Safety-Critical Java Level 2: Motivations, Example Applications and Issues

Andy Wellings
Department of Computer Science
University of York, York, UK
Andy.Wellings@york.ac.uk

Matt Luckcuck
Department of Computer Science
University of York, York, UK
ml881@york.ac.uk

Ana Cavalcanti
Department of Computer Science
University of York, York, UK
ana.cavalcanti@york.ac.uk

ABSTRACT

Safety Critical Java defines three compliance levels: Level 0, Level 1 and Level 2. Applications that can be scheduled using cyclic-executive techniques can be implemented at Level 0. Applications that can use simple analysable fixed-priority scheduling can be implemented at Level 1. However, Level 2 also targets fixed-priority scheduling, so this cannot be used to decide whether to use Level 1 or Level 2. The SCJ specification is clear on what *constitutes* a Level 2 application in terms of its use of the defined API, but not the *occasions* on which it should be used. Hence, it is not clear what application requirements dictate a Level 2 solution. This paper broadly classifies the features that exist only at Level 2 into three groups: support for nested mission sequencers, support for managed threads, including the use of the `Object.wait`, `Object.notify`, `HighResolutionTime. waitForObject` and `Services.delay` methods, and support for global scheduling across multiple processors. In this paper we explore the first two groups to derive possible programming requirements that each group of features support. We identify several areas where the specification needs modifications in order to support fully these derived requirements. These include support for terminating managed threads, the ability to set a deadline on the transition between missions, and augmentation of the mission sequencer concept to support composibility of timing constraints.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming; D.2.11 [**Software Architectures**]: Patterns; D.4.1 [**Process Management**]: Multiprocessing/multiprogramming/multitasking

1. INTRODUCTION

An international effort has produced a specification for a high-integrity real-time version of Java: Safety-Critical Java (SCJ) [13]. SCJ is based on a subset of Java augmented by the Real-Time Specification for Java (RTSJ) [23], which supplements Java's garbage-collected heap memory model with support for memory regions [22] called memory areas.

The programming model of SCJ is based on missions, event handlers and no-heap real-time threads (called managed threads). Each mission consists of a set of periodically (PEH) and aperiodically (APEH) released event handlers, and (potentially) no-heap real-time threads. A mission normally continues to execute until one of its handlers or threads (or a peer mission) requests termination, upon which a cleanup phase is performed and the next mission is prepared. Mission execution is controlled by an application-defined mission sequencer.

Additionally, SCJ restricts the RTSJ memory model to prohibit use of the heap, and defines a policy for the use of RTSJ's immortal and scoped memory areas. Each component of the programming model has an associated memory area, whose lifetime is that of the component. An immortal area holds objects throughout the lifetime of the program: they are never deallocated. A mission scoped area is cleared out at the end of each mission. Each release of a handler has an associated per-release scoped memory area, cleared out at the end of the release. Additionally, during a release, a stack of temporary private scoped memory areas can be used. For a thread, the execution of its associated `run` method is viewed as a single release, and consequently, is performed within its own local memory area.

The current version of SCJ defines three compliance levels (Level 0, 1 and 2), which reflects three supported programming (and execution) models[1].

- A Level 0 application's execution model is essentially a cyclic executive. In this model, only periodic handlers are supported and within a mission these are executed sequentially in a precise, clock-driven timeline[12]. A single mission sequencer is supported that allows the sequential execution of one or more missions.

- At SCJ Level 1, missions are again executed in sequential order by a single mission sequencer. The handlers now can be periodic or aperiodic, and are executed concurrently by a priority-based scheduler; any access to shared data has to be performed by `synchronized` methods to avoid race conditions. A

[1]The goal is that applications conforming to each compliance level should be capable of being certified to the highest level of integrity. However, it is accepted that the effort required to certify at compliance Level 2 may be significantly greater than at Levels 0 and 1.

JTRES '13 October 09-11, 2013, Karlsruhe, Germany

http://dx.doi.org/10.1145/2512989.2512991.

notable restriction of the Level 1 programming model is that use of `Object.wait()` and `Object.notify()` is prohibited. The rationale for this is that Level 1 applications should be analysable using standard schedulability analysis techniques for fixed-priority systems (such as response-time analysis or rate monotonic analysis)[4]. These techniques assume that all concurrent activities have well-defined release patterns, and that the activities (once released) do not self-suspend for any reason. Arbitrary use of `wait` or`notify` methods undermines these assumptions. Note that SCJ aperiodic handlers are assumed to be released sporadically with bounded minimum inter-arrival times (although this is not checked by the SCJ infrastructure).

- The Level 2 programming model supports a top-level single mission sequencer. However, each mission may create and execute additional mission sequencers concurrently with the initial mission sequencer. Computation in a Level 2 mission can be performed by periodic and aperiodic handlers, and no-heap managed real-time threads. Each child mission has its own mission memory, and may also create and execute other child missions. A Level 2 application may use the `Object.wait` and `Object.notify` methods.

It is clear that those applications that can be scheduled using cyclic-executive techniques should be implemented at Level 0. Furthermore, applications that can use simple analysable fixed priority scheduling should use Level 1. Hence, the required scheduling techniques are a primary indicator of whether or not Level 0 should be used. However, Level 2 also targets fixed priority scheduling, so this cannot be used to determine the use of Level 1 or Level 2. Furthermore, it is not clear what generic application-level requirements dictate a Level 2 solution. The SCJ specification is clear on what *constitutes* a Level 2 application in terms of its use of the defined API, but not the *occasions* on which it should be used.

To understand the purpose of Level 2, it is necessary to discover the generic application-level programming requirements for which Level 2 functionality is necessary. In the current version of the specification, this traceability is not provided in the rationale for the three levels.

The following features of SCJ are *only* available at Level 2:

- support for nested mission sequencers;
- support for managed threads;
- support for the `Object.wait`; and `Object.notify` methods;
- support for timeouts on waits and suspension-based delays; and
- support for global scheduling across multiple processors.

We broadly classify these features into three groups:

- support for nested mission sequencers;
- support for managed threads: including the use of the `Object.wait`, `Object.notify`, `HighResolutionTime.waitForObject` and `Services.delay` methods; and
- support for global scheduling across multiple processors.

We explore the first two groups to derive possible programming requirements that each group of features support. We identify several areas where the specification needs modifications in order to support fully these derived requirements. Support for global scheduling only at Level 2 reflects the fact that the state of the art in multiprocessor schedulability analysis is still advancing [7]. We will not address this issue in this paper.

2. NESTED MISSION SEQUENCERS

The ability to construct applications comprising of nested mission sequencers is, perhaps, the most important difference between Levels 0 or 1 and Level 2. In this section we identify two software architecture patterns that require the support of nested missions sequencers. We also sketch an example for each of the patterns. We call these two patterns: the *Multiple-Mode Application Pattern* and the *Independently Developed Subsystem Pattern*.

2.1 The Multiple-Mode Application Pattern

This pattern captures the typical architecture of systems that have to operate in multiple modes. Each mode consist of multiple periodic and sporadic concurrent activities with well-defined released frequencies and deadlines. In addition to these per-mode activities, there may also be concurrent activities that execute in all modes. Well-known schedulability analysis can be used to guarantee the timing properties in the steady state situations of executing in each mode. Analysis techniques also exist for handling the transitions between modes on a single processor [21, 16].

Architecture Components

The software architecture components that characterise this pattern are shown in Figure 1. It shows several types of objects. Tasks represent concurrent activities; a *sequencer* is a component which encapsulates several *subsystems* (modes) and objects that may be required to manage those subsystems. Only one subsystem is active at any one time. Subsystems can consist of tasks and other objects (not shown in the diagram). Consequently, Figure 1 illustrates an application that consists of several tasks that execute in all modes of the application. These tasks can request the Multiple Mode Sequencer to change its mode of operation.

In terms of SCJ, these components can be mapped down to mission and missions sequencers as illustrated in Figure 2. Each sequencer is mapped to a mission sequencer. Each subsystem within a sequencer is mapped to a mission. Control of the missions is subsumed into the sequencer. The tasks are mapped to SCJ schedulable objects. In this example, there are four modes of operation; each one in each mode is represented by a mission and the mission sequencer acts as the mode-change controller. Figure 2 also illustrates the use of two periodic event handlers that must be active in all modes, and of modes that have a single periodic handler.

Example Application

An example application that can use this pattern is an idealized Space Shuttle. It has several modes, each associated with a phase of its operation, as illustrated in Figure 3.

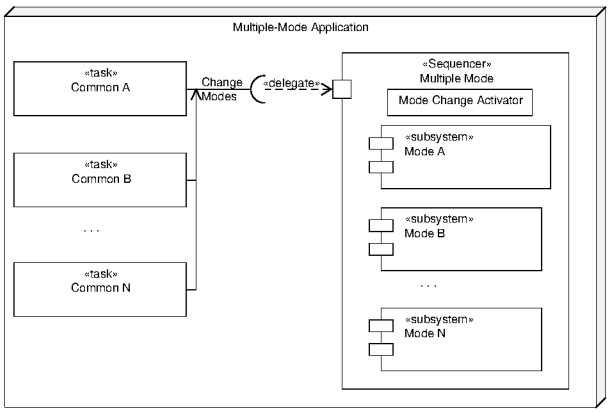

Figure 1: The Multiple Mode Operations Pattern

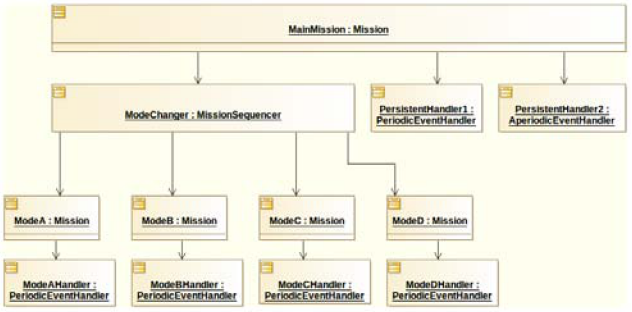

Figure 2: Multiple Mode Operations Pattern: SCJ Representation

Adequacy of SCJ Support

Using missions to support individual modes of operation and mission sequencers to support the mode change controllers has two main advantages. The first is that encapsulating each mode in a mission enhances the modularisation of the SCJ program and the traceability of its structure to its architectural model. This is important when each mode is a significant software component in its own right. The second is that SCJ supports a well-defined process for mission termination, where handlers are allowed to complete their current release before the mission completes. This is usually what is required when mode change requests are *planned* events[2].

There are, however, also some disadvantages and issues:

1. In order to execute a new mode it is necessary to create all the new objects (that are to reside in the mission memory) during the initialization phase of the mission (mode). Hence, for *unplanned* mode changes or applications that require fast and predictable planned changes, there may be some efficiency or latency concerns.

[2]Planned mode changes occur at well-defined points in a system's operation. In contrast, unplanned mode changes usually occur as a result of error conditions being detected. Such errors may be anticipated, but the time of their occurrence can not predicted. Hence the time at which a mode change is required cannot be predicted; they are unplanned.

2. There is no automatic single release time for all the schedulable objects. The schedulable objects in the initial mode start at a different time from the common schedulable objects. To create a single start time, it is necessary to use absolute-time offsets for all periodic handlers.

3. For timing analysis, the mission sequencer (mode changer) must be viewed as an aperiodic activity whose minimum inter-arrival time is equal to the minimum time between mode change requests. Its deadline represents any time constraints on the mode change operation. As an SCJ mission sequencer is a managed event handler, it only has a priority. *It does not have any release parameters.* These must be captured outside of the SCJ program and used in any schedulability analysis to program the appropriate priority. We discuss this concern further in Section 4.

4. It is not possible to do compositional time analysis of the application. The whole application must be analyzed in each mode along with each mode transition. We return to this issue later on in Section 2.2.

2.2 The Independently Developed Subsystem Pattern

Overview

Assembling systems that are comprised of independently developed subsystems is an important approach to developing systems that are more complex than those typically developed for Level 1. The ability to create nested mission sequencers at SCJ Level 2 seems to be key to supporting this compositional approach to constructing systems.

Architecture Components

The software architecture that characterises this pattern is shown in Figure 4. The subsystems are independently developed subsystems and are coordinated and controlled by another subsystem. The SCJ representation of this pattern is illustrated in Figure 5.

Example Application

A good example is the railway system described by Hunt and Nilsen[10] and illustrated in Figure 6. Here, there are multiple levels of nested mission sequencers. Each level essentially represents a subsystem that can be developed independently. We only show the nested level for the navigation services.

Adequacy of SCJ Support

Although missions appear to be an ideal construct for structuring subsystems, there are several limitations of the approach. The first is that in order to compose a system from many subsystems (missions) each of these subsystems (missions) must be controlled by its own mission sequencer. This is acceptable if each mission has multiple modes of operation, but can become cumbersome otherwise.

The second problem has already been mentioned in Section 2.1. When a system is composed of subsystems, there is no automatic common release time for all the schedulable objects. If required, this has to be programmed explicitly. However, for multiple nested sequencers, this can become

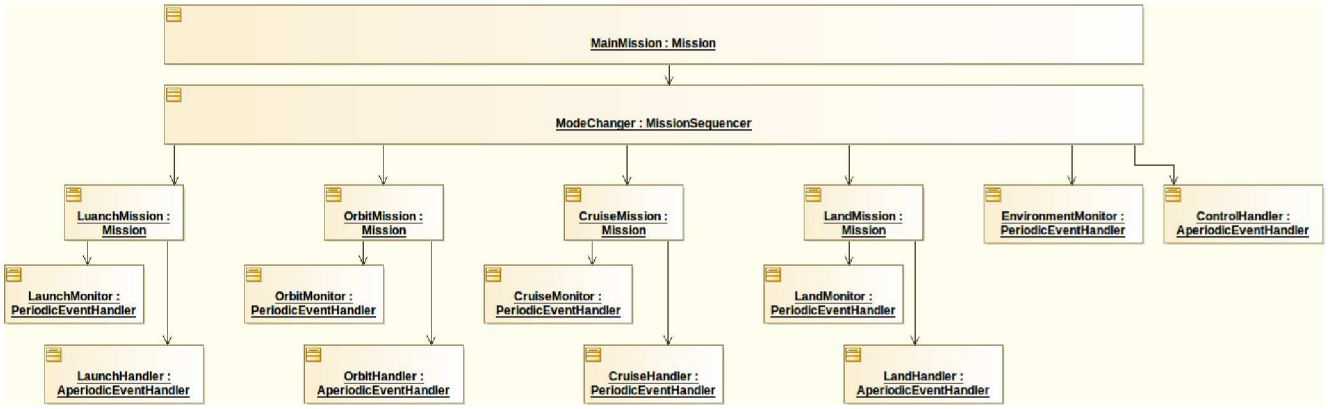

Figure 3: Space Shuttle with Moded Operations

cumbersome because the system start time would need to be passed down to all schedulable objects in the system.

Whilst the above limitations can be seen as minor, the third limitation is more significant. Neither SCJ nor the RTSJ directly support hierarchical scheduling. Hence it is difficult to achieve composability of timing constraints when subsystems are independently developed. The RTSJ does support the notion of processing groups, but these are too general and difficult to use in a multiprocessor environment [3, 24]. Hierarchical scheduling techniques for single processor and partitioned multiprocessor systems are well established [6] and techniques are beginning to emerge for globally scheduled multiprocessor systems [2, 7].

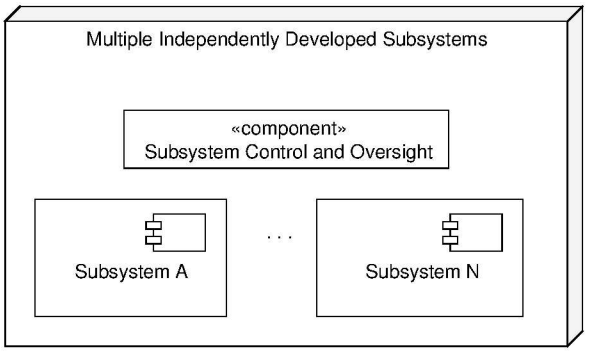

Figure 4: The Independently Developed Subsystem Pattern

The lack of such a facility in SCJ severely limits it use in supporting the timing analysis of applications composed according to the "independently developed subsystem" pattern. We return to this issue in Section 4.

3. MANAGED THREADS, WAIT/NOTIFY, TIMEOUTS AND DELAYS

There are several motivations for allowing managed threads in SCJ Level 2 applications. The first is to cater for schedulable objects that do not have a standard release profile.

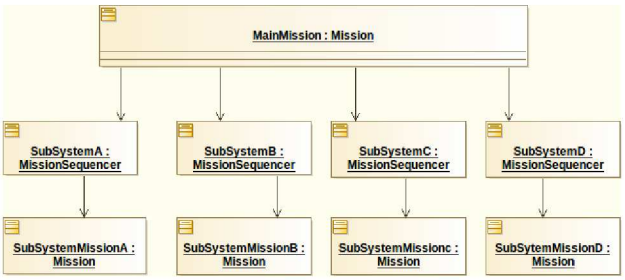

Figure 5: The Independently Developed Subsystem: SCJ Representation

The second is to allow suspension-based waiting for input and output operations to complete. The final motivation is to allow more encapsulation of state information.

3.1 Non-Standard Release Profiles

It is impossible to anticipate every possible scenario in which a schedulable object might need to be released. Here, we consider three common scenarios and discuss the extent to which they are supported at Level 1. We choose one of these to discuss in detail to illustrate how it can be implemented in Level 2 code.

A Periodic Activity Released by an Event

In SCJ, a periodic activity is either released immediately when it is started, or released after an absolute or relative delay from when it is started. There is no possibility of releasing a periodic activity by notification from another schedulable object (for example, by the firing of an event) or, indeed, an interrupt. Simply introducing the `Object.wait` and `Object.notify` methods into Level 2, and allowing the periodic activity to wait for a notification is not sufficient as deadline monitoring of a periodic event handler starts from when the handler is first released. Furthermore, in SCJ, deadlines cannot be dynamically changed, so it is not possible to set an initial deadline and then change it after the notification has occurred.

In the general case, the initial release event for a periodic activity may be: the start of the enclosed mission, a

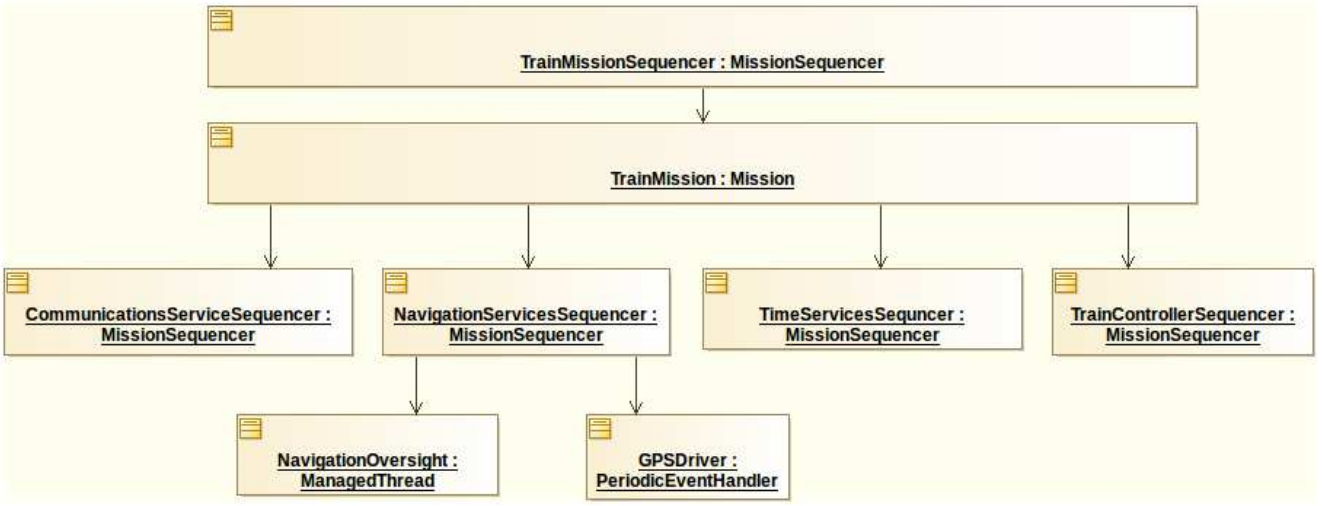

Figure 6: Railway Systems with Multiple Subsystems

timed offset from the mission start, a notification from another schedulable object (or an interrupt) or the absence of such notification. To cope with these latter two scenarios with `Object.wait` and `Object.notify` requires the application designer to equate such events with mode changes and to use the mode change pattern discussed in Section 2.1. This would not be appropriate when there are tight jitter constraints for the periodic activity starting.

The above discussion suggests that the Level 1 support for periodic event handlers is not flexible enough to cope with anything other than simple time-released periodic activities. The introduction of managed threads at Level 2 allows these more general release patterns to be catered for, as they allow the programmers to implement their own release mechanisms. It should be noted that managed threads are a very simplified version of the RTSJ's no-heap real-time thread, with the following restrictions: there is no automatic release mechanism (i.e., no support for `waitForNextPeriod`) and there is no mechanism to add a deadline. Furthermore, in SCJ the per-release memory area is created when the thread starts and cleared when the thread terminates. Consequently, if needed, developers have to program their own support for more sophisticated memory management.

Consider, for example, the periodic managed thread released by software notification in Figure 7, which shows an abstract extension of the `ManagedThread` class. The `firstRelease` method (lines 19-24) is called during the mission to indicate that the periodic activity should now start. The abstract `work` method declared on line 36 should be overridden to provide the functionality to be called each period. The `run` method (lines 39-49) is made final and waits for the initial release before calling the `work` method periodically. The example illustrates the added flexibility that is available at Level 2; it could not be programmed at Level 1.

The generalization of a periodic activity that is released by an event is the case where the periodic activity is released sporadically, that is, it is released by an event. It then executes periodically for a certain time (either determined by time itself or by another event), and then waits to be started again. A good example of this type of activity is a thruster control system (given by Wellings [23] page 235). Here, the astronaut initiates the thruster along with a duration of the engine "burn". The control of the engine itself requires a periodic activity to avoid the mechanical drift of valves.

For the same reasons as those described above for an event-released periodic activity, the only way this release pattern can be supported in SCJ is with managed threads using `Object.wait` and `Object.notify`.

Consumers in a Producer-Consumer System

Another common release pattern is where producer schedulable objects generate data that must be processed by consumer schedulable objects. Typically this data may come in bursts, and the consumer should process all the data as quickly as possible and block when there is no data available. These requirements cannot be met at Level 1, since it does not support a queue of outstanding release events for aperiodic event handlers. Level 2 allows this release pattern to be programmed as managed threads.

Background Activities: Run as Fast as You Can

There are occasions where background activities are required to run a fast as possible. There is no notion of release events for these activities (other than their initial start). These activities could be created using Level 1 functionality by having an aperiodic event handler that is released once, but this would be a misuse of this mechanism (which is essentially there to support sporadic releases). Although there is no negative consequence for this misuse, a managed thread is a better abstraction to support this requirement.

3.2 Suspension-based Waiting for IO where Busy-Waiting is Inappropriate

In many systems, a device driver will busy-wait for its associated device input (or output) to complete. This is because the expected delay is small and context switching away from the driver is considered inefficient. There are ways to

```
1  import javax.realtime.*;
2  import javax.safteycritical.*
3
4  public abstract class PeriodicThread extends ManagedThread {
5
6    public PeriodicThread(int period, int deadline,
7                          PriorityParameters priority,
8                          StorageParameters storage) {
9      super(priority, storage);
10     this.period = period;
11     this.deadline = deadline;
12     deadlineMissDetection = new DeadlineMissHandler();
13       // Where deadline misshandler is an extension of an SCJ OneShotEventHandler
14     nextRelease = new AbsoluteTime(); // created in mission memory
15     nextDeadline = new AbsoluteTime(); // created in mission memory
16     myMission = Mission.getCurrentMission();
17   }
18
19   public synchronized void firstRelease() {
20     notify();
21     nextRelease = Clock.getRealtimeClock().getTime(nextRelease);
22     nextDeadline.set(nextRelease.getMilliseconds() + deadline);
23     deadlineMissDetection.scheduleNextReleaseTime(nextDeadline);
24   }
25
26   private synchronized boolean waitFirstRelease() {
27     try { wait(); }
28     // or HighResultionTime.waitForObject(this, timeout)
29     // if a timeout is also required
30     catch(InterruptedException ie) { // mission is to be terminated
31       return false;
32     }
33     return true;
34   }
35
36   protected abstract void work();
37     // override this to provide the function of the thread
38
39   public final void run() {
40     if (waitFirstRelease()) {
41       while(!myMission.terminationPending()) {
42         nextRelease.add(period,0);
43         work();
44         nextDeadline.add(period,0);
45         deadlineMissDetection.scheduleNextReleaseTime(nextDeadline);
46         Services.delay(nextRelease); // waitForNextPeriod
47       }
48     }
49   }
50   private  AbsoluteTime nextRelease;
51   private  AbsoluteTime nextDeadline;
52   private final int period;
53   private final int deadline;
54   private DeadlineMissHandler deadlineMissDetection;
55   private Mission myMission;
56 }
```

Figure 7: A Periodic Schedulable Object Released by Software Notification

integrate this delay into the scheduling of the driver (see [4, section 14.6]), however allowing the driver to delay when it has no other activity to perform may also be appropriate. On the other hand, when this delay is a relatively significant amount of time, it is necessary to allow the system to schedule some alternative activities. Since it is not possible to have a suspension-based delay at Level 1, this requirement can only be implemented at Level 2.

3.3 Encapsulation of State Information

Another characteristic that differentiates managed threads from aperiodic or periodic event handlers is their use of memory. An event handler has its private memory areas cleared at the end of each release, which means that state that must persist across releases cannot be saved in the event handler's private memory. Outer memory areas must be used instead. A managed thread, however, only has its memory areas cleared when it exits its run method (i.e., it terminates). This means that data can be stored locally and preserved over successive application-implemented 'releases' of the thread. Of course, the effect of these two approaches is the same. The thread's memory area can last for as long as the memory area of its controlling mission, which is where more persistent data used by an event handler must be stored. However, this ability to encapsulate state is important from a software engineering perspective.

As an example, consider several logging schedulable objects that log their local state changes into local bounded buffers. When a buffer becomes full (which may take several releases of its associated schedulable object), the data is copied into a single global buffer in mission memory, which another schedulable object uses to write the system state changes to disk. If the logging schedulable objects are event handlers, the local buffers cannot be stored in their per-release memory areas, as such areas are cleared at the end of each of their releases. They would need to be stored in mission memory itself. This would make them more widely visible than is needed. Using managed threads, the local buffers can be stored in the per-release memory areas, as these will not be cleared until their associated managed threads terminate. Application-implemented releases, such as those programmed in Figure 7, can be augmented to use a nested private memory area for objects that can be cleared at the end of each application-level release. This is illustrated in Figure 8, which just shows the augmented run method (and an associated runnable) of lines 39-49 of Figure 7.

```
1  Runnable R = new Runnable () {
2    public void run() { work(); }
3  };
4
5  public final void run() {
6    if (waitFirstRelease()) {
7      while(!myMission.terminationPending()) {
8        nextRelease.add(period,100);
9        ManagedMemory.enterPrivateMemory(
10             privateMemorySize, R);
11       nextDeadline.add(period,100);
12       deadlineMissDetection.
13        scheduleNextReleaseTime(nextDeadline);
14       Services.delay(nextRelease);
15     }
16   }
17 }
```

Figure 8: Augmented Periodic Schedulable Object

4. SUMMARY OF ISSUES AND PROPOSALS

Sections 2 and 3 have explored some of the applications requirements where the use of Level 2 functionality seems more desirable. Here, we review the issues identified as potential causes for problems, and explore whether there are simple changes that can be made to the SCJ specification.

4.1 Managed Thread Termination

In SCJ, a managed thread terminates when it returns from its `run()` method. In Section 3 we have shown the implementation of a periodic thread that is first released by software notification. The code (see Figure 7) uses the `Object.wait` method (on line 27). Of course, if the enclosing mission has a terminate mission request outstanding, then the thread may never be released before all the other schedulable objects in the mission have terminated.

The SCJ defines the following activities to be performed on receipt of a mission termination request (taken from [13]):

1. disable all periodic event handlers associated with this mission so that they will experience no further firings;
2. disable all aperiodic event handlers so that no further firings will be honored;
3. clear the pending event (if any) for each event handler so that the event handler can be effectively shut down following completion of any event handling that is currently active;
4. wait for all of the managed schedulable objects associated with this mission to terminate their execution;
5. invoke the `ManagedSchedulable.cleanUp` methods for each of the managed schedulable objects associated with this mission, and invoke the `cleanUp` method associated with this mission.

Note that this list does not require calls to the `interrupt` methods of all the managed threads, which would cause all blocked managed schedulables to wake-up with an exception and hence expedite termination. This has to be programmed by application using the `terminationHook` method. This can be inconvenient when the mission has many schedulable objects.

4.2 Deadlines on Mission Sequencers

In Section 2.1, the use of a mission sequencer to implement mode change protocols has been discussed. It has been noted that an SCJ mission sequencer is a managed event handler and, as a consequence, it only has a priority. It does not have any release parameters. Systems that support multiple modes of operations often have deadlines associated with the mode changes. Hence, at Level 2 it would be appropriate to allow some form of deadline-miss handler to execute if the mode change does not occur promptly. Adding aperiodic release parameters to mission sequencers would seem to

undermine the mission programming model particularly for sequencers that support a single non-terminating mission. Instead, what we propose is to provide the following new methods in the `MissionSequencer` class:

```
/**
  * As for Mission.requestTermination
  *
  * In addition, the SCJ infrastructure will
  * set a timer that will fire if mission
  * termination (including any cleanup)
  * has not completed by the
  * deadline. On expiry of the timer, the
  * infrastructure will release the aperiodic
  * event handler passed as a parameter.
  *
  * The timer will be canceled if it has
  * not fired when the mission terminates.
  */
  @SCJAllowed(Level_1)
  public final void
    requestTerminationOfCurrentMission(
       AbsoluteTime deadline,
       AperiodicEventHandler deadlineMiss);

/**
  * As for Mission.requestTermination
  *
  * In addition, the SCJ infrastructure will
  * set a timer that will fire if next mission
  * has not started by the deadline.
  * On expiry of the timer, the
  * infrastructure will release the aperiodic
  * event handler passed as a parameter.
  *
  * The timer will be canceled if it has
  * not fired when the new mission starts.
  *
  * If there is no new mission, the timer is
  * canceled when the call to getNextMission
  * returns null.
  */
  @SCJAllowed(Level_1)
  public final void
    requestMissionChange(
       AbsoluteTime deadline,
       AperiodicEventHandler deadlineMiss);
```

Such a facility might also be a useful addition to Level 1. However, to call the above method from within the enclosed mission requires the current mission sequencer to be obtained. For programming convenience, a static method could be supplied in the mission sequencer class:

```
  @SCJAllowed(Level_1)
  static public MissionSequencer<MissionLevel>
                getCurrentSequencer();
```

This is in line with current facilities to obtain the current mission.

4.3 Support for Compositional Timing Analysis

Section 2.2 has identified the role of mission sequencers as a mechanism that can potentially support the composition of safety-critical systems from independently developed subsystems (or components). Hierarchical scheduling (and its associated schedulability analysis) is a well-established technique that facilitates composition when components have real-time attributes (such as deadlines). Unfortunately, hierarchical scheduling is neither supported by SCJ nor RTSJ. The main reason for this is possibly the lack of support by real-time operating system vendors.

Timing analysis for compositional real-time priority-based systems is usually achieved using a two-stage approach. Typically only two levels in the hierarchy are considered. At the top level, each subsystem is allocated an execution-time server, which is given a capacity, a priority and a replenishment period. Server parameters need to be carefully assigned to obtain good schedulability [5]. Analysis is then performed at the top level to ensure that each server can obtain its full capacity. The individual subsystems can then be analyzed in isolation to determine if they can meet their timing requirements given their allocated server's parameters. The schedulable objects within a subsystem are only scheduled for execution (and in priority order) when their logical server would be scheduled at the top level (and has available capacity).

Hence, there are two aspects of hierarchical scheduling that are needed to support composition of independently developed subsystems. The first aspect is multi-level priorities. In a two-level scheme, a mission (subsystem) is given a priority and the managed schedulable objects within that subsystem are also given a priority. Only when the mission's priority is the highest of all the "executable" missions' priorities are its associated managed schedulable objects executed (in priority order). In SCJ this can be achieved by:

- using the priority associated with the mission sequencer as the priority at which all its encapsulated missions execute; and
- the spreading out of the priorities of the mission sequencers across the full range of priorities. If two mission sequencers ($MS1$ and $MS2$) have priorities x and y respectively, where $y > x$, then all the schedulable objects in $MS1$ have priorities in the range $x..y-1$; thus all schedulable objects in $MS2$ have priorities greater than the schedulable objects in $MS1$.

The second aspect is that each mission has a CPU budget that is consumed whenever one of its schedulable objects is executing. It also has a period after which its budget is replenished. When a processing group's budget has been totally consumed, all its associated schedulable objects are suspended until the next replenishment occurs. In the RTSJ, this functionality can be supported by processing group if all the schedulable objects run on the same CPU.

One possibility is for SCJ to support the following subset of the RTSJ `ProcessingGroupParameters` class

```
package javax.safetycritical;

@SCJAllowed(LEVEL_2)
public class ProcessingGroupParameters {

  public ProcessingGroupParameters (
    HighResolutionTime start,
    RelativeTime period,
    RelativeTime budget)
 ...
}
```

The intention with the above class is to allow SCJ to implement a simple deferrable server [20] – a more elaborate scheme would allow other techniques such as a sporadic server [20] to be supported. The advantage of the latter is that it is supported by the POSIX standard.

New constructors can then be added to the `MissionSequencer` and the `Mission` classes:

```
1   -- in the MissionSequencer class
2   @SCJAllowed(LEVEL_2)
3   public MissionSequencer (PriorityParameters pri,
4         StorageParameters storage,
5         ProcessingGroupParameters params,
6         int priRange);
7
8   -- in the Mission class
9   @SCJAllowed(LEVEL_2)
10  public Mission (PriorityParameters pri,
11        ProcessingGroupParameters params,
12        int priRange);
```

In this constructor, the `pri` parameter indicates the lowest priority level that any encapsulated schedulable object can take, and `pri + priRange` the highest priority it can take. With this approach, the missions encapsulated within a mission sequencer need to execute on the same processor.

5. RELATED WORK

Other efforts have been made to provide safe language subsets for safety critical systems. MISRA C is a restricted subset of standard C that originated in the automotive industry in 1998[14]. In 2004 the standard was updated with a view to widening its scope as well as improving its structure, as opposed to adding many new rules. MISRA C has gained wide popularity in aircraft and medical systems as well as other critical software domains[9].

Several subsets of Ada have been developed since the language was first defined. The most widely used one is SPARK Ada, which highly restricts the amount of language features available to the programmer. The intent is to reduce the risk of failures resulting from errors in the program. This is balanced by ensuring that the language has the level of abstraction to provide the expressive power needed to hide the details of the implementation. SPARK also acknowledges the desire for safety-critical programs to be verifiable and restricts the language with this in mind[1]. SPARK has become one of the most popular choices for high-integrity real-time systems.

The Ravenscar[8] profile is another subset of Ada. It aims to aid program reliability — defined as predictable and consistent functioning. The control flow of a program is divided into two phases: initialisation and execution. All concurrent entities are allocated in the initialisation phase and they are started at the beginning of the execution phase. The concurrent entities in a Ravenscar program may only be periodic — released at regular intervals — or sporadic — released at irregular intervals but with minimum inter-arrival times — aperiodic entities are not supported. These concurrent entities are scheduled by a pre-emptive priority-based scheduler.

Drawing on the restrictions of the Ravenscar profile, a profile was created to aid the reliability of Java-based systems using the Real-Time Specification for Java (RTSJ); Ravenscar-Java[11]. The programs written in the Ravenscar-Java profile conform to the RTSJ standard with restrictions to ensure the program adheres to the Ravenscar rules. Other profiles have been proposed: for example Schoelberl et al. [19], who also considers the possibility of missions as application modes of operations [18].

As far as we are aware, there has been no previous work that has considered how to represent components in SCJ. There have been several approaches suggested for the RTSJ – see [15] as an example and for a review of related approaches. Most of these projects either focus on the functional aspects of component declaration and system composition, or they focus on the use of RTSJ memory areas.

There have been attempts to integrate the OSGi Java-based framework with the RTSJ, but again little attention has been given to the composition of timing constraints. The notable exception is the work by Richardson and Wellings [17], which considers real-time admission control of components within a Real-Time OSGi framework. They recognize the limitation of the RTSJ's processing groups. To achieve the same effect as hierarchical scheduling of execution-time servers they use a combination of processing groups, priority scaling and periodic timers. Essentially each server's priority is represented by a range of RTSJ priorities. A component allocated to a server must use this range when assigning priorities to its schedulable objects. The cost overrun handler that can be assigned to a processing group changes the priorities of its associated schedulable objects to a background priority. A separate periodic event handler is created whose release coincides with the replenish period. This resets the schedulable objects to their original priorities. Effectively, this approach can be used to implement the sporadic-server approach. It forms the basis of the approach that we have proposed in Section 4.3 for SCJ.

6. CONCLUSIONS AND FUTURE WORK

SCJ Level 2 has received little public scrutiny. Most papers that address SCJ consider either Level 0 or 1. Whilst it is clear from the SCJ specification what constitutes a Level 2 application (in terms of its use of the defined API), it is far from clear the occasions on which Level 2 should be used. This paper has explored some of the occasions in which applications cannot be easily implemented at Level 1 and that, therefore, require Level 2 support. In doing so, we have found no redundant features of Level 2. For each feature (only available at Level 2) we can find good examples that require use of that feature.

Unfortunately, our studies also reveals some deficiencies in the provided Level 2 features. These are:

1. lack of convenient support for terminating managed threads,
2. the inability to set a deadline on the transition between missions, and
3. need to enrich the mission sequencer concept to support composibility of timing constraints.

Feature 1 is not controversial and is probably just an omission in the current specification. For feature 2, it can be argued that this is not necessary for safety-critical systems as static analysis should have determined whether deadlines can be met. However, we note that the SCJ does support detection of deadline misses on managed schedulable objects at Levels 1 and 2. Feature 3 is, perhaps, very controversial

as it requires the monitoring of cpu-time usage. Although this is supported by the POSIX standard via sporadic process servers, we are not aware of any interpretation of the approach when the threads within the process can execute in parallel.

Our future work involves further evaluation of the extent to which a sporadic server can be implemented using the approach given in Section 4.3, and the development of libraries that support the various release patterns for managed threads identified in Section 3. In another line of future work, we will define a semantics for SCJ Level 2 programs following the lines of the approach adopted in [25] for Level 1 programs.

Finally, although there are some prototype implementations of SCJ Level 0 and 1, we are are unaware of any Level 2 implementations. Hence it has been difficult to fully evaluate some of the approaches presented in this paper. Once a Level 2 implementation is available we will undertake further testing of the algorithms presented.

7. ACKNOWLEDGEMENTS

This research reported in this paper is funded by the UK EPSRC under grant EP/H017461/1.

8. REFERENCES

[1] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] A. Burmyakov, E. Bini, and E. Tovar. The generalized multiprocessor periodic resource interface model for hierarchical multiprocessor scheduling. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, pages 131–139. ACM, 2012.

[3] A. Burns and A. Wellings. Processing group parameters in the Real-time Specification for Java. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, pages 360–370. Springer, 2003.

[4] A. Burns and A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Addison Wesley, 2009.

[5] R. Davis, A. Burns, et al. An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems. In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.

[6] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10–pp. IEEE, 2005.

[7] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011.

[8] B. Dobbing and A. Burns. The Ravenscar Ttasking Profile for High Integrity Real-Time Programs. *Ada Lett.*, XVIII(6):1–6, 1998.

[9] L. Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology*, 46(7):465 – 472, 2004.

[10] J. Hunt and K. Nilsen. Safety-Critical Java: The mission approach. In M. T. Higuera-Toledano and A. J. Wellings, editors, *Distributed, Embedded and Real-time Java Systems*, pages 199–233. Springer US, 2012.

[11] J Kwon. *Ravenscar-Java: Java Technology for High Integrity Real-Time Systems*. PhD thesis, The University of York, 2006.

[12] C. D. Locke. Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992.

[13] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. Safety critical java specification, version 0.95. Technical report, 6 December 2012.

[14] MIRA. Misra, 1998.

[15] A. Plsek, F. Loiret, and M. Malohlava. Component-oriented development for Real-Time Java. In M. T. Higuera-Toledano and A. J. Wellings, editors, *Distributed, Embedded and Real-time Java Systems*, pages 265–292. Springer US, 2012.

[16] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[17] T. Richardson and A. Wellings. RT-OSGi: Integrating the OSGi framework with the Real-Time Specification for Java. In M. T. Higuera-Toledano and A. J. Wellings, editors, *Distributed, Embedded and Real-time Java Systems*, pages 293–322. Springer US, 2012.

[18] M. Schoeberl. Mission modes for safety critical Java. In *Proceedings of the 5th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, SEUS'07, pages 105–113, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] M. Schoeberl, H. Søndergaard, B. Thomsen, and A. P. Ravn. A profile for safety critical Java. In *ISORC*, pages 94–101. IEEE Computer Society, 2007.

[20] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[21] K. W. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority preemptively scheduled systems. In *Real-Time Systems Symposium, 1992*, pages 100–109. IEEE, 1992.

[22] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[23] A. Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, 2004.

[24] A. Wellings and M. Kim. Processing group parameters in the Real-time Specification for Java. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 3–9. ACM, 2008.

[25] F. Zeyda, A. L. C. Cavalcanti, and A. Wellings. The Safety-Critical Java Mission Model: a formal account. In *International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, 2011.