HIJA
High-Intensity Java Application

Project Number IST-511718

HIJA
Safety Critical Java Proposal

Version 0.5
25 May 2006
Draft

Confidentiality: Public Distribution

Authoring Partner Organisation:
aicas

Every effort has been made to ensure that all statements and information contained herein are accurate, however the Partners accept no liability for any error or omission in the same.

©Copyright in this document remains vested in the Partners
## Project Partner Contact Information

<table>
<thead>
<tr>
<th>Company</th>
<th>Contact Person</th>
<th>Address</th>
<th>Telephone</th>
<th>Facsimile</th>
</tr>
</thead>
<tbody>
<tr>
<td>aicas</td>
<td>Fridtjof Siebert</td>
<td>Haid-und-Neu-Strasse 18</td>
<td>+49 721 663 968-23</td>
<td>+49 721 663 968-93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>76131 Karlsruhe, Germany</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bellstream</td>
<td>Mateusz Radziszewski</td>
<td>ul. Niedzialkowski 26a/11</td>
<td>+48 91 432 56 80</td>
<td>+48 91 432 56 90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>70-440 Szczecin, Poland</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FZI Karlsruhe</td>
<td>Marc Schanne</td>
<td>Haid-und-Neu-Strasse 10–14</td>
<td>+49 721 9654 270</td>
<td>+49 721 9654 271</td>
</tr>
<tr>
<td></td>
<td></td>
<td>76131 Karlsruhe, Germany</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Telecom Italia Learning Services</td>
<td>Cristina Brugnoli</td>
<td>Viale Parco de’ Medici, 61</td>
<td>+39 06 368 70960</td>
<td>+39 06 368 80101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00148 Rome, Italy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trialog</td>
<td>Antonio Kung</td>
<td>25 Rue du General Foy</td>
<td>+33 1 447 0100</td>
<td>+33 1 429 48064</td>
</tr>
<tr>
<td></td>
<td></td>
<td>75008 Paris, France</td>
<td></td>
<td></td>
</tr>
<tr>
<td>University Polytechnic of Madrid</td>
<td>Miguel A. de Miguel</td>
<td>Avenida Ramiro de Maeztu 7</td>
<td>+34 913 367 366</td>
<td>+34 913 367 333</td>
</tr>
<tr>
<td></td>
<td></td>
<td>28040 Madrid, Spain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>University of York</td>
<td>Andrew Wellings</td>
<td>Heslington Hall</td>
<td>+44 1904 432742</td>
<td>+44 1904 432767</td>
</tr>
<tr>
<td></td>
<td></td>
<td>York YO10 5DD, United Kingdom</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Contents

1 Proposal Overview ........................................ 1
   1.1 Definitions, Background, and Scope ................... 1
   1.2 Java Technology and Ground Rules .................... 4
   1.3 API Structuring ...................................... 5

2 Threads and Scheduling .................................. 7
   2.1 Concurrency Model .................................. 8
   2.2 Java and Realtime Threads .......................... 9
   2.3 Asynchronous Event and their handlers ............... 12
   2.4 Scheduling .......................................... 21
   2.5 Other RTSJ restrictions .............................. 23

3 Memory .................................................... 25
   3.1 Allocation Model .................................... 25
   3.2 Memory Model ....................................... 26
   3.3 MemoryArea Class Hierarchy ........................ 27
   3.4 Summary ............................................. 36

4 Synchronization .......................................... 37

5 Clocks, Timers, and Time ................................ 39
   5.1 Definitions .......................................... 39
   5.2 Time ................................................. 40
   5.3 Clocks ............................................... 45
   5.4 Timers ............................................... 45
   5.5 Open Issues ......................................... 47
# Safety Critical Java Proposal

## 6 Java Metadata Annotations
- 6.1 Restricting the Use of RTSJ Features
- 6.2 Annotations for Sharing Objects
- 6.3 Annotations for Scoped Memory
  - 6.3.1 Design
  - 6.3.2 Example: Bounded Object Queue
  - 6.3.3 Vector
- 6.4 Analyzing Code for Resource Usage

## 7 Starting Safety Critical Java Applications

## 8 Class Libraries for Safety Critical Applications
- 8.0.1 Enumerations and Iterators

## 9 Safety Critical Java API

## A Optional Features
- A.1 Pinnable Scopes
- A.2 Networks
  - A.2.1 RTCorba
  - A.2.2 Data Distribution Service
- A.3 Device Control
  - A.3.1 Interrupt Handling
  - A.3.2 Real-Time Data Access

## B An Example Safety Critical Application
## Document Control

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Draft</td>
<td>27 June 2005</td>
</tr>
<tr>
<td>0.2</td>
<td>Draft</td>
<td>10 October 2005</td>
</tr>
<tr>
<td>0.3</td>
<td>Draft</td>
<td>20 October 2005</td>
</tr>
<tr>
<td>0.4</td>
<td>Draft</td>
<td>22 May 2006</td>
</tr>
<tr>
<td>0.5</td>
<td>Draft</td>
<td>25 May 2006</td>
</tr>
</tbody>
</table>
Executive Summary

This document contains the current HIJA proposal for safety critical Java. It should not be seen as a single proposal; rather it is a series of proposals for various aspects of a safety critical java standard. It is organized along similar lines as the Real-Time Specification for Java. The appendix contains some optional standard features.
Chapter 1

Proposal Overview

The Real-Time Specification for Java (RTSJ) was designed to address the general needs of adapting Java for use in realtime applications. As Java has matured, it was become increasingly desirable to use Java in ever more critical applications. However, traditional safety critical application require an exceedingly rigorous validation and certification process. Therefore a tighter and smaller Java standard is needed to support these applications through the validation and certification process.

1.1 Definitions, Background, and Scope

The field of safety critical software development is a bit fragmented with the results that the language used is not always uniform. To facilitate the reader’s understanding of this proposal, some definition are in order. Storey [5] provides a number of useful definitions.

- **Safety** is a property of a system that it will not endanger human life or the environment.
- A **safety-related system** is one by which the safety of the equipment or the plant is assured.
- The term **safety critical system** is normally used as a synonym for a safety-related system, although in some cases it may suggest a system of high criticality (e.g. in DEF STAN 00-55, it relates to SIL 4).
- **Safety integrity** is the likelihood of a safety-related system satisfactorily performing the required safety functions under all the stated conditions within a stated period of time.

Some additional definitions from Burns and Wellings [1] are useful as well.

- **Hard realtime components** are those where it is absolutely imperative that output responses to input stimuli occur within a specified deadline.
• **Soft realtime components** are those where output responses are important but the system will still function correctly if the deadlines are occasionally missed.

• **Firm realtime components** have associated deadlines that can be missed occasionally, but there are no benefit from late delivery of their output responses.

In the aviation industry, the DO-178B standard defines the following Software Integrity Levels (SIL – called Development Assurance Levels) for software [4].

• **Level A**—Software whose anomalous behavior would cause or contribute to a failure of system function resulting in a catastrophic failure condition for the aircraft. Where a catastrophic failure is one which would prevent continued safe flight and landing.

• **Level B**—Software whose anomalous behavior would cause or contribute to a failure of system function resulting in a hazardous/severe-major failure condition for the aircraft. Where 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 would cause or contribute to a failure of system function resulting in a major failure condition for the aircraft. Where 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 would cause or contribute to a failure of system function resulting in a minor failure condition for the aircraft. Where a minor failure is one which would not significantly reduce aircraft safety or functionality.

• **Level E**—Software whose anomalous behavior would cause or contribute to a failure of system function with no effect on aircraft operational capability.

Other standards have similarly defined levels and also add a probability of such a failure occurring. For example, in IEC 61508 the maximum probability of a catastrophic failure (for Level A) is defined to be between \(10^{-5}\) and \(10^{-4}\) per year per system. In DEF STANDARD 00-56, SIL levels are defined in terms of a failure and the resulting severity of any resulting accident (see Figure 1).

The type of verification techniques that must be used to show that a software component meets its specification will depend on the SIL (DAL) that has been assigned to that component. For example, Level A and B software might be constrained so that it can be subjected to various static analyses techniques (such as control flow analysis—whether it is control, data, or information). The type and level of structural testing (within a requirements-based testing framework) might be different. For example in DO178-B, Modified Condition Decision Coverage (MCDC) is compulsory at Level A but optional at Level B; only statement level coverage is required at
Level C. Also whether or not the analysis and testing has to be done independently may vary between levels.

For the purpose of defining a Java standard for safety critical application, the document will not distinguish between integrity Levels A and B and will assume that this is the target integrity levels for the safety critical Java specification. However, where appropriate, the proposal mentions extensions to the specification that may be made for Level C software. The proposal assumes that Level D and E software can use the full blown RTSJ (including realtime garbage collection). From a realtime perspective, one can assume that Level A and B may have hard safety related realtime software components for which there must be a suitable probability (commensurate with the overall system failure probability) that deadlines will be met.

Whilst there is no accepted full definition of mission-critical realtime software, there is broad agreement that mission-critical software is deemed vital for the success of the enterprise using that software, and any failure will have a significant negative impact on the enterprise (possibly even its continuing existence). Software at integrity level A, B, and C is clearly mission-critical software in the sense that failure of safety-related software is unlikely to result in a successful mission. However, in general mission-critical software may not (directly) cause loss of life and may not be subject to the rigorous development and assessment processes of Level A, B, and C software. However, this may not uniformly be the case, and mission-critical software may be subject to the same rigorous development process as safety critical software. Possible extension to the base safety critical for mission-critical will only be considered for Level C applications.¹

In summary, the proposal considers three profiles.

¹Mission critical is much broader than even Level C would allow, so a Java standard for mission critical is beyond the scope of this effort.
Safety Critical—DO178-B Levels A and B, may have hard, firm and soft real-time safety-related components. Implementations should provide a range of static analysis techniques (including Stack and Heap Usage Analysis and Timing Analysis). They should be subject to a verified development process (including MCDC testing). Independence of verification should be used.

Mission Critical—DO178-B Level C, may have hard, firm, and soft mission-related components. Implementations should be subject to a verified development process (including statement-level coverage testing). Independence of verification should be used.

Non-Critical—DO178-B Levels D and E, may have hard, firm, and soft real-time mission-related. Implementations should be subject to a verified development process (using only, perhaps, requirements-based testing). Independence of verification will generally not be used.

Clearly there will cases where a particular system will not fit exactly into one of these profiles. However, they do allow some distinctions to be made between different possible subsets of the RTSJ.

1.2 Java Technology and Ground Rules

Two of the key issues in the use of Java technology in safety critical systems are application memory management and concurrency control. Whilst there are many other issues that need to be considered, agreement on a common approach within the Open Group Real-Time and Embedded Systems Forum to these issues is crucial for progress to be made.

This document proposes an approach based on a very simplified version of the RTSJ memory management and concurrency models. The approach has been constrained by five ground rules.

- Any approach should be conservative (and probably incremental). For example, the safety critical community will not suddenly adopt a programming model that allows general asynchronous transfer of controls even if one can show them that they are safe and predictable. It is only in recent years that they have started to accept (very simple) models of concurrency (e.g. the Ada Ravenscar profile). As confidence is obtained, so over the years more expressive models will be accepted. Similarly, the safety critical community currently does not make use of dynamic memory management at software Integrity Levels A and B (excluding simple stacks for method invocations). Any introduction of such a facility must be similarly limited and constrained.

- In order to avoid fragmenting the market place, any realtime extensions to Java should be the same or a subset of the RTSJ. Ideally, a safety critical program would still correctly execute on an RTSJ platform (perhaps via a simple binding layer) albeit with a different performance.
• If functionality is missing from the proposed 1.0.1 version of the RTSJ, new APIs should be proposed via the RTSJ Technical Interpretation Committee (TIC). Ideally, the TIC will respond positively and the new APIs will be added in a later revision of the RTSJ. Alternately, the TIC will propose an alternative means of attaining the same functionality.

• New classes can be added to the API, if those classes can be implemented by using the facilities of the RTSJ.

• Annotations can be used to allow off-line activities (usually supported by tools) to perform (for example) proof of correctness, to show the absence of runtime errors or to allow better timing analysis. Annotations can also be used to provide a means of documenting the assumptions made by the programmer. The focus is on annotations to support the memory management and concurrency models, rather than annotations for proof of correctness or resource usage determination. The rules of Java 5 metadata provide this annotation facility. However, metadata has limitations and will not be appropriate for complexer annotations such as proof of correctness or worst-case execution time analysis, where the Java Modeling Language is more appropriate.

1.3 API Structuring

There are several different possible ways of structuring the safety critical API to obtain compatibility with the RTSJ.

1. One could produce a version of the RTSJ’s API that is a subset at the method level. The main problems with this approach is that it potentially leads to empty classes in the hierarchy. Also, problems exists where the RTSJ (and Java, in general) throw checked exceptions. Any new classes can be placed in a new package under javax.realtime, e.g. javax.realtime.critical.

2. One could produce a new API and provide an RTSJ-compliant binding between the SC Java API and the RTSJ. The main problem with this approach is that significant divergence may occur between the two APIs and the binding layer may become too “thick”.

3. Attempt to mirror the RTSJ class structure but where appropriate assume the existence of a thin binding (e.g. where checked exceptions are thrown but off-line analysis can show their absence, a thin binding to the RTSJ can catch those exceptions and if necessary produce a standard response).

To ensure compatibility with the RTSJ and to provide for scalability (i.e., a program written according to the safety critical profile should execute on a system implementing the mission critical profile), the first approach is adopted here. Where there are public methods or variables defined in java.lang or in the javax.realtime but only used in the packages implementing SC Java, they will be marked accordingly.
Chapter 2

Threads and Scheduling

Traditionally, safety critical systems have been small and sequential. Cyclic executive scheduling has been used to perform the scheduling of any activities with time constraints. Proof of timeliness constraints has been, essentially, by construction and testing. The limitations of this approach are well known. As systems have become larger, there has been a gradual migration to computational models which support simple concurrent tasks. The Ada programming language has led the way in using tasks for realtime, embedded programs, and the emerging version of Ada (Ada 2005) will include an explicit subset of its tasking constructs called the Ravenscar profile.

As the trend moves from sequential to concurrent safety critical systems, it moves from deterministic scheduling to predictable scheduling. Hence, it is no longer possible to show by construction and testing that timing constraints will be met. Consequently, it has become necessary to support this migration by advances in schedulability analyses techniques (called feasibility analyses by the RTSJ). Aspects of these techniques are now mature and accepted by certification authorities (e.g., simple utilization based analysis or response time analysis both using rate monotonic or deadline monotonic priority ordering of tasks).

The analyses techniques consider a task as executing in response to a sequence of invocation requests; each request is usually called a release request and each invocation of the tasks in response to the release request is called a release. Release requests are usually classified as being:

- periodic—usually time-triggered,
- sporadic—usually event-triggered,

---

1In this document, the terms tasks or threads to represent concurrent entities that shares a memory space with other tasks or threads in the same process. Hence, the document uses the term process to mean multiple-tasks (multiply-threads) sharing the same memory space. Each process has (logically and perhaps physically) a separate memory space.

2While it is necessary to analyze a program to determine its resource usage requirements (particularly, its processor execution time requirements), this issue is orthogonal to agreeing on the concurrency model.

3The reader should see the RTSJ 1.0.1. (Specification at www.rtsj.org) for a rigorous definition.
For predictability, several effects must be considered. Aperiodic tasks need to have associated servers which bound the time allocated to their service (e.g. via the RTSJ processing group parameters or, more traditionally via “Sporadic” or “Aperiodic” server technologies). Similarly, for predictability, sporadic tasks (if they are to have hard deadlines) have minimum inter-arrival time constraints that must be enforced at runtime. Shared objects support communication between tasks, so a priority inversion avoidance strategy is needed as well. This inversion is typically avoided by some form of priority inversion control protocol that is able to bound the time a high priority task can be blocked by a low priority task while waiting for access to the shared object. Another important consideration is the size and complexity of any runtime support system or VM necessary to support the concurrency model. A concurrency model for safety critical Java must cover these issues.

2.1 Concurrency Model

Typically realtime systems can be thread based or event based. The RTSJ supports both paradigms using the notion of a schedulable object. Currently (as of version 1.0.1) there is adequate support for periodic realtime threads. However, the next major version of the RTSJ is likely to support sporadic and aperiodic threads as well.\(^4\)

Since both a thread with an explicit \texttt{waitForNextRelease} call and events are equivalent, for simplicity sake, the safety critical Java concurrency model supports only the event based model.

The following subset of RTSJ (including the extension) is proposed for supporting the simple concurrency model necessary for the schedulability analysis techniques to be acceptable to certification authorities, and which does not result in an overly complex runtime system or VM.

- Only no-heap and non-daemon RTSJ schedulable objects (created in the initialization phase) are supported (Java threads are not supported).
- The initial thread is a realtime thread, all other schedulable objects will be asynchronous event handlers.
- All asynchronous event handlers must be bound (i.e. \texttt{BoundAsyncEventHandlers}).
- All asynchronous event handlers must have periodic and sporadic release parameters (asynchronous event handlers with aperiodic handlers are not supported).

\(^4\)The proposal is for a \texttt{waitForNextRelease} method and a \texttt{release} method. For realtime threads with periodic release parameters, \texttt{waitForNextRelease} has the same semantics as (and is synonymous with) \texttt{waitForNextPeriod}. For realtime threads with sporadic or aperiodic release parameters, the proposal provides a mechanism by which the thread can wait for its next release “event” to occur. The \texttt{release} method provides the mechanism to generate the “event” to release a realtime thread with sporadic or aperiodic release parameters. For realtime threads with periodic release parameters, the method allows the thread to be woken early.
• Each handler can only service one event, but an event may release more than
one handler.\footnote{Allowing the same handler to be released from more than one event would make it very difficult
to reason about the MIT between release requests. One could relax this requirement so long as the
MIT between releases can be shown to be bounded.}
• The only scheduler is the default \texttt{RTSJ} preemptive priority-based scheduler
with exactly 28 priorities. There is no support for dynamic priorities.
• Scheduling within priority is FIFO.
• Deadline misses are detected but there is no support for CPU-time monitoring
and processing group parameters (as would be needed in support of aperiodic
server technologies).
• Shared objects are represented by classes with synchronized method. No use
of the synchronized statement is allowed. No suspension for any reason is al-
lowed within a synchronized method (including due to use of the Java thread-
ing wait or sleep mechanisms).
• Priority inversion is controlled by use of the \texttt{RTSJ} priority ceiling emulation
protocol (called the priority protect protocol by POSIX and immediate priority
ceiling inheritance protocol in Ada). The default ceiling priority is
\begin{verbatim}
PriorityScheduler.instance().getMaxPriority()
\end{verbatim}

2.2 Java and Realtime Threads

Given that \textsc{SC Java} only support one realtime thread, the API for the Java \texttt{Thread}
classes will be very restricted.

```
package java.lang;

@SafetyRestricted
public class Thread implements Runnable
{
    @SafetyRestricted
    public Thread();

    @SafetyRestricted
    public void run();

    @SafetyRestricted
    public void start();
}
```

The goal here is to restrict application programmers so that they cannot create con-
ventional Java threads, yet at the same time allow the classes in the \texttt{javax.realtime}
package to extend from the thread class. Unfortunately, Java does not allow this fine
level of control unless the \texttt{Thread} class is in the \texttt{javax.realtime} package as
well. Here, an optional annotation (see section 6.1) is used to note that no attempt
should be made by an application to create a non-realtime thread, either directly or indirectly by extending from the `Thread` class.

The `Schedulable` interface, and `RealtimeThread` and `NoHeapRealtimeThread` classes are equally simple.

```java
package javax.realtime;

import static javax.safetycritical.annotate.ScopeSafe.NEW;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;
import javax.safetycritical.annotate.ScopeSafe;
import javax.safetycritical.annotate.ScopeSafe.Assignable;

public interface Schedulable extends Runnable {
    @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
    public MemoryParameters getMemoryParameters();
    @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
    public ReleaseParameters getReleaseParameters();
    @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
    public SchedulingParameters getSchedulingParameters();
}

package javax.realtime;

import static javax.safetycritical.annotate.SafetyRestricted.Restriction.EPHEMERAL;
import static javax.safetycritical.annotate.ScopeSafe.NEW;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;
import javax.safetycritical.annotate.SafetyRestricted;
import javax.safetycritical.annotate.ScopeSafe;
import javax.safetycritical.annotate.ScopeSafe.Assignable;

public class RealtimeThread extends Thread implements Schedulable {
    @SafetyRestricted
    public RealtimeThread(SchedulingParameters scheduling,
                           ReleaseParameters release,
                           MemoryParameters memory,
                           MemoryArea area,
                           ProcessingGroupParameters group,
                           java.lang.Runnable logic)
    {
        /*...*/
    }
    @SafetyRestricted
    public RealtimeThread(SchedulingParameters schedule,
                           MemoryArea area)
    {
        /*...*/
    }
    @SafetyRestricted
    public RealtimeThread()
    {
        /*...*/
    }
    @SafetyRestricted
    public void deschedulePeriodic()
    {
        /*...*/
    }
    @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
    public static RealtimeThread currentRealtimeThread()
```
package javax.realtime;
import javax.safetycritical.annotate.SafetyRestricted;

public class NoHeapRealtimeThread extends RealtimeThread {
    @SafetyRestricted
    public NoHeapRealtimeThread(SchedulingParameters schedule, MemoryArea area)
    {
        super(schedule, null, null, area, null, null);
        /*...*/
    }
    @SafetyRestricted
    public NoHeapRealtimeThread(SchedulingParameters schedule, ReleaseParameters release)
    {
        super(schedule, release, null, null, null, null);
        /*...*/
    }
    public void start()
    {
        /*...*/
    }
}

25 May 2006  Version 0.5
Confidentiality: Public Distribution
The above classes provide a simplified version of the RTSJ realtime thread model. However, the proposal provides only for periodic and sporadic activities, which can be most easily described as events. Supporting both only complicates the implementation.

### 2.3 Asynchronous Event and their handlers

The event based paradigm in SC Java is implement using the RTSJ asynchronous event handling mechanisms. In keeping with the approach outlined above for threads, events and their handlers have equally simple APIs. First, the AsyncEvent class is given:

```java
package javax.realtime;
public class AsyncEvent {
    @SafetyRestricted
    public AsyncEvent();

    @SafetyRestricted
    public void addHandler(AsyncEventHandler handler);

    @SafetyRestricted
    public void fire();

    @SafetyRestricted
    public void bindTo(String happening);
}
```

To generate periodic asynchronous events requires use of a Timer

```java
package javax.realtime;

public abstract class Timer extends AsyncEvent {
    protected Timer(HighResolutionTime time,
            Clock clock,
            AsyncEventHandler handler);

    public AbsoluteTime getFireTime();
    public void start();
}
```

```java
package javax.realtime;

public class PeriodicTimer extends Timer {
}
```
public PeriodicTimer(HighResolutionTime start,
                    RelativeTime interval,
                    AsyncEventHandler handler);

public AbsoluteTime getFireTime();
public RelativeTime getInterval();
}

OneShotTimers are not supported and all periodic timers are based on the realtime clock.

Since the SC Java programmer should only be able to create periodic events and sporadic events, special classes are needed for implementing classes for these types of events. SC Java provides a new subpackage javax.realtime.critical. For sporadic events, the proposal distinguishes between software generated and hardware generated events.

package javax.safetycritical;
import javax.realtime.AsyncEvent;

/**
 * SporadicEvent is a class of events that enables Java code to
 * release the execution of SporadicEventHandler. The event is
 * software-triggered, in contrast to {link SporadicInterrupt}, which
 * provides an event that may be caused externally.<p>
 * @author Fridtjof Siebert (siebert@aicas.com)
 */
public class SporadicEvent extends AsyncEvent
{
    /* -------------------------- constructors ------------- ---------------*/
    /**
     * Constructor for a sporadic event that is linked to a given
     * handler.
     *
     * @param handler the handler that is to be added to this event.
     */
    public SporadicEvent(SporadicEventHandler handler)
    {
        super();
        addHandler(handler);
    }
    /* ----------------------------- methods --------------- ---------------*/
Safety Critical Java Proposal

/**
 * fire fires this event, i.e., releases the execution of all
 * handlers that were added to this event.
 */
public void fire()
{
    super.fire();
}
}

package javax.safetycritical;

import javax.realtime.AsyncEvent;

public class SporadicInterrupt extends AsyncEvent
{
    /* ---------------------------- variables ---------------------------- */
    /**
     * The handler that was added to this event.
     */
    private SporadicEventHandler handler_;

    /**
     * The happening this event was bound to.
     */
    private String happening_;

    /**
     * has this been bound to the happening?
     */
    private boolean bound = false;

    /* -------------------------- constructors -------------------------- */
    /**
     * Constructor for a new sporadic interrupt that releases the
     * execution of a given handler and that is bound to a given
     * happening.
     * @param handler the handler that is to be added to this event.
     * @param happening a platform-dependent string identifying the
     */

* happening that causes this event, e.g. "SIGINT" for control-C
* events on Unix systems.
*/

public SporadicInterrupt(SporadicEventHandler handler, String happening)
{
    super();
    addHandler(handler);
    this.handler_ = handler;
    this.happening_ = happening;
    Safelet.instance().addSporadicInterrupt(this);
}

/**
   * start starts this interrupt
   */
void start()
{
    bindTo(happening_);
    bound = true;
}

/**
   * cleanup removes the binding of this interrupt to the external
   * happening and removes the handler from this interrupt.
   */
void cleanup()
{
    if (handler_ != null)
    {
        removeHandler(handler_);
    }
    if (bound)
    {
        unbindTo(happening_);
    }
}

These class are not RTSJ classes but are easily implemented using the RTSJ as illustrated above. The implementation integrates the memory management approach as well. No checked exception can be propagated from the enter method. Any unchecked exception (if allowed in this subset) that attempts to propagate is converted to a preallocated ThrowBoundaryError, which could be caught (but this is not shown above).

All events must have their handlers bound when they are created (during the initialization phase). The binding is permanent. The handler hierarchy is given below.6

6There is some discrepancy here between the constructors and those of the RTSJ.

package javax.realtime;

public class AsyncEventHandler implements Schedulable
{
    @SafetyRestricted
    public AsyncEventHandler(SchedulingParameters schedule,
                               ReleaseParameters release,
                               boolean noHeap);
@SafetyRestricted
public AsyncEventHandler(SchedulingParameters schedule,
        ReleaseParameters release,
        boolean noHeap,
        Runnable logic);

public MemoryParameter getMemoryParameters();
public ReleaseParameter getReleaseParameters();
public SchedulingParameter getSchedulingParameters();

@SafetyRestricted
public void handleAsyncEvent();

@SafetyRestricted
public final void run();

@SafetyRestricted
public final void setDaemon(boolean on);
}

package javax.realtime;

public class BoundAsyncEventHandler extends AsyncEventHandler
{
    @SafetyRestricted
    public BoundAsyncEventHandler(SchedulingParameters schedule,
            ReleaseParameters release,
            boolean noHeap);

    @SafetyRestricted
    public BoundAsyncEventHandler(SchedulingParameters schedule,
            ReleaseParameters release,
            boolean noHeap,
            Runnable logic);
}

Again, the goal is for the programmer not to use these classes; instead they should use the following classes:

package javax.safetycritical;

import javax.realtime.BoundAsyncEventHandler;
import javax.realtime.LTMemory;
import javax.realtime.MemoryParameters;
import javax.realtime.PriorityParameters;

...
import javax.realtime.ReleaseParameters;
import javax.safetycritical.SporadicEventHandler;

/* --------------------------------------------------- ------------------- */
/**
* EventHandler is the common super class of PeriodicEventHandler and
* SporadicEventHandler. It contains all the code required to define
* the scheduling, release, memory, logic and cleanup code of event
* handlers.<p>
* This class is abstract, non-abstract sub-classes must implement the
* methods handleEvent and cleanup.<p>
* No sub-classes of this class other then PeriodicEventHandler and
* SporadicEventHandler are allowed.<p>
* @author Fridtjof Siebert (siebert@aicas.com)
*/
public abstract class EventHandler extends BoundAsyncEventHandler
{
    /*-------------------------------- variable -------------------------------*/
    /**
     * Runnable that executes handleEvent(), the logic contained in the
     * event.
     */
    private Runnable task_ = new Runnable()
    {
        public void run()
        {
            handleEvent();
        }
    };

    /**
     * The scoped memory area that was created for the task to execute.
     */
    private LTMemory scope_;

    /*-------------------------------- constructors --------------------------*/
    /**
     * Constructor to create an event handler.
     * @param priority specifies the priority parameters for this
     * periodic event handler. Must not be null.
     * @param parameters specifies the release parameters, must be
     * either PeriodicParameters or SporadicParameters.
     * @param memSize the size in bytes of the memory area to be used
     * for the execution of this event handler. 0 for an empty memory
     * area. Must not be negative.
     * @throws IllegalArgumentException if priority or parameters is null
     * or if memSize is negative or if this is not a subclass or
     * PeriodicEventHandler or SporadicEventHandler.
     */
    public EventHandler(PriorityParameters priority,
            ReleaseParameters parameters,
            long memSize)
    {
        super(/* scheduling */ priority,
            /* release */ parameters,
            /* memory */ new MemoryParameters(0,0),
            /* area */ null,
            /* group */ null,
        );
    }
}
if (priority == null) {
    throw new IllegalArgumentException("priority is null");
} else if (parameters == null) {
    throw new IllegalArgumentException("parameters is null");
} else if (memSize < 0) {
    throw new IllegalArgumentException("memSize is negative");
} else if (! (this instanceof PeriodicEventHandler) && ! (this instanceof SporadicEventHandler)) {
    throw new IllegalArgumentException("EventHandler must be subclass of PeriodicEventHandler or SporadicEventHandler");
} else {
    setDaemon(true);
    LTMemory s = new LTMemory(memSize);
    scope_ = s;
    Safelet.instance().addScope(s);
    Safelet.instance().addEventHandler(this);
}

/**----------------------------- methods -----------------------------*/

/** handleAsyncEvent runs this event within the dedicated scoped memory. This method is final since subclasses should not overwrite this method, but provide an instance of Runnable as logic to execute. */
public final void handleAsyncEvent() {
    scope_.enter(task_);
}

/** handleEvent is an abstract method that contains the logic to execute periodically. This needs to be implemented by any instance of PeriodicEventHandler. */
public abstract void handleEvent();

/** cleanup is an abstract method that contains cleanup code that needs to be executed when the current mission terminates. */
public abstract void cleanup();
package javax.safetycritical;

import javax.realtime.PriorityParameters;
import javax.realtime.PeriodicParameters;
import javax.realtime.PeriodicTimer;

/**
 * PeriodicEventHandler permits the automatic periodic execution of a
 * task that is bound to a periodic timer.<p>
 * This class is abstract, non-abstract sub-classes must implement the
 * methods handleEvent and cleanup.<p>
 * @author Fridtjof Siebert (siebert@aicas.com)
 */
public abstract class PeriodicEventHandler extends EventHandler {

    /**
     * Constructor to create a periodic event handler.
     * @param priority specifies the priority parameters for this
     * periodic event handler. Must not be null.
     * @param parameters specifies the periodic release parameters, in
     * particular the start time, period and deadline miss and cost
     * overrun handlers. Note that a relative start time is not
     * relative to NOW but relative to the point in time when
     * initialization is finished and the timers are started. This
     * argument must not be null.
     * @param memSize the size in bytes of the memory area to be used
     * for the execution of this event handler. 0 for an empty memory
     * area. Must not be negative.
     * @throws IllegalArgumentException if priority, parameters or if
     * memSize is negative.
     */
    public PeriodicEventHandler(PriorityParameters priority,
                                 PeriodicParameters parameters,
                                 long memSize)
    {
        super(priority, parameters, memSize);

        PeriodicTimer timer = new PeriodicTimer(parameters.getStart(),
                                                parameters.getPeriod(),
                                                this);

        Safelet.instance().addTimer(timer);
    }

}
package javax.safetycritical;

import javax.realtime.PriorityParameters;
import javax.realtime.SporadicParameters;

public abstract class SporadicEventHandler extends EventHandler {
    
    public SporadicEventHandler(PriorityParameters priority,
        SporadicParameters parameters,
        long memSize)
    {
        super(priority,parameters,memSize);
    }
}

Note, there is no programmer access to the RTSJ fireCount mechanisms.
2.4 Scheduling

There is no runtime access to the priority-based scheduler. Hence, the `Scheduler` and the `PriorityScheduler` classes are missing. The only support specified is support for the parameter classes and support for priority inversion control.

It is necessary to have restricted `PriorityParameters` that allow the priority to be set and queried but not changed (assuming no dynamic priorities for mode changes).

```java
package javax.realtime;
public class SchedulingParameters {}

package javax.realtime;
public class PriorityParameters {
    public PriorityParameters(int priority);
    public int getPriority();
}
```

There is no `ImportanceParameters` class.

Given the assumption that schedulability analysis is performed off line and that the on line environment is predictable, one failure hypothesis is that deadlines should not be missed. However, some people are not comfortable with this assumption so it is assumed that deadline monitoring is performed.

**SC Java** provides no mechanisms for coping with cost overruns. Consequently, in **SC Java**, the support for this mechanism is removed. This is reflected in the `ReleaseParameters` class hierarchy. Note also the absence of an `AperiodicParameters` class.

```java
package javax.realtime;

import static javax.safetycritical.annotate.ScopeSafe.NEW;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;
import javax.safetycritical.annotate.ScopeSafe.Assignable;

public class ReleaseParameters {
    private RelativeTime deadline_;
    private AsyncEventHandler handler_;
    protected ReleaseParameters(RelativeTime deadline,
                                 AsyncEventHandler handler)
    {
        deadline_ = deadline;
        handler_ = handler;
    }
    @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
    public RelativeTime getDeadline()
    {
        return deadline_;
    }
}
@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public AsyncEventHandler getDeadlineMissHandler()
{
    return handler_;
}
}

package javax.realtime;

import static javax.safetycritical.annotate.ScopeSafe.NEW;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;

import javax.safetycritical.annotate.ScopeSafe;
import javax.safetycritical.annotate.ScopeSafe.Assignable;

public class PeriodicParameters extends ReleaseParameters
{
    HighResolutionTime start_;
    RelativeTime period_;

    public PeriodicParameters(AbsoluteTime start,
                               RelativeTime period,
                               RelativeTime deadline,
                               AsyncEventHandler handler)
    {
        super(deadline, handler);
        start_ = start;
        period_ = period;
    }

    public PeriodicParameters(HighResolutionTime start,
                               RelativeTime period,
                               RelativeTime cost,
                               RelativeTime deadline,
                               AsyncEventHandler overrunHandler,
                               AsyncEventHandler missHandler)
    {
        super(deadline, overrunHandler);
        start_ = start;
        period_ = period;
    }

    @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
    public HighResolutionTime getStart()
    {
        return start_;
    }

    @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
    public RelativeTime getPeriod()
    {
        return period_;
    }
}

package javax.realtime;

import static javax.safetycritical.annotate.ScopeSafe.NEW;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;

import javax.safetycritical.annotate.ScopeSafe;
import javax.safetycritical.annotate.ScopeSafe.Assignable;

public class SporadicParameters extends ReleaseParameters
{
    private RelativeTime period_;

    public SporadicParameters(RelativeTime period,
                               RelativeTime deadline,
                               AsyncEventHandler handler)
    {
        super(deadline, handler);
        period_ = period;
    }

    @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
    public RelativeTime getPeriod()
    {
        return period_;
    }
}

There is an open issue of what should be done with MIT violations.

### 2.5 Other RTSJ restrictions

The following RTSJ classes/interfaces are not available in the SC Java profile:

- AsynchronouslyInterruptedException,
- Interruptible, and
- Timed—no asynchronous transfer of control is allowed.
Chapter 3

Memory

Java augmented by the RTSJ provides three types of memory: heap, immortal, and scoped memory. Only immortal memory is familiar (in concept) to the SCS community since SCS, as a rule, are not allowed to deallocate objects. Once allocated, an object in immortal memory can never be reclaimed. The object must be reused explicitly by the application programmer. Even if realtime garbage collection can be shown to be safe, efficient, and predictable, the jump from the current status quo to such an environment is too great and would probably not be accepted. The controversy over the complexity, the expressive power, and the need for runtime checks of the full scoped memory model, along with the required programming paradigm shift again suggests that such a “leap of faith” is also beyond the SCS community. Safety critical Java provides only immortal memory and a well defined subset of scoped memory.

3.1 Allocation Model

Traditionally, safety critical applications allocate all their data structures before the run phase of the application begins. One speaks of an initialization phase and a mission phase. Systems may also provide for reinitialization in case of some unforeseen circumstance of operation mode change. Taken together, an application with this model has three phases as in figure 3.1: initialization, mission, and recovery. Only small, ephemeral structures are allocated, typically on the execution stack, during the mission phase. The bulk of the allocation takes place in the initialization phase and those structures persist during the life of the mission phase.

Figure 3.1: Safety Critical Execution Phases
A more complicated application may wish to support reinitialization or even mode changes where a different set of data structures may be necessary. This capability is represented by the arrow back to the initialization phase in the diagram. Still only ephemeral structures are allocated during the mission phase.

For an object oriented system, these structures would be objects. Traditionally, Java stores all objects in a heap that is subject to garbage collection. The RTSJ offers additional memory areas for storing objects: immortal memory and scoped memory. The first is a heap where objects are never deallocated and the later is a memory area where object are deallocated once it is no longer in use by any thread. These two types of memory areas are useful for safety critical applications, but for the sake of safety and efficiency, a more limited use is allowed than what the RTSJ provides.

### 3.2 Memory Model

As already stated, the introduction of dynamic memory into a safety critical environment should be limited. A simple nesting structure should be used such that a given scoped memory can only be entered by a single thread at any given time and a scope may only be entered from the memory area in which it is created. These rules simplify scope entry analysis. Furthermore, though immortal memory is simple to understand, it has the enormous disadvantage that objects in immortal memory are not reclaimable. Therefore a global mission scope will be used in the place of immortal memory to hold global objects used during a mission. The advantage is that all objects allocated in this mission memory can be reclaimed whenever the mission is restarted.

Corresponding to an assumed three phase model of application execution, an SC Java system will allocate objects in mission memory in the initialization phase and then in scoped memory for the mission phase of the application. A method of ensuring that class initialization happens only in the initialization phase is needed.

- A scope memory is provided which is entered at the beginning of mission initialization and exited after mission recovery before the next mission is initialized.
- Objects allocated in the initialization phase are never collected during the duration of a given mission unless they are explicitly allocated in (a limited form of) scoped memory.
- Objects allocated in the mission phase are always allocated in a simplified scoped memory area. The execution of a schedulable object can be viewed as a sequence of non-overlapping releases. Every schedulable object has its own private ScopedMemory and all allocations performed during a release of a schedulable object will be performed in this private scoped memory. The memory allocated to created objects will be freed at the end of the release.
- Objects creation in mission scoped memory or immortal memory during the mission phase is explicitly prohibited.
- Programmer annotations are used to document the programmer’s intention and to guide off-line tools in proving that all runtime assignment checks can be omitted.
- Simple nesting of scoped memory areas inside a single release without sharing scoped memory areas between threads is allowed.

The resulting programming model is quite simple. Furthermore, it avoids fragmentation in the underlying memory management system. This will enable confidence to be obtained with the use of dynamic memory, and for more expressive models to be developed in the future.

### 3.3 MemoryArea Class Hierarchy

The MemoryArea class for a pure SC Java system has only a subset of the methods available to a full RTSJ program.

```java
package javax.realtime;
import javax.safetycritical.annotate.SafetyRestricted;
import javax.safetycritical.annotate.ScopeSafe;
import java.lang.reflect.Constructor;
import static javax.safetycritical.annotate.SafetyRestricted.Restriction.EPHEMERAL;

public abstract class MemoryArea
{
    protected MemoryArea() { /* ... */}
    @SafetyRestricted(EPHEMERAL)
    public static MemoryArea getMemoryArea(Object object)
    {
        /* ... */
        return null;
    }
    @SafetyRestricted
    public abstract void enter(Runnable logic);
    public void executeInArea(Runnable logic)
    throws InaccessibleAreaException
    {
        /* ... */
    }
    public Object newInstance(Class type)
    throws IllegalAccessException,
            IllegalArgumentException,
            InstantiationException,
            OutOfMemoryError,
            ExceptionInInitializerError,
            InaccessibleAreaException
    {
        /* ... */
        return null; // dummy return
    }
    public Object newInstance(Constructor c, Object[] args)
    throws IllegalAccessException,
            IllegalArgumentException,
            InstantiationException,
            OutOfMemoryError,
            ExceptionInInitializerError,
```
The MemoryArea class has been significantly reduced compared with its RTSJ counterpart. Also, static analysis applied to the application program will require memory sizes to be accurately determined. Hence, no OutOfMemoryError should be thrown.

Further restrictions imposed by this model reduce the complexity of the underlying runtime support. For example, disallowing a scoped memory area from being entered from any scope other than the one in which it was created and allowing only one schedulable object to be active in each nested scoped memory area make implementing scoped memory areas quite simple. The remainder of the MemoryArea class hierarchy can now be given.
public Object newInstance(Constructor c, Object[] args)
    throws IllegalAccessException,
            IllegalArgumentException,
            InstantiationException,
            OutOfMemoryError,
            ExceptionInInitializerError,
            InaccessibleAreaException
{    
    /\...*/
    return null; // dummy return
}

public Object newArray(Class type, int count)
{
    /\...*/
    return null; // dummy return
}

@ScopeSafe
public abstract long memoryConsumed();
@ScopeSafe
public abstract long memoryRemaining();
@ScopeSafe
public abstract long size();

package javax.realtime;

import javax.safetycritical.annotate.SafetyRestricted;
import javax.safetycritical.annotate.ScopeSafe;

public final class ImmortalMemory extends MemoryArea
{
    @SafetyRestricted
    public static ImmortalMemory instance()
    {
        /\...*/
        return null; // dummy return
    }

    @SafetyRestricted
    public void enter(Runnable logic)
    {
        /\...*/
    }

    @ScopeSafe
    public long memoryConsumed()
    {
        /\...*/
        return 0L; // dummy return
    }

    @ScopeSafe
    public long memoryRemaining()
    {
        /\...*/
        return 0L; // dummy return
    }

    @ScopeSafe
    public long size()
    {
        /\...*/
        return 0L; // dummy return
    }
}

package javax.realtime;
import javax.safetycritical.annotate.SafetyRestricted;

public abstract class ScopedMemory
    extends MemoryArea
{
    public ScopedMemory(long size)
    { /*...*/
    }
    @SafetyRestricted
    public synchronized Object getPortal()
        throws MemoryAccessError, IllegalAssignmentError
    { /*...*/
        return null; // dummy return
    }
    @SafetyRestricted
    public synchronized void setPortal(Object object)
    { /*...*/
    }
    @SafetyRestricted
    public synchronized void join() throws InterruptedException
    { /*...*/
    }
}

package javax.realtime;

import javax.safetycritical.annotate.SafetyRestricted;
import javax.safetycritical.annotate.ScopeSafe;

public class LTMemory extends ScopedMemory
{
    @SafetyRestricted
    public LTMemory(long size)
    { super(size);
        /*...*/
    }
    @SafetyRestricted
    public void enter(Runnable logic)
    { /*...*/
    }
    @ScopeSafe
    public long memoryConsumed()
    { /*...*/
        return 0L; // dummy return
    }
    @ScopeSafe
    public long memoryRemaining()
    { /*...*/
        return 0L; // dummy return
    }
    @ScopeSafe
    public long size()
    { /*...*/
        return 0L; // dummy return
    }
}

package javax.safetycritical;
import static javax.safetycritical.annotate.SafetyRestricted.Restriction.EPHEMERAL;
import java.lang.reflect.Constructor;
import javax.realtime.InaccessibleAreaException;
import javax.realtime.LTMemory;
import javax.safetycritical.annotate.SafetyRestricted;

public class SafetyMemory extends LTMemory {
    @SafetyRestricted(EPHEMERAL)
    public SafetyMemory(long size) {
        super(size);
        // ...
    }

    public Object newArrayInArea(Object object, Class type, int size) {
        return getMemoryArea(object).newArray(type, size);
    }

    public Object newInstanceInArea(Object object, Class type) throws IllegalArgumentException, OutOfMemoryError, ExceptionInInitializerError, InaccessibleAreaException, IllegalAccessException, InstantiationException {
        return getMemoryArea(object).newInstance(type);
    }

    public Object newInstanceInArea(Object object, Constructor constructor, Object[] arguments) throws IllegalArgumentException, OutOfMemoryError, ExceptionInInitializerError, InaccessibleAreaException, IllegalAccessException, InstantiationException {
        return getMemoryArea(object).newInstance(constructor, arguments);
    }

    public void executeInArea(Object object, Runnable logic) {
        getMemoryArea(object).executeInArea(logic);
    }
}

Only SafetyMemory may be instantiated directly by a safety critical program. It is based on LTMemory, whose allocation timing characteristics must be documented. Furthermore, finalizers should not be allowed in user code, since finalization is poorly defined in Java and the same effect can be obtained with try statement that includes a finally clause for any given scope. The schedulable object should not return from the enter method until finalization is completed and its memory is reclaimed. Finalization should be performed at the same priority as the schedulable object. Finalizers are only allowed in system libraries.

Mission memory is nothing other than a SafetyCriticalScope which is provided for the application during startup for holding objects that have a mission life span. This acts like an immortal memory area, except that it can be reinitialized at the end of each mission. All objects needed by the mission are allocated in the mission memory area by the runnable given to the initialize the application. The area is exited only after all tasks have terminated. Optionally, some cleanup may be performed. A typical usage might be as follows.

/* --------------------------------------------------- ----------------- */
import javax.safetycritical.Safelet;
import javax.safetycritical.PeriodicEventHandler;
import javax.safetycritical.SporadicEventHandler;
import javax.safetycritical.SporadicInterrupt;
import javax.realtime.LTMemory;
import javax.realtime.MemoryArea;
import javax.realtime.PeriodicParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;
import javax.realtime.SporadicParameters;

/**
 * test_critical is an example application for the HIJA
 * safety-critical Java profile.
 * If property "TASKS" is set, this property must be a string of
 * digits 1 through 5. All the tasks corresponding to these digits
 * will be started.
 * @author Fridtjof Siebert (siebert@aicas.com)
 */
public class SafetyCriticalExample extends Safelet
{
    /*-------------------------- variables ---------------*/

    /**
     * counter for how often control+C was hit.
     */
import java.util.concurrent.TimeUnit;

public class Application {
  private static final int MAX_CONTROL_C = 5;

  public long missionMemorySize() {
    return 200000;
  }

  public void initialize() {
    System.out.println("Initialzing:");

    String tasks = System.getProperty("TASKS", "12345");
    if (tasks.contains("1")) {
      new PeriodicEventHandler
        (new PriorityParameters(20),
         new PeriodicParameters(null, // start
                                  new RelativeTime(1000,0), // period
                                  new RelativeTime(50,0), // cost
                                  new RelativeTime(1000,0), // deadline
                                  null, // overrunHandler
                                  null), // missHandler
          100000)
        .long start;

      public void handleEvent() {
        if (start == 0) {
          start = System.currentTimeMillis();
        } else {
          System.out.println("beep: ");
        }
      }
    }
  }

  public static void main(String[] args) {
    new Application().initialize();
  }
}
public void cleanup()
{
    System.out.println("cleanup beep!");
};
}
if (tasks.contains("2")
{
    /* another periodic task that prints "PENG!" every 400ms */
    new PeriodicEventHandler
    (new PriorityParameters(20),
    new PeriodicParameters(/* start */ null,
        /* period */ new RelativeTime(400,0),
        /* cost */ new RelativeTime(50,0),
        /* deadline */ new RelativeTime(500,0),
        /* overrunHandler */ null,
        /* missHandler */ null),
        100000)
    {
        long start;
    public void handleEvent()
    {
        if (start == 0)
        {
            start = System.currentTimeMillis();
        }
        else
        {
            System.out.println("PENG! "+(System.currentTimeMillis() - start));
        }
    }
    public void cleanup()
    {
        System.out.println("cleanup PENG!");
    }
    }
}if (tasks.indexOf("3") >= 0)
{
    /* An interrupt handler that reacts on control-C, terminates the
     * mission and starts a new mission or terminates the
     * application. */
    new SporadicInterrupt
    (new SporadicEventHandler
    (new PriorityParameters(25),
    new SporadicParameters(new RelativeTime(20,0), // interarrival
        new RelativeTime(50,0), // cost
        new RelativeTime(500,0), // deadline
        null, // overrunHandler
        null), // missHandler
        20000)
    {
        public void handleEvent()
        {
            control_C_count_++;
            System.out.println("**** INTERRUPT # " + control_C_count_ + " (terminating after " + MAX_CONTROL_C + " interrupts)");
        }
    }
}
if (control_C_count_ < MAX_CONTROL_C)
    restart();
else
terminate();
}

public void cleanup() {}

if (tasks.indexOf("4") >= 0)
{
    /* A sporadic event that is fired on window size change: */
    new SporadicInterrupt
    (new SporadicEventHandler
    (new PriorityParameters(30),
     new SporadicParameters(new RelativeTime(100,0), // interarrival
                             new RelativeTime(50,0), // cost
                             new RelativeTime(500,0), // deadline
                             null), // overrunHandler
      null), // missHandler
10000)
    public void handleEvent()
    {
        System.out.println("WINDOW CHANGED");
    }

    public void cleanup() {}
    "SIGWINCH";
}

if (tasks.indexOf("5") >= 0)
{
    /* a periodic event that performs a lot of memory allocation */
    new PeriodicEventHandler
    (new PriorityParameters(20),
     new PeriodicParameters(new RelativeTime(2000,0), // start
                             new RelativeTime(4000,0), // period
                             new RelativeTime(500,0), // cost
                             new RelativeTime(4000,0), // deadline
                             null), // overrunHandler
      null), // missHandler
10000)
    public void runlocal.run()
    {
        "memory intensive function creating a string of length desired_length and printing this string to System.out if its length is print_length.
        */
        Runnable runlocal = new Runnable()
        {
            public void run()
            {
                /* length of string that should be created by this call to runlocal.run()*/
                int desired_length;
                /* memory intensive function creating a string of length desired_length and printing this string to System.out if its length is print_length.*/
                memory intensive function creating a string of length desired_length and printing this string to System.out if its length is print_length.*/
                MEMORY intensive function creating a string of length desired_length and printing this string to System.out if its length is print_length.*/
            }
        }
    }
}
3.4 Summary

The base memory system is quite simple. Only linear nesting is permitted. Other scoped memory type may be provided as options. Finally, mission memory enables the system to be warm restarted or even change between run modes.
Chapter 4

Synchronization

As in the RTSJ, synchronization between realtime threads can be done via Java monitors. SC Java uses a minimum of the RTSJ interfaces for priority inversion control.

```java
package javax.realtime;

import static javax.safetycritical.annotate.ScopeSafe.PARAMETER;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;

import javax.safetycritical.annotate.ScopeSafe;
import javax.safetycritical.annotate.ScopeSafe.Assignable;

public abstract class MonitorControl
{
    /**
     * Note here that parameter 1 is not actually assigned to parameter 0 but
     * parameter 1 must be in the same scope as or in an outer scope of the scope
     * of parameter 0.
     *
     * @param monitor is the object whose monitor behavior is to be specified.
     * @param control is the specification for monitor behavior
     */
    @ScopeSafe({Assignable(to = PARAMETER + 0, from = { PARAMETER + 1 })})
    public static void setMonitorControl(Object monitor, MonitorControl control)
    {
        /* ... */
    }

    @ScopeSafe({Assignable(to = RETURN, from = { PARAMETER + 0 })})
    public static MonitorControl getMonitorControl(Object monitor)
    {
        /* ... */
        return null;
    }
}
```

```java
package javax.realtime;

import javax.safetycritical.annotate.ScopeSafe;

public class PriorityCeilingEmulation extends MonitorControl
{
    @ScopeSafe
    public static int getMaxCeiling()
    {
```
Only priority ceiling emulation is supported. Ceiling priorities should be static (assuming no, or very limited, support for mode changes); hence, a call to setMonitorControl throws an exception if the ceiling priority has already been set. Off-line tools should show the absence of this error condition.

The intention is that SC Java should facilitate a lockless implementation of synchronized methods.
Chapter 5

Clocks, Timers, and Time

Safety Critical application require precise timing mechanisms for maintaining real-time response. Time is the basis of any realtime system. The RTSJ handles these issues reasonably well, both for keeping time and for measuring time. A safety critical system may use the timing facilities of the RTSJ except those that are deprecated in version 1.0.2 as outlined here.

5.1 Definitions

The RTSJ uses the following definitions when talking about time and for consistencies sake, this specification does the same.

- A timing mechanism is either a Clock or a Timer, capable of representing and following the progress of time, by means of time values.
- A monotonic clock is a clock whose time values are monotonic, and a monotonic non-decreasing clock is a clock whose time values are monotonic non-decreasing. Monotonicity is a boolean property, while time synchronization, uniformity and accuracy are characteristics that depend on agreed tolerances.
- Time synchronization is a relation between two clocks. Two clocks are synchronized when the difference between their time values is less than some specified offset. Synchronization in general degrades with time, and may be lost, given a specified offset.
- Resolution is the minimal time value interval that can be distinguished by a timing mechanism.
- Uniformity, in this context, refers to the measurement of the progress of time at a consistent rate, with a tolerance on the variability. Uniformity is affected by two other factors, jitter and stability.
- Jitter is a short term and non-cumulative small variation caused by noise sources, like thermal noise. More practically, jitter refers to the distribution of the differences between when events are actually fired or noticed by the software and when they should have really occurred according to time in the real world.
• (Lack of) stability accounts for large and often cumulative variations, due to e.g. supply voltage and temperature.

5.2 Time

Three time classes from the RTSJ are available for use in safety critical programs: AbsoluteTime, RelativeTime, and HighResolutionTime. As in the RTSJ, the base time class is HighResolutionTime as follows.

```java
class HighResolutionTime implements Comparable{
  HighResolutionTime(long ms, int ns, Clock clock)
  {
    /*...*/
  }
  @ScopeSafe({@Assignable(to = RETURN, from = {NEW})})
  public abstract AbsoluteTime absolute(Clock clock);
  @ScopeSafe({@Assignable(to = RETURN, from = {PARAMETER+1})})
  public abstract AbsoluteTime absolute(Clock clock, AbsoluteTime dest);
  @ScopeSafe
  public int compareTo(HighResolutionTime time)
  {
    /*...*/
    return -1; // dummy return
  }
  @ScopeSafe
  public int compareTo(java.lang.Object object)
  {
    /*...*/
    return -1; // dummy return
  }
  @ScopeSafe
  public boolean equals(HighResolutionTime time)
  {
    /*...*/
    return false; // dummy return
  }
  @ScopeSafe
  public boolean equals(java.lang.Object object)
  {
    /*...*/
    return false; // dummy return
  }
  @ScopeSafe({@Assignable(to = RETURN, from = {NEW})})
  public Clock getClock()
  {
    /*...*/
    return null; // dummy return
  }
  @ScopeSafe
  void setClock(Clock clock)
  {
```

Confidentiality: Public Distribution
Both `AbsoluteTime` and `RelativeTime` are subclasses thereof.

```java
package javax.realtime;

import static javax.safetycritical.annotate.ScopeSafe.NEW;
import static javax.safetycritical.annotate.ScopeSafe.PARAMETER;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;
import javax.safetycritical.annotate.ScopeSafe;
import javax.safetycritical.annotate.ScopeSafeAssignable;

public class AbsoluteTime extends HighResolutionTime {
    public AbsoluteTime() {
        this(0, 0, null);
    }
}
```
public AbsoluteTime(AbsoluteTime time) {
    this(time, time.getClock());
}

public AbsoluteTime(long millis, int nanos) {
    this(millis, nanos, null);
}

public AbsoluteTime(Clock clock) {
    this(0, 0, clock);
}

public AbsoluteTime(AbsoluteTime time, Clock clock) {
    super(0, 0, clock);  // dummy call
    /* ... */
}

public AbsoluteTime(long millis, int nanos, Clock clock) {
    super(millis, nanos, clock);
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
public AbsoluteTime absolute(Clock clock, AbsoluteTime dest) {
    /* ... */
    return null;  // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public AbsoluteTime absolute(Clock clock) {
    return this;
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public AbsoluteTime add(long millis, int nanos) {
    return add(millis, nanos, null);
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 2 })})
public AbsoluteTime add(long millis, int nanos, AbsoluteTime dest) {
    /* ... */
    return null;  // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public AbsoluteTime add(RelativeTime time) {
    return add(time, null);
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
public AbsoluteTime add(RelativeTime time, AbsoluteTime dest) {
    /* ... */
    return null;  // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public RelativeTime subtract(AbsoluteTime time) {
    return subtract(time, null);
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
public RelativeTime subtract(AbsoluteTime time, RelativeTime dest) {
    /* ... */
    return null;  // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public AbsoluteTime subtract(RelativeTime time) {
    return subtract(time, null);
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
public AbsoluteTime subtract(RelativeTime time, AbsoluteTime dest) {
    /* ... */
    return null;  // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public RelativeTime subtract(RelativeTime time) {
    return subtract(time, null);
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
public RelativeTime subtract(RelativeTime time, RelativeTime dest) {
    /* ... */
    return null;  // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public AbsoluteTime subtract(RelativeTime time) {
    return subtract(time, null);
}
return subtract(time, null);
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
public AbsoluteTime subtract(RelativeTime time, AbsoluteTime dest)
{
    /*...*/
    return null; // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public RelativeTime relative(Clock clock)
{
    return relative(clock, new RelativeTime());
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
public RelativeTime relative(Clock clock, RelativeTime dest)
{
    /*...*/
    return null; // dummy return
}

package javax.realtime;

import static javax.safetycritical.annotate.ScopeSafe.NEW;
import static javax.safetycritical.annotate.ScopeSafe.PARAMETER;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;
import javax.safetycritical.annotate.ScopeSafe;
import javax.safetycritical.annotate.ScopeSafe.Assignable;

public class RelativeTime extends HighResolutionTime
{
    public RelativeTime()
    {
        this(0, 0, null);
    }

    public RelativeTime(long ms, int ns)
    {
        this(ms, ns, null);
    }

    public RelativeTime(RelativeTime time)
    {
        this(time, null);
    }

    public RelativeTime(Clock clock)
    {
        this(0, 0, clock);
    }

    public RelativeTime(long ms, int ns, Clock clock)
    {
        super(ms, ns, clock);
    }

    public RelativeTime(RelativeTime time, Clock clock)
    {
        super(0, 0, clock);
        /*...*/
    }

    @ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
    public AbsoluteTime absolute(Clock clock, AbsoluteTime destination)
    {
        /*...*/
        return null; // dummy return
    }

    @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
    public AbsoluteTime absolute(Clock clock)
    {
return absolute(clock, null);
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public RelativeTime relative(Clock clock)
{
    //...
    return null; // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
public RelativeTime relative(Clock clock, RelativeTime destination)
{
    //...
    return null; // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public RelativeTime add(long millis, int nanos)
{
    return add(millis, nanos, null);
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
public RelativeTime add(long millis, int nanos, RelativeTime dest)
{
    //...
    return null; // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public RelativeTime add(RelativeTime time)
{
    return add(time, null);
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
public synchronized RelativeTime add(RelativeTime time, RelativeTime dest)
{
    //...
    return null; // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public RelativeTime getInterarrivalTime()
{
    return getInterarrivalTime(null);
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 0 })})
public RelativeTime getInterarrivalTime(RelativeTime dest)
{
    //...
    return null; // dummy return
}

@ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
public RelativeTime subtract(RelativeTime time)
{
    return subtract(time, null);
}

@ScopeSafe({@Assignable(to = RETURN, from = { PARAMETER + 1 })})
public synchronized RelativeTime subtract(RelativeTime time, RelativeTime dest)
{
    //...
    return null; // dummy return
}

@ScopeSafe
boolean / pure @/ isNegativeRelativeTime()
{
    //...
    return false; // dummy return
}

@ScopeSafe
boolean / pure @/ isGreaterZero()
{
5.3 Clocks

The clock interface is the same as in the RTSJ.

```java
package javax.realtime;

import static javax.safetycritical.annotate.ScopeSafe.NEW;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;
import java.safetycritical.annotate.ScopeSafe.Assignable;

public abstract class Clock {
    public Clock() {
        return false; // dummy return
    }
}
```

5.4 Timers

Only `Timer` and `PeriodicTimer` are supported in safety critical programs.

```java
package javax.realtime;

import static javax.safetycritical.annotate.ScopeSafe.NEW;
import static javax.safetycritical.annotate.ScopeSafe.PARAMETER;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;
import static javax.safetycritical.annotate.ScopeSafe.THIS;
import java.safetycritical.annotate.ScopeSafe.Assignable;
```
public abstract class Timer extends AsyncEvent
{
    protected Timer(HighResolutionTime time,
                        Clock cclock,
                        AsyncEventHandler handler)
    {
        /*...*/
    }
    @ScopeSafe
    public boolean isRunning()
    {
        /*...*/
        return false; // dummy return
    }
    @ScopeSafe
    public void start()
    {
        /*...*/
    }
    @ScopeSafe
    public void start(boolean disabled)
    {
        /*...*/
    }
    @ScopeSafe
    public boolean stop()
    {
        /*...*/
        return false; // dummy return
    }
    @ScopeSafe
    public ReleaseParameters createReleaseParameters()
    {
        /*...*/
        return null; // dummy return
    }
    @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
    public void enable()
    {
        /*...*/
    }
    @ScopeSafe
    public void disable()
    {
        /*...*/
    }
    @ScopeSafe
    public void destroy()
    {
        /*...*/
    }
    @ScopeSafe
    public void fire() throws UnsupportedOperationException
    {
        /*...*/
    }
    @ScopeSafe({@Assignable(to = RETURN, from = { THIS })})
    public Clock getClock()
    {
        /*...*/
        return null; // dummy return
    }
    @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
    public AbsoluteTime getFireTime()
    {
        /*...*/
        return null; // dummy return
    }
@ScopeSafe({@Assignable(to = THIS, from = { PARAMETER + 0 })})
public void reschedule(HighResolutionTime time)
{ /* ... */
  @ScopeSafe
  void fireIt()
  { /* ... */
  }
}

package javax.realtime;

import static javax.safetycritical.annotate.ScopeSafe.NEW;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;
import javax.safetycritical.annotate.ScopeSafe;
import javax.safetycritical.annotate.ScopeSafe.Assignable;

public class PeriodicTimer extends Timer
{
  public PeriodicTimer(HighResolutionTime start,
           RelativeTime interval,
           AsyncEventHandler handler)
  { this(start, interval, Clock.getRealtimeClock(), handler);
  }
  public PeriodicTimer(HighResolutionTime start,
           RelativeTime interval,
           Clock clock,
           AsyncEventHandler handler)
  { super(start, clock, handler);
  }
  @ScopeSafe
  void checkPeriodic()
  { /* ... */
  }
  @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
  public ReleaseParameters createReleaseParameters()
  { /* ... */
    return null; // dummy return
  }
  @ScopeSafe({@Assignable(to = RETURN, from = { NEW })})
  public RelativeTime getInteval()
  { /* ... */
    return null; // dummy return
  }
  @ScopeSafe
  public void setInterval(RelativeTime newinterval)
  { /* ... */
  }
}

5.5 Open Issues

There are two issues with regard to Clocks that are under discussion in the JSR 282 Working group: how to handle passive clocks and to what extent accuracy, stabil-
ity, and jitter should be represented. Passive clocks are clocks that do not support Timers. This has an impact on how resolution should be defined, since even active clocks, those which can support timers, may have a different resolution for reading and for setting timers. As for the representation of accuracy, stability, and jitter, documentation in an implementation is surely necessary, but reliable comparison of time values from two different clocks would require that at least some of these parameters be explicitly represented in the clock and timer APIs. Close work with the JSR 282 working group is in order here.
Chapter 6

Java Metadata Annotations

A safety critical subset of Java and the RTSJ is necessary since it is too hard to verify the correctness both of a general RTSJ program and to verify that an RTSJ implementation is correct. It is not reasonable to require that the definition of a legal program only encompasses correct programs. To do this would either mean restricting programs to a language subset that is much less capable than a Turing machine, e.g. a state machine, or requiring the compiler to prove program correctness. The first option would not allow the programmer to express most interesting programs at all, and the second approach would require a compiler that is computationally far too complex. The most successful way to restrict a program but still maintain Turing machine equivalence is using a typing system. A safety critical subset of Java with the RTSJ could restrict the set of legal program to those that are legal Java programs but also pass some additional type checking. Other tools would be still needed to help a programmer demonstrate that a given program is correct and some coding conventions would also be helpful, but this goes beyond the scope of what is reasonable to do in language design. There are however some useful extensions to the type system that can be made without violating any Java principles.

Java 5 adds many new capabilities to the Java language that were not available to the designers of the RTSJ. Of particular interest is the new Metadata annotations facility. Metadata annotations enable a tool developer to add additional type-like information to a Java program to enable the more detailed functional and non-functional analyses, both for ensuring program consistency and for aiding the runtime system to produce more efficient code. New metadata annotations can provide a basis for addition checks for ensuring the correctness and efficiency of safety critical Java programs.

Java has a relatively strong type system, but there are still aspects of safety critical code that one would like to specify more tightly. In particular, non functional attributes are difficult to capture in a standard typing system. Metadata annotations and additional checking programs (perhaps also class loaders) could address some of these concerns. This definition of SC Java uses metadata for supporting program correctness analysis, documentation, and usage in three areas:
• usage restrictions on RTSJ features,
• proper use of scoped memory, and
• sharing data between threads.

Usage restrictions are for marking aspects of RTSJ classes that are needed for the implementation of SC Java, but should not be used directly by SC Java programs. Annotations for scoped memory are for determining which classes can be safely passed between scopes and to minimize the need for runtime assignments checks in code that uses scoped memory. Finally, annotations can be used to better document and enforce the proper sharing of objects between threads.

Of course, these are not the only additional feature that one might like to check. One might also like to check more complex features, but for many interesting features Java 5 metadata annotations are not powerful enough. For example, a full set of Hoare clauses for preconditions, postconditions, and invariants as specified by the Java Modeling Language (JML)[3] would be extremely useful for both analyzing functional and non functional correctness. Unfortunately, metadata class definitions do not allow recursive definition, but recursive definitions are needed for defining the full syntax of expression. Not all attributes that might be desirable are applicable for encoding in metadata classes; therefore only relatively simple attributes are captured here.

6.1 Restricting the Use of RTSJ Features

There are classes defined in the RTSJ have quite flexible interfaces. Often, the correct use of their methods can only be ensure by using runtime checks. For a safety critical subset, this may not be desirable in user code, but some of these methods may still be needed internally to implement SC Java. Java provides a mechanism for restricting the visibility of methods to a given package; however SC Java should define all its classes in its own package. This means that any methods from RTSJ classes and used by SC Java must be public.

Where there are public methods or variables defined in javax.realtime but only used by the javax.realtime.critical package, they will be marked with the following annotation with the default value:

```java
package javax.safetycritical.annotate;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.RetentionPolicy.CLASS;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.CONSTRUCTOR;

@Retention(CLASS)
@Target({TYPE, METHOD, CONSTRUCTOR})
public @interface SafetyRestricted {
```
public Restriction value() default Restriction.PROHIBITED;
public static enum Restriction { PROHIBITED, EPHEMERAL; }

Likewise, this annotation may also be used to mark classes in the java.lang package which should only be used in the javax.realtime and javax.realtime.critical packages.

There is also a less restrictive form where the value is set to EPHEMERAL. The object returned by any so marked method may not be stored in any non local variable. The returned object can only be used for calling its methods. This is particularly useful in conjunction with memory scoping.

### 6.2 Annotations for Sharing Objects

When analyzing which objects are shared between threads, there is no generic way of knowing whether or not a given shared object was intended to be shared. One could analyze each object being shared and require that all public methods are synchronized; however this is both time consuming and error prone. A better solution would be to give the class designer the possibility of marking classes as being sharable. Then a compiler could check that classes so marked fulfill the criteria required for proper operation.

In **SC Java**, a sharable class is to be marked by the following annotation class.

```java
package javax.safetycritical.annotate;
import java.lang.annotation.Documented;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.RetentionPolicy.CLASS;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.METHOD;

/**
 * An annotation to mark a class as safe to share between threads.
 */
@Documented
@Inherited
@Retention(CLASS)
@Target({TYPE})
public @interface Sharable {}
```

A sharable class, one marked with this annotation is defined as a class where all public methods are either synchronized or do not have access to the mutable internal state of the class. All subclasses must also fulfill this requirement as well.

Checking for synchronization is an easy check, but checking that a method does not access the immutable state of an object is computationally more complex. Therefore, **SC Java** provides an annotation to mark a method that do not change the state of its enclosing object:
The name Pure is used as an analogue to a mathematically pure function. There are various grades of pure that are interesting for safety critical code. For sharable objects, pure methods should not access any mutable state: @Pure(IMMUTABLE) or simply @Pure. For use in assertions (and also in JML), it is sufficient that calling a pure method does not change the visible state of the object: @Pure(READ) or @Pure(CACHED); whereby showing that a method only reads is easier than demonstrating that it makes use of a private cache object that does not effect the visible state of the object.

With the @Pure notation, checking that a class is sharable reduces to checking that all public methods are either synchronized or @Pure. Other methods are then limited to internal use of the object. A class file user can also more easily know whether or not a class is intended for sharing between threads.

There is an important special case of a sharable class: an immutable class. An immutable class has not mutable state and therefore all methods are pure. Immutable classes are also interesting for use by compilers for optimization and for use with scoped memory. SC Java offers a special notation for immutable classes:
6.3 Annotations for Scoped Memory

One of the concerns with the safety critical subset is the avoidance of runtime checks for memory violations. Annotations can be used to provide a means of documenting for the developer when it is safe to use a certain class or method and what rules apply to the implementation of subclasses thereof for the subclass to be usable in the same contexts as the original method. Static analysis techniques enable these annotations to be verified and, therefore, the absence of runtime memory violation exceptions to be demonstrated.

Consider a method call on an object. The following factors determine whether that call may result in a memory violation occurring.

- The memory area where “this” object (i.e., the called method’s object) was created. If the defining class has any instance reference variables, then these can be used by the object to hold references to other objects which may reside in different memory areas.
- The memory area(s) where any parameter objects reside. A method or constructor may assign references to parameter objects in the defining class’ field variables (instance or static), or field variables reachable from the defining class’s reference variables. Also it may assign references to objects reachable from parameter objects.
- The memory area of the current allocation context of the calling and caller method. The called method may allocate new objects and assign references to them in the defining class’ field variables (instance or static) or field variables reachable from the defining class’s reference variables or parameter objects.
- The memory area of objects whose references are returnable. Returned object references may be assigned by the calling method. These references may refer to objects newly created within the current allocation context (using new), or they may refer to pre-allocated objects reachable from the defining class’s reference variables (instance or static) or they may refer to pre-allocated objects reachable from method’s parameter objects.
- The memory area of any objects whose references may be thrown by exceptions. Thrown exception object references may be assigned by the calling method. These references may refer to objects newly created within the current allocation context (using new), or they may refer to pre-allocated objects reachable from the defining class’ field variables (instance or static) or they may refer to preallocated objects reachable from method’s parameter objects.

Static references, by definition, reside in immortal memory.

There are six possible combinations of memory areas in the mission phase, where the allocation context must always be within scoped memory.

Case 1. The reference variables of “this” object are held in the current scoped memory. The parameter objects are also in the current scoped memory.
Case 2. The reference variables of “this” object are held in the current scoped memory. The parameter objects are in an outer memory area.

Case 3. The reference variables of “this” object are held in the current scoped memory. Some of the parameter objects are in the current scoped memory, others are in an outer memory area.

Case 4. The reference variables of “this” object are held in an outer memory area. The parameter objects are in the current scoped memory.

Case 5. The reference variables of “this” object are held in an outer memory. The parameter objects are in an outer memory area.

Case 6. The reference variables of “this” object are held in an outer memory area. Some of the parameter objects are in the current scoped memory, others are in an outer memory area.

In the initialization phase, the allocation context is a global scoped memory area. Until the mission phase begins, all allocation take place in this memory area. The goal is to enable static analysis tools to expose potential illegal assignment and reference relationships between objects, thus eliminating the need for runtime checks and exceptions.

### Design

Scoped memory was introduced in the RTSJ to enable more flexible allocation without using a garbage collected heap. The advantage of the way scoped memory is defined in the RTSJ is that it can be used quite flexibly, but there is a cost in both efficiency and in the need for additional runtime consistency checks. In a safety critical system, one would still like some of this flexibility, but one would like to have been performance and were possible no runtime checks. Therefore SC Java places some restrictions on the use of scoped memory. Some of these restriction can be made by reducing the available classes and methods, but other require more support from the user. For this support, SC Java supports a new annotation @ScopeSafe.

This new annotation is designed not only to support the base memory model given in chapter 3 with singly nested scopes for each schedulable object. They enable the programmer to provide a simplified view of the allowed data flow within the object with respect to the relative allocation depth in memory. Not every possible legal use of object in nested scopes can be expressed with this notation, but it gives a clear indication of how and when a class can be used across scopes and immortal memory.

```java
package javax.safetycritical.annotate;
import java.lang.annotation.Documented;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.RetentionPolicy.CLASS;
import static java.lang.annotation.ElementType.METHOD;
```
The annotation @ScopeSafe() without any arguments indicates that the given method can be used in any scoped memory context. In particular, this means that the parameters given to the method do not escape the method in any way and the return value does not reside in scoped memory. Adding @AssignableFrom clauses gives the method more freedom of assignment, but it also limits how the method can be used.

An @AssignableFrom clause allows a given kind of assignment to be performed in the method it annotates, but it also indicates under what conditions the method may be safely used. @AssignableFrom has the sense of allowing an assignment from its “from” parameter to its “to” parameter, but it also means that the caller must ensure that the object referred to by “from” resides in a memory area whose depth is less than or equal to that of the object referred by “to”. For the simple SC Java case, immortal memory can be thought of as having a depth of zero, the global scope a depth of one, and the scope of any given release a depth of two. If nesting is allowed, then each nested scoped memory has a depth of one more than its parent as in the RTSJ.¹

Each @AssignableFrom clause may have multiple “from” parameters, but only one “to” parameter. This reflect the fact that any given “to” might be assigned differ-

¹By extension, heap memory could be considered to have a depth between immortal memory and all scoped memory areas.
ently depending on the code execution. Each “to” value may occur at most once in
@ScopeSafe. This simplifies reading for both humans and for tools.

The constants represent each of the interesting types of assignable objects. RETURN,
EXCEPTION, and NEW are notable in that NEW can only be used as a “from” value
and the other three can only be used as “to” values. Parameters can either be consid-
ered collectively with ANY_PARAMETER or individually with PARAMETER+index,
where index is the zero based position of the parameter in the methods argument
list. THIS refers to the methods “this” object and to any object contained by “this”
but not visible from outside the object in any way. CONTEXT describes assignments
and references to any object that is contained within the object but can be obtained
from outside the object. Assignments to other objects contained in CONTENT are not
allowed in a @ScopeSafe object.

CONTENT is special in that an assignment to content means that the “from” object
will become part of the content of the object. This means that any structure between
content and the object must be at the same scoped depth as the object itself. For
example, if an object contains a set of values and this set is contained in an internal
hash table, the hash table must also be allocated in the same memory context as the
object itself. An assignment from NEW to THIS differs from an assignment from
NEW to CONTENT in that the first is used to build up internal structure that may not
be visible from the outside and the second augments the visible content of the object.

For NEW to be useful with EXCEPTION and RETURN with nested scopes, the method
must have some way of obtaining the outermost allocation context where the excep-
tion or return will be referenced. For RETURN, this might be done over a thread local
variable that the caller is required to set. Handling exceptions is more complex. The
simpler model avoids this problem, since not EXCEPTION or RETURN is allowed to
escape its release scope.

As an example, one could take a method that assigns a static field variable:

```java
@ScopeSafe(@Assignable(from = PARAMETER+0, to = STATIC))
public method setStatic(Object value);
```

The caller must ensure that value resides in immortal memory. Since all class objects
are allocated in immortal memory, no object residing in scoped memory may be
assigned to it. Furthermore, if “from” is augmented so that STATIC could be assign
a new object as below, the method could not be called during the mission phase of a
SC Java application.

```java
@ScopeSafe(@Assignable(from = NEW, PARAMETER+0, to = STATIC))
public method setStatic(Object value);
```

Another interesting case is the following:

```java
@ScopeSafe(@Assignable(from = NEW, PARAMETER+0, to = CONTENT))
public method put(Object value);
```
This might be the case in a collection class where either the structure that holds the elements in the collection are allocated lazily as in a vector or the structure is built up over time as in a tree or linked list. Here, the method must ensure that the call to “new” is executed in the same context as the object itself. An analogous rule applies when \texttt{STATIC} is the destination of an \texttt{@Assignable} clause.

### 6.3.2 Example: Bounded Object Queue

The following class illustrates a simple use of annotations.

```java
import static javax.safetycritical.annotate.ScopeSafe.CONTENT;
import static javax.safetycritical.annotate.ScopeSafe.EXCEPTION;
import static javax.safetycritical.annotate.ScopeSafe.NEW;
import static javax.safetycritical.annotate.ScopeSafe.PARAMETER;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;
import javax.safetycritical.annotate.ScopeSafe;
import javax.safetycritical.annotate.ScopeSafe.Assignable;

public class Queue {
    // This would probably be in another class.
    public final static int DEFAULT_QUEUE_SIZE = 100;
    Object [] data;
    int insertIndex, extractIndex = 0;
    int currentSize = 0;
    int queueSize = DEFAULT_QUEUE_SIZE; // default queue size

    // Constructor
    public Queue(int size) {
        if (size > 0) { queueSize = size; }
        data = new Object[queueSize];
    }

    // This is okay with Case 1, 2, 4, or 5
    // There is only one parameter, so it can not be Case 3 or 6.
    @ScopeSafe({@Assignable(from = { PARAMETER + 0 }, to = CONTENT)})
    public synchronized boolean deposit(Object element) {
        if (currentSize < queueSize) {
            data[insertIndex] = element;
            insertIndex = (insertIndex + 1) % queueSize;
            currentSize = currentSize + 1;
            return true;
        }
        else { return false; }
    }

    // This is okay with Case 1, 2, 4, or 5
    // Return object is pre-allocated.
    @ScopeSafe({@Assignable(from = { CONTENT }, to = RETURN),
                @Assignable(from = { NEW }, to = EXCEPTION)})
    public synchronized Object extract() throws QueueEmptyException {
        Object element;
        // Throw the following exception when a thread attempts to extract
        // an element from the queue that is empty at the time of request.
        if (currentSize == 0) { throw new QueueEmptyException(); }
        else {
            element = data[extractIndex];
            data[extractIndex] = null;
            extractIndex = (extractIndex + 1) % queueSize;
            currentSize = currentSize - 1;
            return element;
        }
    }
}
```
Now, consider all the cases and see when illegal assignments will occur using the Deposit method as an illustration.

Deposit assigns from PARAMETER to CONTENT. This is okay in Case 1 since it does not assign anything to static. The situation in Case 2 and 3 are also okay as CONTENT and PARAMETER are within the same scope. However, in Case 4, illegal assignments can occur when PARAMETER is in scoped memory and CONTENT in immortal. In other words, the method Deposit is not scope safe in the situation represented by Case 4. Likewise in Cases 5 and 6, PARAMETER cannot be assigned to CONTENT, so the checker must flag this error.

Checking methods with no parameters needs to consider Case 1 and 4 only. Likewise, methods with one parameter should be checked against Case 1, 2, 4, and 5. Only when there are two or more parameters in different memory areas, should the method be checked against Case 3 and 6.

In summary only the extract, empty, and full methods are safe to use in all contexts.

### 6.3.3 Vector

Annotations can also be used for nested scopes. Proper annotations on library classes can help find problems with standard implementations. Below is an abbreviated implementation of java.util.Vector that can be used with nested scopes.
import static javax.safetycritical.annotate.ScopeSafe.THIS;
import static javax.safetycritical.annotate.ScopeSafe.CONTENT;
import static javax.safetycritical.annotate.ScopeSafe.RETURN;
import static javax.safetycritical.annotate.ScopeSafe.PARAMETER;

public class Vector
{
    private Object[] elements_;
    private int count_;
    private int increment_;;

    public Vector() { this(1, 8); }
    public Vector(int capacity, int increment)
    {
        if (capacity < 0) throw new IllegalArgumentException();
        elements_ = new Object[capacity];
        increment_ = increment;
    }

    private void setCapacity(int value)
    {
        // area may not be assigned outside of this
        // methods lexical scope.
        MemoryArea area = MemoryArea.getMemoryArea(this);
        // Note that the use of new Object[value]
        // would be incorrect here!
        Object[] new_array = (Object[])area.newArray(Object.class, value);
        System.arraycopy(elements_, 0, new_array, 0, elements_.length);
        elements_ = new_array;
    }

    private void checkBounds(int index)
    {
        if (index >= count_)
            throw new ArrayIndexOutOfBoundsException(index + " > " + count_);
    }

    @ScopeSafe({@Assignable(to = RETURN, from = { CONTENT })})
    public synchronized Object elementAt(int index)
    {
        checkBounds(index);
        return elements_[index];
    }

    @ScopeSafe({@Assignable(to = CONTENT, from = { PARAMETER + 0 })),
                @Assignable(to = THIS, from = { NEW }))}
    public synchronized void addElement(Object obj)
    {
        if (count_ == elements_.length)
            ensureCapacity(count_ + 1);
        elements_[count_++] = obj;
    }
checkBounds(index);
Object temp = elements_[index];
elements_[index] = element;
return temp;
}
}

6.4 Analyzing Code for Resource Usage

This proposal does not include any Java 5 Metadata annotations for helping the programmer to determine resource usage. The requirements for such annotations can not be accommodated within the current structure of Java annotations. A parameterized notation would be needed to ensure that general purpose libraries could be build.

The best candidate for this at the moment is the Java Modeling Language (JML). For instance, JML enables the developer to write parameterized descriptions of loop bounds. These exact bounds can then be determined at link time with techniques such as data flow analysis. Usage analysis tools can then use the information for worst case execution and worst case memory use analysis.

```java
/*@ decreasing array.length - index; */
/*@ maintaining index >= 0 && index < array.length; */
for (int index = 0; index < array.length; index++)
{
    done_something(array[index]);
}
```

Unlike using modes or writing loop bounds directly in code, the result is both of general use and accessible for formal verification with tools such as KeY and ESC Java. No refactoring is necessary to use a library in a different domain. Formal verification tool can be used provide additional evidence of correctness for certification. Using JML to help support resource usage analysis provides leverages work being done in functional verification to make sure that use input is actually correct. The HI-JA project is working on demonstrating this capability for an aeronautic navigation system with both level A and level C components.

Unfortunately, including JML notations in the SC Java standard would go beyond the scope of the current mandate. For this reason and because the current state of the art is still in flux, it is advantageous to leave support for resource analysis out of the SC Java. This kind of analysis is not normally part of the compiler. Since JML can be used without change the Java language, this need not inhibit its use. In fact, leaving usage analysis out of the specification may produce better tools, since an immature standard would inhibit further advances.
Chapter 7

Starting Safety Critical Java Applications

The initial startup environment for a purely safety critical system will differ to that of an RTSJ system. In RTSJ, the system starts with a normal Java thread in heap memory. In SC Java, this will not be possible, since SC Java will not have heap memory and therefore only NoHeapRealtimeThreads can be used. To overcome this seaming incompatibility, a startup class is defined analogues to an Applet for web applications or a Midlet for MIDP applications.

A SC Java application is written by extending the Safelet class. The application is defined by overriding the initialize() and the cleanup() methods. The initialize method is responsible for setting up all periodic and sporadic event handlers needed by the application. It is called as soon as the base environment is set up. The cleanup method is called at the end of the mission phase to do any cleanup or shutdown activities.

There are two advantages of setting up an application this way. Firstly, the uses is spared a good bit of startup work. Secondly, the Safelet class can ensure that the initial thread and memory are set up correctly for the application for both RTSJ and SC Java environments. A possible implementation follows.

```java
/* --------------------------------------------------- ------------------ */
* aicas GmbH, Karlsruhe, FRG *
* $Source: /export/home/cvsroot/HIJA/wp8/scj/javax/safetycritical/Safelet.java,v $ *
* $Revision: 1.1 $ *
* $Author: jjh $ *
* Contents: Java source of HIJA SC-Java class Initializer *
* --------------------------------------------------- ------------------ */
package javax.safetycritical;

/]------------------------------------------------------------------------*/
import java.util.Vector;
```
import javax.realtime.AsyncEventHandler;
import javax.realtime.ImmortalMemory;
import javax.realtime.MemoryParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.PriorityScheduler;
import javax.realtime.NoHeapRealtimeThread;
import javax.realtime ScopedMemory;
import javax.realtime.Timer;

/* --------------------------------------------------- ------------------- */
/** * Initializer must be the super class of the main class in a * safety-critical Java application. * * A subclass must implement the method initialize() that sets up the * safety-critical system. The initial instance of the subclass will * be allocated in immortal memory by the safety-critical Java * system. * * initialize() is executed in a realtime thread that is running in * mission memory. By default, mission memory is immortal memory. * However, if method missionMemorySize() is overwritten to return a * value larger than 0, an instance of scoped memory of this size is * created and used as mission memory. * * This class contains a main that enables staring a safety-critical * Java application from a normal RTSJ implementation. * * Usage: <javavm> javax.safetycritical.Initializer <main class> * * @author Fridtjof Siebert (siebert@aicas.com) */
public abstract class Safelet extends NoHeapRealtimeThread
{
    /*----------------------------- variables -------------------*/
    /** * The singleton instance of the initializer. */
    private static Safelet instance_;
    /** * Flag used to indicate the current status of the mission. */
    private boolean running_;
    /*----------------------------- constructors ----------------*/
    /** * Constructor for the initializer. */
    * @throws IllegalStateException if an instance of Initializer has * already been created. */
    public Safelet()
    {
        super(new PriorityParameters(PriorityScheduler.
        instance().getMaxPriority()), // scheduling
        ImmortalMemory.instance()); // area
        // ...
    }
    /*----------------------------- methods ---------------------*/
    /** * instance returns the singleton instance of Initializer. */
static Safelet instance() { return instance_; }

/**
 * addTimer is used during initialization to add a timer that needs
 * to be started before the mission phase and that needs to be
 * destroyed after the mission.
 * @param t the timer, must not be null.
 */
void addTimer(Timer t) {}

/**
 * addEventHandler is used during initialization to add an async
 * event handler that needs to be destroyed after the mission.
 * @param h the async event handler, must not be null
 */
void addEventHandler(AsyncEventHandler h) {}

/**
 * addScope is used during initialization to add a scoped memory area
 * that needs to be freed after the mission.
 * @param s the scoped memory area, must not be null
 */
void addScope(ScopedMemory s) {}

/**
 * addSporadicInterrupt is used during initialization to add a
 * sporadic interrupt that needs to be cleaned up after the mission.
 * @param s the sporadic interrupt, must not be null
 */
void addSporadicInterrupt(SporadicInterrupt s) {}

/**
 * initialize performs the required initialization to start all the
 * activities that need to be performed during the mission phase.<p>
 * This method is abstract since it needs to be implemented by the
 * main class of the safety critical application. <p>
 * During initialization, all resources such as PeriodicEventHandler,
 * SporadicEventHandler, etc. need to be created. <p>
 * initialize will be called automatically by the runtime system of
 * the safety critical Java on an instance that was allocated in
 * immortal memory. The allocation context during the execution of
 * initialize() will be the mission memory.<p>
 */
public abstract void initialize();

/**
 * postMortem can be overridden to provide application dependent
 * cleanup after all handlers have been stop. For Safelets that use
 * missionMemory, an new mode may be set here for use when restarting
 * the application.
 */
public void postMortem() {}
private void startSystem() {}

/**
 * terminate asks for termination of the current mission. If the
 * given boolean argument is false or if mission memory is immortal
 * memory, the safety critical Java system will be terminated
 * completely. Otherwise, the mission will be restarted, i.e., all
 * activities will be stopped, all scoped memory areas including
 * mission memory will be reclaimed, and the system will be started
 * again by a call to initialize().
 */
public void terminate() {}

/**
 * terminate asks for termination of the current mission. If the
 * given boolean argument is false or if mission memory is immortal
 * memory, the safety critical Java system will be terminated
 * completely. Otherwise, the mission will be restarted, i.e., all
 * activities will be stopped, all scoped memory areas including
 * mission memory will be reclaimed, and the system will be started
 * again by a call to initialize().
 */
public void restart() {}

public boolean running()
{
    return running_;
}

/**
 * run is the main routine that performs initialization, starting of
 * the system, cleanup and eventually restring of a mission.<p>
 * This method is final, subclasses cannot overwrite it. Instead,
 * subclasses can overwrite method {initialize}, which is
 * called from run().<p>
 */
public final void run() {}

/**
 * main is the main routine that enables staring a safety-critical
 * Java application from a normal RTSJ implementation.<p>
 * Usage: <javavm> javax.safetycritical.Initializer <main class>
 * @param args array of command line arguments.
 */
public static void main(String[] args) {}
package javax.safetycritical;

/**
   * ClassInitializer provides a simple interface to initialize classes. <p>
   * A safety critical Java application must ensure that all classes are
   * initialized during the initialization phase. This can be performed by
   * calls to ClassInitializer.initialize() as follows:<p>
   *
   * <pre>
   * public void initialize()
   * {
   *  ClassInitializer.initialize(com.mycomany.thisproject.A.class);
   *  ClassInitializer.initialize(com.mycomany.thisproject.B.class);
   *  ClassInitializer.initialize(com.mycomany.thisproject.C.class);
   *  ClassInitializer.initialize(com.mycomany.library1.P.class);
   *  ClassInitializer.initialize(com.mycomany.library1.Q.class);
   *  ClassInitializer.initialize(org.opensource.toolset.X.class);
   *  ClassInitializer.initialize(org.opensource.toolset.Y.class);
   *  ClassInitializer.initialize(org.opensource.toolset.Z.class);
   *  ...
   *  start event handlers etc...
   *  ...
   * } ...
   * </pre>
   *
   * @author Fridtjof Siebert (siebert@aicas.com)
   */
   public class ClassInitializer
   {
   /*----------------------------- methods -----------------------------*/
   /**
   * initialize initializes the given class.
   *
   * @param cl the class to be initialized
   *
   * @throws NullPointerException if cl is null.
   */
   public static void initialize(Class cl)
   {
   try
   {
   Class ignore = cl.forName(cl.getName(),
   true,
   cl.getClassLoader());
   } catch (ClassNotFoundException e)
   {
   throw new InternalError("Loading of already loaded class >>" + cl.getName() +
   "<< failed. This should never happen.");
   }
   }
   }
Chapter 8

Class Libraries for Safety Critical Applications

For safety critical systems, any libraries that the system uses must also be certifiable. Given the costs of the certification process, it is desirable to keep the size of any standard library as small as possible. On the other hand, usability for a wide variety of application would favor a large library set. Highly critical applications still tend to be quite small, so SC Java should start with a small library applicable to DO178B level A applications. Additional supersets can then be added later for applications at levels B and C.

The designers of Java 2 Micro Edition (J2ME), went through a similar process, although for a somewhat different application domain. J2ME defines two configurations and a number of profiles for each configuration. The smaller of the two—Connected Limited Device Configuration (CLDC)—is aimed at devices with very limited resource support, and the larger—Connected Device Configuration (CLDC)—is meant for larger devices. Though many safety critical applications are built on platforms with significantly more resources than a typical CLDC application, the cost of certification would also dictate toward a smaller VM.

Whereas CDC is the same as Java 2 Standard Edition (J2SE) at the language level, CLDC omits support for the following features:

- floating point calculations,
- object finalization, and
- error handling (as opposed to interrupt handling).

Of these, only finalization is not essential to applications in SC Java target domains. A compliant CLDC lack of the following as well:

- Java Native Interface (JNI),
- user-defined class loaders,
- reflection,
- thread groups and thread daemons,
• Object.finalize() method in CLDC libraries,
• weak references, and
• class file verification.

Of these, only JNI is required for SC Java. Thus, CLDC with floating point, error handling, and JNI is a good minimum for an SC Java implementation.

Currently, CLDC has three profiles:
• IMP—Information Module Profile,
• MIDP—Mobile Information Device Profile, and
• STB—Set-Top Box profile.

Of these, IMP is the smallest. It is also closest in application domain to what a typical D0178B level application might need. Thus IMP is a good start for defining a safety critical profile.

Though this is not a complete analysis of all the classes and methods that need be contained in all SC Java implementations, the IMP profile on a CLDC VM extended with floating point, error handling, and JNI should be the basis.

8.0.1 Enumerations and Iterators

Enumerations and iterators are bad citizens in a multi threaded environment. They already make nested scoping difficult because the creation of an enumeration or iteration object can often have a much deeper scope than its eventual point of use. Iterators and Enumerators are difficult in a multi threaded environment because the typical way of use is not thread safe even when all methods are synchronized.

```java
while (someIterator.hasNext())
{
    doSomething(someIterator.next());
}
```

The problem is that some other thread can change the context of the object from which the iterator has been obtained between the call to hasNext() and getNext() causing next() to throw NoSuchElementException. One way to fix this is with a synchronized block:

```java
while (someIterator.hasNext())
{
    // This is often a collection
    synchronized (objectBeingIteratedOver)
    {
        if (someIterator.hasNext())
            doSomething(someIterator.next());
    }
}
```
But this violated the no synchronized block rule and it only works when the iteration is not sensitive to the exact state of the set of objects being acted upon. It does not appear that the extended for syntax, which only works for collections, solves this problem. Of course one could write the following:

```java
try {
    while (someIterator.hasNext()) {
        doSomething(someIterator.next());
    }
} catch (NoSuchElementException error) {}
```

thereby accepting the indeterminate iteration.

When a consistent state is needed, there is no way around something like the following:

```java
synchronized (objectBeingIteratedOver) // This is often a collection {
    while (someIterator.hasNext()) {
        doSomething(someIterator.next());
    }
}
```

Here function passing works better, which in Java means passing an object containing a callback method. The class of the object containing the set of object to be iterated can have a synchronized method that iterates over the set for the caller. For each element, the callers callback method is called:

```java
public class SomeClass ...
{
    ...
    public synchronized doAllSomeSet(Walker callbackObject) {
        while (someIterator.hasNext()) {
            callbackObject.doOne(someIterator.next());
        }
    }
    ...
}

public interface Walker {
    public void doOne(Object element);
}
```
Here the doOne() method in callbackObject might, for example, call the private do-
Something() method.

Ideally, doAll() methods would be the standard way of using collections, but this
proposal can not change the collection framework. It might be possible to provide
wrapper classes that make implementing this functionality easier. It would make
writing scope safe code easier as well.
Chapter 9

Safety Critical Java API

The following list defines the API for SC Java.

Package java.lang

- java.lang.AbstractMethodError
- java.lang.ArithmeticException
- java.lang.ArrayIndexOutOfBoundsException
- java.lang.ArrayStoreException
- java.lang.Boolean
- java.lang.Byte
- java.lang.Class
- java.lang.ClassCastException
- java.lang.ClassCircularityError
- java.lang.ClassFormatError
- java.lang.ClassLoader
- java.lang.ClassNotFoundException
- java.lang.CloneNotSupportedException
- java.lang.Cloneable
- java.lang.Comparable
- java.lang.Compiler
- java.lang.Double
- java.lang.IllegalAccessError
- java.lang.IllegalArgumentException
- java.lang.IllegalMonitorStateException
- java.lang.IllegalStateException
- java.lang.IllegalThreadStateException
- java.lang.IncompatibleClassChangeError
• java.lang.IndexOutOfBoundsException
• java.lang.InheritableThreadLocal
• java.lang.InstantiationException
• java.lang.InterruptedException
• java.lang.LinkageError
• java.lang.Long
• java.lang.Math
• java.lang.NegativeArraySizeException
• java.lang.NoClassDefFoundError
• java.lang.NoSuchFieldError
• java.lang.NoSuchFieldException
• java.lang.NoSuchMethodError
• java.lang.NoSuchMethodException
• java.lang.NullPointerException
• java.lang.Number
• java.lang.NumberFormatException
• java.lang.Object
• java.lang.OutOfMemoryError
• java.lang.Package
• java.lang.Process
• java.lang.Runnable
• java.lang.Runtime
• java.lang.RuntimeException
• java.lang.RuntimePermission
• java.lang.SecurityException
• java.lang.Short
• java.lang.StackOverflowError
• java.lang.System
• java.lang.Thread
• java.lang.ThreadDeath
• java.lang.ThreadGroup
• java.lang.ThreadLocal
• java.lang.Throwable
• java.lang.UnknownError
• java.lang.UnsatisfiedLinkError
• java.lang.UnsupportedClassVersionError
• java.lang.UnsupportedOperationException
• java.lang.VerifyError
• java.lang.VirtualMachineError
• java.lang.Void
• java.lang.SecurityManager
Package java.lang.ref

- java.lang.ref.PhantomReference
- java.lang.ref.Reference
- java.lang.ref.ReferenceQueue
- java.lang.ref.SoftReference
- java.lang.ref.WeakReference

Package java.lang.reflect

- java.lang.reflect.AccessibleObject
- java.lang.reflect.Array
- java.lang.reflect.Constructor
- java.lang.reflect.Field
- java.lang.reflect.InvocationTargetException
- java.lang.reflect.Member
- java.lang.reflect.Method
- java.lang.reflect.Modifier
- java.lang.reflect.ReflectPermission

Package java.io

- java.io.BufferedInputStream
- java.io.BufferedOutputStream
- java.io.BufferedReader
- java.io.BufferedWriter
- java.io.ByteArrayInputStream
- java.io.ByteArrayOutputStream
- java.io.CharArrayReader
- java.io.CharArrayWriter
- java.io.CharConversionException
- java.io.DataInput
- java.io.DataInputStream
- java.io.DataOutput
- java.io.DataOutputStream
- java.io.EOFException
- java.io.Externalizable
- java.io.File
- java.io.FileDescriptor
- java.io.FileFilter
- java.io.FileInputStream
- java.io.FileNotFoundException
- java.io.FileOutputStream
- java.io.FilePermission
- java.io.FileWriter
• java.io.FilenameFilter
• java.io.FilterInputStream
• java.io.FilterOutputStream
• java.io.FilterReader
• java.io.FilterWriter
• java.io.IOException
• java.io.InputStream
• java.io.InputStreamReader
• java.io.InterruptedIOException
• java.io.InvalidClassException
• java.io.InvalidObjectException
• java.io.LineNumberInputStream
• java.io.LineNumberReader
• java.io.NotActiveException
• java.io.NotSerializableException
• java.io.ObjectInput
• java.io.ObjectInputStream
• java.io.ObjectInputValidation
• java.io.ObjectOutput
• java.io.ObjectOutputStream
• java.io.ObjectStreamClass
• java.io.ObjectStreamConstants
• java.io.ObjectStreamException
• java.io.ObjectStreamField
• java.io.OptionalDataException
• java.io.OutputStream
• java.io.OutputStreamWriter
• java.io.PipedInputStream
• java.io.PipedOutputStream
• java.io.PipedReader
• java.io.PipedWriter
• java.io.PushbackInputStream
• java.io.PushbackReader
• java.io.Reader
• java.io.Serializable
• java.io.SerializablePermission
• java.io.StreamCorruptedException
• java.io.StreamTokenizer
• java.io.SyncFailedException

Package java.net

• java.net.Authenticator
• java.net.BindException
• java.net.ConnectException
• java.net.ContentHandler
• java.net.ContentHandlerFactory
• java.net.DatagramPacket
• java.net.DatagramSocket
• java.net.DatagramSocketImpl
• java.net.DatagramSocketImplFactory
• java.net.FileNameMap
• java.net.HttpURLConnection
• java.net.Inet4Address
• java.net.Inet6Address
• java.net.InetAddress
• java.net.InetSocketAddress
• java.net.JarURLConnection
• java.net.MalformedURLException
• java.net.MimeTypeMapper
• java.net.MulticastSocket
• java.net.NetPermission
• java.net.NetworkInterface
• java.net.NoRouteToHostException
• java.net.PasswordAuthentication
• java.net.PortUnreachableException
• java.net.ProtocolException
• java.net.ServerSocket
• java.net.Socket
• java.net.SocketAddress
• java.net.SocketException
• java.net.SocketImpl
• java.net.SocketImplFactory
• java.net.SocketInputStream
• java.net.SocketOptions
• java.net.SocketOutputStream
• java.net.SocketPermission
• java.net.SocketTimeoutException
• java.net.URI
• java.net.URISyntaxException
• java.net.URL
• java.net.URLClassLoader
• java.net.URLConnection
• java.net.URLDecoder
• java.net.URLEncoder
• java.net.URLStreamHandler
• java.net.URLStreamHandlerFactory
• java.net.UnknownHostException
• java.net.UnknownServiceException
Safety Critical Java Proposal

Package java.util

- java.util.AbstractCollection
- java.util.AbstractMap
- java.util.AbstractSequentialList
- java.util.AbstractSet
- java.util.ArrayList
- java.util.Arrays
- java.util.BitSet
- java.util.Collection
- java.util.Collections
- java.util.Comparator
- java.util.ConcurrentModificationException
- java.util.Dictionary
- java.util.EmptyStackException
- java.utilEnumeration
- java.util.EventListener
- java.util.EventObject
- java.util.HashMap
- java.util.HashSet
- java.util.Hashtable
- java.util.Iterator
- java.util.LinkedList
- java.util.List
- java.util.ListIterator
- java.util.Map
- java.util.MissingResourceException
- java.util.NoSuchElementException
- java.util.Observable
- java.util.Observer
- java.util.Properties
- java.util.Random
- java.util.ResourceBundle
- java.util.Set
- java.util.SortedMap
- java.util.SortedSet
- java.util.Stack
- java.util.StringTokenizer
- java.util.TimeZone
- java.util.TooManyListenersException
- java.util.Vector
- java.util.WeakHashMap
- java.util.jar.JarEntry
- java.util.jar.JarException
- java.util.jar.JarFile
• java.util.jar.JarInputStream
• java.util.jar.Manifest

Package javax.realtime

• javax.realtime.AEHThreadPool
• javax.realtime.AbsoluteTime
• javax.realtime.AperiodicParameters
• javax.realtime.ArrivalTimeQueueOverflowException
• javax.realtime.AsyncEvent
• javax.realtime.AsyncEventHandler
• javax.realtime.AsynchronouslyInterruptedException
• javax.realtime.BoundAsyncEventHandler
• javax.realtime.CeilingViolationException
• javax.realtime.Clock
• javax.realtime.DuplicateFilterException
• javax.realtime.FinalizeNode
• javax.realtime.GarbageCollector
• javax.realtime.HeapMemory
• javax.realtime.HighResolutionTime
• javax.realtime.IOPageTypeFilter
• javax.realtime.IllegalAssignmentError
• javax.realtime.ImmortalMemory
• javax.realtime.ImmortalPhysicalMemory
• javax.realtime.ImportanceParameters
• javax.realtime.InaccessibleAreaException
• javax.realtime.Interruptible
• javax.realtime.JamaicaGC
• javax.realtime.LTMemory
• javax.realtime.LTPhysicalMemory
• javax.realtime.List
• javax.realtime.MITViolationException
• javax.realtime.MarkAndSweepCollector
• javax.realtime.MemoryAccessError
• javax.realtime.MemoryArea
• javax.realtime.MemoryInUseException
• javax.realtime.MemoryParameters
• javax.realtime.MemoryScopeException
• javax.realtime.MemoryTypeConflictException
• javax.realtime.MonitorControl
• javax.realtime.NoHeapRealtimeThread
• javax.realtime.NotRemovableTypeFilter
• javax.realtime.OffsetOutOfBoundsException
• javax.realtime.OneShotTimer
• javax.realtime.POSIXSignalHandler
• javax.realtime.PeriodicParameters
• javax.realtime.PeriodicTimer
• javax.realtime.PhysicalMemoryManager
• javax.realtime.PhysicalMemoryTypeFilter
• javax.realtime.PriorityCeilingEmulation
• javax.realtime.PriorityInheritance
• javax.realtime.PriorityParameters
• javax.realtime.PriorityScheduler
• javax.realtime.ProcessingGroupParameters
• javax.realtime.RAMTypeFilter
• javax.realtime.RationalTime
• javax.realtime.RawMemoryAccess
• javax.realtime.RawMemoryFloatAccess
• javax.realtime.RealtimeClock
• javax.realtime.RealtimeSecurity
• javax.realtime.RealtimeSystem
• javax.realtime.RealtimeThread
• javax.realtime.RelativeTime
• javax.realtime.ReleaseParameters
• javax.realtime.ResourceLimitError
• javax.realtime.Schedulable
• javax.realtime.SchedulableObjectParameters
• javax.realtime.Schedulables
• javax.realtime.Scheduler
• javax.realtime.SchedulingParameters
• javax.realtime.ScopedCycleException
• javax.realtime.ScopedMemory
• javax.realtime.SizeEstimator
• javax.realtime.SizeOutOfBoundsException
• javax.realtime.SporadicParameters
• javax.realtime.ThrowBoundaryError
• javax.realtime.Timed
• javax.realtime.Timer
• javax.realtime.UnknownHappeningException
• javax.realtime.UnsupportedPhysicalMemoryException
• javax.realtime.VTMemory
• javax.realtime.VTPhysicalMemory
• javax.realtime.WaitFreeDequeue
• javax.realtime.WaitFreeReadQueue
• javax.realtime.WaitFreeWriteQueue
Appendix A

Optional Features

There are some facilities that are needed in some, but not all safety critical applications. It is useful to standardize them so that all environments that support the options provide them in a standard way. Three areas where this would be useful are providing nested scopes, support for networking, and support for controlling devices.

A.1 Pinnable Scopes

Though not necessarily needed for level A applications, there are certainly safety critical applications that are best implemented with a producer consumer pattern. When significant data reduction or transformation is required a scoped memory region would be helpful. In standard RTSJ, this can be accomplished with the wedge thread pattern, where an extra thread is used to hold a scope alive while it is being transferred from the producer to the consumer. Using a wedge thread is undesirable, particularly in a safety critical system. The following class provides better solution.

```java
package javax.safetycritical;
public class PinnableMemory extends SafetyMemory {
    public PinnableMemory(long size) {
        super(size);
    }
    public void Pin() {}
    public void Unpin() {}
}
```

The `Pin()` method is used to keep the given scope alive and the `Unpin()` is used to release it. Pinning and unpinning can only be done by a thread in the scope. In the implementation, pinning and unpinning have the same effect on the scope in terms of preserving its contents as creating a thread in the scope and destroying it respectively.
A.2 Networks

Networking support is necessary for many safety critical applications. A base level interface would include at least some subset of java.net where Socket and DatagramSocket are the most useful. Alternatively, the J2ME connection framework could be used. These classes would need to be adapted for use with realtime buses. All more complicated interfaces can be based there upon.

A.2.1 RTCorba

Corba provides a language independent interface for communicating between processes across a network. There is now a realtime variant with Java bindings from the OMG. A compliant VM which includes Corba should implement RTCorba according to the OMG specification.

A.2.2 Data Distribution Service

DDS is another possible data exchange protocol that can be built on top of a socket interface. Again, it is beyond the scope of SC Java to define bindings for DDS. SC Java should endorse bindings defined by the OMG.

A.3 Device Control

Safety critical applications traditionally use polling to exchange information with the environment. RawMemory along with a PeriodicThread is enough to provide this base functionality. For applications that use an event driven model, the BoundedAsynchronousEvent class supports am means of reacting to happenings.

A.3.1 Interrupt Handling

In RTSJ, the firing of an asynchronous event can be triggered by the occurrence of an interrupt. Consequently, asynchronous event handlers can only be used for “second-level” interrupt handling. While it is true that many safety critical systems do not allow interrupts (or only allow very specific system interrupts), the authors opine that greater flexibility can be obtained by providing support for application-level interrupt-handling; however, the support offered by the RTSJ needs enhancing to support first-level interrupt handlers. Furthermore, the RTSJ does not provide any means of binding “happenings” to physical devices. Ideas from “Real-Time Data Access” may be of use here.
A.3.2 Real-Time Data Access

The RTDA defines a means of providing platform independent, type safe access to device registers and mapping devices to interrupts. Some of the ideas here could be taken to make it easier to write platform independent device drivers. In particular, RTDA itself could be implemented in RTSJ provided that there were a standard way to map happenings to device interrupts.
Appendix B

An Example Safety Critical Application

import javax.safetycritical.Safelet;
import javax.safetycritical.PeriodicEventHandler;
import javax.safetycritical.SporadicEventHandler;
import javax.safetycritical.SporadicInterrupt;
import javax.realtime.LTMemory;
import javax.realtime.MemoryArea;
import javax.realtime.PeriodicParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;
import javax.realtime.SporadicParameters;

/**
 * test_critical is an example application for the HIJA
 * safety-critical Java profile.
 * If property "TASKS" is set, this property must be a string of
 * digits 1 through 5. All the tasks corresponding to these digits
 * will be started.
 * @author Fridtjof Siebert (siebert@aicas.com)
 */
public class SafetyCriticalExample extends Safelet
{
  /**
   * counter for how often control+C was hit.
   */
  int control_C_count_;
}
Safety Critical Java Proposal

/**
 * Maximum number of control+C hits before this example is terminated.
 */
static final int MAX_CONTROL_C = 5;

/**
 * missionMemorySize returns the desired size in bytes of a scoped
 * mission memory. A result of 0 indicates that immortal memory
 * shall be used as mission memory.<p>
 * A result greater than 0 returned by a redefinition of this method
 * requests mission memory to be scoped memory with the size that is
 * returned by this method.<p>
 * Note that using scoped memory as mission prohibits storing
 * references to any objects allocated during initialization phase
 * into static fields or other object that were allocated in
 * immortal memory (such as objects created by static initializers).
 * @return the desired size of a scoped memory to be used as mission
 * memory. 0 to use immortal memory as mission memory.
 */
public long missionMemorySize()
{
    return 200000;
}

/**
 * initialize sets up all the tasks in this application.
 */
public void initialize()
{
    System.out.println("Initializing: ");

    String tasks = System.getProperty("TASKS", "12345");

    if (tasks.contains("I"))
    {
        /* A periodic task that prints "beep" every second: */
        new PeriodicEventHandler
        (new PriorityParameters(20),
        new PeriodicParameters(null, // start
            new RelativeTime(1000,0), // period
            new RelativeTime( 50,0), // cost
            new RelativeTime(1000,0), // deadline
            null, // overrunHandler
            null), // missHandler
            100000)
        {  
            long start;

            public void handleEvent()
            {
                if (start == 0)
                {  
                    start = System.currentTimeMillis();
                }
                else
                {  
                    System.out.println("beep: " + (System.currentTimeMillis() - start));
                }
            }
        }
    }
}
public void cleanup()
{
    System.out.println("cleanup beep!");
}

if (tasks.contains("2"))
{
    /* another periodic task that prints "PENG!" every 400ms */
    new PeriodicEventHandler
    (new PriorityParameters(20),
     new PeriodicParameters(100000)
     {long start;

    public void handleEvent()
    {
        if (start == 0)
        {
            start = System.currentTimeMillis();
        }
        else
        {
            System.out.println(" PENG! " + (System.currentTimeMillis() - start));
        }
    }

    public void cleanup()
    {
        System.out.println("cleanup PENG!");
    }
    }
    }

if (tasks.indexOf("3") >= 0)
{
    /* An interrupt handler that reacts on control-C, terminates the
     * mission and starts a new mission or terminates the
     * application. */
    new SporadicInterrupt
    (new SporadicEventHandler
    (new PriorityParameters(25),
     new SporadicParameters(new RelativeTime(20,0), // interarrival
     new RelativeTime(50,0), // cost
     new RelativeTime(500,0), // deadline
     null, // overrunHandler
     null), // missHandler
     20000)
    {public void handleEvent()
    {
        control_C_count_++;
        System.out.println("**** INTERRUPT # " + control_C_count_ + " (terminating after " + MAX_CONTROL_C + " interrupts)");
        if (control_C_count_ < MAX_CONTROL_C)
            restart();
        else
            System.out.println("**** INTERRUPT # " + control_C_count_ + " (runnable)");
    }
    }
    }
terminate();
}

public void cleanup() {}

"SIGINT";

if (tasks.indexOf("4") >= 0)
{
    /* A sporadic event that is fired on window size change: */
    new SporadicInterrupt
        (new SporadicEventHandler
            (new PriorityParameters(30),
             new SporadicParameters(new RelativeTime(100,0), // interarrival
                                 new RelativeTime( 50,0), // cost
                                 new RelativeTime(500,0), // deadline
                                 null, // overrunHandler
                                 null), // missHandler
                                 10000)
        public void handleEvent()
        {
            System.out.println("WINDOW CHANGED");
        }
        public void cleanup() {}
        "SIGWINCH";
}

if (tasks.indexOf("5") >= 0)
{
    /* a periodic event that performs a lot of memory allocation
    * using a nested scope:
    */
    new PeriodicEventHandler
        (new PriorityParameters(20),
         new PeriodicParameters(new RelativeTime(2000,0), // start
                               new RelativeTime(4000,0), // period
                               new RelativeTime(500,0), // cost
                               new RelativeTime(4000,0), // deadline
                               null, // overrunHandler
                               null), // missHandler
                               10000)
        { /* the local memory area */
            MemoryArea local = new LTMemory(5000);

            /* length of string that creates output */
            int print_length = 0;

            /* length of string that should be created by this call to
            * runlocal.run()
            */
            int desired_length;

            /* memory intensive function creating a string of length
            * desired_length and printing this string to System.out if its
            * length is print_length.
            */
            Runnable runlocal = new Runnable()
            { public void run()
                {
                    String teststring = "=======";
                    while (teststring.length() < desired_length)
                        {
public void handleEvent()
{
    print_length = (print_length + 10) % 100;

    /* create 100 long strings: */
    for(desired_length=0; desired_length<100; desired_length++)
    {
        /* enter nested scope 100 times: */
        local.enter(runlocal);
    }
}

public void cleanup() {}
Bibliography


