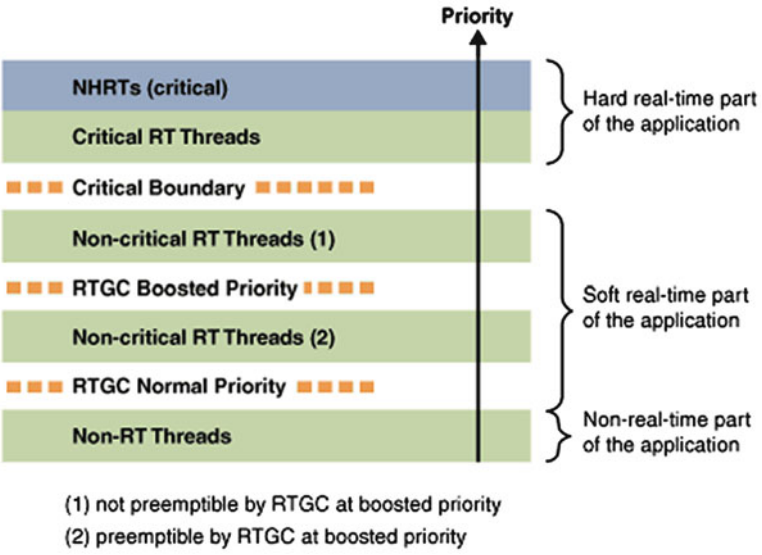


Fig. 12.6 Sun Java RTS model of garbage collection

it takes to return the free level of memory above the unsafe threshold. Therefore, such application threads cannot have real-time guarantees; i.e., to such threads GC appears to be stop-the-world. Threads with a priority above the "RTGC Boosted Priority" but below the "Critical Boundary" will not be pre-empted by the GC thread but may have to perform demand GC if memory becomes exhausted. Finally threads with a priority above the "Critical Boundary" will not be pre-empted by the GC and will not have to perform demand GC because they have an area of the heap exclusively reserved.

In order for the Java RTS GC model to function correctly, i.e., to prevent memory exhaustion, the GC must be able to keep pace with the rate of garbage creation. This typically means dividing the application threads so that the majority are non-critical and can be pre-empted by the GC. This allows the GC thread to be given a large fraction of the CPU utilisation when the free memory level drops below the safe threshold. If the majority of application threads have a priority above the "RTGC Boosted Priority" level then, even when free memory is low, the GC would still be running at background priority as threads with a priority above the "RTGC Boosted Priority" level cannot be pre-empted by the GC. In this case, the GC would be less effective at preventing memory exhaustion.

Clearly, the Sun Java RTS GC model is only useful when an application has a small number of critical threads. However, many applications require that the majority of their threads not suffer unbounded interference from the GC thread, and thus the default behaviour of Sun Java RTS is unsuitable. A more suitable approach to GC would be to have the GC keep pace with garbage creation such that the free memory should never fall below the free memory threshold, and thus there would

be no need to have the GC monopolise the CPU by assigning the GC thread a large CPU utilisation in an attempt to frantically "catch up" with the application's garbage creation. Essentially, a preventative approach to reaching low free memory levels is better than Sun Java RTS GC's default curative approach.

We, therefore, propose that since the Sun Java RTS is dynamically reconfigurable in the sense that its priority can be manipulated during run-time, it is possible to implement a pacing garbage collector by using analysis to determine the pace of GC based on the rate of garbage creation. As there are a number of research papers dedicated to the topic of configuring a time-based GC, we implement time-based GC using Sun Java RTS's GC, using accompanying analysis to determine in what way the GC parameters need to be changed based on the changing application architecture.

Time-based GC can be implemented on Sun Java RTS's GC by having a GC control thread which can modify the GC thread's priority so as to simulate the GC thread running with a computation time per period (as happens in time-based GC). The GC thread is assigned a background priority (thus appearing inactive), and the GC control thread is assigned whatever period the GC thread is required to run at, and as the period is typically small, according to Rate Monotonic Analysis, the GC control thread will run at a very high priority.

As the GC control thread is released, it periodically raises the priority of the GC thread to a priority higher than all application threads. The GC control thread can manipulate the amount of time that the GC thread is permitted to run at high priority during its period by using a timer. Once the GC thread has used its allocated computation time per period (i.e., the timer expires), the GC control thread will then lower the GC thread's priority back to its background level.

The actual online reconfiguration analysis to be used with RT-OSGi and the modified Sun Java RTS GC is a modified version of Kim's analysis [236]. The analysis is performed at the same time as CPU admission control (discussed in Sect. 12.4). In the GC reconfiguration analysis, it is necessary to determine new C,T,D parameters for the GC thread and to check whether there is enough free memory in the system to support the changing application configuration.

This reconfiguration analysis is now discussed in more detail. The notation used in the reconfiguration analysis is given in Table 12.2.

12.6.1 Estimating GC Work

At admission control time, the increase in GC work must be estimated. GC work includes the cost of reference traversal (an upper bound on the number of reference fields to scan (in Words) multiplied by the execution-time required for the GC to scan one word (c_1)), the cost of object evacuation (the maximum live memory (in Bytes) multiplied by the cost of evacuating one Byte (c_2)), and the cost of memory initialisation (an upper bound on the amount of memory to be initialised (in Bytes) multiplied by the cost of initialising one Byte (c_3)). The equation used to derive an

Table 12.2 Reconfiguration analysis notations

Symbol	Meaning
C_i	Thread i's computation time (ms)
T_i	Thread i's period (ms)
D_i	Thread i's deadline (ms)
A_i	Thread i's memory allocation per period (MB)
R	Size of root-set (bytes)
τ_{GC}	Time taken to complete a GC cycle
$\langle\rangle$	Set of tasks
n	Number of application threads
H	Heap size (H/2) = semi space size (MB)
T_{GC}	Period of GC thread's controller thread (ms)
C_{GC}	Budget at which GC thread can run at high priority for (ms)
W_{GC}	The amount of GC work (ms)
M	Application memory requirement

$$W_{GC} = c_1\left(\frac{R + \sum_{i=1}^{n} A_i}{sizeof(word)}\right) + c_2 \sum_{i-1}^{n} A_i + c_3\left(\frac{H}{2}\right)$$

Fig. 12.7 Estimating GC work

estimate of GC work (W_{GC}) is given in Fig. 12.7. As a note, it is the scanning and copying phases of garbage collection which are the major factors in determining the GC work. The time taken in these phases is dependent on the amount of an application's live memory and not on the amount of dead (or garbage) memory.

In the equation in Fig. 12.7, it is pessimistically assumed that a thread's live memory equals the memory allocated by the thread (A_i). The implication of this is that the amount of GC work will be overestimated and, thus, the GC will be assigned more CPU-time than is necessary to complete a garbage collection cycle. The reason for this pessimistic assumption is because lifetime analysis (which is used to determine whether an object is live or not) is well known to be a very difficult task for developers to perform.

In Fig. 12.7, it is also assumed for simplicity that the total live memory allocated by the application is the sum of the memory allocated in one release of each application thread, i.e., it is assumed that after the first period, any memory that a thread allocates is garbage. The result of this is that the total live memory allocated by the application does not increase regardless of the number of releases of threads during a GC cycle. It is important that the total memory allocated by an application does not grow beyond that specified in Fig. 12.7. If the total live memory allocated by the application does increase beyond the value used in Fig. 12.7, the GC reconfiguration analysis becomes invalidated because the GC work, cycle length, and associated application free memory requirement will increase beyond what

$$C_{GC} = \max\left\{ x \mid \forall t \in \tau_{\{i=0\ldots n\}} : \left\lceil \frac{T_i}{T_{GC}} \right\rceil x + \sum_{j=1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j \leq D_i \right\}$$

Fig. 12.8 Calculating C_{GC}

was estimated. As a result, the application may have falsely passed the memory admission control and memory exhaustion may occur.

From the discussion above, it is clear that allowing threads to allocate live memory in multiple periods whilst using Fig. 12.7 may cause memory exhaustion because the application's memory allocation requirement will be greater than that assumed in Fig. 12.7. Therefore if it is a requirement for the application's threads to allocate live in more than just their first release, Fig. 12.7 can easily be modified to accommodate this. This is achieved by replacing the "sum of one release of each application thread live memory" term in Fig. 12.7 with the worst-case live memory of the application in a steady state (i.e., whatever the live memory will be once the application threads stop allocating live memory).

12.6.2 Calculating GC Parameters

As the behaviour of the Sun Java RTS's GC has been modified to provide time-based GC, the only parameters that are required are computation time, period, and deadline.

The GC budget (C_{GC}) is calculated by starting with a low base value and iteratively increasing it until the GC utilisation (cost divided by period) is so large as to cause excessive interference to application threads causing the system to become unschedulable. C_{GC} can be determined using the equation in Fig. 12.8. Note that in Fig. 12.8, priority 1 is the highest.

The GC period/deadline (T_{GC}) is assigned to be equal to the application thread with the smallest period, and thus, the GC thread will be given highest priority according to RMA priority assignment. The reason why the GC must run at the highest priority is because the adapted behaviour of the Sun Java RTS GC is to provide time-based GC (as explained earlier in this section), which typically requires the GC to run at the highest priority. Therefore the GC reconfiguration analysis presented here must also assume this.

According to Rate Monotonic priority assignment, the GC's period could be arbitrarily smaller than the application thread with the smallest period, and still have a higher priority than any application threads. Therefore, ideally, in order to compute the GC's period, the period of the GC would be iteratively decreased (much like the GC cost is iteratively increased in Fig. 12.8). The result of this combined with Fig. 12.8 would essentially find the maximum CPU utilisation that can be assigned

$$R_{GC} = T_{GC} + \frac{W_{GC}}{C_{GC}} * T_{GC}$$

Fig. 12.9 Determining the GC cycle time

to the GC without breaking application schedulability. However, as discussed further below, it would be too time consuming to try and find both the minimum period and maximum cost (i.e., the maximum CPU utilisation) for the GC in dynamic environments such as RT-OSGi.

By iteratively increasing the GC cost and keeping the GC period static, it is fairly quick to compute an approximation of the maximum CPU utilisation which can be assigned to the GC thread whilst keeping the application schedulable.

As a note, regardless of whether the maximum or an approximation of the maximum GC CPU utilisation is calculated, the CPU utilisation may be so small as to be ineffective. For example, if the system is heavily loaded with application components, there may be so little CPU time left over for the GC thread such that the context switch and other overheads may have an overruling effect. It is therefore necessary to check that the GC parameters computed are above what is considered an effective threshold i.e., the C_{GC} and T_{GC} parameters calculated act as an acceptance test for memory admission control.

12.6.3 Estimating GC Cycle Time

The GC cycle time is the time it takes the GC to complete all of its work, i.e., the work estimated using the equation in Fig. 12.7. During a GC period (T_{GC}), the GC thread can only perform an amount of work equal to the C_{GC} (calculated using Fig. 12.8). Therefore, a number of GC periods will be required to complete a GC cycle. The amount of time it takes (The GC response time – R_{GC}) can be determined using the equation in Fig. 12.9.

12.6.4 Determining the Free Memory Requirement of the Application

It is important to estimate the amount of memory allocated by application threads during a GC cycle, i.e., during the time-frame calculated in Fig. 12.9. If the free memory in the system is less than the application's memory requirements during a GC cycle, it is inevitable that memory exhaustion will occur. The reason for this is because, in the class of GCs known as Copying GCs (as with Sun Java RTS), garbage memory is not made available for use again until after the GC has completed its cycle. Therefore, it is known that the free memory available in the system will not increase until after a GC cycle has completed.

$$M = 2\left(\sum_{i=1}^{n}\left(\left(\left\lceil \frac{R_{GC}}{T_i} \right\rceil + 1\right)A_i\right) + \sum_{i=1}^{n} A_i\right)$$

Fig. 12.10 Determining free memory requirement of the application

Figure 12.10 is used as an acceptance test for memory admission control during application reconfiguration. It gives a safe free-memory threshold – if there is at least M free-memory in the system, the application is guaranteed to not experience garbage-related memory exhaustion, and the application can therefore be reconfigured.

If, on the other hand, there is insufficient free memory, the application reconfiguration must be rejected because the maximum amount of CPU-time that the GC can be assigned without making the application unschedulable is insufficient to recycle memory at the required rate. Note how this behaviour differs from that of the default behaviour of Sun Java RTS. The Sun Java RTS GC, without a time-based behaviour and the associated reconfiguration analysis, at admission control-time would always permit a RT-OSGi application to be reconfigured even if it means that the GC would eventually make the application unschedulable (by running at a priority higher than one or more application threads for as long as necessary to return the level of free memory above the safe threshold).

If application reconfiguration passes the free memory acceptance test and the GC parameters acceptance test, then the GC can finally be reconfigured to run at the new pace dictated by the parameters previously calculated. More specifically, on the time-based modifications to Sun Java RTS's GC, the GC controller thread's period is set to T_{GC}, and the time for which the GC controller thread permits the GC thread to run at high priority for is set to C_{GC}.

As a note, in Fig. 12.10, the application's memory requirement is calculated as the worst-case live memory plus garbage allocation in a GC cycle of worst-case length multiplied by two. The reason for this is that at end of a GC cycle, before the semispace flip, both semispaces will have copies of live memory and a GC cycle's worth of garbage allocation.

12.6.5 Memory Allocation Enforcement

If threads allocate more memory than the amount used for calculating the GC work (Fig. 12.7), then the rest of the GC reconfiguration analysis discussed above becomes invalidated. More specifically, the GC cycle length will be longer than was estimated in Fig. 12.9, and as a result, the free memory requirement of the application will be greater than that estimated in Fig. 12.10. The implication of this is that components may be erroneously admitted into the system because the actual free memory will be less than the amount used in the acceptance test for component

deployment. In order to prevent this situation, the memory allocation (as specified in Fig. 12.7) of threads per period is enforced.

To achieve memory allocation enforcement, the memory allocation monitoring of the RTSJ is used. In memory allocation monitoring, the application developer places a bound on the total memory allocation that can be made by a thread during its lifetime. If the allocation bound is exceeded, an exception is thrown. This behaviour is extended for RT-OSGi such that it appears that the memory bound is per period rather than a lifetime total. For example, this is achieved by periodically increasing the lifetime total permitted allocation of a thread by the amount of allocation per period. Also, the memory allocation overrun handling is extended by creating subclasses of the RTSJ classes which implement the Schedulable Object interface specifically for RT-OSGi. Upon detecting a memory allocation overrun, three models of handling the overrun are proposed:

1. Hard enforcement – provides the application with no means of recovery and simply blocks the thread until its next period without notification
2. Soft enforcement – fires an asynchronous "overrun memory budget" event. The application's event handler is released which allows the application to attempt to recover from the overrun
3. Hybrid enforcement – threads have two memory budgets, a soft and a hard budget.

Overrunning the soft budget will cause the event handling mechanism to be used. Continuing to allocate memory despite the soft budget overrun may cause a hard budget overrun. Overrunning the hard budget causes the thread to be blocked for its next period

12.7 Overheads of RT-OSGi

There are a number of overheads introduced by extending the OSGi Framework with Real-Time capabilities These include:

1. CPU Admission Control – the RT-OSGi life cycle operations have a longer execution time than the standard OSGi life cycle operations because they have been augmented with admission control. For example, the component-derived install operation must perform server parameter selection, response time analysis, and priority range assignment. Regarding the percentage increase in execution-time of the "install" operation when augmented with CPU admission control, it is quite high at 47.6%. However, the measured execution-time overhead in our prototype is 21.2 ms, which is acceptable for the types of real-time applications that RT-OSGi targets.
2. User-Derived Life Cycle Operation Processing – In order to guarantee user-derived life cycle operations a minimum level of service, one or more servers are employed that reserve a quantity of CPU time for processing such requests. This RT-OSGi approach of using servers will reduce the amount of CPU time that is

available for use by the application. However, the actual overhead is application dependent, if a user requires a high level of responsiveness to user-derived life cycle operations, the server will be assigned a large share of the CPU.

3. Cost Enforcement – Whether at a high or low level, the provision of cost enforcement entails overheads. If the OS provides cost enforcement, it must periodically reduce the currently executing thread's remaining CPU budget and re-evaluate whether the thread is still the most eligible for execution, i.e., whether or not the thread is the highest priority and whether it has any CPU budget remaining. If a thread's remaining budget equals zero, it should be descheduled and should not be made eligible for execution until its budget has been replenished. Clearly there is overhead in the OS scheduler keeping track of each thread's CPU budget.

   If the cost enforcement is provided by the Java Virtual Machine (JVM), there is likely to be even greater overhead than the overhead incurred by providing cost enforcement at the OS level. For example, it may be necessary to create a high priority thread to keep track of the amount of CPU-budget available for each application thread, i.e., the high priority thread would perform the cost monitoring of tasks that would otherwise have been performed by the OS scheduler. In addition, there is also overhead in handling cost overruns. For example, upon detecting a cost-overrun, the cost-monitoring thread may notify the JVM by sending a signal to the JVM's process. The JVM may then in turn convey the cost-overrun to the application by firing a "Cost-Overrun" asynchronous event and releasing any associated handlers created by the application. Furthermore, cost-overrun handling provides overhead in the sense that the application-defined "Cost-Overrun" handlers must execute at a higher priority level than the application threads for which they are attempting to provide cost-enforcement for.
4. RT-GC Reconfiguration Analysis (Memory Admission Control) – The GC reconfiguration analysis is performed when life cycle operations take place. Such operations will, therefore, have a longer execution-time; keeping in mind that the execution-time for RT-OSGi life cycle operations is already larger than their standard OSGi counterparts due to the CPU-admission control discussed above.
5. Application-Level Time-Based GC – Providing Time-Based GC on top of the standard Java RTS involves the overhead of creating a high priority "GC Control" thread which manipulates the GC thread's priority to have it provide time-based GC.

12.8 Related Work

In [9], Rodrigues Americo provides a discussion of a subset of the issues that are solved in this chapter, that is, a discussion of some of the issues of integrating the OSGi Framework and RTSJ. It also gives a detailed discussion of the underlying

concepts of OSGi i.e., SOA and CBSE, as well as details about real-time systems and the RTSJ.

In [243], Kung et al., describe ideas for providing cost enforcement in the OSGi Framework, but at the JVM level. Like our work, they wish to provide resource guarantees to each component. However, unlike the work by Kung et al., we chose to modify the OSGi Framework to provide a suitable real-time environment.

Miettinen et al. [274] modified the OSGi Framework so as to enable the monitoring of a component's resource consumption. Essentially, they add all of the threads in a component to a thread group, and provide a monitoring agent to collect resource usage information. This work is similar to ours in that they are providing cost monitoring at the component level, however the motivation for our works differ. Miettinen et al. are interested in improving the performance of standard Java applications. Their monitoring tools are intended to be used during testing so as to identify inefficient components before the application is finally deployed. Since they are not using the RTSJ, they do not attempt to provide cost enforcement nor temporal isolation among components.

In [180], Gui et al., looked at using the OSGi Framework in the development of real-time systems. The motivation for their work is the same as this work, to develop dynamically reconfigurable real-time systems. They propose a real-time component model over the OSGi Framework. In their model, XML is used by component developers to declaratively configure real-time tasks. The functionality of real-time tasks is developed in native code, and non-real-time tasks are developed using Java. This approach gives a "split" architecture, real-time tasks are under the control of the real-time operating system, and non-real-time tasks run in the OSGi Framework. This is different from our work. Instead of using native code and a real-time component model, we are using the RTSJ to write real-time components.

12.9 Conclusions

In this chapter, the OSGi Framework has been introduced and motivation provided for using OSGi with the RTSJ in order to develop real-time systems. A prototype real-time version of the OSGi Framework (RT-OSGi) has been implemented (see [323]) and applied to a case study (see [326]). The prototype has been produced by augmenting the Apache Felix OSGi Framework implementation with the following features:

- Temporal isolation – application level cost enforcement and application level hierarchical scheduling;
- CPU admission control – server parameters selection, efficient RTA for component groups, and asynchronous thread termination through asynchronous transfer of control;

- WCET calculation – service execution-time contracts with associated service discovery, and asynchronous event handling;
- Memory management – reconfigurable time-based GC, GC reconfiguration analysis, memory admission control and memory allocation enforcement.

In conclusion, RT-OSGi, with the above features, provides a suitable environment for guaranteeing the timing requirements of application components written in the RTSJ. Such RT-OSGi applications can be reconfigured i.e., new real-time components can be added, and currently deployed components can be updated or removed whilst maintaining application availability. This ability for dynamic reconfiguration allows for applications to be built with high availability requirements which are also resource efficient.

In terms of future work, there are many directions in which RT-OSG could be extended. For example, the current RT-OSGi implementation is for uniprocessor systems, but as multicore/multiprocessor systems are now becoming widespread, it would be beneficial to be able to deploy RT-OSGi applications on such platforms. However, there are a number of issues with running the current implementation of RT-OSGi on a multiprocessor platform. Issues include cost enforcement and its affect on schedulability analysis, the schedulability analysis algorithm used in admission control, and the RT-GC reconfiguration analysis.

Acknowledgements This work is supported by the UK EPSRC and IBM through case award 0700092X.

Chapter 13
JavaES, a Flexible Java Framework for Embedded Systems

Juan Antonio Holgado-Terriza and Jaime Viúdez-Aivar

Abstract Today, several strategies can be adopted in order to build a complete embedded application, including the use of software and hardware components with a reusable base to satisfy the increasing demands and requirements of actual embedded systems with shorter time to market. A new flexible Java framework is presented in this chapter for the development and deployment of functional embedded systems. It may be adapted to different existing embedded target devices, providing at the same time a common programming environment based on Java with well-defined semantics. This framework makes possible the portability of applications among embedded targets without any re-implementation according to the WORA principle – write once, run anywhere – and gives also a reliable and sound platform that may extend the capabilities of an embedded target by the integration of hardware components without requiring the implementation of any device driver.

13.1 Introduction

The development of embedded systems requires a good integration between the hardware and software components in order to achieve a product with a low unit cost, since it can be produced in large quantities. Java has drawn attention of embedded developers, since it can simplify the design, development, testing, and maintenance of embedded systems, especially in the case of larger applications [18, 125, 386]. In addition, the independence of machine architecture gives the Java programs the capability to run on practically any platform according to WORA principle – write once, run anywhere – whenever an implementation of the

J.A. Holgado-Terriza (✉) • J. Viúdez-Aivar
Software Engineering Department, University of Granada, Granada 18071, Spain
e-mail: jholgado@ugr.es; jviudez@ugr.es

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_13,

execution platform, *the Java Virtual Machine* (JVM), is present. Many commercial [4, 147, 218, 266, 287, 291, 402] and academic proposals [2, 352, 363, 378] are currently available to carry out the development of embedded applications based on the Java programming language.

But Java has some disadvantages and drawbacks that should be considered in the programming of embedded systems. First, differences on performance can be found depending on whether the JVM is implemented on top of an *Operating System* (OS), without an OS, directly on native code such as Squawk [378] or by means of a Java Processor such as JOP [352] or aJile [4]. The execution model, garbage collector, support for multithreading, or memory model are some of the issues that determines the overall performance on the execution of Java programs [125, 234, 352, 386].

Second, over the years, a variety of *Java Configurations* (JC) applicable to the embedded domain have been defined and released. These address the specific characteristics of embedded applications based on the *Java Standard Edition* (Java SE) or, more usually, on the *Java Micro Edition* (Java ME), providing a customized set of Java libraries coupled with the JVM. Some JCs are very popular, for example, *Connected Limit Device Configuration* (CLDC) for small, resource-constrained devices; others are near to the end of their lives or are based on different Java platform versions, for example, *EmbeddedJava* or *PersonalJava*. Recently, new configurations have been defined, for example, *JavaTV* for TV set-top boxes. The large number of *Java Configurations* for the embedded domain, is because embedded environments impose rigid constraints in performance, power consumption, costs or memory resources. Consequently, the only way to satisfy those constraints is by applying some limitations on the Java programming language, the JVM or the supported classes libraries, which in turn may increase the differences and incompatibility among them (e.g., CDC vs CLDC). Furthermore, the JC may include additional selected Java libraries conforming to Java standard specifications from *Java Community Process* (JCP) [224] to fulfill the specific features of an embedded target; e.g., the *Information Module Profile* (JSR 195) adds to CLDC specific libraries for networked devices with very limited displays.

Third, Java does not provide access to low-level facilities and hardware abstractions as pointers, memory address, interrupt management, I/O devices, in order not to break the Java's security model [117]. An extension, *Java Native Interface* (JNI) [250], can be used to run a portion of code written in other programming languages within the Java infrastructure for accessing and controlling the hardware by the invocation of Java native methods that include external procedure calls. However, this mechanism is dependent on the execution environment and the mismatch between languages could cause a major impact on performance [117]. The lack of low-level facilities is resolved by most Java embedded device manufacturers by implementing a proprietary interface to native methods plus adding a new library specific to the target capabilities.

Fourth, the Java platform has poor real-time behavior and this feature is necessary in many embedded applications. The Java platform includes some dynamic features such as garbage collector, dynamic binding, dynamic class loading, or

reflection which are unpredictable in nature. In addition, the Java semantics for multi-threading and real-time execution are weak and underspecified [425]. Diverse solutions have been provided by implementing a JVM without garbage collection [249], or to run on top of a real-time operating system [147, 218] or on bare metal [4, 287]. However, the above solutions do not resolve the lack of predictability induced by the non-bounded latencies in context switching or interrupt handling and the possible priority inversion issues in resource sharing [425]. The more obvious alternatives are to adopt *Real Time Specification Java* (RTSJ) [65] or *Safety Critical Java* (SCJ) [192] for the JVM implementation [301].

Embedded manufacturers solve some of the Java weaknesses developing their own specific JVM implementations that satisfy a concrete Java Configuration (e.g., CDC) and, in addition, include a proprietary library to fulfil the software development environment of their embedded targets providing an API to support specific hardware components [266, 291]. The problem is that we have found different models to perform the same function; i.e., the timers are implemented in multiple ways without satisfying the same semantics.

In this work, we have developed a new flexible framework based on Java to enable the portability and adaptability of embedded appliances among different heterogeneous embedded targets. In our opinion, two strategies are possible in order to achieve this goal. One approach would require the design of a new JVM with a suitable API from scratch for specific hardware architecture, including a predictable scheduler model, memory model, I/O management and interrupt handling. This strategy requires a high effort for the development of a complete solution that is valid for different hardware architectures. SimpleRTJ [337] is an example of a clean room JVM implementation that has been specifically designed to run on devices with scarce memory with 68HC11, i386, H8/300, 68K and M. Core processors. The other approach involves building a new software infrastructure that acts as an intermediate software layer between the application and the execution environment. The development of an extra software layer promotes the definition of a common API, but also causes a penalty on performance that can affect the reliability of applications and increase memory demands. There are several Java libraries and frameworks based on this scheme available for different *Java Configurations*. Javolution [223] is an example of the above kind of library that provides a high-performance implementation of standard Java packages such as java.util collections, making the execution more time-predictable.

The proposed framework in this work combines both strategies described above but it is inspired by platform based design [342]. According to the above approach the embedded device is considered an operative hardware-software entity at a suitable abstraction level for easing the applications development. Instead of building a JVM from scratch for a particular embedded target to enable Java, we conceptualize an abstract model of an overall embedded system that includes the existent hardware components and software infrastructure (JVM among others) for that particular embedded target. At a high level, the abstract model provides the abstractions, structures, properties, mechanisms and basic primitives to assist

the development of embedded applications, as in the second strategy described above. At a low level, the abstract model should be implemented by using the underlying JVM plus the additional libraries including the implementation of low-level facilities in some cases, to assure the system performance, as in the first above strategy. In addition, at low-level it may be also necessary to extend the hardware capabilities to keep guaranteeing the integrity of the abstract model of the embedded system.

This chapter addresses a new general approach to manage the heterogeneity existing on Java embedded devices in a flexible way. In spite of the claimed portability from the WORA principle, Java today is still difficult to apply widely in the embedded domain because of the current differences in *Java Configurations*, Java platform versions, thread models, memory models and JVM implementations, making the portability a really difficult goal to achieve. A new flexible framework based on Java, *Java Embedded System* (JavaES), has been developed for a wide range of heterogeneous resource-constrained embedded devices or microcontrollers to show the benefits of our general approach, covering two main aspects:

- A general flexible approach for building customized, optimized and compact software adaptations to the execution environment of the embedded target devices, according to developer's needs, by the definition of an abstract model of the embedded system that must also guarantee the application portability.
- A general programming model based on a non-standard API which is valid on any *Java Configuration*, and which makes easier the handling of hardware abstractions, the hardware interface to hardware devices, and a transparent extension of the hardware base of any embedded target device by selecting the best combination (hardware base plus hardware components).

The framework has successfully been applied for the programming of embedded devices on a home-automation platform [329], on a wireless sensor network [13] and for instruments of a physics laboratory [207].

13.1.1 Chapter Organization

Section 13.2 of the chapter gives an overview of the framework, its features and how the flexibility can be achieved at different levels. Section 13.3 presents the main components of the JavaES framework and the software design flow of an application. Section 13.4 describes the generation process to obtain a customizable runtime environment for a concrete embedded target. Section 13.5 shows the building and deployment process of an application. Section 13.6 presents the JavaES architecture and how the abstraction layer is implemented. Section 13.7 details an application example in order to verify the benefits that the JavaES framework could provide to developers. Section 13.8 is focused on a study of the incurred overhead. Finally, we present the conclusions and some research directions.

13.2 JavaES Framework

JavaES (Java Embedded System) is a framework based on Java, which provides an adaptable, flexible and sound platform for the development of applications on heterogeneous embedded target devices, especially for microcontrollers and resource constrained embedded systems. This platform provides an abstraction of an overall embedded system, including hardware and software components. In addition, a programming model with a well-defined semantics is given for the Java development on the limited resources of an embedded system, abstracting the complexity and heterogeneity that might exist on the physical embedded devices from different execution environments, hardware architectures and manufacturers. For instance, JavaES affords mechanisms and high-level facilities that makes possible the safe, reliable access and management of the underlying hardware level (I/O ports, timers, counters, PWM signal, or memory).

The machine independence is achieved by virtualizing both, system hardware resources (processing capability, memory size, or peripheral devices) and available software infrastructures (OS, JVM or drivers) in an abstraction layer that decouples the Java applications with respect to the physical embedded target, preserving the code portability at the same time. This in turn allows developers to control the hardware resources of the platform, and allows them to modify, update or extend their hardware capabilities according to the needs of the application. It allows not only the integration of software components, but almost any peripheral device at the hardware level, customizing in this way the embedded system without programming any additional driver or native library. Therefore, JavaES can build a compact, flexible and lightweight runtime environment compatible with a variety of *Java Configurations* (JavaSE, JavaME-CDC, JavaME, CLDC, PersonalJava or EmbeddedJava), which can be adapted to the specific embedded target according to the application needs.

A list of embedded targets currently supported by the JavaES framework is shown in the Table 13.1. Some of the most important features and characteristics included in JavaES are the following:

- Full support for several embedded targets independently of the supported *Java Configuration*.
- A common hardware abstraction layer for the access and management of hardware components in a transparent and homogeneous way, regardless of the specific embedded target.
- A built-in hardware extension mechanism to integrate new hardware components without JVM reimplementation.
- Memory management schemes based on heap and memory regions protected from the garbage collector.
- Flexible access to synchronous serial buses such as I2C, SPI or 1-Wire.
- Efficient programming schemes for communication protocols based on connection-oriented and connectionless communications.
- An asynchronous events model for handling interrupts and software events.

Table 13.1 Features and characteristics of embedded target devices supported by JavaES

	Processor	Memory	Connectivity support	Java execution model	Java standard compliance	Real-time support	JavaES support
Javelin stamp [291]	RISC Ubicom SX48AC (25 MHz)	32 kB RAM 32 kB EEPROM	16 GPIOs, PWM, RS-232	JVM with AOT compiler on the bare-metal	Non-standard compliant Java 1.2 subset	Real time capable (low-level programming) no schedulers	Partial
SNAP [215]	Imsys Cjip (66.67 MHz)	1 kB on-chip 8 MB RAM 2 MB Flash	8 I/O pins, SPI, I2C, 1-Wire, CAN, Ethernet, RS-232	JVM with JIT compiler with partial on-chip implementation	JavaME CLDC 1.1	Real-time OS deterministic timers support	Complete
IM3910 [218]	IM3910 (167 MHz max.)	32 MB RAM 8 MB Flash	30 GPIOs, 8 ADCs, 2 DACs, SPI, I2C, CAN, Ethernet, RS-232, JTAG	JVM with JIT compiler and partial on-chip implementation	JavaME CLDC 1.0 Subset JDK 1.1.8 javax.comm	Real-time OS deterministic timers support	Complete
EC200 (EJC) [147]	Cirrus logic EP7312 (+ slave PIC 16LF872) (74 MHz)	8 kB on-chip Up to 64 MB RAM 16 MB Flash	23 GPIOs, 3 ADCs, SPI, I2C, 1-Wire, Ethernet, RS-232, RS-485	JVM with JIT compiler	Personal Java compliant (Java 1.1.8-based)	Real-time OS, thread RTOS-Java priority mapping support	Complete

SC-210 (EJC) [147]	Cirrus logic EP7312 (+ slave PIC 16LF872) (74 MHz)	8 kB on-chip Up to 64 MB RAM 16 MB Flash	50 GPIOs, 3 ADCs, SPI, I2C, 1-Wire, Ethernet, RS-232, RS-485, 5.7″ color LCD	JVM with JIT compiler	Personal Java compliant (Java 1.1.8-based)	Real-time OS, thread RTOS-Java priority mapping support	Complete
JStik (aJile) [403]	aJile aJ-100 (100 MHz)	48 kB on-chip 2 MB RAM 4 MB Flash	23 GPIOs, SPI, I2C, 1-Wire, CAN, Ethernet, RS-232	Direct on-chip Java bytecodes execution (Java processor)	JavaME CLDC 1.0	Real-time threads and scheduling schemes (Piano roll, periodic threads) support	Complete
JNIB (aJile) [4]	aJile aJ-100 (100 MHz)	48 kB on-chip 16 MB RAM 16 MB Flash	8 GPIOs, SPI, I2C, 1-Wire, CAN, Ethernet, RS-232, USB, JTAG	Direct on-chip Java bytecodes execution (Java processor)	JavaME CLDC 1.1 CDC 1.1	Real-time threads and scheduling schemes (Piano roll, periodic threads) support	Complete
TSTIK (TINI) [402]	Intel 8051 (75 MHz)	1 MB RAM 2 MB Flash	24 GPIOs, SPI, I2C, 1-Wire, CAN, Ethernet	JVM with AOT compiler	JDK 1.1.8 Subset JDK 1.3 Subset JDK 1.4	Real-time OS, but no Java real-time support.	Complete

(continued)

Table 13.1 (continued)

	Processor	Memory	Connectivity support	Java execution model	Java standard compliance	Real-time support	JavaES support
DS80C410 (TINI) [267]	Intel 8051 (75 MHz)	1 MB RAM 64 kB ROM 1 MB Flash	64 GPIOs, 1-Wire, CAN, Ethernet	JVM with AOT compiler	JDK 1.1.8 Subset JDK 1.3 Subset JDK 1.4	Real-time OS, but no Java real-time support.	Complete
SunSpot [287]	AT91RM9200 (180 MHz) + Avr Atmega88 (20 MHz)	512 kB RAM 4 MB Flash	5 GPIOs, 6 ADCs, 4 output pins, 2G/6G 3D accelerometer, temperature, light sensor, 8 tri-color leds, I2C (TWI), RS-232, USB, IEEE 802.15.4	JVM with AOT compiler on the bare-metal	JavaME CLDC 1.1 MIDP 1.0	No RTOS 3 high precision hardware counter	Complete
TC65 [266]	ARM7 Core (52 MHz) + Blackfin DSP (208 MHz)	400 kB RAM 1.7 MB Flash	10 GPIOs, 2 ADCs, 1 DAC, I2C, SPI, RS-232, USB, GSM/EGPRS	JVM with JIT compiler	JavaME CLDC 1.1 HI IMP-NG	Operative system without real-time support	Complete

- Specific primitives and mechanisms to handle time services such as watchdogs, timers and system clocks.
- A common remote monitor for system debugging.
- A unified toolchain to compile, build and deploy applications on any supported embedded target.

13.2.1 The Flexibility Levels of JavaES Framework

Our starting point is a heterogeneous computing device not only with respect to hardware architecture, but also with respect to execution environment, deployment procedures, different programming schemes and different debug mechanisms. The question is how we can deal with this heterogeneous domain and how we can unify the programming, deployment and maintenance of these devices within a common method. In our opinion, the key factor should be the flexibility. JavaES is able to manage the flexibility at different levels:

- *Programming Level*. There are multiple choices for programming embedded applications. The developer should have many alternative abstractions and programming facilities with the corresponding reliability and efficiency level, combining the high-performance achieved by hardware components assembly with the flexibility of software implementations. Java comprises a rich built-in API in any *Java Configuration* to address important issues such as concurrency, networking or file management at a high-level in the software domain, but it fails to provide low-level facilities to interact with the hardware layer. JavaES overcomes some of these difficulties by means of the JavaES hardware extension mechanism. This mechanism exposes a hardware abstraction layer at the application level that allows the interfacing and direct handling of hardware devices in a way that developers are unaware of whether the software abstraction is implemented with hardware or software components. For example, a watchdog is used to protect the execution of a code section with an auto-reset facility whenever the execution time for that code section exceeds a predefined time. JavaES provides a flexible implementation with two watchdog types: one is based on a hardware device (if available on the embedded target device), and the other is a software component (for the case where no hardware support is available). Depending on application requirements and the target capabilities, the developer may select the preferred watchdog type. In addition to the above, JavaES also improves the watchdog semantics by allowing the possibility to produce regular kicking as long as the code section is not functioning properly.
- *Virtual Machine Level*. The JVM is the component of Java infrastructure responsible for the execution of Java programs. It includes several facilities to improve the compilation and execution of Java programs such as a fixed data type model, a selected execution model, a task scheduler, a preverify mechanism, a garbage collector, a class loader or a secure sandbox model. However, all

these characteristics could involve high memory resources, low speed and non-predictable execution that should be optimized on a JVM implementation. In the embedded domain the physical restrictions of embedded targets (i.e., scarce memory or processing capabilities) can impose in addition severe limitations for the execution of a JVM. Hence, some JVM implementations are provided by manufacturers to address the suitable functionality for specific purposes; e.g., TINI is specific for embedded internet devices [267], while Lejos was designed for robotic Lego Systems [249]. Other JVM implementations provide different programming models or extensions to handle the same feature [195, 378], or restrict the semantics of Java language [249, 347]. JavaES adds a flexibility level with respect to the JVM, adapting the JavaES software infrastructure to the several *Java Configurations* that could be found in an embedded domain such as CDC, CLDC or JavaSE (and old deprecated variants, PersonalJava or EmbeddedJava, still available on the market). In this way, the code portability is guaranteed, making the implementation of the WORA principle possible.

- *Hardware Level*. The flexibility on hardware can be achieved by selecting mechanisms that makes easier the integration of the new hardware devices and peripheral units into hardware base, preferably on top of the JVM layer, and by defining a consistent hardware abstraction layer that unifies the access to hardware. JavaES applies a strategy that combines the use of standard I/O buses available on the current low-end microcontroller with additional chips and coprocessors to extend the hardware base in order to, for instance, augment the input/output capabilities of embedded targets. Each change in hardware is controlled by the JavaES framework by adding specific classes and software components that directly manage the new hardware. Thus, for instance, analog input/output are often scarce on embedded targets; a simple way to extend the analog capability on hardware is to connect an external *Analog-to-Digital Converter* (ADC) or *Digital-to-Analog Converter* (DAC) to the master processor through a synchronous serial bus such as I2C or SPI.
- *Deployment Level*. The application deployment is the activity at which the application is loaded and stored into the embedded target before its execution. There are several mechanisms for deploying an application into an embedded target. In many cases, the deployment involves the build of an application image; that is, the application plus the execution environment (e.g., operating system plus JVM). These mechanisms depend directly on communication interfaces used to download the application image, the presence of an embedded loader or monitor capable to load it onto the embedded target, the method used to store it, and the type of initialization and execution of this application image. JavaES unifies the different methods used to deploy an application by selecting automatically the required mechanisms for each embedded target.
- *Debugging Level*. Debugging is the traditional stage used to test the program by finding and reducing defects and bugs that could be causing the malfunctioning of a program. A good debugger and profiler may then increase the productivity, especially in the embedded domain in which applications should operate during long periods of time. The debugging process is applied at two

levels. The first debugging level is useful to test the first versions of the application and it is applied on host machine using emulators. The second debugging level is more realistic and it is applied directly on the target device. In this case, the debugging process allows us to obtain remotely the trace of program execution and variable watching, dumping data to a host computer console. Although the employed methods on embedded targets may be very different, JavaES provides a common mechanism that generalizes the debugging process at the two levels.

13.3 Components in JavaES

JavaES provides suitable abstractions, programming models and mechanisms to make easier the development of portable embedded applications over a variety of resource constrained embedded target devices using Java as the main programming language. It includes a set of components and helper tools, the *JavaES Toolchain*, to assist the design, building and compilation, local and remote debugging, deployment, maintenance and execution of programs on these embedded target devices.

Figure 13.1 shows the software design flow of a JavaES application. The main component of JavaES is the *Virtual Embedded Platform* (VEP), which defines a functional abstract platform of the overall embedded target device, including abstract components for both, hardware and software components. The VEP is machine independent and completely configurable depending on the application requirements by means of the VEP Specification document. In addition, an abstract API is provided for the programming of the applications.

Besides the VEP, JavaES builds a custom runtime environment, *VEP-Runtime Environment* (VEP-RE), specific for each VEP, which contains the software infrastructure with the required abstractions and components, adjusted for the specific embedded target device. This process is addressed automatically by a generative tool that will be described later, taking into account the *JavaES Specifications Repository* (JESR). JESR maintains the full set of specification and configuration documents necessary for the generation of the VEP-RE from an abstract VEP. It also adds the implementations (e.g., proprietary libraries, JavaES components) afforded by the framework to build the runtime environment that can be loaded onto a specific embedded target.

The building and deployment tasks are carried out by a standalone tool based on the Ant tool, the Java-based build utility [15], which could run within an Integrated Development Environment (IDE) such as Netbeans or Eclipse. The build process of an application involves the compilation by using the specific libraries of a VEP-RE, packaging the application into a single compressed .jar file. However, the deployment may be dependent on the particular embedded target. Hence, the tool should select the most appropriate deployment mechanism for each case.

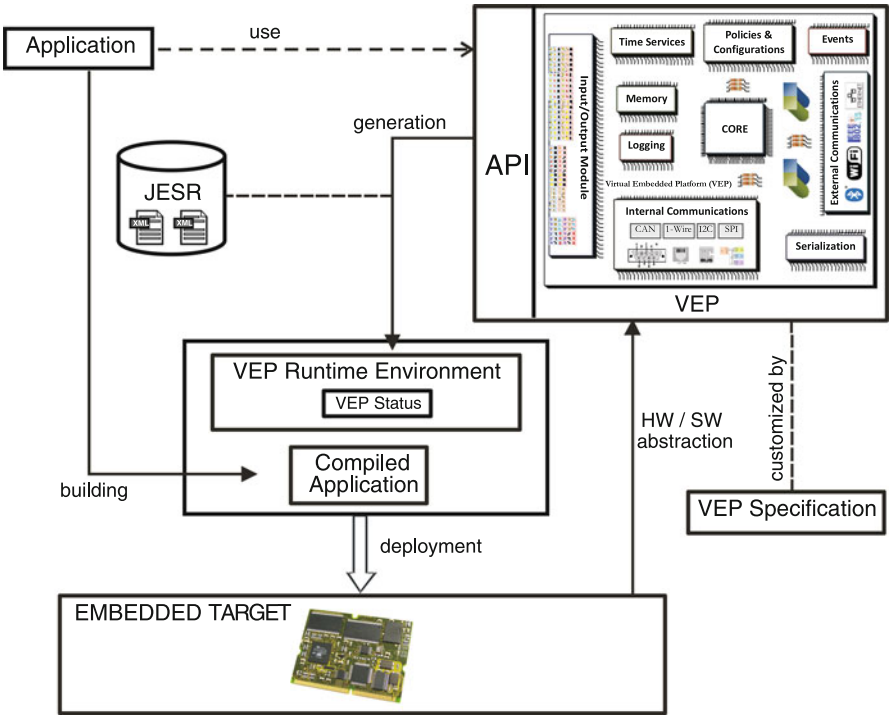


Fig. 13.1 Software design flow of a JavaES application

In addition to the above, the framework comprises other significant components for the application development summarized in Fig. 13.2. The JavaES Simulator allows us to emulate the program's execution directly on host computer with the restrictions imposed by the VEP, making testing and local debugging of the programs possible just before their deployment onto the specific embedded target. The remote debugging directly on the embedded target is also possible by means of the JaCoDES (Java Corleone Debugger for Embedded Systems) debugger. It adds a lightweight remote monitor (agent) placed on the specific target, on which we can trace the program execution and check the system state at runtime of all the virtual components (VEP Status); i.e., the I/O pins values, free VEP memory, number of active counters and their configurations, and execution traces. This monitor is similar to the *Squawk Debug Agent* (SDA) of the *Squawk Debugging Environment* (SDWP) [378] but with the difference that JaCoDES cannot access to JVM status or control the execution flow.

Another important component is the Benchmark Suite. JavaES can guarantee the code portability of programs over the set of compliant heterogeneous embedded targets. Although the program execution will have the same logic behavior, unfortunately, it cannot assure the predictability, efficiency or power consumption,

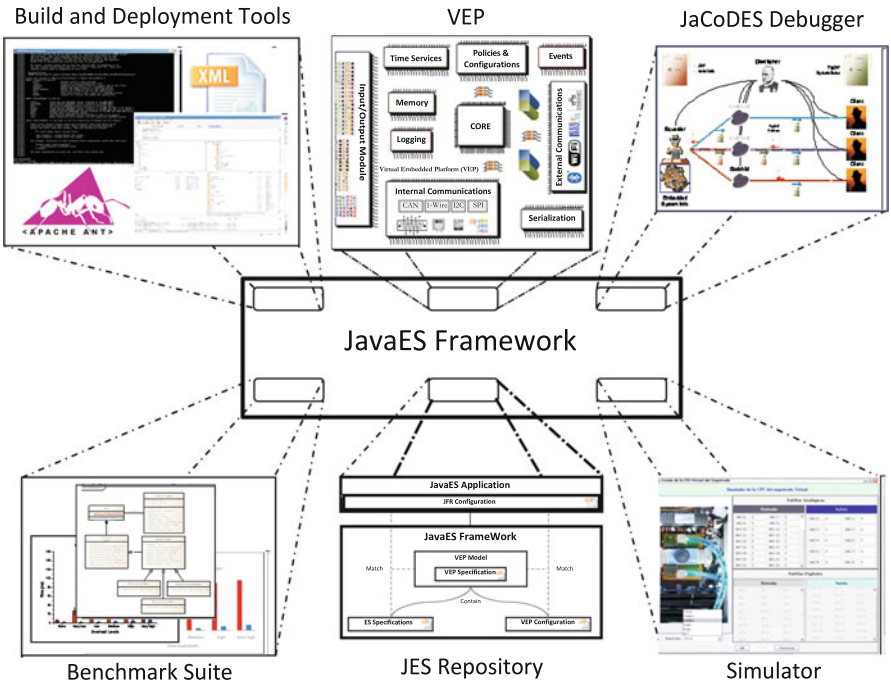


Fig. 13.2 Components of JavaES framework

since there are important differences in JVM implementations and hardware architectures that produce different execution traces. This, obviously, might indicate that, depending on the application, one embedded device is more appropriate than others. For this reason, a test bench is a necessary tool in the framework. It supplies a set of tools and test programs that can be executed automatically on any embedded target device in order to determine its capabilities in terms of performance, predictability and dependability among others. Thus, a comparative evaluation of targets may be performed, and consequently this information may be used to determine the most appropriate physical target device for the application requirements.

13.3.1 Virtual Embedded Platform

Unlike desktop applications, an embedded system needs to address software and hardware issues at the same time, since its functionality depends specifically on the available capabilities of the hardware resources in terms of input/output, memory or processing capabilities, and the programmability supported by the software infrastructure for an optimized control of these resources. In addition, in an embedded computing the software level (e.g., the kernel and drivers) that interacts

directly with the hardware resources is particularly important, since embedded systems often have scarce resources and stringent requirements.

Hence, the *Virtual Embedded Platform* (VEP) conceptualizes a functional abstract model of the overall embedded system, both hardware and software, but at a suitable abstraction level for making easier the application development. For this reason, it must contain not only a detailed description of the hardware devices, but also a description of the software components and its infrastructure, including the Java platform. Obviously, the fact that the VEP is defined as an abstract platform on top of a JVM imposes certain constraints and limitations derived from the Java semantics, the implemented features on the supported libraries and the JVM. Hence, the VEP should include the desired features and properties required from the underlying JVM or even the execution environment, for example, the presence of a garbage collector or the selected task scheduler. Subsequently, JavaES will search the more appropriate software component according to the VEP in the generation process of the VEP-RE.

From a hardware perspective, for example, a description of the peripheral devices and their input/output capabilities for exploring the physical environment can be specified at this level, such as the type of pins (analog or digital), the maximum voltage supported, the input or output use, or the resolution of ADC or DAC converters. JavaES will supply the required high-level abstractions and programming schemes in Java to interact with them. Moreover, it is also possible to decide if a hardware device (e.g., a clock, a counter or a watchdog) is to be implemented at hardware level or emulated at software level.

The VEP is machine independent and user configurable. Consequently, it should include a description of the minimum hardware-software components actually needed independently of the physical target device. Afterwards, JavaES may assist the developers and recommend the best embedded target or a set of targets that satisfies their demands.

The abstractions, properties and features relevant to specify an abstract VEP are described in a document, the VEP Specification, stored in a XML file, whose structure and constraints are validated by an XML Schema. It is organized in seven sections as it is shown in Fig. 13.3. Each section contains specific features, properties and parameters in separate fields, which are mandatory in some cases and optional in others. The sections of a VEP specification are the following:

- *Execution Environment*. This includes the features and properties that characterize the software execution environment with respect to Java semantics, the *Java Configuration*, the supported additional libraries, and the JVM.
- *Memory*. This section contains the physical memory features (size, types), but also the memory model defined on the Java platform such as the presence of memory regions, or the activation of the garbage collector.
- *Time Services*. This contains the services related to time such as clocks, timers, stopwatches or watchdogs, that are often used for measuring time or defining periodic tasks or timeouts. The developer can choose the most appropriate time services based on the resolution, clock rate or accuracy among others.

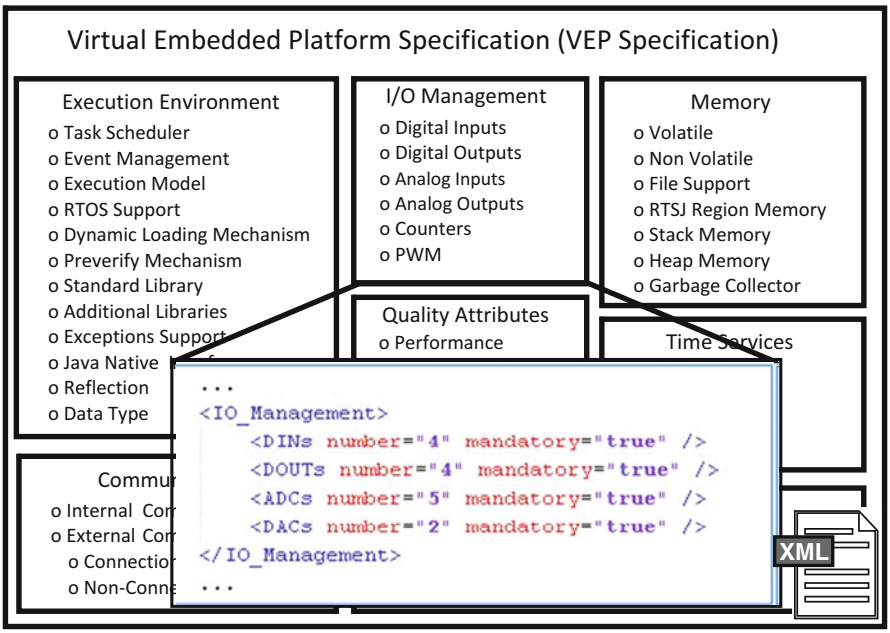


Fig. 13.3 An example of the structure of a VEP specification

- *I/O Management*. This section includes a description of the I/O facilities and abstractions available on the VEP. It may specify the I/O port types, the number of I/O pins, the number of event counters, whether a pin has a PWM signal associated to it or the maximum voltage of a pin.
- *Communications Support*. In this section, the communication ports and communication interfaces needed for a VEP are specified and classified depending on the scope, distance, and nature of the communication medium. JavaES distinguishes between internal and external communication support. Internal communications refer to short range communications, normally among boards or chips, such as synchronous serial buses (I2C, SPI or 1-Wire). External communications describe communications with peripheral units or other computing devices including asynchronous serial ports such as UART, RS232 or Ethernet. Each communication port should be characterized by adding information about its nature, name, configuration, etc.
- *Debug Toolset*. This contains information for customizing the JaCoDES debugger. Here the time interval for remote polling, or the communication link used to transfer debugging data obtained by the remote monitor of the embedded target can be specified.
- *Quality Attributes*. In this section, parameters used to compare the system behavior (performance, reliability or predictability) among target embedded devices are captured. The satisfaction of quality attributes on each target can be

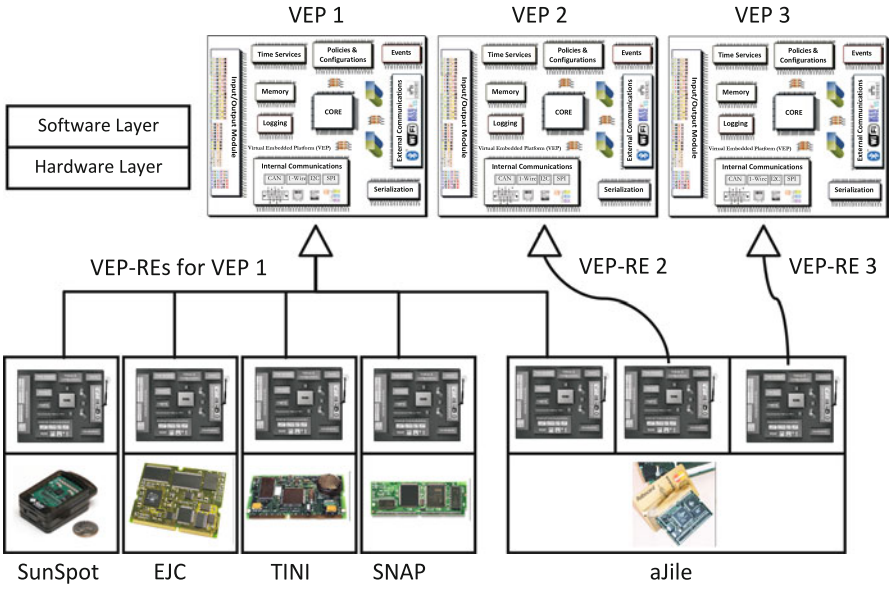


Fig. 13.4 A graphical representation of the possible ways to generate a VEP-RE from a VEP for different embedded target devices

measured by means of the JavaES Benchmark suite, which provides a test bench for defining the corresponding metrics. The inclusion of quality attributes in a VEP can limit the set of valid embedded target devices compatible to VEP.

13.3.2 VEP Runtime Environment

The *VEP Runtime Environment* (VEP-RE) is the customizable lightweight software infrastructure that JavaES generates for each VEP, adjusted for a specific embedded target device. The VEP-RE includes the set of the required objects, classes, data structures, routines and protocols organized in packages, and the programming schemes with a well-defined API, in order to assist the developing of embedded programs. Moreover, it is customized, on the one hand, by the features, properties and abstractions of the corresponding VEP and, on the other hand, by the restrictions imposed by that VEP.

The VEP-RE is built by means of the JavaES generation scheme to be detailed later. It represents an instance of a VEP or, in other words, a particular implementation of the JavaES framework according to a VEP for a specific embedded target. This means that each VEP might have an optimized VEP-RE for a selected target. By extension, multiple VEP-REs can be obtained from a particular VEP for different target devices. This is shown schematically in the Fig. 13.4.

The VEP-RE does not interfere with the standard libraries of the *Java Configuration* and the proprietary libraries added by the manufacturer on the execution environment of the embedded target device, but it provides an add-on that complements the Java platform. Then, the VEP-RE acts as a middleware between the application layer and the underlying execution environment that may include the JVM and operating system, or a JVM on the bare metal. For instance, the implementation of a customized VEP-RE for a specific VEP requires a *Java Configuration* based on CLDC 1.1 for a Snap or Sunspot embedded target, whereas in the case of an EJC embedded target it requires a *Java Configuration* based on PersonalJava (subset of JDK 1.1.8). The *Java Configurations* are somewhat different. Therefore, they may not provide the same facilities for a specific programming issue. Nevertheless, it is still possible to implement the semantics of a particular facility from a specific *Java Configuration* in any embedded target device in order to guarantee the portability as JavaES does. Thus, for example, the data storage mechanism on a flash memory might be implemented by using the generic connector framework (GCF) in CLDC, or by the file management facilities available in the java.io package on PersonalJava. However, JavaES homogenizes the data storing by implementing the GCF in any *Java Configuration* as the only facility for the file management, even for PersonalJava. Consequently, the VEP-RE for the EJC embedded target requires an implementation of the GCF based on the classes and interfaces of java.io package. This implementation is hidden by the VEP-RE and transparent to the running application.

The VEP-RE runtime obviously increases the overhead with respect to the underlying software infrastructure. However it is optimized in space in order to decrease its size in memory and it is optimized in time in order to reduce the penalty in performance. Whenever possible, we have attempted to improve the predictability of the JavaES framework in the implementation of VEP-RE for each embedded target device, for example, by controlling the additional resources of the required loaded objects, or by reducing the number of call invocations. But it depends essentially on the real-time capabilities of the underlying execution environment, JVM and OS.

13.4 VEP-RE Generation

The VEP-RE Generation is the process in which a VEP-RE is built in a consistent way. The resulting VEP-RE is specific for an embedded target device, but conforming to the specified abstract VEP. The VEP-RE generation requires the participation of a set of specification and configuration documents available on the JESR repository, and a method that addresses the generation process. These specification and configuration documents are represented with XML files. First, a brief description of these specification and configuration documents is given before explaining in more detail the generation process.

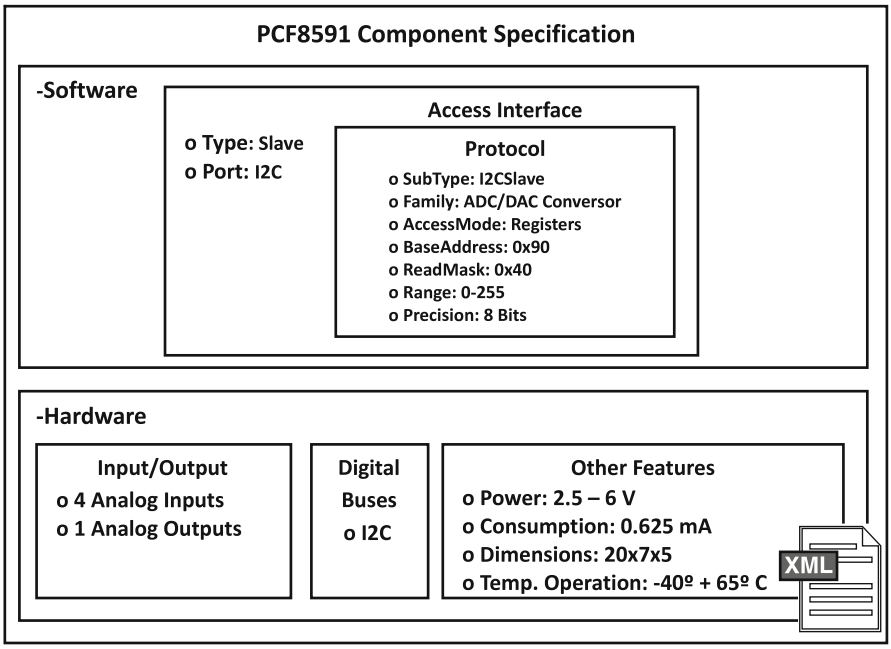


Fig. 13.5 The component system specification of PCF8591

The *Embedded System Specification*(ESS) contains complete information about the overall physical embedded target device, including hardware and software. This information should be complete, since it is required to determine whether the embedded target device is compatible with the VEP and is also used to configure JavaES at startup. The physical capabilities are, for instance, the processor features, the memory resources, the presence of hardware clocks and timers, the power consumption or the input-output capabilities. Likewise the software section includes features, properties and constraints related with the Java semantics, the supported libraries including the *Java Configuration*, and finally the JVM and the underlying operating system when it is present.

The ESS is composed of one or more specifications. The *Base Embedded System Specification* (BESS) is mandatory and captures the features of the base embedded target device. The BESS structure distinguishes the hardware level from the software level and it should be defined for each embedded target supported by JavaES. In addition, a set of *Component Embedded System Specifications* (CESS) may be optionally defined to describe specific components that could extend the capabilities of the base physical embedded device with respect to hardware, software, or both. The CESS has usually two sections: one section for describing the component interface and other section for describing the characteristics of the component system. Figure 13.5 shows the CESS of a PCF8591 chip that extends the analog input/output of an embedded system by means of an I2C bus.

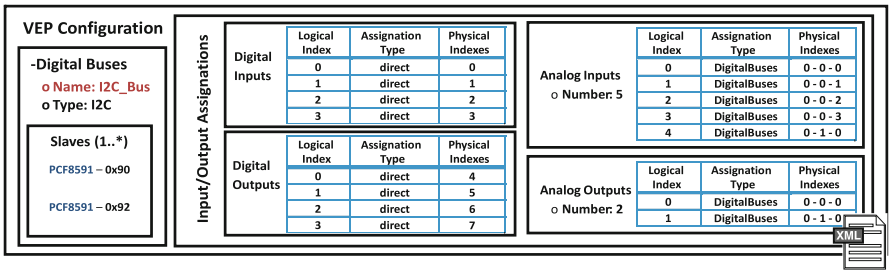


Fig. 13.6 An excerpt of the VEP configuration for IM3910 showing the I/O assignations

The *VEP Configuration* document sets the mapping rules between a VEP and the corresponding ESS, revealing specifically how VEP elements should be associated with ESS elements in order to achieve a valid VEP-RE for an embedded target device. For instance, a developer may configure the topology and configuration of supported digital buses, or the assignments of logical VEP inputs/outputs to physical ones in ESS. The Fig. 13.6 shows an example applicable later to the case study.

The *JavaES Framework Application Requirements Configuration* (JFAR Configuration) document captures the specific requirements of an embedded application, and it is more restrictive than the VEP. Usually the purpose of this configuration document is to determine some particular requirements of the application that may have an impact on the total size of the application or its performance. Thus, only minimum characteristics that need to be satisfied in order to run the application should be described. There are two requirements types: hard requirements and soft requirements. Hard requirements are mandatory requirements that should be fulfill to run the application such as minimum RAM memory or code version among others. Soft requirements are requirements whose satisfaction is not mandatory for application execution such as, for example, the clock resolution.

In order to find an optimized VEP-RE, first, a verification of the required specification and configuration documents of JESR repository should be performed. Figure 13.7 shows how the specification and configuration documents are related. First, the verification involves the checking of the VEP with respect to the ESS, the base embedded device defined on *BESS* (e.g., SNAP) and the set of component systems described on *CESS* (e.g., I/O expander). The next stage consists of applying the mapping rules defined on the *VEP Configuration* document, which provides assignments between logical models and physical ones. Finally, if required, the *JFAR Configuration* document is checked to include the restrictions imposed by the application. Next, the VEP-RE can be built by selecting the suitable packages and components applicable to a specific embedded target device.

The VEP-RE generation process is addressed by a generative tool based on Ant following the five step approach as shown in Fig. 13.8. This process is driven by the VEP Specification since, once prepared after the first step, it should be checked

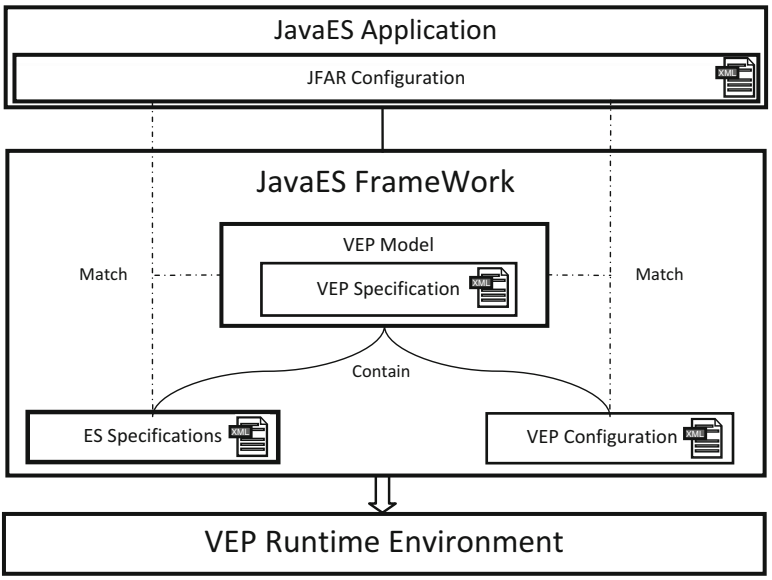


Fig. 13.7 Relationship between the specification and configuration documents for the VEP-RE generation

in the remaining steps in order to validate the correctness of the process. The first step, the VEP modeling, is currently carried out manually by preparing a XML file to contain the *VEP Specification*, although we expect to have a graphical tool (*VEP Composer*) to assist the modeling.

The second step, the VEP Configurator, searches for the best BESS-CESS combination compatible with the VEP, and then determines how the mappings between the VEP abstractions and the physical components are performed according to the *VEP Configuration*. This step is still manual and, consequently, the developer should select the profile corresponding to the embedded target device (e.g., SunSpot or Tini) on which the VEP-RE is going to be generated; that is, the BESS is fixed from the profile, while the additional CESS components are free to be incorporated for each hardware component integrated onto the physical embedded target. However, we expect to add a way to automatically select the best embedded target or BESS-CESS combination from a design space exploration such as CoDesign methodologies [123]. The remaining steps are carried out automatically.

The VEP-RE libraries selector chooses the common part of VEP-RE libraries according to the VEP to be added to the final VEP-RE. The Target Libraries Selector finds the required implementation of the selected embedded target device in the JESR repository and only includes the necessary ones. Finally, the VEP-RE packager is responsible for packaging the VEP-RE into a single jar file.

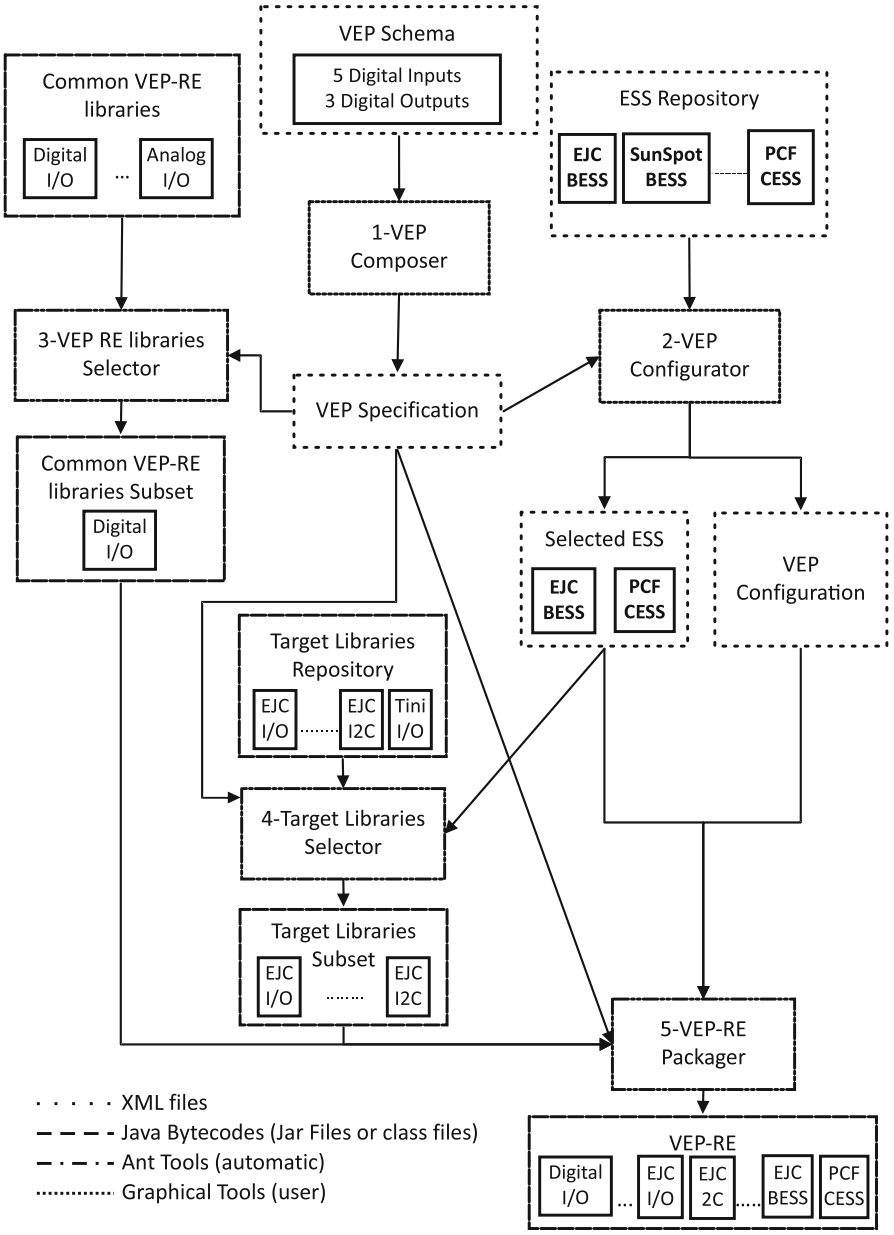


Fig. 13.8 Generation process of a VEP-RE with an example with EJC target and digital I/O VEP

13.5 Application Building and Deployment

The execution of the JavaES infrastructure requires a typical environment for embedded development according to a host-target scheme. The JavaES framework can be executed on a standard PC with any current JVM implementation. But it is also required to include the development tools of the embedded target manufacturer, since the JavaES tools make an intensive use of them by integrating them into the JavaES toolchain. According to the software design flow shown in Fig. 13.1, the JavaES toolchain is composed of three independent processes: (a) the VEP-RE generation process, (b) the application building process, and (c) the deployment process. Only the latter two processes are detailed in this section.

The application building process (shown in Fig. 13.9) allows us to construct an embedded application, packaged in an single application jar. First, the VEP-RE could be integrated into an IDE to help the implementation of an application. If necessary, a JFAR specification may be carried out to specify some additional restrictions of the application under development. Finally, the application builder compiles the application and links the application classes and packages into the jar file.

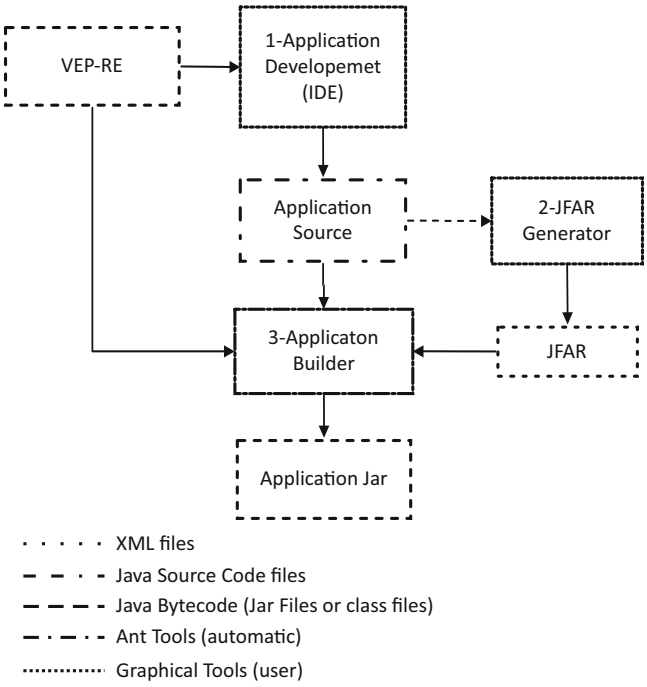


Fig. 13.9 The application building process

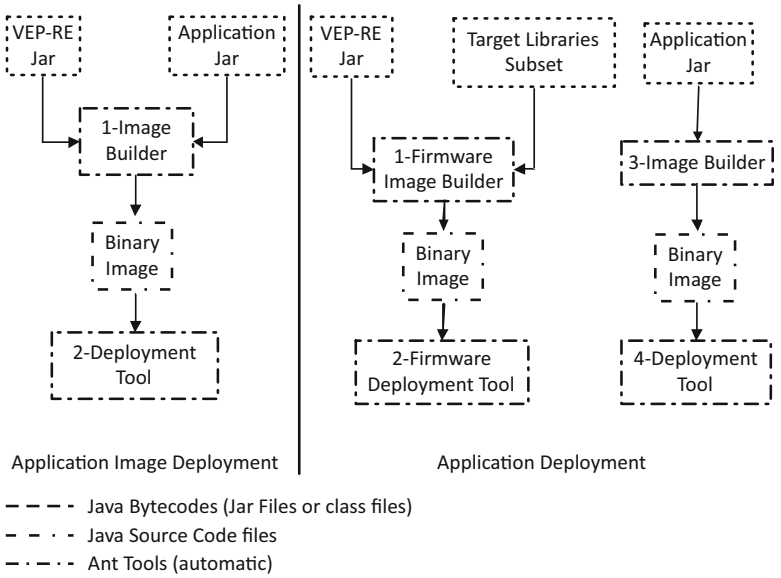


Fig. 13.10 JavaES deployment process

Two methods are usually employed for the deployment process: (a) application deployment, and (b) application image deployment (Fig. 13.10). The application deployment assumes that the developer should package the application and libraries (generally in a jar file with additional information in a manifest), which is downloaded into target to proceed to run. This requires the execution environment to be previously installed on the embedded target device. In contrast, the application image method requires both the application and the complete execution environment to be built and deployed as a single image. The latter method is slower but it is usually less resource-intensive than the former. The generative tool based on Ant follows the suitable method depending on the selected embedded target device.

13.6 JavaES Architecture

The JavaES architecture is basically the VEP-RE architecture on which programs are implemented and executed. It is composed of two separate hierarchical software layers as it is shown in Fig. 13.11. Each layer contains several interrelated components. Each component has an associated package that groups all classes, packages and interfaces related to it.

The low-level layer, *JavaES Abstraction Layer* (JAL), is responsible for encapsulating the abstractions and components defined on the JavaES programming model with respect to underlying infrastructure provided by embedded manufacturers;

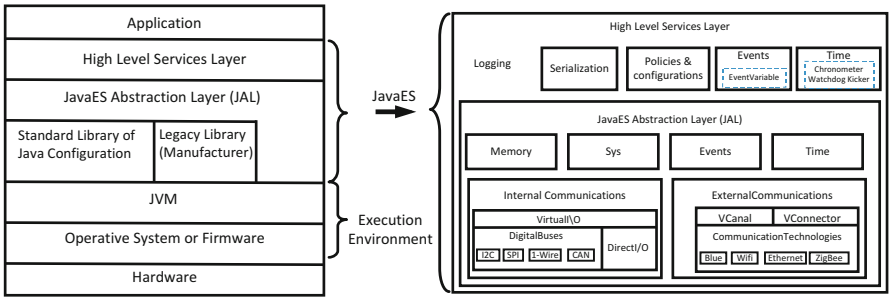


Fig. 13.11 VEP runtime environment architecture

i.e., the JVM plus the standard library compliant to a *Java Configuration*, and the proprietary libraries for accessing and dealing with additional hardware devices. The structure of this layer is similar to an operating system, including several abstractions and components to handle the hardware devices, although it is based on the software infrastructure in the layer below. It is composed of the following components:

The Sys package abstracts the hardware base and the execution environment, providing a general model of the embedded system. It contains the low-level implementation of JavaES, which is adapted to the execution environment of the specific embedded target. This package also has the responsibility of initializing the VEP-RE according to the underlying embedded target and loading the VEP-Status for the monitoring of system state. During the initialization phase, JavaES identifies the I/O peripheral units and hardware devices, including physical communication ports, and then loads the manager objects. Finally, it includes an implementation of the supported Debug Toolset, providing support for remote outputs or JaCoDES.

The Memory package includes facilities for accessing memory resources; most of them depend directly on the JVM. However, JavaES also provides different access modes to cover the applications requirements such as heap memory, objects pools or immortal memory if it is supported.

The Events package comprises an asynchronous model of an event paradigm borrowed from RTSJ, that includes in some cases interrupt handling when the JVM captures the interrupts. It can be useful for time services implementations, networking or I/O handling in general.

The Time package encapsulates the time services such as a system clock, counters, timers (periodic, one shot, cycled), watchdogs, or stopwatches for time measurements. All of these mechanisms allow for a flexible and efficient application programming relative to time tasks. Some of these time services have been implemented with the RTSJ semantics in mind.

The Internal Communications package includes classes and interfaces to deal with short range communication protocols, usually according to a master-slave

protocol such as the I2C, SPI or 1-Wire buses. This package abstracts the access to these buses, offering a homogeneous protocol which is underlying-technology independent, and providing a common interface for reading/writing to buses.

The External Communications package groups communication protocols and interfaces for communication between the embedded system and external peripherals or autonomous systems. This package offers several communication models independent of underlying technology (Ethernet, RS-232, Wifi, Bluetooth or IEEE 802.15.4) such as connection-oriented channels and non-connection-oriented transmissions (in datagrams) with an implementation of the GCF included on JavaME-CLDC.

The high level services layer contains important services and high-level functionalities that may help the application programming. They are hardware independent and therefore JAL independent. Many important issues are covered in this section such as serialization, event logging, or time service for measuring the WCET (Worst Case Execution Time). For instance, the serialization in JavaES is the main mechanism for marshalling data that may be used to transfer data in communications protocols or to store data on a non-volatile memory such as flash memory; JavaES includes several kinds of optimized serializations based on String, XML tags and bitmasks.

13.6.1 The Hardware Abstraction in JavaES Abstraction Layer

The VEP-RE implementation is fully composable in order to simplify the insertion or deletion of any component in the VEP-RE generation process. The VEP-RE is entirely implemented in Java on top of the JVM provided by the manufacturer along with the embedded target device. This means that VEP-RE cannot change the implicit JVM behavior, for example, thread scheduling, hardware device access, signals and interrupt handling or shared resources. Since the Java specifications are still obscure or weak in many of these concerns [425], there are multiple ways to address these issues [2,301,367], making it difficult to have a unique model; in fact, many Java specifications from JCP (e.g., RTSJ [65] or SCJ [192]) advocate to clarify some of them.

For this reason, JavaES must manage the heterogeneity by abstracting many of these concerns relying on Java semantics. For example, the JavaES event management generalizes the interrupt or signal handling whenever the interrupt or signal can be captured by the JVM, or the JVM can expose its handling at Java level, but it depends on the specific embedded target. Thus, JavaES makes a clear separation between the JavaES abstractions and the corresponding implementation over the concrete JVM implementation by means of the abstract factory pattern [163]. Then, each JavaES abstraction is encapsulated as an abstract class, the abstract factory, which includes the platform-independent implementation to give support to applications, and a concrete factory with respect to the embedded target

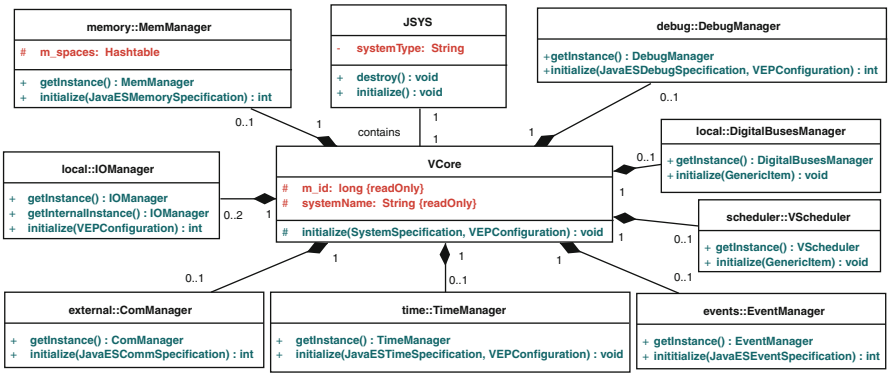


Fig. 13.12 Class diagram of JavaES manager classes

with the platform-dependent implementation. The concrete factory implementations are stored in special packages within the *Hardware Development Kit* (HDK) in the *Target Libraries Repository*, and they are added to the VEP-RE in the building phase of the VEP-RE generation process. For example, the `IOManager` object can manage analog or digital pins virtually, providing access to them via a public interface with abstract methods. However, the concrete `IOManager` gives an implementation to the abstract factory based on the specific objects available in the software infrastructure to get access to physical I/O pins. In our case, the implementation of the concrete `IOManager` provides access to physical I/O pins of the embedded target, and also to I/O pins available in an external hardware device connected to the embedded target through a digital bus. This implementation is then hidden to the abstract factory of `IOManager`, giving transparent access to the physical pins regardless of where they are.

Figure 13.12 shows the first order objects that give support to applications. The main class, `JSYS`, is a static class responsible for initializing the VEP-RE when the application is started. Then, the `VCore` object is loaded into memory, taking charge of the initialization process. The initialization first loads the aforementioned specification and configuration documents for the VEP-RE generation process as configuration files, since those provide relevant information about the features and properties of the specific embedded target and external hardware devices (BESS and CESS), and the mapping between the VEP abstractions and physical ones (VEP Configuration). In addition, during the initialization the basic services are loaded as Manager objects such as the memory controller, I/O and communication devices, time services, the task scheduler and the asynchronous event controller. Each Manager object is an abstract factory with the corresponding concrete factory in each embedded target, and is implemented as a singleton object in order to make it accessible from any invocation of the application.

For example, Fig. 13.13 shows that `IOManager` supervises the analog and digital I/O pins through method invocation (i.e., `getPinADC` and `setPinDAC`)

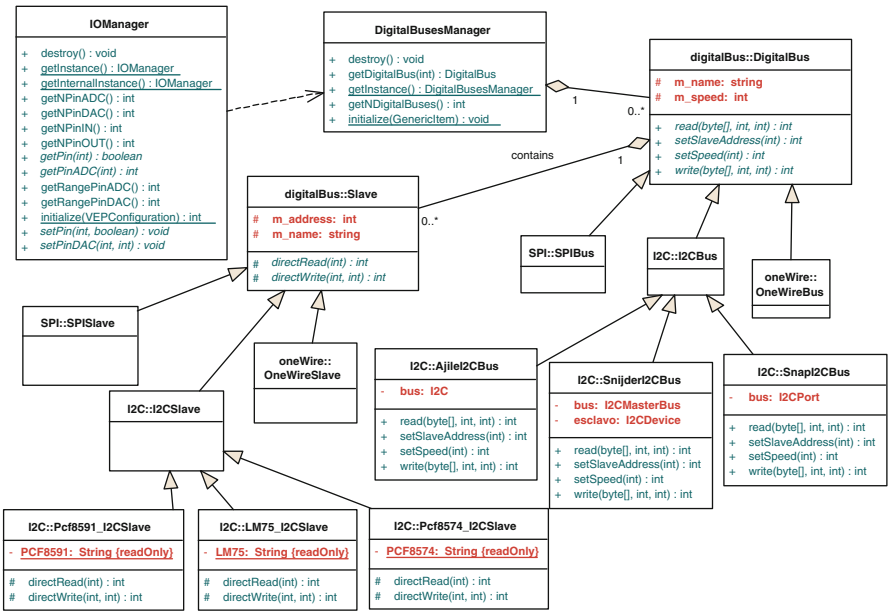


Fig. 13.13 Class diagram of I/O management

for analog I/O pins. Internally, `IOManager` can have a link with an external analog pin according to the VEP configuration at the JavaES initialization. Then a `getPinADC` call can make use of the native I/O methods provided by the concrete target implementation or the `DigitalBus` abstraction, in order to access an external ADC pin through a digital bus such as I2C, SPI or 1-Wire. In this case, the read method of the `DigitalBus` class is called using the indexes defined into the VEP Configuration. The read method implementations are hidden by the VEP-RE and uses the primitives provided by the target manufacturer to access the digital bus as shown in Fig. 13.13.

13.7 Case Study

A case study is presented in this section in order to show the JavaES adaptability to heterogeneous embedded target devices, while maintaining the code portability of the program. A temperature regulator has been developed for an instrument device, which is used in a high performance laboratory for surface physics experiments. The regulator device must control and maintain a constant temperature value inside the measurement cell of a pendant drop tensiometer [207], which tends to fluctuate during the measurement process. Basically, the regulator device requires a simple feedback control embedded application in which the control loop manages in each

iteration an analog input to obtain a temperature value from a temperature sensor, two analog outputs to control the heating and cooling level, in our case on a Peltier cell, and two LEDs to show the regulator activation mode.

Many JavaES compliant embedded target devices could be suitable for this application. But we have selected two typical Java-based embedded target devices, the IM3910 from IMSys [218] and the EJC200 from Snijder [147]. Both embedded target devices are limited in the number of available analog I/O pins, requiring the addition of I/O expanders for augmenting the I/O capabilities. Moreover, they are essentially different with respect to the API and the implementation of the software infrastructure (i.e., JVM and RTOS) provided by the manufacturer; for instance, IM3910 provides a JVM on a proprietary RTOS based on a JIT compiler with a Java Configuration based on CLDC 1.0, plus a set of non-standard libraries to give support to specific hardware devices. The EJC, meanwhile, provides a JVM also over a proprietary RTOS based on a JIT compiler but with a Java Configuration based on PersonalJava (JDK 1.1.8).

From a JavaES perspective, each embedded target has a specific BESS. Then, after preparing the VEP specification of the VEP platform with the hardware and software requirements of the embedded application, JavaES can generate a customized VEP-RE from this VEP by selecting the most suitable BESS-CESS combination. Since the BESS is fixed (a profile is chosen), we can select the required CESS components for each additional software component or hardware device in order to obtain the final VEP-RE. Particularly, a standard VEP is prepared as shown in Fig. 13.3.

Figure 13.14 illustrates the hardware diagram of the temperature regulator for both embedded target devices. For IM3910 we have added two additional PCF8591 chips [282] connected to the I2C bus to provide support for analog inputs and outputs; i.e., two DACs for controlling the heating and cooling level, and one ADC for obtaining the temperature values. Each PCF8591 will have an associated CESS as shown in Fig. 13.5, but other embedded target devices might not require additional hardware components to handle the analog I/O. On the other hand, the EC200 can directly read the analog input of the temperature sensor without any other external I/O device, but we need to include a MPC4922 chip to make two analog output channels available, in this case, through a SPI digital bus.

Although the VEP-RE implementation for each embedded target device is essentially different, the provided API is the same and, consequently, the program developed on JavaES would be exactly the same. The differences are then managed by the *VEP Configuration document*, in which the developers must model the mappings between the virtual and abstract primitives of VEP implemented on the VEP-RE and the physical ones defined in the ESS (BESS and CESS). Figure 13.7 shows an extract of the VEP Configuration for the VEP with the IM3910 profile. In this case, four digital inputs and four digital outputs defined on the VEP are

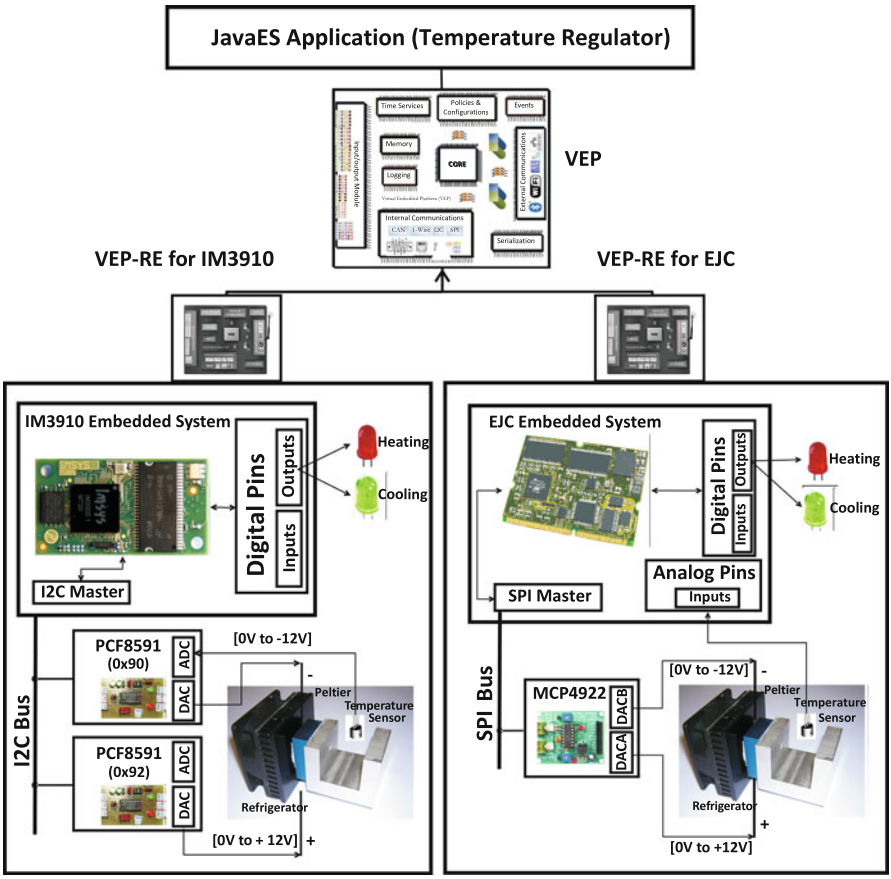


Fig. 13.14 Schematic design of the temperature regulator on two embedded target devices with the hardware diagram (*below*) and the corresponding required VEP-RE (*up*)

assigned directly to physical pins on IM3910 (defined by a number), while the five ADCs and two DACs are mapped to two external PCF8591 devices. The mapping to external I/O is achieved by the definition of three indexes: the digital bus index, the slave index (specifies the slave chip address connected to the master microcontroller) and the internal slave index (identifies the internal address of the pin within the external slave chip). With this information JavaES can create two PCF8591-I2CSlave objects according to the hierarchy of Fig. 13.13 connected to the I2Bus object, which is handled by the singleton object, `DigitalBusManager`. Each PCF8591-I2CSlave encapsules its hardware device with the corresponding address (taken from the VEP Configuration) and overrides the `read` and `write` methods with the specific data protocol provided by the manufacturer of PCF8591.

```
public class PeltierRegulator2 {
  static double desiredTemp=25; //Desired temperature
  static long delayTime=1000; // One second of delay.
  static double actualTemp=-1;
        //Auxiliary variable for actual temperature.
  static int value=-1,dacIndex=-1,dacRange=-1;
       //Auxiliary variables for DACs
  static double constant=-1;
  // Main method
  public static void main (String []args){
    constant=100.0/(double)JSYS.IOManager.getRangePinADC(0);
       //100ºC/maxRange
    JSYS.initialize(); //Start JavaES Framework
    Runnable logic = new Runnable() {
      public void run() {
        //Reads from sensor
        actualTemp = JSYS.IOManager.getPinADC(0) * constant;
        if (actualTemp == desiredTemp) {
          //No temperature variation
          JSYS.IOManager.setPinDAC(0, 0); //Turn off cooling.
          JSYS.IOManager.setPinDAC(1, 0); //Turn off heating.
        } else {
          //Temperature variation
          //Check if it must heat or cool.
          dacIndex = (actualTemp > desiredTemp) ? 0 : 1;
          dacRange = JSYS.IOManager.getRangePinDAC(dacIndex);
          //Calculates the DAC value to set with a Fuzzy function.
          value =
          Fuzzy(Math.abs(actualTemp - desiredTemp), 100, dacRange);
          //Sets the calculated value to the correct DAC.
          JSYS.IOManager.setPinDAC(dacIndex, value);
        }
      }
    };
    VAsyncEventHandler handler=JSYS.getAsyncEventHandler(0);
    handler.setLogic(logic);
    VPeriodicTimer timer=
    JSYS.getPeriodicTimer(null,VClock.getTime(delayTime),
                          null,handler);
    timer.start(); //Start the PeriodicTimer.
    JSYS.core.delay(3600000); //Sleeps while an hour.
    JSYS.destroy(); //Stop and destroy JavaES Framework resources.
  }
}
```

Fig. 13.15 The regulator device implementation

The control loop of the regulator requires a reading of the temperature sensor and subsequently sending actuator signals to the Peltier cell when the temperature is above or below the desired temperature value. The code in Fig. 13.15 shows the regulator device implementation. As a typical program in JavaES, first an initialization procedure (i.e., JSYS.initialize) starts the JavaES platform. The initialization loads all the needed objects to manage underlying execution environment and hardware devices. In the same way at the end of the program a delete facility (i.e., JSYS.destroy) closes securely the access to hardware devices and communications ports and then releases the memory. The program core shows how the access to analog pins is invoked, and how the timer is implemented. The access for ADC and DAC converters is carried out by a single invocation to `getPinADC` or `setPinDAC` to the `IOManager` singleton object without knowledge of whether they are handled on the external chip (e.g., PCF8591) or on the internal microcontroller. The code shows also the periodic timer implementation for the control loop. The timer semantics are similar to the one defined in the RTSJ specification, where a periodic time interval, a specific clock and an event handler for managing the time expiration of the timer are required.

13.8 Evaluation

Although the test bench is not discussed in this paper, a preliminary study is presented in order to obtain evidence about the order of magnitude of the overhead incurred by the JavaES framework on the embedded targets. An experiment is performed to measure the memory and performance overhead, whose results are shown in Table 13.2.

The memory overhead is measured after the JavaES initialization, since the most demands of memory resources are reserved in this stage. The memory over-head value is obtained by measuring the increase in consumed memory after the JavaES initialization. The results show that memory overhead is moderate in general except for the EJC target with a magnitude order about 1 MB. There are two reasons for this value. First, the VEP-RE implementation is more complex than the implementation of other embedded targets. Second, the JVM is based on a JIT compiler and it reserves large amounts of memory for class loading. But, afterwards, it achieves shorter response times when native calls are executed.

For the I/O performance study we employed a digital I/O pin of the embedded target and an external PFC8591 converter through an I2C digital bus for the analog I/O. Then, the I/O calls were evaluated by comparing the average time needed for reading or writing an analog or digital pin by a JavaES invocation and the average time required by a native call, an invocation using the underlying JVM. The code for this experiment is similar to that presented in Fig. 13.14.

The overhead of digital I/O calls ranges from a few microseconds up to three times more than a native call, depending once again on the VEP-RE implementation. Moreover, there are important differences in the digital I/O call magnitude order

Table 13.2 JavaES memory and I/O performance overhead

Embedded device	Memory overhead (kB)	Digital I/O native call (μs)	Digital I/O JavaES call (μs)	Analog inputs native call (μs)	Analog inputs JavaES call (μs)	Analog outputs native call (μs)	Analog outputs JavaES call (μs)
EJC	1,120	1.68 ± 0.12	8.21 ± 0.11	941 ± 2	1,164 ± 4	458 ± 2	628 ± 4
SNAP	151	105.8 ± 0.3	277.8 ± 0.3	1009.0 ± 0.6	5,654 ± 13	421.7 ± 0.6	4,632 ± 15
TStik	279	1,115 ± 11	2,701 ± 83	4,920 ± 2	13592.7 ± 1.4	2,157 ± 2	7,514 ± 4
Sunspot	63	284.68 ± 0.14	285.8 ± 0.3	6,323 ± 2	6,343 ± 8	1965.2 ± 1.2	1,972 ± 2
IM3910	150	71.5 ± 0.2	127.6 ± 0.3	1142.7 ± 0.3	2,415 ± 2	518.5 ± 0.4	1936.8±1.1

among target devices, depending mainly on the target JVM implementation and its hardware architecture; i.e., in some cases the digital I/O calls involve additional communication through a slow synchronous bus. The overhead of analog I/O calls is higher than in the case of digital ones due to the synchronous digital bus, but it is insignificant on powerful target devices as SunSpot and EJC and moderate on the others. All the obtained results are relatively stable at a 95% confidence interval. Moreover, the maximum values of the samples (not shown in the table) are close to the average values.

13.9 Conclusions

This chapter shows the methodological approach of JavaES framework for the development of Java applications for a wide range of heterogeneous resource constraint embedded systems. The main benefit of our proposal is that it provides a portable and flexible solution to the existing variety of Java technologies applicable on the embedded domain. The heterogeneity and the differences with respect to the hardware architecture, memory and peripheral devices require adjusting the design of virtual machines and libraries specifically to the particular restrictions of every embedded target. This implies significant differences among the JVM implementations and the supported libraries that hinder the development of Java applications compliant to the WORA principle, not only due to the API compatibility, but also due to the differences in performance, predictability and memory consumption. In this sense, JavaES tends to standardize the different Java technologies, providing an API based on Java J2ME-CLDC with the addition of RTSJ semantics whenever it has been possible. Although we have kept in mind the standard specifications from the JCP during the API design of JavaES, i.e., the use of GCF for the communication and file management, there is not currently any specification in JCP that could provide an API for access to the hardware level. Therefore, we have defined and adopted a new one in order to simplify the development of portable applications on diverse embedded targets.

JavaES makes it possible to generate specific VEP-REs for each embedded target according to application requirements. The VEP-RE acts as an add-on over the underlying software infrastructure of an embedded target, optimized in time and space in order to minimize the impact of the JavaES overhead.

We believe that the presented approach is valid and applicable in other domains such as the mobile domain, in which it is very common to find that a Java application may fail to run when the mobile phone does not have a specific functionality or resource.

Acknowledgements This paper is an extended version of the paper “A flexible Java framework for embedded systems” published in the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES, ACM Digital Library, 2009.

References

1. A. Corsaro, C. Santoro. The Analysis and Evaluation of Design Patterns for Distributed Real-Time Java Software. *16th IEEE International Conference on Emerging Technologies and Factory Automation*, 2005.
2. G. Agosta, S. Crespi, and G. Svelto. Jetatine: A virtual machine for small embedded systems. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 170–177, 2006.
3. aicas. Hija safety critical Java proposal. Available at http://www.aicas.com/papers/scj.pdf, May 2006.
4. aJile Systems. ajile systems: Home site. Available at http://www.ajile.com/.
5. aJile Systems. aj-100 real-time low power Java processor. preliminary data sheet, 2000.
6. aJile Systems. aj-102 technical reference manual v2.4. Available at http://www.ajile.com/, 2009.
7. aJile Systems. aj-200 technical reference manual v2.1. Available at http://www.ajile.com/, 2010.
8. L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM international conference on Embedded software*, EMSOFT'04, pages 95–103, New York, NY, USA, 2004. ACM.
9. J.C.R. Americo. A study of the impact of real-time constraints on Java/OSGi applications. Available at http://hdl.handle.net/10183/26347, 2010.
10. J.S. Anderson and E.D. Jensen. Distributed real-time specification for Java: a status report (digest). In *JTRES'06: Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 3–9, New York, NY, USA, 2006. ACM Press.
11. J.S. Anderson, B. Ravindran, and E.D. Jensen. Consensus-driven distributable thread scheduling in networked embedded systems. In *EUC'07: Proceedings of the 2007 international conference on Embedded and ubiquitous computing*, pages 247–260, Berlin, Heidelberg, 2007. Springer-Verlag.
12. 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 Syst.*, 37(1):1–44, 2007.
13. J. Andreu, J. Videz, and J.A. Holgado. An ambient assisted-living architecture based on wireless sensor networks. *Advances in Soft Computing, Springer*, 51:239–248, 2008.
14. Aonix. Aonixperc-ultra. Available at http://www.atego.com/downloads/support/data-sheets/aonixperc-ultra.pdf.
15. Apache. Apache Ant tool. Available at http://ant.apache.org/.
16. Apogee. Aphelion. Available at http://www.apogee.com/aphelion.html, 2004.
17. ARM. Jazelle technology: ARM acceleration technology for the Java platform. white paper, 2004.

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5,

18. A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A real-time Java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007.
19. P.J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
20. C.R. Attanasio, D.F. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collectors. In H. G. Dietz, editor, *Proceedings of the Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science*, pages 177–192, Cumberland Falls, Kentucky, August 2001. Springer-Verlag.
21. J.S. Auerbach, D.F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. G. Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In C.M. Kirsch and R. Wilhelm, editors, *EMSOFT*, pages 249–258. ACM, 2007.
22. J.S. Auerbach, D.F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone. Tax-and-spend: democratic scheduling for real-time garbage collection. In L. de Alfaro and J. Palsberg, editors, *EMSOFT*, pages 245–254. ACM, 2008.
23. D.F. Bacon, P. Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL'03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.
24. H.G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.
25. H.G. Baker. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Not.*, 27(3):66–70, 1992.
26. T.P. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Syst.*, 32(1–2):49–71, 2006.
27. U. Balli, H. Wu, B. Ravindran, J.S. Anderson, and E.D. Jensen. Utility accrual real-time scheduling under variable cost functions. *IEEE Trans. Comput.*, 56(3):385–401, 2007.
28. John Barnes. *High Integrity Software, The SPARK Approach to Safety and Security*. Praxis Critical Systems Limited, 2006.
29. M. Barr. Embedded systems memory type. *Embedded Systems Programming*, pages 103–104, May 2001.
30. P. Basanta-Val. *Techniques and Extensions for Distributed Real-Time Java*. PhD thesis, Universidad Carlos III de Madrid, 2007.
31. P. Basanta-Val, L. Almeida, M. Garcia-Valls, and I. Estevez-Ayres. Towards a synchronous scheduling service on top of a unicast distributed real-time Java. In *13th IEEE Real Time and Embedded Technology and Applications Symposium, 2007*, pages 123–132, Apr. 2007.
32. P. Basanta-Val, I. Estevez-Ayres, M. Garcia-Valls, and L. Almeida. A synchronous scheduling service for distributed real-time Java. *Parallel and Distributed Systems, IEEE Transactions*, 21(4):506, Apr. 2010.
33. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Avres. Real-time distribution support for residential gateways based on osgi. In *11th IEEE Conference on Consumer Electronics*, 2011.
34. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Avres. Extending the concurrency model of the real-time specification for Java. *Concurrency and Computation: Practice and Experience*, accepted [2010] for publication.
35. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Avres. Non-functional information transmission patterns for distributed real-time Java. *Software: Practice and Experience. Accepted on March 2011.*, accepted [2011] for publication.
36. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. No heap remote objects: Leaving out garbage collection at the server side. In *OTM Workshops*, pages 359–370, 2004.
37. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Agcmemory: A new real-time Java region type for automatic floating garbage recycling. *ACM SIGBED*, 2(3), July 2005.
38. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Enhancing the region model of real-time Java for large-scale systems. In *2nd Workshop on High Performance, Fault Adaptative, Large Scale Embedded Real-Time Systems*, May 2005.

39. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Towards the integration of scoped memory in distributed real-time Java. In *ISORC'05: Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 382–389, Washington, DC, USA, 2005. IEEE Computer Society.
40. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Extendedportal: violating the assignment rule and enforcing the single parent one. In *4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, page 37, October 2006.
41. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Simplifying the dualized threading model of RTSJ. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2008*, pages 265–272, May 2008.
42. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Simple asynchronous remote invocations for distributed real-time Java. *IEEE Transactions on Industrial Informatics*, 5(3):289–298, Aug. 2009.
43. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. A neutral architecture for distributed real-time Java based on RTSJ and rmi. In *15th IEEE Conference on Emerging Technologies and Factory Communication*, pages 1–8, 2010.
44. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Towards a cyber-physical architecture for industrial systems via real-time Java technology. *International Conference on Computer and Information Technology*, 0:2341–2346, 2010.
45. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Towards propagation of non-functional information in distributed real-time Java. In *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010*, pages 225–232, May 2010.
46. P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Using switched-ethernet and linux tc for distributed real-time Java infrastructures. In *Work-in-Progress Proceedings IEEE RTAS 2010*, 2010.
47. P. Basanta-Val, M. Garcia-Valls, I. Estevez-Ayres, and J. Fernandez-Gonzalez. Integrating multiplexing facilities in the set of jrmp subprotocols. *Latin America Transactions, IEEE (Revista IEEE America Latina)*, 7(1):107–113, March 2009.
48. P. Basanta-Val, M. Garcia-Valls, J. Fernandez-Gonzalez, and I. Estevez-Avres. Fine tuning of the multiplexing facilities of javas remote method invocation. *Concurrency and Computation: Practice and Experience*, accepted [2010] for publication.
49. P. Basanta-Val, Ma. Garcia-Valls, and I. Estevez-Ayres. No-heap remote objects for distributed real-time Java. *ACM Trans. Embed. Comput. Syst.*, 10(1):1–25, 2010.
50. A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
51. D.M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *Proc. USENIX Tcl/Tk Workshop, 1996 - Volume 4*, pages 15–15, 1996.
52. A. C. Beck and L. Carro. Low power Java processor for embedded applications. In *Proceedings of the 12th IFIP International Conference on Very Large Scale Integration*, pages 213–228, Darmstadt, Germany, December 2003.
53. W. S. Beebee and M. C. Rinard. An implementation of scoped memory for real-time Java. In *EMSOFT*, pages 289–305, 2001.
54. 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.
55. G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proceedings of the Real-Time Systems Symposium*, pages 68–78, 1999.
56. E. Bini and S.K. Baruah. Efficient computation of response time bounds under fixed-priority scheduling. In *15th International Conference on Real-Time and Network Systems*, pages 95–104, 2007.
57. K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21:123–138, November 1987.

58. B. Blanchet. Escape analysis for JavaTM: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, November 2003.
59. G.E. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 1999. ACM Press.
60. A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57, 2007.
61. M. Boger. *Java in Distributed Systems: Concurrency, Distribution, and Persistence*. John Wiley and Sons, Inc, New York, NY, USA, 2001.
62. T. Bogholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K.G. Larsen. Model-based schedulability analysis of safety critical hard real-time Java programs. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, pages 106–114, New York, NY, USA, 2008. ACM.
63. G. Bollella, T. Canham, V. Carson, V. Champlin, D. Dvorak, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz. Programming with non-heap memory in the real time specification for Java. In *OOPSLA Companion*, pages 361–369, 2003.
64. G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain. Mackinac: Making HotSpot^{TM} real-time. In *ISORC*, pages 45–54. IEEE Computer Society, 2005.
65. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.
66. G. Bollella, S. Graham, and T. J. Lehman. Real-time tspaces. In *IECON'99 Proceedings. The 25th Annual Conference of the IEEE Industrial Electronics Society 1999.*, volume 2, pages 837–842, 1999.
67. G. Bollella and K. Reinholtz. Scoped memory. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 23–25, 2002.
68. M. Bordin and T. Vardanega. Real-time Java from an Automated Code Generation Perspective. In *JTRES'07: Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 63–72, New York, NY, USA, 2007. ACM.
69. A. Borg. A real-time RMI framework for the RTSJ. Available at http://www.cs.york.ac.uk/ftpdir/reports/, 2003.
70. A. Borg, R. Gao, and N. Audsley. A co-design strategy for embedded Java applications based on a hardware interface with invocation semantics. In *Proc. JTRES*, pages 58–67, 2006.
71. A. Borg and A. J. Wellings. Reference objects for RTSJ memory areas. In *OTM Workshops*, pages 397–410, 2003.
72. A. Borg and A.J. Wellings. A real-time RMI framework for the RTSJ. In *Proceedings. 15th Euromicro Conference on Real-Time Systems, 2003*, pages 238–246, 2003.
73. B. Bouyssounouse and J. Sifakis. *Embedded systems design: the ARTIST roadmap for research and development*. Springer, 2005.
74. M. Bowen. *Handel-C Language Reference Manual, 2.1 edition*. Embedded Solutions Limited, 1998.
75. C. Boyapati, A. Salcianu, Jr. W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI'03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM Press.
76. V. A. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *ISMM'08: Proceedings of the 7th international symposium on Memory management*, pages 141–150, New York, 2008. ACM.
77. V. A. Braberman, D. Garbervetsky, and S. Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.
78. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java programming language. In *OOPSLA'98 Proceedings of the*

13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, October 1998.
79. B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353, April 2008.
80. C.P. Bridges and T. Vladimirova. Agent computing applications in distributed satellite systems. In *International Symposium on Autonomous Decentralized Systems, 2009. ISADS'09*, pages 1–8, March 2009.
81. Brigham Young University. JHDL: FPGA CAD Tools. http://www.jhdl.org/, 2006.
82. A.R. Brodtkorb. The Graphics Processor as a Mathematical Coprocessor in MATLAB. In *Proc. CISIS*, pages 822–827, 2008.
83. R.A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, TX, August 1984. ACM Press.
84. B.M. Brosgol, R.J. Hassan II, and S Robbins. Asynchronous transfer of control in the real-time specification for Java. *Ada Lett.*, XXII:95–112, April 2002.
85. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software: Practice and Experience*, 36:1257–1284, 2006.
86. F. Budinsky, D. Steinberg, R. Ellersick, E. Merks, S.A. Brodsky, and T.J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2004.
87. T. Bures, P. Hnetynka, and M. Malohlava. Using a Product Line for Creating Component Systems. In *Proceedings of ACM SAC 2009, Honolulu, Hawaii, U.S.A., Mar 2009*, March 2009.
88. T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA '06: Proc. of the 4th International Conference on Software Engineering Research, Management and Applications*, pages 40–48, USA, 2006. IEEE Computer Society.
89. A. Burns, B. Dobbing, and T. Vardanega. Guide to the use of the ada ravenscar profile in high integrity systems. Technical Report Technical Report YCS-2003-348, University of York (UK), 2003.
90. A. Burns and A.J. Wellings. Processing group parameters in the real-time specification for Java. In *On the Move to Meaningfull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume LNCS 2889, pages 360–370. Springer, 2003.
91. A. Burns and A.J. Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.
92. A. Burns and A.J. Wellings. *Real-Time Systems and Programming Languages:*. Addison Wesley, 4th edition, 2009.
93. C. Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd ed.* Addison-Wesley Professional, Boston, 2002.
94. H. Cai and A.J. Wellings. Temporal isolation in Ravenscar-Java. *The Eighth IEEE International Symposium On Object-Oriented Real-Time Distributed Computing*, pages 364–371, May 2005.
95. S.M. Carta, D. Pani, and L. Raffo. Reconfigurable coprocessor for multimedia application domain. *J. VLSI Signal Process. Syst.*, 44:135–152, August 2006.
96. F.C. Carvalho, C.E. Pereira, T. Silva Elias Jr, and E.P. Freitas. A practical implementation of the fault-tolerant daisy-chain clock synchronization algorithm on can. In *DATE'06: Proceedings of the conference on Design, automation and test in Europe*, pages 189–194, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.
97. V. Cechticky, P. Chevalley, A. Pasetti, and W. Schaufelberger. A Generative Approach to Framework Instantiation. *Proceedings of GPCE*, pages 267–286, September 2003.
98. Z. Chai, W. Zhao, and W. Xu. Real-time Java processor optimized for RTSJ. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC'07, pages 1540–1544, New York, NY, USA, 2007. ACM.

99. Z. Chen. *Java Card technology for Smart Cards: architecture and programmer's guide*. Addison-Wesley, 2000.
100. P. Cheng and G.E. Blelloch. A parallel, real-time garbage collector. In *PLDI'01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 125–136, New York, NY, USA, 2001. ACM.
101. S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *ISMM*, pages 85–96, 2004.
102. N. Cheung, J. Henkel, and S. Parameswaran. Rapid Configuration and Instruction Selection for an ASIP: A Case Study. In *Proc. DATE - Volume 1*, 2003.
103. W-N. Chin, F. Craciun, S. Qin, and M. C. Rinard. Region inference for an object-oriented language. In *PLDI*, pages 243–254, 2004.
104. J-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.
105. M. Clarke, G.S. Blair, G. Coulson, and N. Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. *Lecture Notes in Computer Science*, 2218:160, 2001.
106. P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *International Conference on Supercomputing*, pages 278–285, 1996.
107. C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In Michael Hind and Jan Vitek, editors, *Proceedings of the 1st International Conference on Virtual Execution Environments, VEE 2005, Chicago, IL, USA, June 11–12, 2005*, pages 46–56. ACM, 2005.
108. A. Corsaro. Jrate. Available at http://jrate.sourceforge.net/, 2004.
109. A. Corsaro and R. Cytron. Efficient memory-reference checks for real-time Java. In *LCTES*, pages 51–58, 2003.
110. A. Corsaro and C. Santoro. Design patterns for RTSJ application development. In *OTM Workshops*, pages 394–405, 2004.
111. A. Corsaro and D.C. Schmidt. The design and performance of the jrate real-time Java implementation. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 900–921. Springer-Verlag, 2002.
112. A. Corsaro and D.C. Schmidt. The design and performance of real-time Java middleware. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1155–1167, November 2003.
113. G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A Generic Component Model for Building Systems Software. *ACM Trans. Comput. Syst.*, 26(1):1–42, 2008.
114. I. Crnkovic, M. R. V. Chaudron, and S. Larsson. Component-Based Development Process and Component Lifecycle. In *ICSEA*, page 44, 2006.
115. E. Curley, J. Anderson, B. Ravindran, and E.D. Jensen. Recovering from distributable thread failures with assured timeliness in real-time distributed systems. In *SRDS'06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 267–276, Washington, DC, USA, 2006. IEEE Computer Society.
116. K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
117. Frampton D., Blackburn S.M., Cheng P., Garner R. J., Grove D., Moss J.E.B., and Salishev S.I. Demystifying magic: high-level low-level programming. In *Proceedings of the ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 81–90, 2009.
118. R.I Davis. *On Exploiting Spare Capacity in Hard Real-Time Systems*. PhD thesis, University of York, 1995.
119. R.I. Davis and A. Burns. Hierarchical fixed priority scheduling. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 389–398. IEEE Computer Society, 5–8 Dec 2005.

120. R.I. Davis and A. Burns. An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems. In *Proccedings of Real-Time and Network Systems, RTNS*, 2008.
121. R.I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *Computer Surveys*, 2011 (to appear).
122. R.I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computers*, Apr (Preprint) 2008.
123. G. De Micheli, R. Ernst, and W. Wolf, editors. *Readings in Hardware/Software Co-design*. Kluwer Academic Publishers, 2002.
124. M.A. de Miguel. Solutions to make java-rmi time predictable. In *Proceedings. Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2001. ISORC 2001*, pages 379–386, 2001.
125. M. Debbabi, A. Mourad, C. Talhi, and Yahyaoui H. Accelerating embedded Java for mobile devices. *IEEE Communications Magazine*, 1:79–85, 2005.
126. M. Deters and R. Cytron. Automated discovery of scoped memory regions for real-time Java. In *MSP/ISMM*, pages 132–142, 2002.
127. S. Dey, P. Sanchez, D. Panigrahi, L. Chen, C. Taylor, and K. Sekar. Using a soft core in a SOC design: Experiences with picoJava. *IEEE Design and Test of Computers*, 17(3):60–71, July 2000.
128. J.A. Dianes, M. Diaz, and B. Rubio. ServiceDDS: A framework for real-time p2p systems integration. In *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010*, pages 233–237, May 2010.
129. P. Dibble. RTSJ 1.1 alpha 6 release notes. Available at http://www.rtsj.org/specjavadoc/book_index.html.
130. P. Dibble. The current status of the RTSJ and jsr 282. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '06, pages 1–1, New York, NY, USA, 2006. ACM.
131. P. Dibble and et al. Java Specification Request 282 (RTSJ 1.1). Available at http://jcp.org/en/jsr/detail?id=282.
132. P. Dibble and et al. The Real-Time Specification for Java 1.0.2. Available at http://www.rtsj.org/.
133. P. Dibble and A.J. Wellings. Jsr-282 status report. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES'09, pages 179–182, New York, NY, USA, 2009. ACM.
134. P.C. Dibble. *Real-Time Java Platform Programming*. Amazon, 2nd edition, 2008.
135. S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmark. In *ECOOP*, pages 92–115, 1999.
136. S-T. Dietrich and D. Walker. The evolution of real-time Linux. *Seventh Real-Time Linux Workshop*, http://www.osadl.org/Papers.rtlws-2005-papers.0.html, 2005.
137. E.W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.
138. DIN. *Bahnanwendungen - Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme - Software fĀr Eisenbahnsteuerungs- und Āberwachungssysteme*, deutsche fassung edition, 2001. No. EN 50128; VDE 0831-128:2001-11.
139. DOC. Rtzen project home page. Available at http://doc.ece.uci.edu/rtzen/, 2005.
140. D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, January 1994. ACM Press.
141. D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123, January 1993. ACM Press.

142. T. Domani, E. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.
143. T. Domani, E. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for Java. Technical Report 88.385, IBM Haifa Research Laboratory, 2000. Fuller version of [142].
144. O. Marchi dos Santos and A. J. Wellings. Cost monitoring and enforcement in the Real-Time Specification for Java - a formal evaluation. In *Proceedings of the 26th Real-Time Systems Symposium*, pages 177–186. IEEE Computer Society Press, 2005.
145. D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz. Project Golden Gate: Towards Real-Time Java in Space Missions. In *ISORC*, pages 15–22, 2004.
146. W.K. Edwards. *Core Jini with Book*. Prentice Hall Professional Technical Reference, 1999.
147. EJC. The ejc (embedded Java controller) platform. Available at http://www.embedded-web.com/index.html.
148. T. Endo. A scalable mark-sweep garbage collector on large-scale shared-memory machines. Master's thesis, University of Tokyo, February 1998.
149. T. Endo, K. Taura, and A. Yonezawa. Predicting scalability of parallel garbage collectors on shared memory multiprocessors. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA*, pages 43–43. IEEE Computer Society, 2001.
150. T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.
151. I. Estevez-Avres, M. Garcia-Valls, P. Basanta-Val, and J. Diez-Sanchez. Overall approach for the selection of deterministic service-based real-time composition algorithms in heterogeneous environments. *Concurrency and Computation: Practice and Experience. Accepted on March 2011.*, accepted [2011] for publication.
152. I. Estevez-Ayres, L. Almeida, M. Garcia-Valls, and P. Basanta-Val. An architecture to support dynamic service composition in distributed real-time systems. In *ISORC'07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 249–256, Washington, DC, USA, 2007. IEEE Computer Society.
153. D. Holmes et al. The OVM project. Available at http://www.ovmj.org/, 2004.
154. J.P. Etienne, J. Cordry, and S. Bouzefrane. Applying the CBSE Paradigm in the Real-Time Specification for Java. In *JTRES'06: Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 218–226, USA, 2006. ACM.
155. EUROCAE. *Object-Oriented Technology Supplement to ED-12C and ED-109A*. No. ED-217.
156. EUROCAE. *Software Considerations in Airborne Systems and Equipment Certification*. No. ED-12B.
157. EUROCAE. *Software Considerations in Airborne Systems and Equipment Certification*. No. ED-12C.
158. EUROCAE. *Software Standard for Non-Airborne Systems*. No. ED-109.
159. EUROCAE. *Final Annual Report for Clarification of ED-12B*, 2001. No. ED-94B.
160. S.F. Fahmy, B. Ravindran, and E.D. Jensen. Scheduling distributable real-time threads in the presence of crash failures and message losses. In *Proceedings of the 2008 ACM symposium on Applied computing - SAC'08*, pages 294–301, 2008.
161. A. Ferrari, D. Garbervetsky, V. A. Braberman, P. Listingart, and S. Yovine. Jscoper: Eclipse support for research on scoping and instrumentation for real time Java applications. In *ETX*, pages 50–54, 2005.
162. C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM'01)*, Monterey, CA, April 2001.
163. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley Professional Computing Series. Addison-Wesley, 1995.

164. D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program instrumentation and run-time analysis of scoped memory in Java. *Electr. Notes Theor. Comput. Sci.*, 113:105–121, 2005.
165. D. Garbervetsky, S. Yovine, V. A. Braberman, M. Rouaux, and A. Taboada. Quantitative dynamic-memory analysis for Java. *CCPE*, (doi: 10.1002/cpe.1656), Nov 2010.
166. M. Garcia-Valls, P. Basanta-Val, and I. Estevez-Ayres. Adaptive real-time video transmission over DDS. In *8th IEEE International Conference on Industrial Informatics (INDIN), 2010*, pages 130–135, Jul. 2010.
167. M. Garcia-Valls, I. Estevez-Ayres, P. Basanta-Val, and Carlos Delgado-Kloos. Cosert: A framework for composing service-based real-time applications. In *Business Process Management Workshops 2005*, pages 329–341, October 2005.
168. M. Garcia-Valls, I. Rodriguez-Lopez, L. Fernandez-Villar, I. Estevez-Ayres, and P. Basanta-Val. Towards a middleware architecture for deterministic reconfiguration of service-based networked applications. In *15th IEEE Conference on Emerging Technologies and Factory Communication*, pages 1–4, 2010.
169. A. Garg. Real-time linux kernel scheduler. *Linux Journal* http://www.linuxjournal.com/article/10165, 2009.
170. D. Gay and A. Aiken. Memory management with explicit regions. In *PLDI*, pages 313–323, 1998.
171. D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC*, pages 82–93, 2000.
172. B. Goetz, T. Peierls, J. Bloch, J. Bowbeeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison Wesley, 2006.
173. J. Goldberg, I. Greenberg, R. Clark, E.D. Jensen, K. Kim, and D.M. Wells. Adaptive fault-resistant systems. Available at http://www.csl.sri.com/papers/sri-csl-95-02/, jan 1995.
174. S. Gorappa, J.A. Colmenares, H. Jafarpour, and R. Klefstad. Tool-based configuration of real-time corba middleware for embedded systems. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 342–349, Washington, DC, USA, 2005. IEEE Computer Society.
175. S. Gorappa and R. Klefstad. Empirical evaluation of openccm for java-based distributed, real-time, and embedded systems. In *SAC*, pages 1288–1292, 2005.
176. F. Gruian and Z. Salcic. Designing a concurrent hardware garbage collector for small embedded systems. In *Proceedings of Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC 2005*, pages 281–294. Springer-Verlag GmbH, October 2005.
177. F. Gruian and M. Westmijze. Bluejamm: A bluespec embedded Java architecture with memory management. In *SYNASC'07: Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 459–466, Washington, DC, USA, 2007. IEEE Computer Society.
178. F. Gruian and M. Westmijze. Bluejep: a flexible and high-performance Java embedded processor. In *JTRES'07: Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 222–229, New York, NY, USA, 2007. ACM.
179. F. Gruian and M. Westmijze. Investigating hardware micro-instruction folding in a Java embedded processor. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES'10, pages 102–108, New York, NY, USA, 2010. ACM.
180. N. Gui, V. De Flori, H. Sun, and C. Blondia. A framework for adaptive real-time applications: the declarative real-time OSGi component model. In *Proceedings of the 7th workshop on Reflective and adaptive middleware*, ARM'08, pages 35–40, New York, NY, USA, 2008. ACM.
181. J.C. Palencia Gutierrez and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *IEEE Real-Time Systems Symposium*, pages 26–35, 1998.
182. T.R. Halfhill. Imsys hedges bets on Java. *Microprocessor Report*, August 2000.

183. R.S. Hall and H. Cervantes. Challenges in building service-oriented applications for osgi. *Communications Magazine, IEEE*, 42(5):144–149, May 2004.
184. R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
185. D.S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 53. IEEE Computer Society, 2001.
186. T. Harmon. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California, Irvine, 2009.
187. D.J. Hatley and I.A. Pirbhai. *Strategies for real-time system specification*. Dorset House Publishing Co., Inc., New York, NY, USA, 1987.
188. S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao. The Chimaera Reconfigurable Functional Unit. *IEEE Trans. Very Large Scale Integr. Syst.*, 12:206–217, February 2004.
189. J.R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *Proc. FCCM*, pages 12–, 1997.
190. R. Henriksson. Scheduling real-time garbage collection. In *Proceedings of NWPER'94*, Lund, Sweden, 1994.
191. R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.
192. T. Henties, J.J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. *Electronic Notes in Theoretical Computer Science*, 2009.
193. T. Henzinger and J. Sifakis. The embedded systems design challenge. *FM 2006: Formal Methods*, pages 1–15, 2006.
194. M. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture, 1993*, pages 289–300, 1993.
195. M.T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.P. Lesot, and F. Parain. Java embedded real-time systems: An overview of existing solutions. In *In Proc. 3rd International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 392–399, 2000.
196. M.T. Higuera-Toledano. Hardware-based solution detecting illegal references in real-time Java. In *ECRTS*, pages 229–237, 2003.
197. M.T. Higuera-Toledano. Hardware-based solution detecting illegal references in real-time Java. In *Proceedings. 15th Euromicro Conference on Real-Time Systems (ECRTS 2003)*, pages 229–337, July 2003.
198. M.T. Higuera-Toledano. Towards an understanding of the behavior of the single parent rule in the RTSJ scoped memory model. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 470–479, 2005.
199. M.T. Higuera-Toledano. Hardware support for detecting illegal references in a multiapplication real-time Java environment. *ACM Trans. Embed. Comput. Syst.*, 5:753–772, November 2006.
200. M.T. Higuera-Toledano. Towards an analysis of race carrier conditions in real-time Java. In *IPDPS*, 2006.
201. 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.
202. M.T. Higuera-Toledano, V. Issarny, M. Banâtre, G. Cabillic, JP. Lesot, and F. Parain. Region-based memory management for real-time Java. In *ISORC*, pages 387–394, 2001.
203. 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.
204. G.H. Hilderink, A.W.P. Bakkers, and J.F. Broenink. A distributed real-time Java system based on csp. In *ISORC'00: Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 400–407, Washington, DC, USA, 2000. IEEE Computer Society.

205. M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *MSP/ISMM*, pages 143–156, 2002.
206. C.A.R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
207. J.A. Holgado, A. Moreno, and Capel M.I. Java-based adaptable middleware platform for virtual instrumentation. In *IEEE Symposium on Virtual Environments, Human-Computer Interfaces and Measurement Systems*, pages 144–149, 2007.
208. 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 Proc. ACM/IFIP/USENIX 8th Int'l Middleware Conference (Middleware 2007)*, Vol. 4834:41–59, 2007.
209. B. Huber. Worst-case execution time analysis for real-time Java. Master's thesis, Vienna University of Technology, Austria, 2009.
210. B. Huber, W. Puffitsch, and M. Schoeberl. Worst-case execution time analysis driven object cache design. *Concurrency and Computation: Practice and Experience*, doi: 10.1002/cpe.1763, 2011.
211. R.L. Hudson, R. Morrison, J.E.B. Moss, and D.S. Munro. Garbage collecting the world: one car at a time. *SIGPLAN Not.*, 32:162–175, October 1997.
212. L. Huelsbergen and P. Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In Jones [230], pages 166–175.
213. IBM. Ibm websphere real-time. Available at http://www-306.ibm.com/sotware/webservers/realtime/, 2006.
214. iLAND. middleware for deterministic dynamically reconfigurable networked embedded systems. Available at at http://www.iland-artemis.org, 2010.
215. Imsys. Snap, simple network application platform. Available at http://www.imsys.se/.
216. Imsys. ISAJ reference 2.0, January 2001.
217. Imsys. Im1101c (the Cjip) technical reference manual / v0.25, 2004.
218. Imsys AB. Imsys. Available at http://www.imsystech.com/.
219. Inc Objective Interface Systems. Jcp RTSJ and real-time corba synthesis: Request for proposal, 2001,.
220. Inc Objective Interface Systems. Jcp RTSJ and real-time corba synthesis: Initial submision. Available at http://www.omg.org/docs/realtime/02-06-02.pdf, 2002.
221. Intel. *Intel386 DX Microprocessor Hardware Reference Manual, number 231630, ISBN 1-55512-153-5*, 1991.
222. International Electrotechnical Commission. *IEC61508. Standard for Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems (E/E/PES)*, 1998. No. IEC 61508.
223. Javolution. Available at http://javolution.org.
224. JCP. Java Community Process Program. Available at http://jcp.org/en/home/index.
225. E.D. Jensen. A proposed initial approach to distributed real-time Java. In *Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pages 2–6, March 2000.
226. E.D. Jensen. The distributed real-time specification for java: an initial proposal. *Comput. Syst. Sci. Eng.*, 16(2):65–70, 2001.
227. JEOPARD Project. Deliverable 2.3: Jamaica and FPGA Integration Report. FP7 Project Report, 2010.
228. JEOPARD Project. Deliverable 7.2: Evaluation Results. FP7 Project Report, 2010.
229. M.S. Johnstone and P.R. Wilson. The memory fragmentation problem: solved? In *Proceedings of the 1st international symposium on Memory management*, ISMM'98, pages 26–36, New York, NY, USA, 1998. ACM.
230. R. Jones, editor. *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, October 1998. ACM Press.
231. R.E. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

232. M. Joseph and P.K. Pandya. Finding response times in a real-time system. *Computer Journal*, 29(5):390–395, 1986.
233. T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. Cdx: a family of real-time Java benchmarks. In *JTRES'09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 41–50, New York, NY, USA, 2009. ACM.
234. I.H. Kazi, H.H. Chen, B. Stanley, and D.J. Lilja. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys*, 32(3):213–240, 2000.
235. A. Kim and J.M. Chang. Designing a Java microprocessor core using fpga technology. *IEE Computing & Control Engineering Journal*, 11(3):135–141, June 2000.
236. T. Kim, N. Chang, N. Kim, and H. Shin. Scheduling garbage collector for embedded real-time systems. *SIGPLAN Not.*, 34:55–64, May 1999.
237. H-J Ko and C-J Tsai. A double-issue Java processor design for embedded applications. In *IEEE International Symposium on Circuits and Systems, 2007. ISCAS 2007*, pages 3502–3505, May 2007.
238. H. Kopetz. TTA supported service availability. In *Second International Service Availability Symposium ISAS 2005*, pages 1–14, 2005.
239. H. Kopetz. The complexity challenge in embedded system design. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2008*, pages 3–12. IEEE, 2008.
240. J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.
241. J. Kreuzinger, R. Marston, T. Ungerer, U. Brinkschulte, and C. Krakowski. The komodo project: thread-based event handling supported by a multithreaded Java microcontroller. In *EUROMICRO Conference, 1999. Proceedings. 25th*, volume 2, pages 122–128 vol.2, 1999.
242. A.S. Krishna, D.C. Schmidt, K. Raman, and R. Klefstad. Enhancing real-time corba predictability and performance. In *CoopIS/DOA/ODBASE*, pages 1092–1109, 2003.
243. A. Kung, J.J. Hunt, L. Gauthier, and M. Richard-Foy. Issues in building an ANRTS platform. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES'06, pages 144–151, New York, NY, USA, 2006. ACM.
244. R. Lassalle, D. Masson, S. Midonnet, and et al. LejosRT. Available at http://lejosrt.org.
245. Lattice Semiconductor Corporation. LatticeSC/M HTX Evaluation Board and Reference Design. http://www.hypertransport.org/default.cfm?page=ProductsViewProduct&ProductID=94, 2010.
246. Edward A. Lee. Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, May 2008. Invited Paper.
247. J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed priority preemptive systems. In *proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 110–123, Phoenix, Arizona, December 1992.
248. J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the Real-Time Systems Symposium*, pages 110–123, San jose, California, December 1987. IEEE Computer Society.
249. Lejos. Available at http://lejos.sourceforge.net/.
250. S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. The Java Series. Prentice Hall, 1999.
251. Linux Manual Page. sched_setaffinity(). Available at http://www.die.net/doc/linux/man/man2/sched_setaffinity.2.html, 2006.
252. C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.
253. J.W.S.W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

254. D. Locke, B.S. Andersen, B. Brosgol, M. Fulton, T. Henties, J.J. Hunt, J.O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A.J. Wellings. Safety-critical Java technology specification, public draft. Available at http://www.jcp.org/en/jsr/detail?id=302, 2011.
255. Lockheed Martin. *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, December 2005.
256. F. Loiret, M. Malohlava, A. Plsek, P. Merle, and L. Seinturier. Constructing Domain-Specific Component Frameworks through Architecture Refinement. In *Euromicro SEAA 2009*, 2009.
257. F. Loiret, R. Rouvoy, L. Seinturier, D. Romero, K. Sénéchal, and A. Plšek. An aspect-oriented framework for weaving domain-specific concerns into component-based systems. *To appear in Journal of Universal Computer Science (J.UCS), Special Issue : Software Components, Architectures and Reuse*, pages 709–724, December 2010.
258. F. Loiret, L. Seinturier, L. Duchien, and D. Servat. A Three-Tier Approach for Composition of Real-Time Embedded Software Stacks. In *Proc. of CBSE*, 2010.
259. M.P. Lun and A.S. Fong. Introducing pipelining technique in an object-oriented processor. In *TENCON'02. Proceedings. 2002 IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering*, volume 1, pages 301–305 vol.1, Oct. 2002.
260. D. Masson. Real-Time Systems Simulator. Available at http://igm.univ-mlv.fr/~masson/Softwares/Realtime%20Systems%20Simulator/.
261. D. Masson. RTSJ event manager. Available at http://igm.univ-mlv.fr/~masson/Softwares/RTSJEventManager. on-line reference.
262. D. Masson. Simulation results for advanced mixed traffic scheduling mechanisms. Available at http://igm.univ-mlv.fr/~masson/Softwares/Realtime%20Systems%20Simulator/curves/. on-line reference.
263. D. Masson and S. Midonnet. RTSJ Extensions: Event Manager and Feasibility Analyzer. In *6th Java Technologies for Real-Time and Embedded Systems (JTRES'08)*, volume 343 of *ACM International Conference Proceeding*, pages 10–18, Santa Clara, California, September 2008. (9 pp.).
264. D. Masson and S. Midonnet. Userland Approximate Slack Stealer with Low Time Complexity. In *16th Real-Time and Network Systems (RTNS'08)*, pages 29–38, Rennes, France, October 2008. (10 pp.).
265. D. Masson and S. Midonnet. The jointly scheduling of hard periodic tasks with soft aperiodic events within the Real-Time Specification for Java (RTSJ) . Technical Report hal-00515361, L.I.G.M., Université de Marne-la-Vallée, June 2010. electronic version (34 pp.) Preprint IGM-2010-04.
266. Matrix. Mtx65. Available at http://www.matrix.es/.
267. Maxim. Tini. Available at http://www.maxim-ic.com/products/microcontrollers/tini/.
268. M.C. Merten, A.R. Trick, C.N. George, J.C. Gyllenhaal, and W.W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. *SIGARCH Comput. Archit. News*, 27:136–147, May 1999.
269. M. Meyer. A novel processor architecture with exact tag-free pointers. In *2nd Workshop on Application Specific Processors*, pages 96–103, San Diego, CA, 2003.
270. M. Meyer. A novel processor architecture with exact tag-free pointers. *IEEE Micro*, 24(3):46–55, 2004.
271. M. Meyer. An on-chip garbage collection coprocessor for embedded real-time systems. In *RTCSA'05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, pages 517–524, Washington, DC, USA, 2005. IEEE Computer Society.
272. Michael Gonzalez Harbour. Supporting SMPs in POSIX, private communication, 2006.
273. G. Michel and J. Sachtleben. An integrated gyrotron controller. *Fusion Engineering and Design*, In Press, Corrected Proof:–, 2011.
274. T. Miettinen, D. Pakkala, and M. Hongisto. A method for the resource monitoring of osgi-based software components. In *Software Engineering and Advanced Applications, 2008. SEAA'08. 34th Euromicro Conference*, pages 100–107, September 2008.
275. MIRA Limited, Warwickshire, UK. *MISRA-C: 2004 Guidelines for the use of the C language in critical systems*, October 2004.

276. G.A. Moreno. Creating custom containers with generative techniques. In *GPCE'06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 29–38, New York, NY, USA, 2006. ACM.
277. Nazomi. JA 108 product brief. Available at http://www.nazomi.com.
278. P. Nicolas, S. Lionel, D. Laurence, and C. Thierry. A Component-based and Aspect-Oriented Model for Software Evolution. *Int. Journal of Computer Applications in Technology*, 31(1/2):94–105, 2008.
279. K. Nilsen. Quantitative Analysis of Developer Productivity in C vs. Real-Time Java. *Defense Advanced Research Projects Agency Workshop on Real-Time Java*, 2004.
280. K. Nilsen. A type system to assure scope safety within safety-critical Java modules. In *Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2006.
281. K. Nilsen and W.J. Schmidt. Cost-effective object space management for hardware-assisted real-time garbage collection. *ACM Letters on Programming Languages and Systems*, 1(4):338–354, December 1992.
282. NXP. Pcf 8591. Available at http://www.nxp.com/.
283. J.M. O'Connor and M. Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.
284. V. Olaru, A. Hangan, G. Sebestyen-Pal, and G. Saplacan. Real-time Java and multi-core architectures. In *4th International Conference on Intelligent Computer Communication and Processing, 2008. ICCP 2008*, page 215, Aug. 2008.
285. OMG. Corba component model, 2002.
286. Open Group/IEEE. The open group base specifications issue 6, ieee std 1003.1, 2004 edition. IEEE/1003.1 2004 Edition, The Open Group, 2004.
287. Oracle Labs. Sunspot. Available at http://www.sunspotworld.com/.
288. OSGi Alliance. About the OSGi service platform. Available from: www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf, cited February 2011.
289. OSGi Alliance. OSGi service platform core specification, release 4. Available at http://www.osgi.org/Specifications/HomePage, cited February 2011.
290. Y. Ossia, O. Ben-Yitzhak, I. Goft, E.K. Kolodner, V. Leikehman, and A. Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 129–140, Berlin, June 2002. ACM Press.
291. Parallax. Javelin stamp. Available at http://www.parallax.com.
292. G. Pardo-Castellote. Omg data-distribution service: Architectural overview. In *ICDCS Workshops*, pages 200–206, 2003.
293. PCI-SIG. PCIe Base Specification. http://www.pcisig.com/specifications/pciexpress/, 2010.
294. P.P. Pirinen. Barrier techniques for incremental tracing. In Jones [230], pages 20–25.
295. C. Pitter and M. Schoeberl. Towards a Java multiprocessor. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 144–151, Vienna, Austria, September 2007. ACM Press.
296. C. Pitter and M. Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.
297. F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: Design patterns and semantics. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 101–110, 2004.
298. F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM'07: Proceedings of the 6th international symposium on Memory management*, pages 159–172, New York, NY, USA, 2007. ACM.
299. F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 33–44, New York, NY, USA, 2008. ACM.
300. F. Pizlo, L. Ziarek, P. Maj, A.L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. *SIGPLAN Not.*, 45:146–159, June 2010.

301. F. Pizlo, L. Ziarek, and J. Vitek. Real time Java on resource-constrained platforms with fiji vm. In *JTRES'09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 110–119, New York, NY, USA, 2009. ACM.
302. A. Plsek, F. Loiret, P. Merle, and L. Seinturier. A Component Framework for Java-based Real-time Embedded Systems. In *Proceedings of ACM/IFIP/USENIX 9th International Middleware Conference*, volume 5346/2008, pages 124–143, Leuven, Belgium, December 2008. IEEE Computer Society.
303. A. Plsek, P. Merle, and L. Seinturier. A Real-Time Java Component Model. In *Proceedings of the 11th International Symposium on Object/Component/Service-oriented Real-Time Distributed Computing (ISORC'08)*, pages 281–288, Orlando, Florida, USA, May 2008. IEEE Computer Society.
304. A. Plsek, L. Zhao, V.H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing safety critical Java applications with oSCJ/L0. In *JTRES*, pages 95–101, 2010.
305. T.B. Preusser, P. Reichel, and R.G. Spallek. An embedded GC module with support for multiple mutators and weak references. In Christian Müller-Schloer, Wolfgang Karl, and Sami Yehia, editors, *Architecture of Computing Systems - ARCS 2010, 23rd International Conference, Hannover, Germany, February 22–25, 2010. Proceedings*, volume 5974 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2010.
306. T.B. Preusser, M. Zabel, and R.G. Spallek. Bump-pointer method caching for embedded Java processors. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 206–210, New York, NY, USA, 2007. ACM.
307. T.B. Preusser, M. Zabel, and P. Reichel. The SHAP microarchitecture and Java virtual machine. Technical Report TUD-FI07-02, Fakultaet Informatik, TU Dresden, April 2007.
308. PRISMTech. DDS implementation, 2010. Available on-line at www.prismtech.com/opensplice.
309. M. Prochazka, S. Fowell, and L. Planche. DisCo Space-Oriented Middleware: Architecture of a Distributed Runtime Environment for Complex Spacecraft On-Board Applications. In *4th European Congress on Embedded Real-Time Software (ERTS 2008)*, Toulouse, France, 2008.
310. W. Puffitsch and M. Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 213–221, Vienna, Austria, September 2007. ACM Press.
311. QNX. QNX neutrino RTOS 6.3.2. Available at www.qnx.com/download/download/16841/multicore_user_guide.pdf, 2007.
312. R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 259–269, 1988.
313. K. Raman, Y. Zhang, M. Panahi, J.A. Colmenares, and R. Klefstad. Patterns and tools for achieving predictability and performance with real-time Java. In *RTCSA'05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, pages 247–253, Washington, DC, USA, 2005. IEEE Computer Society.
314. K. Raman, Y. Zhang, M. Panahi, J.A. Colmenares, R. Klefstad, and T. Harmon. Rtzen: Highly predictable, real-time Java middleware for distributed and embedded systems,. In *Middleware*, pages 225–248, 2005.
315. S. Ramos-Thuel and J.P. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS'93)*, 1993.
316. B. Ravindran, J.S. Anderson, and E.D. Jensen. On distributed real-time scheduling in networked embedded systems in the presence of crash failures. In *SEUS'07: Proceedings of the 5th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, pages 67–81, Berlin, Heidelberg, 2007. Springer-Verlag.

317. B. Ravindran, E. Curley, J.S. Anderson, and E.D. Jensen. Assured-timeliness integrity protocols for distributable real-time threads with in dynamic distributed systems. In *EUC'07: Proceedings of the 2007 conference on Emerging direction in embedded and ubiquitous computing*, pages 660–673, Berlin, Heidelberg, 2007. Springer-Verlag.
318. B. Ravindran, E. Curley, J.S. Anderson, and E.D. Jensen. On best-effort real-time assurances for recovering from distributable thread failures in distributed real-time systems. In *ISORC'07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.
319. Recursion Software. Voyager, 2010.
320. J. Regehr. *Handbook of Real-Time and Embedded Systems*, chapter Safe and Structured Use of Interrupts in Real-Time and Embedded Software, pages 16–1 – 16–12. CRC Press, 2007.
321. S. Rho. A distributed hard real-time Java for high mobility components, December 2004.
322. S. Rho, B.K. Choi, and R. Bettati. Design real-time Java remote method invocation: A server-centric approach. In *International Conference on Parallel and Distributed Computing Systems, PDCS 2005, November 14–16, 2005, Phoenix, AZ, USA*, pages 269–276, 2005 2005.
323. T. Richardson. *Developing Dynamically Reconfigurable Real-Time Systems with Real-Time OSGi (RT-OSGi)*. PhD thesis, University of York, 2011.
324. T. Richardson and A. Wellings. An admission control protocol for real-time OSGi. In *The 13th IEEE International Symposium on Object/component/service-oriented Real-time distributed computing*, 2010.
325. T. Richardson and A.J. Wellings. Providing temporal isolation in the OSGi framework. In *7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2009.
326. T. Richardson and A.J. Wellings. On the road to real-time osgi: Extending osgi with an admission control protocol. *International Journal of Computer Systems Science and Engineering (IJCSSE)*, to appear.
327. M. Rinard. Flex compiler infraestructure, 2004.
328. T. Ritzau and P. Fritzson. Decreasing memory overhead in hard real-time garbage collection. In *EMSOFT*, pages 213–226, 2002.
329. S.S. Rodrguez and J.A. Holgado. A home-automation platform towards ubiquitous spaces based on a decentralized p2p architecture. *Advances in Soft Computing, Springer*, 50:304–308, 2008.
330. R. Rouvoy and P. Merle. Leveraging Component-Based Software Engineering with Fraclet. *Annals of Telecommunications, Special Issue on Software Components – The Fractal Initiative*, 64(1–2):65, jan–feb 2009.
331. RTCA. *Software Considerations in Airborne Systems and Equipment Certification*. No. DO-178C.
332. RTCA. *Software Standard for Non-Airborne Systems*. No. DO-278.
333. RTCA. *Final Annual Report for Clarification of DO-178B*, 2001. No. DO-248B.
334. RTCA and EUROCAE. *Software considerations in airborne systems and equipment certification.* Radio Technical Commision for Aeronatics (RTCA), European Organization for Civil Aviation Electronics (EUROCAE), DO178-B, 1992.
335. RTCA/DO-178B. *Software Considerations in Airborne Systems and Equipment Certification*, December 1992.
336. RTI. DDS implementation, 2010. Available on-line at www.rti.com/downloads/.
337. RTJ Computing. simpleRTJ a small footprint Java VM for embedded and consumer devices. Available at http://www.rtjcom.com/, 2000.
338. S. Saewong, R. Ragunathan, J.P. Lehoczky, and M.H. Klein. Analysis of hierar hical fixed-priority scheduling. *Euromicro Conference on Real-Time Systems*, 0:173, 2002.
339. G. Salagnac, C. Rippert, and S. Yovine. Semi-automatic region-based memory management for real-time Java embedded systems. In *RTCSA*, pages 73–80, 2007.
340. G. Salagnac, S. Yovine, and D. Garbervetsky. Fast escape analysis for region-based memory management. *Electr. Notes Theor. Comput. Sci.*, 131:99–110, 2005.

341. A. Salcianu and M. C. Rinard. Pointer and escape analysis for multithreaded programs. In *PPOPP*, pages 12–23, 2001.
342. A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test Comput.*, 18(6):23–33, 2001.
343. Schaumont, P.R. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010.
344. D.C. Schmidt and F. Kuhns. An overview of the real-time corba specification. *IEEE Computer*, 33(6):56–63, 2000.
345. W.J. Schmidt and K. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 76–85, New York, NY, USA, 1994. ACM Press.
346. M. Schoeberl. JOP: A Java optimized processor. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889 of *LNCS*, pages 346–359, Catania, Italy, November 2003. Springer.
347. M. Schoeberl. Restrictions of Java for embedded real-time systems. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 93–100, Vienna, Austria, May 2004. IEEE.
348. M. Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.
349. M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.
350. M. Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006. IEEE.
351. M. Schoeberl. Application experiences with a real-time Java processor. In *Proceedings of the 17th IFAC World Congress*, pages 9320–9325, Seoul, Korea, July 2008.
352. M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.
353. M. Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a JavaProcessor*. CreateSpace, August 2009. Available at http://www.jopdesign.com/doc/handbook.pdf.
354. M. Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, pages 11–16, Tokyo, Japan, March 2009. IEEE Computer Society.
355. M. Schoeberl. A time-predictable object cache. In *Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, pages 99–105, Newport Beach, CA, USA, March 2011. IEEE Computer Society.
356. M. Schoeberl, F. Brandner, and J. Vitek. RTTM: Real-time transactional memory. In *Proceedings of the 25th ACM Symposium on Applied Computing (SAC 2010)*, pages 326–333, Sierre, Switzerland, March 2010. ACM Press.
357. M. Schoeberl and P. Hilber. Design and implementation of real-time transactional memory. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL 2010)*, pages 279–284, Milano, Italy, August 2010. IEEE Computer Society.
358. M. Schoeberl, S. Korsholm, T. Kalibera, and A.P. Ravn. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.*, accepted, 2010.
359. M. Schoeberl, S. Korsholm, C. Thalinger, and A.P. Ravn. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, pages 445–452, Orlando, Florida, USA, May 2008. IEEE Computer Society.

360. M. Schoeberl and W. Puffitsch. Non-blocking real-time garbage collection. *ACM Trans. Embedded Comput. Syst.*, 10(1), 2010.
361. M. Schoeberl, W. Puffitsch, R.U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.
362. T. Schoofs, E. Jenn, S. Leriche, K. Nilsen, L. Gauthier, and M. Richard-Foy. Use of perc pico in the AIDA avionics platform. In *JTRES*, pages 169–178, 2009.
363. U.P. Schultz, K. Burgaard, F.G. Christensen, and J.L. Knudsen. Compiling java for low-end embedded systems. *ACM SIGPLAN Notices*, 38(7):42–50, 2003.
364. D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
365. L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A Component Model Engineered with Components and Aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *Lecture Notes in Computer Science*, pages 139–153, Vasteras, Sweden, Jun 2006. Springer.
366. J. Seligmann and S. Grarup. Incremental mature garbage collection using the train algorithm. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP'95, pages 235–252, London, UK, UK, 1995. Springer-Verlag.
367. N. Shaylor, D. Simon, and W. Bush. A Java virtual machine architecture for very small devices. *ACM SIGPLAN Notices*, 38(7):34–41, 2003.
368. F. Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In Jones [230], pages 130–137.
369. F. Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.
370. F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems (CASES 2000)*, pages 9–17, New York, NY, USA, 2000. ACM.
371. F. Siebert. Constant-time root scanning for deterministic garbage collection. In *Tenth International Conference on Compiler Construction (CC2001)*, Genoa, April 2001.
372. F. Siebert. The jamaica vm. Available at http://www.aicas.com, 2004.
373. F. Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *JTRES*, pages 94–103, 2007.
374. F. Siebert. Limits of parallel marking garbage collection. In *ISMM'08: Proceedings of the 7th international symposium on Memory management*, pages 21–29, New York, NY, USA, 2008. ACM.
375. F. Siebert. Concurrent, parallel, real-time garbage-collection. In *Proceedings of the 2010 international symposium on Memory management*, ISMM'10, pages 11–20, New York, NY, USA, 2010. ACM.
376. E.T. Silva, E.P. Freitas, F.R. Wagner, F.C. Carvalho, and C.E. Pereira. Java framework for distributed real-time embedded systems. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC 2006*, pages 85–92, Apr. 2006.
377. E.T. Silva Jr, D. Andrews, C.E. Pereira, and F.R. Wagner. An Infrastructure for Hardware-Software Co-Design of Embedded Real-Time Java Applications. In *Proc. ISORC*, pages 273–280, 2008.
378. D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk Java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE 2006)*, pages 78–88, New York, NY, USA, 2006. ACM Press.
379. H. Simpson. Four-slot fully asynchronous communication mechanism. In *IEE Proceedings*, pages 17–30, 1990.
380. M.J.S. Smith. *Application-specific Integrated Circuits, ISBN 978-0201500226*. FreeTechBooks, 1997.

381. B. Sprunt, J.P. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings of the Real-Time Systems Symposium*, pages 251–258, Huntsville, AL, USA, December 1988.
382. B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1:27–60, 1989. 10.1007/BF02341920.
383. M. Stanovich, T.P. Baker, A-I. Wang, and M. Gonzalez Harbour. Defects of the posix sporadic server and how to correct them. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:35–45, 2010.
384. G.L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.
385. D.B. Stewart, R.A. Volpe, and P.K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23(12):759–776, December 1997.
386. O. Strom, K. Svarstad, and E. J. Aas. On the utilization of Java technology in embedded systems. *Design Automation for Embedded Systems*, 8(1):87–106, 2003.
387. J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, January 1995.
388. K.N. Su and C.J. Tsai. Fast host service interface design for embedded Java application processor. In *IEEE International Symposium on Circuits and Systems, 2009. ISCAS 2009*, pages 1357–1360, May 2009.
389. Sun. *picoJava-II Microarchitecture Guide*. Sun Microsystems, March 1999.
390. Sun. *picoJava-II Programmer's Reference Manual*. Sun Microsystems, March 1999.
391. J. Sun. Fixed-priority end-to-end scheduling in distributed real-time systems, 1997.
392. J. Sun, M.K. Gardner, and J.W.S. Liu. Bounding completion times of jobs with arbitrary release times, variable execution times, and resource sharing. *IEEE Trans. Softw. Eng.*, 23(10):603–615, 1997.
393. Sun Microsystems. Enterprise JavaBeans. Available at http://java.sun.com/ejb.
394. Sun Microsystems. picojava-ii(tm) microarchitecture guide, March 1999.
395. Sun Microsystems. Java messaging system, 2002.
396. Sun Microsystems. The Javahotspot virtual machine, v1.4.1, 2002.
397. Sun Microsystems. Connected limited device configuration 1.1. Available at http://jcp.org/aboutJava/communityprocess/final/jsr139/, March 2003.
398. Sun Microsystems. Java remote method invocation. Available at http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf, 2004.
399. Sun MicroSystems. The real-time Java platform- a technical white paper. Available at http://labs.oracle.com/projects/mackinac/, cited February 2011.
400. Sun MicroSystems. Sun Java real-time system 2.2 technical documentation. Available from: http://download.oracle.com/javase/realtime/rts_productdoc_2.2.html, Last accessed: February 2011.
401. Sun Microsystems. Javaspaces service specification, version 1.1. Available at http://java.sun.com, October 2000.
402. Systronix. Tstik. Available at http://www.tstik.com/.
403. Systronix. Jstik. Available at http://www.systronix.com/jstik/, 2010.
404. Jr Silva Elias T., D. Barcelos, F. Wagner, and C.E. Pereira. A virtual platform for multi-processor real-time embedded systems. In *JTRES'08: Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 31–37, New York, NY, USA, 2008. ACM.
405. Jr Silva Elias T., F. Wagner, E.P. Freitas, and C.E. Pereira. Hardware support in a middleware for distributed and real-time embedded applications. In *SBCCI'06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 149–154, New York, NY, USA, 2006. ACM.
406. Y.Y. Tan, C.H. Yau, K.M. Lo, W.S. Yu, P.L. Mok, and A.S. Fong. Design and implementation of a Java processor. *Computers and Digital Techniques, IEE Proceedings-*, 153:20–30, 2006.

407. D. Tejera, A. Alonso, and M.A. de Miguel. Predictable serialization in Java. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing. ISORC 2007.*, pages 102–109, May 2007.
408. D. Tejera, R. Tolosa, M.A. de Miguel, and A. Alonso. Two alternative RMI models for real-time distributed applications. In *ISORC'05: Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 390–397, Washington, DC, USA, 2005. IEEE Computer Society.
409. T.S. Tia, J. W.S. Liu, and M. Shankar. Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems. *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 10(1):23–43, 1996.
410. TimeSys. Real-time specification for Java, reference implementation. Available at http://www.timesys.com/.
411. K. Tindell, A. Burns, and A.J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, 1995.
412. M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.
413. M. Tofte and JP. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.
414. 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.
415. S. Uhrig. Evaluation of different multithreaded and multicore processor configurations for soPC. In Koen Bertels, Nikitas J. Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation, 9th International Workshop, SAMOS*, volume 5657 of *Lecture Notes in Computer Science*, pages 68–77. Springer, 2009.
416. S. Uhrig and J. Wiese. jamuth: an IP processor core for embedded Java real-time systems. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 230–237, New York, NY, USA, 2007. ACM Press.
417. United States Government. *Ada'83 Language Reference Manual*, 1983.
418. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.
419. N. Vijaykrishnan and N. Ranganathan. Supporting object accesses in a Java processor. *Computers and Digital Techniques, IEE Proceedings-*, 147(6):435–443, 2000.
420. N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla. Object-oriented architectural support for a Java processor. In Eric Jul, editor, *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 330–354. Springer, 1998.
421. M. Ward and N. C. Audsley. Hardware Compilation of Sequential Ada. In *Proc. CASES*, pages 99–107, 2001.
422. M. Ward and N. C. Audsley. Hardware implementation of the ravenscar ada tasking profile. In *Proc. CASES*, pages 59–68, 2002.
423. M. Ward and N. C. Audsley. A deterministic implementation process for accurate and traceable system timing and space analysis. In *Proc. RTCSA*, pages 432–440, 2007.
424. A. Welc, A.L. Hosking, and S. Jagannathan. Preemption-based avoidance of priority inversion for Java. In *ICPP'04: Proceedings of the 2004 International Conference on Parallel Processing*, pages 529–538, Washington, DC, USA, 2004. IEEE Computer Society.
425. A.J. Wellings. *Concurrent and real-time programming in Java*. John Wiley and Sons, 2004.
426. A.J. Wellings. Multiprocessors and the real-time specification for Java. In *Proceedings of the 11th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing ISORC-2008*, pages 255–261. Computer Society, IEEE, IEEE, May 2008.
427. A.J. Wellings, G. Bollella, P. Dibble, and D. Holmes. Cost enforcement and deadline monitoring in the Real-Time Specification for Java. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2004*, pages 78–85. Computer Society, IEEE, IEEE, May 2004.

428. A.J. Wellings, Y. Chang, and T. Richardson. Enhancing the platform independence of the Real-Time Specification for Java. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ACM International Conference Proceeding Series, pages 61–69, New York, NY, USA, 2009. ACM.
429. A.J. Wellings, R. Clark, D. Jensen, and D. Wells. A framework for integrating the real-time specification for Java and Java's remote method invocation. In *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, pages 13–22, April 2002.
430. A.J. Wellings, R. Clark, E.D. Jensen, and D. Wells. The distributed real-time specification for java: A status report. In *Embedded Systems Conference*, pages 13–22, 2002.
431. A.J. Wellings and MS. Kim. Processing group parameters in the real-time specification for Java. In *Proceedings of JTRES*, 2008.
432. J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, pages 187–206, 1999.
433. J. Whitham, N.C. Audsley, and M. Schoeberl. Using hardware methods to improve time-predictable performance in real-time Java systems. In *Proceedings of the 7th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2009)*, pages 130–139, Madrid, Spain, September 2009. ACM Press.
434. Wikipedia. Programmable Logic Device. http://en.wikipedia.org/wiki/Programmable_logic_device, 2010.
435. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.
436. I. Williams and M. Wolczko. An object-based memory architecture. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 114–130, Martha's Vineyard, MA (USA), September 1990.
437. I.W. Williams. *Object-Based Memory Architecture*. PhD thesis, Department of Computer Science, University of Manchester, 1989.
438. P. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer Berlin / Heidelberg, 1992. 10.1007/BFb0017182.
439. G. Wright, M.L. Seidl, and M. Wolczko. An object-aware memory architecture. Technical Report SML–TR–2005–143, Sun Microsystems Laboratories, February 2005.
440. G. Wright, M.L. Seidl, and M. Wolczko. An object-aware memory architecture. *Sci. Comput. Program*, 62(2):145–163, 2006.
441. K-C. Wu and Y-W. Tsai. Structured ASIC, evolution or revolution? In *Proc. ISPD*, pages 103–106, 2004.
442. Xilinx. Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel. Application Note XAPP529, Xilinx Corporation, 2004.
443. Xilinx. Getting Started with FPGAs - FPGA vs. ASIC. http://www.xilinx.com/company/gettingstarted/fpgavsasic.htm, 2010.
444. T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.
445. M. Zabel, T.B. Preusser, P. Reichel, and R. G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *Prceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62, Lübeck, Germany, Aug. 2007.
446. M. Zabel and R.G. Spallek. Application requirements and efficiency of embedded Java bytecode multi-cores. In *JTRES'10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 46–52, New York, NY, USA, 2010. ACM.
447. A. Zabos and A. Burns. Towards bandwidth optimal temporal partitioning. Available at http://www.cs.york.ac.uk/ftpdir/reports/2009/YCS/442/YCS-2009-442.pdf, 2009.

448. A. Zemva and M. Verderber. FPGA-oriented HW/SW implementation of the MPEG-4 video decoder. *Microprocessors and Microsystems*, 31(5):313–325, 2007.
449. T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 241–251, Washington, DC, USA, 2004. IEEE Computer Society.
450. B. Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, November 1990.