
Article:
Luckcuck, Matt, Wellings, Andy orcid.org/0000-0002-3338-0623 and Cavalcanti, Ana orcid.org/0000-0002-0831-1976 (2016) Safety-Critical Java::level 2 in practice. Concurrency and Computation: Practice and Experience. pp. 1-27. ISSN: 1532-0634

https://doi.org/10.1002/cpe.3951

Safety-Critical Java: Level 2 in Practice

Matt Luckcuck, Andy Wellings and Ana Cavalcanti

16th September 2016

Abstract

Safety Critical Java (SCJ) is a profile of the Real-Time Specification for Java that brings to the safety-critical industry the possibility of using Java. SCJ defines three compliance levels: Level 0, Level 1 and 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. This paper broadly classifies the features that are only available at Level 2 into three groups: nested mission sequencers, managed threads, and global scheduling across multiple processors. We explore the first two groups to elicit programming requirements that they support. We identify several areas where the SCJ specification needs modifications to support these requirements fully; 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. We also propose simplifications to the termination protocol of missions and their mission sequencers. To illustrate the benefit of our changes, we present excerpts from a formal model of SCJ Level 2 written in *Circus*, a state-rich process algebra for refinement.

1 Introduction

An international effort has produced a specification for a high-integrity real-time version of Java: Safety-Critical Java (SCJ) [24]. SCJ is based on a subset of Java augmented by the Real-Time Specification for Java (RTSJ) [38], which supplements Java's garbage-collected heap memory model with support for memory regions [37] called memory areas.

The SCJ programming model is based on the notion of a mission. Each mission consists of a set of periodic (PEH), aperiodic (APEH), and one-shot (OSEH) event handlers, and no-heap managed real-time threads. The execution of a mission progresses through an initialisation, execution, and cleanup phase (see Figure 1). A mission's handlers and threads are created and registered during its initialisation phase. A mission continues to execute until one of its handlers, threads, or a peer mission requests termination, causing control to flow into a mission cleanup phase. An application-defined mission sequencer determines the sequence of missions to be executed.

SCJ restricts the RTSJ memory model to prohibit use of the heap, and defines a policy for the use of the RTSJ's immortal and scoped memory areas. Each managed thread and event handler of the programming model described above has an associated memory area, that holds temporary objects created during execution of that component. When a managed thread terminates, and each time an event handler finishes its handling of and event, the memory area is exited and all of the memory allocated for the component's temporary objects is reclaimed. An immortal memory area holds objects throughout the lifetime of the program: they are never deallocated. The scoped memory

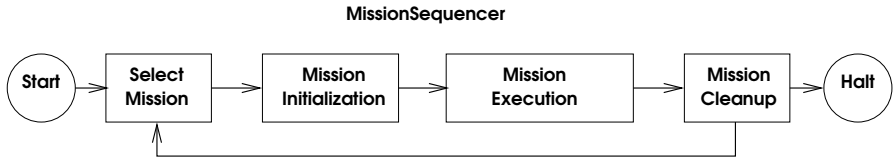

Figure 1: Safety Critical Mission Phases (taken from [24])

area of a mission 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. In the case of a thread, the execution of its associated `run()` method is viewed as a single release, and consequently, it is associated with its own local scoped memory area. Additionally, during a release, a stack of temporary private scoped memory areas can be used.

The SCJ language specification defines three compliance levels (Levels 0, 1 and 2), which reflect three supported programming and execution models. The compliance levels reflect increased levels of complexity in terms of the available programming features and, therefore, of the supported programs, with direct impact on the effort required for certification. It is accepted that the effort required to certify a program that exploits the generality of Level 2 capabilities may be significantly greater than that required to certify programs that use only the more limited capabilities of the lower compliance levels.

The differences between the three compliance levels are summarised in Table 1. The schedulable objects available at each level include those listed for that level and those listed for the previous levels. While mission sequencers are schedulable objects and are used at all compliance levels, they can only be registered to a mission at Level 2, as we explain below. The Suspension column refers to the availability of features like `Object.wait()`, `Object.notify()`, and `Services.delay()`.

	Execution Model	Schedulable Objects	Suspension	Platform
Level 0	Cyclic Executive	Periodic Event Handler	No	Single Processor
Level 1	Preemptive Priority Scheduling	Aperiodic and One-Shot Event Handlers	No	Multi-Processor
Level 2	Preemptive Priority Scheduling	Mission Sequencers and Managed Threads	Yes	Multi-Processor Global Scheduling

Table 1: Comparison of SCJ Compliance Level Features

A Level 0 application's execution model is essentially a cyclic executive. In this model, only periodic handlers are supported; they are executed sequentially in a precise, clock-driven time line [23]. A single mission sequencer controls the sequential execution of one or more missions.

At SCJ Level 1, missions are controlled by a single mission sequencer. The available schedulable objects are periodic, aperiodic, and one-shot event handlers. At Level 1, schedulable objects are executed concurrently by a preemptive priority-based scheduler; any access to shared data has to be performed by `synchronized` methods to avoid race conditions and to assure that the compiler generates code that forces that changes made

to shared variables by one thread to propagate to all other threads that share access to those same variables. A notable restriction of the Level 1 programming model is that use of `Object.wait()` and `Object.notify()` is prohibited. Arbitrary use of such methods complicates the ability to perform schedulability analysis.

At Level 2, missions are executed sequentially by a top-level mission sequencer, as with Level 1. In addition, each mission may register nested mission sequencers during its initialisation phase. Once these nested mission sequencers begin running, they each execute a sequence of child missions, independently of the top-level mission sequencer. Computation in a Level 2 mission can be performed by periodic, aperiodic, and one-shot handlers, and no-heap managed real-time threads. Each child mission has its own mission memory, distinct from its parent's mission memory. A Level 2 application may use Java suspension features.

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 decide between using Level 1 or Level 2.

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 is not provided in the rationale for the three compliance levels.

We broadly classify the additional functionality provided at Level 2 into three groups:

1. nested mission sequencers;
2. managed threads: including the use of the `Object.wait()`, `Object.notify()`, `HighResolutionTime.waitForObject()` and `Services.delay()` methods; and,
3. global scheduling across multiple processors.

We explore the first group in Section 2, showing how they provide support for two example applications: a Space Shuttle, which has several modes of operation, with mode-specific schedulable objects and persistent schedulable objects running throughout all modes; and a Train Control system, which has multiple independent subsystems, each implemented using a nested mission sequencer. Programming several modes of operation is possible at Level 1, but combining this behaviour with tasks running throughout all modes (without restarting that task in each mode) is only possible at Level 2. Moreover, programming multiple subsystems is not possible at Level 1, due to nested mission sequencers being unavailable.

In Section 3 we focus on the second group of features above, examining the benefits of the `ManagedThread` class and presenting three motivating scenarios that show where they are useful: non-standard release profiles, suspension-based waiting, and encapsulation of local state.

The availability of global scheduling only at Level 2 reflects the fact that the state of the art in multiprocessor schedulability analysis is still advancing [12]. Future safety-critical systems may be able to execute on multiprocessor platforms supported by new analysis techniques. We, however, do not address global scheduling in this paper.

We identify several areas where the SCJ specification needs modifications in order to fully support the programming requirements identified in Sections 2 and 3; these are summarised and expanded on in Section 4. The added functionality of Level 2

warrants a more formal description of the programming model and its required run-time support. In particular, the starting and termination of nested mission sequencers, and their associated missions, is much more complex than at Levels 0 and 1. In Section 5, we present a formal model of the termination protocol and show that significant simplification to this aspect of the specification can be achieved with a simple change to the API. Related work is given in Section 6, and we draw our conclusions in Section 7.

2 Nested Mission Sequencers

The ability to construct applications composed of nested mission sequencers is, perhaps, the most important aspect to be considered when choosing between Levels 0 or 1 and Level 2. In this section we identify two software architectural patterns that require the support of nested mission sequencers. We also sketch an example application 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

Overview

This pattern captures the typical architecture of systems that have to operate in multiple modes. Each mode consists of multiple persistent activities with well defined release frequencies and deadlines. In addition to these per-mode activities, there may also be persistent concurrent activities, which execute in all modes. Well known schedulability analysis techniques can be used to guarantee the timing properties in the steady-state situations of execution in each mode. Analysis techniques also exist for handling the transitions between modes, but only on a single processor [36, 30].

Architecture Components

The components that characterise this pattern are shown in Figure 2. Tasks represent concurrent activities. A mode changer encapsulates several modes, and each mode encapsulates several mode-specific tasks. Only one mode per mode changer is active at any one time. There may also be persistent tasks, which are required to operate during all modes. The mode changer and any persistent tasks are controlled by a coordinator. Mode changes are typically requested by tasks from the currently active mode.

In terms of SCJ, a mode changer can be conveniently implemented as a mission sequencer, and each mode as a mission. The tasks can be realised as SCJ schedulable objects. The coordinator component also has a natural correspondence with a mission, often the main mission, which registers the persistent tasks and the mode changer, and controls their operation.

Example Application

An example application that uses this pattern is an idealised Space Shuttle[1], as illustrated in Figure 3. It has three modes, each associated with a phase of its operation. Each mode has several schedulable objects that are only active during that mode – only two are shown for each mode in Figure 3. In addition to the mode-specific schedulables, there are two persistent schedulables shown, `EnvironmentMonitor` and

[1]The code for this example can be found at `http://www.cs.york.ac.uk/circus/hijac/case.html`

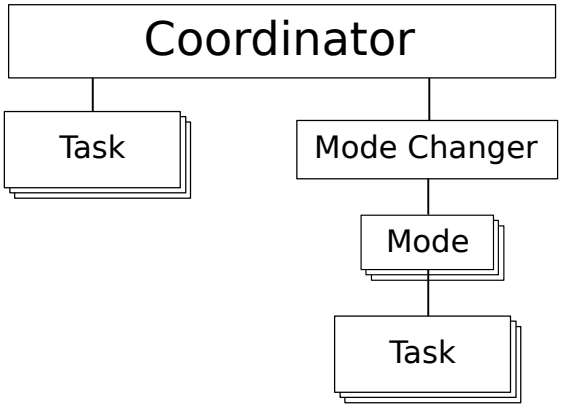

Figure 2: Multiple-Mode Operations Pattern

`ControlHandler`, which are active throughout all the modes.

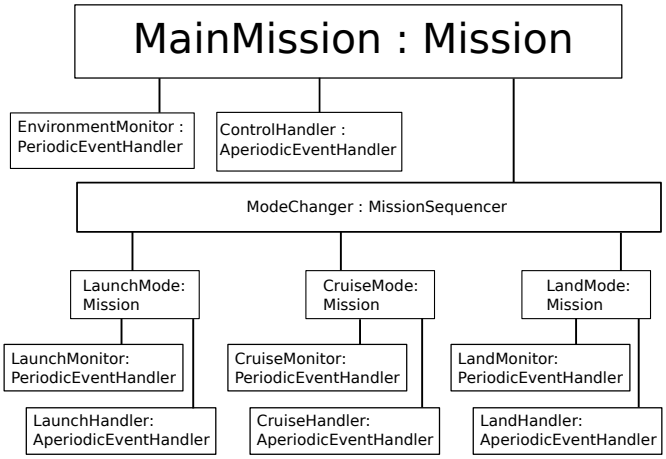

Figure 3: Space Shuttle with Moded Operations

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 advantage is that SCJ supports a well-defined (if somewhat complicated – see Section 5) process for mission termination, where schedulables can complete their current release before the mission completes. This is usually what is required when mode change requests are *planned* events. (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 be predicted. Hence the time at which a mode change is required cannot be predicted.)

On the other hand, in adopting the multiple-mode application pattern in the context of SCJ, there are issues of timing that need to be considered. First of all, 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.

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

For timing analysis, the mission sequencer, implementing the 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. We discuss this concern further in Section 4.

Finally, it is not easy to provide the runtime scheduling needed to support compositional time analysis of the application, as SCJ does not support hierarchical scheduling. SCJ schedules persistent schedulable objects in competition with mode-specific schedulable objects. Hence, the whole application must be analysed in each mode along with each mode transition. We discuss this issue in Sections 2.2 and 4.3 below.

2.2 The Independently Developed Subsystem Pattern

Overview

Assembling systems that are composed of independently developed subsystems, each encapsulating related behaviours, 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 is the key to supporting this approach to constructing systems.

Architecture Components

The software architecture that characterises this pattern is shown in Figure 4. The subsystems are all controlled by a coordinator. Subsystems may contain other subsystems. Typically, each subsystem is, or can be, independently developed and contains several tasks that perform related behaviours.

In terms of SCJ each subsystem can be decomposed into a mission sequencer and a single mission that manages the tasks within that subsystem. Each task can then be implemented by an appropriate managed schedulable: mission sequencer, thread, or handler. In this setting the coordinator component corresponds naturally to a mission, often the main mission, that registers the mission sequencers of each subsystem.

Example Application

A good example of this pattern is the railway system described by Hunt and Nilsen [20]:

> "Collision avoidance in rail systems is a representative safety-critical application. A common approach to the challenge of avoiding train system collisions divides all tracks into independently governed segments. A central rail traffic control system takes responsibility for authorizing particular trains to occupy particular rail segments at a given time. Each train is individually

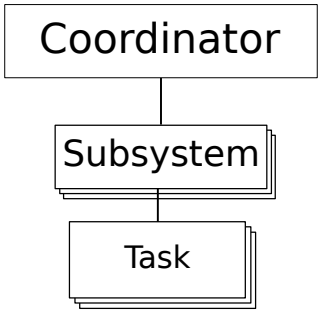

Figure 4: The Independently-Developed Subsystem

> responsible for honouring the train segment authorizations that are granted to it. Note that rail segment control addresses multiple competing concerns. On the one hand, there is a desire to optimize utilization of railway resources. This argues for high speeds and unencumbered access. On the other hand, there is a need to assure the safety of rail transport. This motivates lower speeds, larger safety buffers between travelling trains, and more conservative sharing of rail segments."

The example considers the structure of the on-board software (illustrated in Figure 5), which supports the following requirements:

- maintain reliable and secure communication with the central rail traffic control authority — the *CommunicationService* subsystem;
- monitor progress of the train along its assigned route — the *NavigationService* subsystem;
- control the train's speed in accordance with scheduled station stops, rail segment authorizations, local speed limit considerations, and fuel efficiency objectives — the *TrainController* subsystem;
- infrastructure support for maintaining global time — the *TimeServices* subsystem.

In the implementation described in [20], each of these subsystems is realised as a nested mission sequencer registered to the main mission (`TrainMission`), and each subsystem controls a single mission that registers the subsystem-specific managed schedulables. There are multiple tiers of nested mission sequencers within the subsystems. Each tier represents further subsystems that can be developed independently. For brevity, Figure 5 only shows two of the subsystem-specific managed schedulables and omits the deeper tiers of nested mission sequencers.

Adequacy of SCJ Support

Although the encapsulation provided by missions is ideal for structuring subsystems, as illustrated above there are issues that need to be addressed when adopting this 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 natural if each mission has multiple modes of operation, but can become cumbersome otherwise.

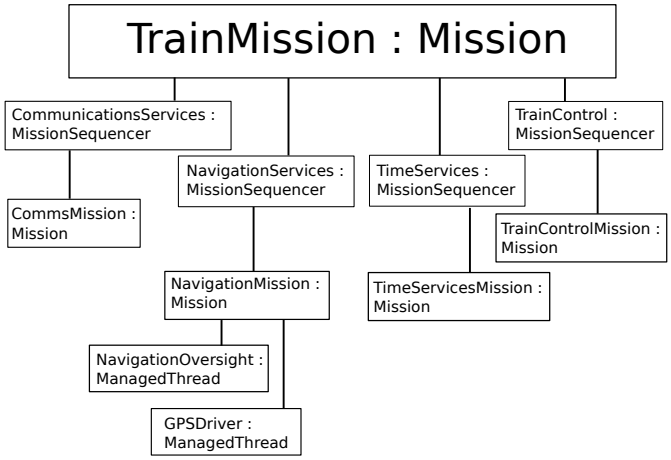

Figure 5: Railway System with Multiple Subsystems

The second issue 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. For multiple nested sequencers, this can become cumbersome because the system start time needs to be passed down to all schedulable objects in the system.

Whilst the above limitations can be seen as minor, the third limitation, which we discuss next, is more significant. Neither SCJ nor the RTSJ directly support hierarchical scheduling. Hence it is difficult to achieve decomposability of timing constraints when subsystems are independently developed. The RTSJ does support the notion of processing groups, which allow several schedulable objects to share a CPU budget, but these are too general and difficult to use in a multiprocessor environment [6, 39]. Hierarchical scheduling techniques for single processor and partitioned multiprocessor systems are well established [11] and techniques are beginning to emerge for globally scheduled multiprocessor systems [5, 12]. The lack of such a facility in SCJ severely limits its 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.

The final issue with the functionality of SCJ, in relation to this programming pattern, is that the API supports a request to terminate a mission sequencer. The intention of this facility is to allow a schedulable object within a mission to request not only its mission to be terminated, but also the whole sequence of missions (of which it is part) to be terminated. The concern with the facility is that it can be misused by a schedulable to terminate an arbitrary mission sequencer. This complicates the semantics of the termination protocol needed to support mission termination and breaks the encapsulation of the mission concept. We return to this issue in Sections 4.4 and 5.

3 Using Managed Threads

There are several motivations for supporting managed threads in SCJ Level 2 applications. The first is to serve the needs of schedulable objects that do not have a standard release profile. The second is to allow suspension-based waiting for input and output operation completion. The final motivation is to allow more encapsulation of state information. We consider each of these, in turn, in this section.

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 difficulties of implementing them at Level 1. We choose one of these to illustrate how implementation using Level 2 is possible.

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 via notification from another schedulable object or, indeed, an interrupt. However, it can be desirable for the initial release of a periodic activity to be triggered by a notification from another schedulable object, the absence of such a notification, or an interrupt. For example, the implementation of a task controlling a mechanical system that requires periodic updates but is started by an aperiodic button press can benefit from such a release pattern.

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. Simply introducing the `Object.wait()` and `Object.notify()` methods into Level 2, to allow a periodic handler to wait for a notification, is not sufficient. The problem is that deadline monitoring of event handlers starts from when the handler is first released. 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.

The introduction of managed threads at Level 2, on the other hand, allows these more general release patterns to be addressed, as managed threads allow programmers to implement their own release mechanisms. We consider, for example, the periodic managed thread released by software notification illustrated in Figure 6, which shows an abstract extension of the `ManagedThread` class. The `firstRelease` method (lines 18-23) is called during the mission to indicate that the periodic activity should now start. The abstract `work` method declared on line 35 must be overridden to provide the functionality to be called each period. The `run` method (lines 38-47) is final and waits for the initial release before calling the `work` method periodically. This example illustrates the added flexibility that is available at Level 2; the periodic thread in Figure 6 can not be programmed at Level 1.

Another example of a more complicated release pattern that can only be programmed in SCJ at Level 2 is the thruster control system described by Wellings [38, Page 235]. Here, an astronaut activates the thruster and supplies a duration for the engine "burn". The control of the engine itself requires a periodic activity to avoid the mechanical drift of valves. This requires an activity that is released by an event, executes periodically for a certain duration (determined either by time itself or by another event), and then waits to be started again. 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()`.

On the other hand, it should be noted that managed threads are a simplified version of the RTSJ's no-heap real-time thread, with the following restrictions: there is no automatic release mechanism (that is, 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.

```
1  public abstract class PeriodicThread extends ManagedThread {
2
3    private AbsoluteTime nextRelease; //the next release time of this thread
4    private AbsoluteTime nextDeadline; // the next deadline of this thread
5    private final int period;
6    private final int deadline;
7    private DeadlineMissHandler deadlineMissDetection;
8    private Mission myMission; //this thread's controlling mission
9    private boolean hadFirstRelease = false;
10
11   public synchronized void firstRelease() {
12     hadFirstRelease = true;
13     notify();
14     nextRelease = Clock.getRealtimeClock().getTime(nextRelease);
15     nextDeadline.set(nextRelease.getMilliseconds() + deadline);
16     deadlineMissDetection.scheduleNextReleaseTime(nextDeadline);
17   }
18
19   private synchronized boolean waitFirstRelease() {
20     while(!hadFirstRelease){
21        try { wait(); }
22        // or HighResolutionTime.waitForObject(this, timeout)
23            // if a timeout is also required
24        catch(InterruptedException ie) { // mission is to be terminated
25        return false;
26         }
27     }
28     return true;
29   }
30
31   protected abstract void work();
32     // override this to provide the function of the thread
33
34   public final void run() {
35     if (waitFirstRelease()) {
36       while(!myMission.terminationPending()) {
37         nextRelease.add(period,0);
38         work();
39         nextDeadline.add(period,0);
40         deadlineMissDetection.scheduleNextReleaseTime(nextDeadline);
41         Services.delay(nextRelease); // waitForNextPeriod
42       }
43     }
44   }
45 }
```

Figure 6: A Periodic Schedulable Object Released by Software Notification

Consumers in a Producer-Consumer System

Another common release pattern is where a producer schedulable object generates data that must be processed by a consumer schedulable object. 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 using managed threads.

Background Activities: Run as Fast as You Can

There are occasions where background activities are required to run as fast as possible. This is the case, for example, of a logging task that is required to process data from application logs whenever the scheduler allows it access to the processor. There is no notion of release events for these activities (other than their initial start). These activities can be programmed with Level 1 functionality using either an aperiodic event handler that is released only once, or a one-shot event handler with no start wait time. Both of these options, however, are a misuse of these mechanisms. 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 busy-waits 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 integrate this delay into the scheduling of the driver (see [7, Section 14.6]), and 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 event handlers is their use of memory. An event handler has its private memory area cleared at the end of each release, which means that state that must persist across releases must be saved in an outer memory area. A managed thread, however, only has its memory area cleared when it exits its `run()` method (that is, 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 persistent data used by an event handler is normally stored. However, this ability to encapsulate state is important from a software engineering perspective, since storing data that is private to a schedulable object in the mission memory of its controlling mission makes this data more widely visible than it should be.

As an example, we consider several 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

```
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(privateMemorySize, R);
10       nextDeadline.add(period,100);
11       deadlineMissDetection.scheduleNextReleaseTime(nextDeadline);
12       Services.delay(nextRelease);
13     }
14   }
15 }
```

Figure 7: Augmented Periodic Schedulable Object

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 need to be stored in the mission memory. Using managed threads, the local buffers can be stored in the per-release memory areas, as these are not cleared until their associated managed threads terminate. In addition, the local buffers do not become exposed to access by other schedulables.

Application-implemented releases, such as those programmed in Figure 6, can also be augmented to use a nested private memory area for objects that can be cleared at the end of each application-implemented release. This is illustrated in Figure 7, which just shows the augmented `run()` method (and an associated runnable) of lines 38-47 of Figure 6.

4 Revisiting the SCJ Level 2 Support

Sections 2 and 3 have explored some of the application requirements where the use of Level 2 functionality is desirable. Here, we review the issues identified as potential causes of problems, and explore changes that can be made to the SCJ specification to avoid these problems.

4.1 Managed Thread Termination

In SCJ, a managed thread terminates when it returns from its `run()` method. In Section 3 we illustrate how this simple release pattern can be adapted, using Level 2 features, to provide more complicated release patterns. Figure 6 shows the example of a periodic thread that is first released by software notification using the `Object.wait()` method (on line 18). With this approach, however, if our periodic thread is released and is waiting for its software notification when its controlling mission begins termination, then the thread may not finish its current release – its `run()` method may remain active. More generally, this applies to any schedulable object that is using the suspension features available at Level 2; if they are waiting when their controlling mission begins termination, then their release may not finish.

SCJ [24] defines the following activities to be performed on receipt of a mission termination request:

- invoke this mission's `terminationHook()` method;

- invoke `signalTermination()` on each managed schedulable object that is registered for execution within this mission;
- disable all periodic event handlers associated with this mission so that no further firings occur;
- disable all aperiodic event handlers, so that no further firings are honoured;
- clear the pending release 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;
- wait for all of the managed schedulable objects associated with this mission to terminate their execution;
- invoke the `ManagedSchedulable.cleanUp()` method for each of the managed schedulable objects associated with this mission; and,
- invoke the `cleanUp()` method associated with this mission.

We note that this list does not require invoking the `interrupt()` methods of all the managed schedulables, which would cause all blocked managed schedulables to wake up with an exception and hence expedite termination. This has to be programmed by applications using the `Mission.terminationHook()` method, and can be inconvenient when the mission has many schedulable objects.

To aid the termination of managed threads and schedulables that are suspended when a termination request is received, we propose that either the SCJ infrastructure interrupts all schedulable objects associated with the mission or that all the schedulable objects associated with the mission are informed of a pending termination request. The latter proposal can be achieved via a new method (`terminationSignalled()`), which each managed schedulable object must implement. The intention of this method is to allow the programmer to manually interrupt those schedulable objects that may be blocked when mission termination is signalled.

4.2 Deadlines on Mission Sequencers

As discussed in Section 2.1 an SCJ mission sequencer does not have any release parameters. Therefore, it cannot have an associated deadline or deadline-miss handler. Systems that support multiple modes of operations often have deadlines associated with the mode changes. Hence, at Level 2 it is 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 seems to undermine the mission programming model, particularly for sequencers that support a single non-terminating mission. Instead, what we propose is to add the methods shown in Figure 8 to the `MissionSequencer` class. These methods identify deadline-miss handlers for mission termination and start. We note that, since mission changes can also occur in Level 1, these facilities might also prove useful in that context.

4.3 Support for Compositional Timing Analysis

Section 2.2 identifies a role for mission sequencers as a mechanism that can support the composition of safety-critical systems from independently-developed subsystems (or components). We represent a subsystem in SCJ with a mission sequencer controlling

```
1  /**
2    * As for Mission.requestTermination
3    *
4    * In addition, the SCJ infrastructure will set a timer that will fire if mission
5    * termination (including any cleanup) has not completed by the
6    * deadline. On expiry of the timer, the infrastructure will release the aperiodic
7    * event handler passed as a parameter.
8    *
9    * The timer will be cancelled if it has not fired when the mission terminates.
10   */
11   @SCJAllowed(Level_1)
12   public final void requestTerminationOfCurrentMission(AbsoluteTime deadline,
13        AperiodicEventHandler deadlineMiss);
14
15 /**
16   * As for Mission.requestTermination
17   *
18   * In addition, the SCJ infrastructure will set a timer that will fire if next mission
19   * has not started by the deadline. On expiry of the timer, the
20   * infrastructure will release the aperiodic event handler passed as a parameter.
21   *
22   * The timer will be cancelled if it has not fired when the new mission starts.
23   *
24   * If there is no new mission, the timer is cancelled when the call to getNextMission
25   * returns null.
26   */
27   @SCJAllowed(Level_1)
28   public final void requestMissionChange(AbsoluteTime deadline,
29        AperiodicEventHandler deadlineMiss);
```

Figure 8: Proposed New Methods for the `MissionSequencer` Class

a single mission, which controls that subsystem's schedulable objects, as detailed in Section 2.2. Hence, we consider the mission sequencer as the top of the subsystem.

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 supported by neither SCJ nor the RTSJ. This is possibly because of the lack of support by real-time operating system vendors. We propose incorporating two elements of hierarchical scheduling into SCJ to improve its support for independently-developed subsystems and components: CPU budgets, to implement execution-time servers; and multi-level priorities, to isolate the scheduling of subsystems.

In the proposed approach, constructing a system made of subsystems can be achieved broadly in three steps. First, each subsystem is allocated an execution-time server, which is given a capacity, a priority order, and a replenishment period. These parameters need to be assigned carefully to obtain good schedulability [10]. Next, the priority ordering of the schedulable objects in each subsystem must be determined. Finally, an integration step assigns concrete priorities to the schedulable objects based on their priority ordering and the priority order of their server.

The schedulable objects within a subsystem are only scheduled for execution (in priority order) when their execution-time server server would be scheduled at the top level (and has available capacity). Once the parameters of the execution-time servers and the schedulables are set, the program needs to be analysed to determine schedulability at both the system and subsystem levels. We note that there is a relationship between the priorities of the execution-time server and of the subsystem's schedulable objects.

In the rest of this section we describe the integration of our proposal into SCJ. We consider only two tiers in the program hierarchy here, for brevity.

4.3.1 CPU Budgets

The first aspect of hierarchical scheduling we require is that each subsystem is allocated a budget, which is consumed whenever one of its schedulable objects is executing, and a period after which its budget is replenished. When a subsystem's budget has been totally consumed, all of its associated schedulable objects are suspended until its next replenishment occurs. In the RTSJ, this functionality can be supported by processing groups, if all the schedulable objects run on the same CPU.

Implementations of the RTSJ that support processing groups ensure that members of a group, collectively, are not be given more CPU time per period than their group's budget. When supported, the RTSJ implements the `ProcessingGroupParameters` class, which is associated with each schedulable object in the processing group. This allows the RTSJ's schedulable objects to share a budget while retaining their individual priorities, deadlines, and periods.

Because processing groups support the requirements for compositional timing analysis, one possible solution is for SCJ to implement the following restricted version of the RTSJ `ProcessingGroupParameters` class, where the deadline of the processing group is equal to its replenishment period.

```
1 package javax.safetycritical;
2
3 public class ProcessingGroupParameters {
4         public ProcessingGroupParameters (HighResolutionTime start,
5             RelativeTime replenishmentPeriod, RelativeTime budget){
6             ...
7         }
8         ...
9 }
```

However, this technique inherits the limitation that the missions encapsulated within a mission sequencer need to execute on the same processor.

4.3.2 Simulating Multi-Level Priorities

The second aspect of hierarchical scheduling that we require is multi-level priorities, which can be simulated in SCJ by manipulating the priorities of mission sequencers and schedulable objects. We propose:

- using the priority of each mission sequencer to define a priority range: from the priority of this mission sequencer to the priority of the next highest priority mission sequencer, and;
- transposing the priorities of all the schedulable objects in this subsystem into this range, while maintaining their original priority order, to ensure that they only run when their subsystem has the highest priority of all the subsystems.

This priority manipulation is performed statically, before the program is executed, in the integration step mentioned above. It may be the case that mission sequencer's priorities must be changed during integration to accommodate the schedulable objects. This is allowed as long as the priority order of the mission sequencers is maintained.

For example, we consider below a simple two-subsystem application using rate-monotonic scheduling. The parameters of the execution-time servers of each subsystem are shown in Table 9.

	Period (ms)	Budget (ms)
Server 1	100	40
Server 2	50	15

Figure 9: Execution-Time Server Parameters

At the top level, the execution-time server the subsystems associated with *Server*1 has a replenishment period of 100 milliseconds and a budget of 40 milliseconds. The execution-time server of the subsystem associated with *Server*2 has a replenishment period of 50 milliseconds and a budget of 15 milliseconds. The top-level is schedulable when the priority of *Server* 2 is greater than the priority of *Server* 1.

Now, we suppose that the subsystem associated with *Server* 1 contains three schedulable objects, $S1$, $S2$, and $S3$, which require a priority ordering where $S3$ has a higher priority than $S2$, which has a higher priority than $S1$. During system integration, the priorities of the servers and schedulables could be assigned so that the priority of *Server* 2 is greater than that of *Server*1 plus 3 in order to allow the priorities of the schedulable objects to be assigned between those of the two servers. An example of the priorities that can be assigned is shown in Table 10.

	Schedulable	Priority
Server 1		10
	S1	10
	S2	11
	S3	12
Server 2		20

Figure 10: Execution-Time Server and Schedulable Priorities

This concrete priority assignment simulates multi-level priorities, because the schedulables of the subsystem associated with *Server* 1 are not able to run if *Server* 2 is executing.

4.3.3 Incorporation into SCJ

As detailed above, to support CPU budgets, SCJ needs to implement processing groups, and SCJ can already support multi-level priorities, by manipulating the priorities of an application's schedulable objects and mission sequencers. To aid integration of these two aspects of hierarchical scheduling into SCJ applications, a new subclass of mission sequencer can be added to encapsulate the concerns of a subsystem, as shown in the example below.

```
1 public class Subsystem extends MissionSequencer{
2     public Subsystem (PriorityParameters pri, StorageParameters storage,
3                 ProcessingGroupParameters params, int priRange){
4             ...
5         }
6         ...
7 }
```

The constructor above takes a `ProcessingGroupParameters` object, as described in Section 4.3.1. To encapsulate the information needed for the priority manipulation, described in Section 4.3.2, the values of `pri` (which is the priority of this mission sequencer)

and of `pri + priRange` define the prioirity range for schedulable objects encapsulated by this subsystem.

4.4 Mission Sequencer Termination

In Section 2.2, we argue that allowing arbitrary schedulables to request the termination of an arbitrary mission sequencer violates the encapsulation supported by missions. We propose that schedulable objects are only allowed to request that their controlling mission is terminated. The mission itself then has the responsibility of deciding whether to request its sequencer to terminate. This gives a more structured approach to termination.

We propose removing the `requestSequencerTermination()` method, which allows a request to terminate a mission sequencer, to enforce this more structured termination policy. Instead, we recommend that the mission cleanup phase indicates whether its sequencer should continue with the next mission or terminate. For that, we propose that `Mission.cleanUp()` return a boolean value, which is passed to the mission sequencer to determine if the mission sequencer should continue or terminate.

We investigate the impact of this change in the next section. As the termination protocol is one of the more complex features of SCJ Level 2 programs, we consider formal models of the SCJ termination protocol for both the current specification and for the protocol that we propose here.

5 Formalisation of Level 2

In this section we present: a formal model of the current termination protocol as presented in the SCJ draft specification [24], in Section 5.2; a formal model of the termination protocol incorporating our proposed changes, in Section 5.3; and a comparison of these two models, in Section 5.4. Our models are written in the state-rich process algebra *Circus*, for which a model of SCJ Level 1 already exists [40]. In Section 5.1 we give a brief overview of the *Circus* notation.

With these models we show that the current termination protocol is more complicated than necessary. Indeed, it was the process of formally modelling SCJ Level 2 that first illuminated the complexities of the mission sequencer termination protocol. These complications only become apparent at Level 2 because of its capacity to nest mission sequencers arbitrarily deeply, which means that mission sequencers can be terminated by schedulables both above and below themselves in the program's hierarchy at any point during the execution phase.

5.1 *Circus* Introduction

Circus [8] combines elements from CSP [19], Z [34], and a refinement calculus [25] to allow modelling of both state and patterns of interaction. Figure 11 sketches the BNF description of the syntax of *Circus*. Below, we describe the elements of the syntax pertinent to the discussion of our formal model. A comprehensive account of *Circus* can be found in [27].

Circus programs, defined in Figure 11 by the syntactic category Program, are formed by a sequence of *Circus* paragraphs. Each *Circus* paragraph, defined in Figure 11 as elements of CircusPar, may be either a Z paragraph (the Par category), a channel declaration, a channel set declaration, or a process declaration. The syntactic category N contains the valid Z (and *Circus*) identifiers.

Circus programs use bi-directional channels to allow their processes to communicate; we discuss the different types of communications later in the section. All of the channels used in a *Circus* program must be declared. Channel declarations are defined by the CDecl syntactic category in Figure 11. Here, Exp is the category of Z expressions. If a channel takes any parameters, their types must be declared. For convenience, channels my be collected into a channel set – defined by the CSExp category. Channel sets allow easy specification of the channels used to interact with a process.

Each *Circus* process has a name and a body (**process** N $\widehat{=}$ ProcDef) and may take parameters (Decl • ProcDef). In our model, this is used where, for example, the process modelling a mission or a mission sequencer takes a parameter representing its unique identifier. Hence, for example, **process** *MissionFW* $\widehat{=}$ *mission* : *MissionID* • ... declares the mission process with a parameter *mission* of type *MissionID*.

The body of a Circus process (**begin** PPar* **state** SchemaExp PPar* • Action **end**) is delimited by the **begin** and **end** keywords; it may contain a state, which is modelled using a Z schema; and some actions, modelled using a free combination of Z state operations, constructs of a simple imperative language, and CSP constructs (PPar*). While Z schemas can be used to define data operations over the state of a *Circus* process using predicates, assignments to variables can also be made directly ($\mathsf{N}^+ := \mathsf{Exp}^+$, from the Command category in Figure 11).

A *Circus* process always has a main action at the end of the process after a •, that dictates the combination of Z schemas and CSP actions that define the behaviour of the process; these actions may reference other local actions for the purpose of structure. Both the state and actions of a *Circus* process are local to that process. This makes *Circus* processes similar to classes in object-oriented programming, where a class has some local variables and methods.

CSP has many operators that are adopted in *Circus*, which all belong to the syntactic category CSPAction in Figure 11. Table 2 provides a description of the operators in this category that we use in our model, some of which are omitted in Figure 11. We describe these in more detail below to support the following discussion of our model.

Action	Syntax	Description
Skip	**Skip**	A simple operator that terminates
Simple Prefix	$c \longrightarrow A$	Simple synchronisation with no data
Prefix	$c.x \longrightarrow A$	Synchronisation with some data x
Input Prefix	$c?x \longrightarrow A$	Synchronisation with a value bound to x
Output Prefix	$c!x \longrightarrow A$	Synchronisation outputting the value of x
External Choice	$A \Box B$	Offers a choice between two actions A and B
Sequence	$A \,;\, B$	Executes A then B in sequence
Parallelism	$A \llbracket ns_a \mid cs \mid ns_b \rrbracket B$	Parallelism, synchronising on the channels in c
Interleaving	$A \llbracket ns_a \mid ns_b \rrbracket B$	Parallelism with no synchronisation between
Iterated Interleaving	$\Vert\Vert\Vert\, x : S \bullet A(x)$	Interleaving of all actions $A(x)$ where $x \in S$

Table 2: Syntax of *Circus* operators derived from CSP

A simple operator is **Skip**, which terminates and does nothing else. A prefix $c \longrightarrow A$ waits for a communication on the channel c and then proceeds to behave like the action A. If a channel has a parameter then this must be provided. The parameter can be included as either an input ($c?x \longrightarrow A$), an output ($c!x \longrightarrow A$), or added to the channel name to indicate a specific communication on that channel ($c.x \longrightarrow A$). This latter form is often used in our models to restrict an action to synchronise on a channel only if

Program	::=	CircusPar*
CircusPar	::=	Par $\mid$ **channel** CDecl $\mid$ **channelset** N $\widehat{=}$ CSExp $\mid$ ProcDecl
CDecl	::=	SimpleCDecl $\mid$ SimpleCDecl; CDecl
SimpleCDecl	::=	N$^+$ $\mid$ N$^+$: Exp $\mid$ [N$^+$]N$^+$: Exp $\mid$ ...
CSExp	::=	$\{\!\mid$ $\mid\!\}$ $\mid$ $\{\!\mid$ N$^+$ $\mid\!\}$ $\mid$ N $\mid$ CSExp $\cup$ CSExp $\mid$ CSExp $\cap$ CSExp
	$\mid$	CSExp $\setminus$ CSExp
ProcDecl	::=	**process** N $\widehat{=}$ ProcDef $\mid$ ...
ProcDef	::=	Decl $\bullet$ ProcDef $\mid$ Proc ...
Proc	::=	**begin** PPar* **state** SchemaExp PPar* $\bullet$ Action **end**
...		
NSExp	::=	$\{\}$ $\mid$ $\{$N$^+\}$ $\mid$ N $\mid$ NSExp $\cup$ NSExp $\mid$ NSExp $\cap$ NSExp
	$\mid$	NSExp $\setminus$ NSExp
PPar	::=	Par $\mid$ N $\widehat{=}$ ParAction $\mid$ **nameset** N $\widehat{=}$ NSExp
ParAction	::=	Action $\mid$ Decl $\bullet$ ParAction
Action	::=	$\mid$ Command $\mid$ N $\mid$ CSPAction ...
CSPAction	::=	$Stop$ $\mid$ $Chaos$ $\mid$ Pred $\&$ Action $\mid$ Action $\sqcap$ Action
	$\mid$	Action $\setminus$ CSExp$\mid$; Decl $\bullet$ Action ...
Comm	::=	N CParameter* ...
CParameter	::=	?N $\mid$?N : Pred $\mid$!Exp $\mid$ .Exp
Command	::=	N$^+$:= Exp$^+$ $\mid$ **if** GActions **fi** $\mid$ **var** Decl $\bullet$ Action
	$\mid$	**val** Decl $\bullet$ Action ...
GActions	::=	Pred $\longrightarrow$ Action $\mid$ Pred $\longrightarrow$ Action $\Box$ GActions

Figure 11: Partial BNF Syntax of *Circus*

it is parametrised by the identifier of the *Circus* process to which it belongs. A related operator is sequential composition ; , which connects any two processes, instead of just a channel communication and a process like the prefix operator $\longrightarrow$. Hence $A ; B$ executes the action A until it terminates and then continues on to execute B.

The external choice operator $\Box$ allows an action to offer its environment the choice of two different channel communications. Hence $c_1 \longrightarrow A \Box c_2 \longrightarrow B$ proceeds to A if there is a communication on c_1 or B if there is a communication on c_2. *Circus* also contains a simple conditional statement as shown in the definition of the syntactic category Command in Figure 11. It takes a familiar if... then... else form. Hence **if** $(x = TRUE) \longrightarrow A \; [] \; (x = FALSE) \longrightarrow B$ **fi** performs the action A if $x = TRUE$ and the action B if $x = FALSE$.

Two actions A and B may be placed in parallel: $A \; [\![\; ns_a \mid cs \mid ns_b \;]\!] \; B$, specifies a synchronisation set of channels cs over which they both have to agree to communicate, and name sets describing the variables that each side of the parallelism may alter that must be disjoint to avoid write conflicts. Hence, in the execution of $A [\![\varnothing \mid \{\!| \, a, b \, |\!\} \mid \varnothing]\!] B$, A and B perform their actions in parallel with each other, but they must both agree to communicate on the channels a and b at the same time; further, the use of the empty set ($\varnothing$) indicates that neither A nor B can alter any variables.

5.1.1 SCJ in *Circus*

A formal model of SCJ Level 1 has been produced [40] to allow the translation of arbitrary SCJ programs into *Circus* in order to facilitate analysis. The *Circus* models are composed of a model of the infrastructure classes of SCJ – the 'Framework Model' – which remains the same and is reused for each translation, and a model of the code provided by the application – the 'Application Model' – which changes for each new program translated. The Framework Model encapsulates the unchanging aspects of *any* SCJ program, whereas the Application Model is generated afresh each time to model a *specific* SCJ Level 1 program.

A *Circus* model of SCJ Level 2 is in development, based on the approach taken by the Level 1 model; therefore, it has a Framework and an Application component.

The Level 2 model has *Circus* processes for each of the main infrastructure classes in SCJ, and each object in the program is represented by its own instance of the relevant *Circus* process. Generally speaking, each *Circus* process retains the name of the SCJ class it models, suffixed with "FW" for framework processes and "App" for application processes.

The methods of each of the objects that we model are represented by *Circus* actions. Ordinarily, a call to a method is modelled by two channels: a channel modelling the call to the method, suffixed by *Call*; and a channel modelling the return from the method, suffixed by *Ret*. For example, mission sequencers need to know that their current mission's `initialize()` method has completed before continuing. The action modelling the `initialize()` method, therefore, starts with a synchronisation on *initializeCall* and terminates with a synchronisation on *initializeRet*. However if, in our model, the caller of the method does not require that the method returns before it continues, then a call to that method is modelled by a single channel.

Figure 12 shows the processes that model the SCJ infrastructure interacting within the Framework Model and some of the communications between the Framework and Application models. For brevity, omitted communication channels are represented by

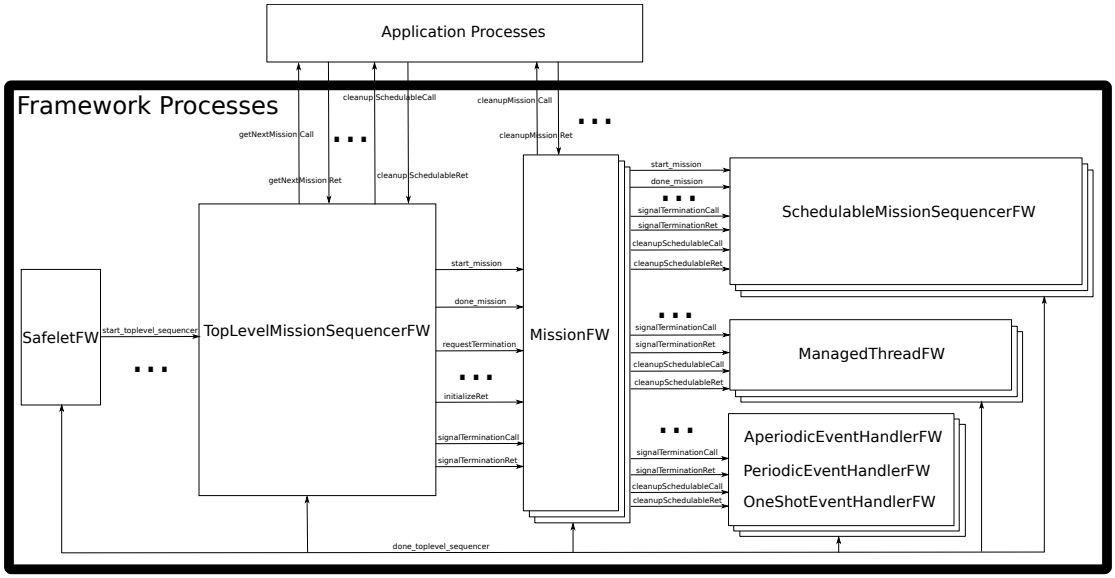

Figure 12: Level 2 Model Structure

ellipses, the three types of event handlers are represented by the single component at the bottom right of the figure, and all application processes are represented by the single component shown at the top of the figure. Because a mission sequencer can be used in two contexts at Level 2 – both as a mission sequencer at the top of the program hierarchy and as a schedulable object nested inside a mission – this class is modelled by two processes: one for the top-level mission sequencer, *TopLevelMissionSequencerFW*; and one for a schedulable mission sequencer, *SchedulableMissionSequencerFW*. This simplifies both processes because they each only have to be involved in communications relevant to their context. The *MissionFW* and the processes representing the schedulable objects may have multiple instances in one model. Each of the SCJ methods that we model is represented by a CSP action in the relevant *Circus* process.

5.2 Model of the Current Termination Protocol

The current termination protocol requires very complex models of the mission sequencer process. This holds for both the *TopLevelMissionSequencerFW* and the *SchedulableMissionSequencerFW*. Complexity arises because mission sequencers may be terminated at arbitrary times during their execution. In this section we describe our model of the protocol and explain the source of its complexity.

5.2.1 Top Level Mission Sequencer

The *TopLevelMissionSequencerFW* process has one parameter, *sequencer*, which is the identifier of this mission sequencer process, and two state components: *currentMission*, which holds the identifier of the mission this sequencer is currently executing; and *terminating*, which is a boolean value that records if this mission sequencer has been asked to terminate. The *GetNextMission* action models the `getNextMission()` method

and is shown below.

$$
\begin{array}{l}
GetNextMission \mathrel{\widehat=} \\
\quad getNextMissionCall\,.\,sequencer \longrightarrow \\
\quad getNextMissionRet\,.\,sequencer\,?\,next \longrightarrow \\
\quad currentMission := next; \\
\quad StartMission; \\
\quad \textbf{if}\ terminating = FALSE \longrightarrow \\
\qquad GetNextMission \\
\quad [] \ terminating = TRUE \longrightarrow \\
\qquad \textbf{Skip}
\end{array}
$$

It communicates with the application model using the channels *getNextMissionCall* and *getNextMissionRet* to get the identifier of the next mission that this mission sequencer should execute. This identifier is stored in the variable *currentMission*. Then the *StartMission* action is called; it uses the *start_mission* channel to start the current mission. This action is defined as follows.

$$
\begin{array}{l}
StartMission \mathrel{\widehat=} \\
\quad \textbf{if}\ currentMission \neq nullMissionId \longrightarrow \\
\qquad \left(\begin{array}{l}
start\_mission\,.\,currentMission \longrightarrow \\
initializeRet\,.\,currentMission \longrightarrow \\
\left(\begin{array}{l}
RequestSequenceTermination \\
\qquad [\![\{ terminating \} \mid \{\!| \, end\_termination \, |\!\} \mid \varnothing]\!] \\
\left(\begin{array}{l}
done\_mission\,.\,currentMission \longrightarrow \\
end\_termination\,.\,sequencer \longrightarrow \textbf{Skip}
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\quad [] \ currentMission = nullMissionId \longrightarrow \\
\qquad terminating := TRUE \ ; \ \textbf{Skip} \\
\quad \textbf{fi}
\end{array}
$$

Once the current mission enters its execution phase (indicated by the communication on the *initializeRet* channel) the *RequestSequenceTermination* action is offered in parallel with a communication on the *done_mission* channel, which is used by the current mission to indicate that it has terminated.

The parallelism here specifies that *RequestSequenceTermination* and the communications in the brackets below the parallel operator synchronise on *end_termination*, that *RequestSequenceTermination* alters the *terminating* variable, and the behaviours below the parallel operator do not alter any variables.

Once a communication on *done_mission* occurs, *StartMission* waits for the action *RequestSequenceTermination* to be ready to engage in *end_termination*; this ends both sides of the parallelism and control returns to the *GetNextMission* action. *GetNextMission* then checks the value of the variable *terminating*, which is set

in the *RequestSequenceTermination* action shown in Section 5.2.3, to determine whether it should recurse or exit.

5.2.2 Schedulable Mission Sequencer

The *SchedulableMissionSequencerFW* process represents a mission sequencer nested within a mission; it is slightly more complicated than the *TopLevelMissionSequencerFW*.

It has a *sequencer* parameter and a *currentMission* state component, like the top-level mission sequencer process. Instead of *terminating* it has two state components: *terminatingAbove*, which indicates if this mission sequencer's controlling mission has asked it to terminate, and *terminatingBelow*, which indicates if this mission sequencer's current mission (or one of its managed schedulables) has asked it to terminate. Two variables are required because we have to treat the requests for termination differently, depending on their source. With two variables, we can model the different protocols separately.

Here, the *GetNextMission* action behaves identically to that used in the process *TopLevelMissionSequencerFW* (Section 5.2.1) aside from the conditional statement below, which checks both *terminatingAbove* and *terminatingBelow* to handle the possibility of the nested mission sequencer being asked to terminate from above or below itself in the program hierarchy.

$$
\begin{array}{l}
\textbf{if } terminatingAbove = FALSE \land terminatingBelow = FALSE \longrightarrow \\
\quad GetNextMission \\
[] \; terminatingAbove = TRUE \lor terminatingBelow = TRUE \longrightarrow \\
\quad \textbf{Skip}
\end{array}
$$

The *StartMission* action of the *SchedulableMissionSequencerFW*, shown below, contains a parallelism of three actions that offer the choice of waiting for its controlling mission to signal its termination (handled by the *SignalTermination* action), the *RequestSequenceTermination* action (which we discuss in Section 5.2.3), and waiting for its current mission to communicate its termination on *done_mission*.

$$
\begin{array}{l}
StartMission \mathrel{\widehat{=}} \\
\textbf{if } currentMission \neq nullMissionId \longrightarrow \\
\left(\begin{array}{l}
start\_mission \,.\, currentMission \longrightarrow \\
initializeRet \,.\, currentMission \longrightarrow \\
\left(\begin{array}{l}
\left(\begin{array}{l}
SignalTermination \\
[\![\{ terminatingAbove \} \mid \{\!| \, end\_terminations \, |\!\} \mid terminatingBelow]\!] \\
RequestSequenceTermination
\end{array} \right) \\
[\![\{ terminatingAbove, terminatingBelow \} \mid \{\!| \, end\_terminations \, |\!\} \mid \varnothing]\!] \\
\left(\begin{array}{l}
done\_mission \,.\, currentMission \longrightarrow \\
end\_terminations \,.\, sequencer \longrightarrow \\
\textbf{Skip}
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
[] \; currentMission = nullMissionId \longrightarrow \\
\quad terminating := TRUE \\
\textbf{fi}
\end{array}
$$

The *SignalTermination* action below handles the interaction with the controlling mission of this nested mission sequencer when it indicates that this mission sequencer should

terminate.

$$
\begin{array}{l}
\mathit{SignalTermination} \mathrel{\widehat{=}} \\
\left(\begin{array}{l}
\left(\ \mathit{end\_terminations}\,.\,\mathit{sequencer} \longrightarrow \mathbf{Skip}\ \right) \\
\Box \\
\left(\begin{array}{l}
\mathit{signalTerminationCall}\,.\,\mathit{sequencer} \longrightarrow \\
\mathit{terminatingAbove} := \mathit{TRUE}; \\
\mathit{requestTermination}\,.\,\mathit{currentMission} \longrightarrow \\
\mathit{signalTerminationRet}\,.\,\mathit{sequencer} \longrightarrow \\
\mathbf{Skip}
\end{array}\right) \\
;\ \mathit{end\_terminations}\,.\,\mathit{sequencer} \longrightarrow \\
\mathbf{Skip}
\end{array}\right)
\end{array}
$$

The *SignalTermination* action handles the nested mission sequencer being terminated from above and the *done_mission* communication handles the nested mission sequencer's current mission telling it to terminate from below. The *RequestSequenceTermination* action handles the nested mission sequencer being told to terminate its sequence of missions by a managed schedule. We discuss this action next.

5.2.3 Request Sequence Termination

The *RequestSequenceTermination* action, shown below, waits for a communication on the *requestSequenceTermination* channel. After this, the value of *terminating* is set to *TRUE* and the mission is queried to see if it is active and has not been asked to terminate already – using the channels *terminationPending* and *missionActive*. If these conditions are met, the action communicates on *requestTermination*, which tells the current mission to begin terminating. Then *RequestSequenceTermination* recurses, so that subsequent calls to `requestSequenceTermination()` in the SCJ application can be handled, and so that the action can be terminated using *end_termination*.

$$
\begin{array}{l}
\mathit{RequestSequenceTermination} \mathrel{\widehat{=}} \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{requestSequenceTermination}\,.\,\mathit{sequencer} \longrightarrow \\
\mathit{terminating} := \mathit{TRUE}; \\
\mathit{terminationPending}\,.\,\mathit{currentMission}\,?\,\mathit{missionTerminating} \longrightarrow \\
\mathit{missionActive}\,.\,\mathit{currentMission}\,?\,\mathit{missionIsActive} \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ \mathit{missionTerminating} = \mathit{FALSE} \wedge \mathit{missionIsActive} = \mathit{TRUE} \longrightarrow \\
\quad \mathit{requestTermination}\,.\,\mathit{currentMission} \longrightarrow \\
\quad\quad \mathbf{Skip} \\
[]\ \mathit{missionTerminating} = \mathit{TRUE} \vee \mathit{missionIsActive} = \mathit{FALSE} \longrightarrow \\
\quad \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
;\ \mathit{RequestSequenceTermination}
\end{array}\right) \\
\Box \\
\left(\ \mathit{end\_termination}\,.\,\mathit{sequencer} \longrightarrow \mathbf{Skip}\ \right)
\end{array}\right)
\end{array}
$$

In the *SchedulableMissionSequencerFW* process, the *RequestSequenceTermination* action differs only in that, where *terminating* is set to *TRUE*, the variable *terminatingBelow* is altered instead. This is to handle the schedulable mission sequencer being terminated from a schedulable that is above it in the program hierarchy using *SignalTermination*, or below it, using *RequestSequenceTermination*. This can be seen in the excerpts presented in Section 5.2.2 where *SignalTermination* sets *terminatingAbove* and *GetNextMission* checks both of these variables.

5.2.4 Clean Up

Our model of a mission uses three actions to model its three phases of operation: initialisation, execution, and clean up. As soon as one phase ends, the mission transitions to the next phase. Hence, the mission's *Cleanup* action begins directly after its *Execute* action has finished. First, the *CleanupSchedulables* action is called, which iterates over the set *schedulables* and executes the `cleanUp()` application method using synchronisations on the *cleanupSchedulableCall* channel followed by the *cleanupSchedulableRet* channel for each schedulable using its identifier s as a parameter. The interleave operator ($|||$) is used to interleave all of the clean up phases.

$$
\begin{aligned}
&\mathit{CleanupSchedulables} \mathrel{\widehat=} \\
&\quad ||| \; s : \mathit{schedulables} \bullet \\
&\qquad \mathit{cleanupSchedulableCall} \,.\, s \longrightarrow \\
&\qquad \mathit{cleanupSchedulableRet} \,.\, s \longrightarrow \mathbf{Skip}
\end{aligned}
$$

Once the clean up of each managed schedulable registered to this mission has completed, the *Cleanup* action executes the `cleanUp()` method of the mission itself using the *cleanupMissionCall* and *cleanupMissionRet* channels. Afterwards, the *Finish* action is executed; it informs the mission's application process to terminate (*end_mission_app*) and then uses *done_mission* to inform the mission's controlling mission sequencer that is has finished.

$$
\begin{aligned}
&\mathit{Finish} \mathrel{\widehat=} \\
&\quad \mathit{end\_mission\_app} \,.\, \mathit{mission} \longrightarrow \\
&\quad \mathit{done\_mission} \,.\, \mathit{mission} \longrightarrow \mathbf{Skip}
\end{aligned}
$$

This model captures the termination protocol as it currently stands. While the model is tractable, we argue that the same functionality can be achieved with a simpler termination protocol. In Section 5.3 we describe our model of the termination protocol including our proposed changes.

5.3 Model of Proposed Changes to Termination Protocol

This section describes a new model for the SCJ termination protocol incorporating our proposed changes. As explained in Section 4.4, we propose the removal of the `requestSequenceTermination()` method to prevent mission sequencers from being terminated by arbitrary schedulables. To enforce an organised termination of mission sequencers we propose that the `mission.cleanUp()` method return a boolean value which

is passed to the mission sequencer to determine if the mission sequencer should continue or terminate.

In adapting our model to our proposed protocol, the state of both flavours of mission sequencer process have been altered. In the *TopLevelMissionSequencerFW* process, *terminating* has been replaced with *continue*. In the *SchedulableMissionSequencerFW* process, both the *terminatingAbove* and *terminatingBelow* variables have been replaced with *continueAbove* and *continueBelow*. These variables indicate to the sequencer that it should continue executing its sequence of missions (if they are both *TRUE*).

If *continue* is *FALSE*, or either *continueAbove* or *continueBelow* is *FALSE* in the case of the *SchedulableMissionSequencerFW*, then the mission sequencer does not execute any more missions. In the *SchedulableMissionSequencerFW* the variable *ContinueBelow* holds the return value from the current mission that is communicated to the mission sequencer at the end of the cleanup phase on the *done_mission* channel – in the *TopLevelMissionSequencerFW* process this value is held in the *continue* variable. The *SchedulableMissionSequencerFW*'s *continueAbove* variable is only changed during the *SignalTermination* action, which handles the mission sequencer's controlling mission requesting it to terminate.

Removing the *RequestSequenceTermination* action is a clear simplification of the model; the *requestSequenceTermination* channel is no longer needed and is removed from the model entirely. Besides this, the actions (in the model of the current termination protocol) that use the *RequestSequenceTermination* action are also simplified in our new model. We give more details of these simplifications in the following three sections.

5.3.1 Top Level Mission Sequencer

The *StartMission* action in the *TopLevelMissionSequencerFW* process is simplified in comparison to the previous version in Section 5.2.1, as can be seen from the excerpt presented below.

$$
\begin{array}{l}
\mathit{StartMission} \mathrel{\widehat{=}} \\
\quad \textbf{if } \mathit{currentMission}! = \mathit{nullMissionId} \longrightarrow \\
\qquad \left(\begin{array}{l}
\mathit{start\_mission} \,.\, \mathit{currentMission} \longrightarrow \\
\mathit{done\_mission} \,.\, \mathit{currentMission} \,?\, \mathit{continueReturn} \longrightarrow \\
\mathit{continue} := \mathit{continueReturn} \,;\ \textbf{Skip}
\end{array} \right) \\
\quad [] \ \mathit{currentMission} = \mathit{nullMissionId} \longrightarrow \\
\qquad \mathit{continue} := \mathit{FALSE} \,;\ \textbf{Skip} \\
\quad \textbf{fi}
\end{array}
$$

This action simply starts the current mission using the *start_mission* channel and then waits for it to terminate and communicate on the *done_mission* channel.

5.3.2 Schedulable Mission Sequencer

The *StartMission* action in the *SchedulableMissionSequencerFW* process (which models a nested mission sequencer) is shown below. It is necessarily more complex than that of the *TopLevelMissionSequencerFW* process, but still simpler than the previous version in Section 5.2.2.

$$
\begin{array}{l}
\mathit{StartMission} \mathrel{\widehat{=}} \\
\quad \textbf{if } \mathit{currentMission}! = \mathit{nullMissionId} \longrightarrow \\
\qquad \left(\begin{array}{l}
\mathit{start\_mission} \,.\, \mathit{currentMission} \longrightarrow \\
\mathit{initializeRet} \,.\, \mathit{currentMission} \longrightarrow \\
\left(\begin{array}{l}
\mathit{SignalTermination} \\
\qquad [\![\varnothing \mid \{\!| \mathit{end\_terminations} |\!\} \mid \{\mathit{continueBelow}\}]\!] \\
\left(\begin{array}{l}
\mathit{done\_mission} \,.\, \mathit{currentMission} \,?\, \mathit{continueReturn} \longrightarrow \\
\mathit{continueBelow} := \mathit{continueReturn}; \\
\mathit{end\_terminations} \longrightarrow \textbf{Skip}
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\quad [] \; \mathit{currentMission} = \mathit{nullMissionId} \longrightarrow \\
\qquad \mathit{continueBelow} := \mathit{FALSE}; \\
\qquad \textbf{Skip} \\
\quad \textbf{fi}
\end{array}
$$

After the mission has been initialised (indicated by the *initializeRet* channel) this action proceeds to a parallelism that offers *SignalTermination* to handle this mission sequencer's controlling mission being terminated and a communication on *done_mission* that indicates that the mission sequencer's current mission has terminated.

5.3.3 Clean Up

To model `Mission.cleanUp()`, which now returns a boolean value, the *MissionFW* process's *cleanupMissionRet* channel takes a boolean parameter.

$$
\begin{array}{l}
\mathit{Cleanup} \mathrel{\widehat{=}} \\
\quad \mathit{CleanupSchedulables}; \\
\quad \mathit{cleanupMissionCall} \,.\, \mathit{mission} \longrightarrow \\
\quad \mathit{cleanupMissionRet} \,.\, \mathit{mission} \,?\, \mathit{continueSequencer} \longrightarrow \\
\quad \mathit{Finish}(\mathit{continueSequencer})
\end{array}
$$

This value is communicated to the *MissionSequencer* process via the *done_mission* channel. This channel is the means of communication that allows a mission to inform its controlling mission sequencer of its completion, and, as revised, also communicates continuation information.

$$
\begin{array}{l}
\mathit{Finish} \mathrel{\widehat{=}} \\
\quad \mathit{end\_mission\_app} \,.\, \mathit{mission} \longrightarrow \\
\quad \mathit{done\_mission} \,.\, \mathit{mission} \,?\, \mathit{continueSequencer} \longrightarrow \\
\quad \textbf{Skip}
\end{array}
$$

When the *MissionSequencer* receives the boolean value from *done_mission*, it stores it in the variable *continue*, which is checked by the *GetNextMission* action after the *StartMission* action finishes. This variable is used to decide whether the *MissionSequencer* should continue its execution and get another mission or terminate. This minor addition to the model presents little extra complexity, while supporting our proposal to simplify the termination protocol significantly.

Section 5.4 compares the two termination protocols in more detail, using our formal models.

5.4 Comparison of Termination Protocols

The current termination protocol allows any schedulable object to call the method `requestSequenceTermination()` (the *Circus* action for which is presented in Sect. 5.2.3) of any mission sequencer in the program, regardless of its place in the hierarchy. The mission sequencer that receives this call informs its current mission to terminate. This is captured in the excerpt in Sect. 5.2.3 by the communication on the *requestTermination* channel, which indicates to the mission that is should terminate. The mission, once it is instructed to terminate, passes this on to its schedulables – at least one of which may have called the `requestSequenceTermination()` method of the mission sequencer in the first place. This creates a needless cycle of termination requests.

In the new termination protocol that we propose, the instigation of termination still begins in a schedulable object, but this request is only passed up one tier at a time. For example, if a reason to terminate the application is detected by a schedulable object, this is passed to its controlling mission – by setting some flag in the mission for example. Once it has terminated, the mission communicates this request for termination to the mission sequencer controlling it, during the mission's clean up phase. This is captured in our models by the communication on the *done_mission* channel of the boolean parameter *continueSequencer* to the *MissionSequencer* process that controls the mission. In this way, the request for termination passes up the program hierarchy, with each tier terminating before the next tier begins handling its termination.

This prevents the situation present in the current protocol in which a schedulable object that initially discovers the need for termination is requested to terminate later when the termination request cascades back down from the mission sequencer it had called `requestSequenceTermination()` on initially. Our new approach does create a small amount of programmer overhead, since the programmer must ensure that schedulable objects can inform their controlling mission that it should return `false` from its `cleanUp()` method. A simple way to remove this small overhead is for the default implementation of `Mission.cleanUp()` to return `false`.

We note that even in the new termination protocol, the schedulable object that discovers the need for termination triggers the termination of its controlling mission and then is asked to terminate itself. To avoid this, the schedulable objects can be programmed to check for the termination of their controlling mission periodically and begin to shut themselves down; this is in fact the only way to terminate a `ManagedThread`. Another solution is to have the schedulable that discovers the need for termination to terminate itself after it has triggered the termination of its controlling mission.

Our changes have a subtle effect on the termination order of the objects in the program. As an example, we consider a program with two nested subsystems. With the current protocol, a schedulable object within one of them may call the method `requestSequenceTermination()` on the top-level mission sequencer and begin a cascade of termination requests that leads to the nested sub-systems terminating in parallel. In the same situation, using the new termination protocol, the termination requests must pass up the hierarchy from the schedulable object that initiates termination to the top-level mission sequencer. This means that the subsystem that contains the schedulable object that requested termination has to terminate before the request for termination passes to the top-level mission and the termination of the other subsystem – and any schedulable objects started by the top-level mission – begins.

In summary, the `requestSequenceTermination()` method complicates the SCJ termination protocol by allowing arbitrary termination of mission sequencers. Our models, while tractable, are complex when modelling this feature of the language at Level 2.

With our proposed changes incorporated, our models become much simpler and are easier to analyse. Our proposed changes to the SCJ termination protocol represent a positive simplification of the language while retaining the ability to terminate a mission sequencer from the application.

In order to show how far our proposed changes simplify the model of SCJ, we have constructed two specifications based on a simple example program. This program contains a single mission, controlling two managed threads that share a one-place buffer in the mission's memory. One specification uses the model of the current framework and the other specification uses the model of the new framework, with our proposed changes. Both of these specifications have been translated to CSPm, the machine-readable version of CSP, in order to utilise the Failures Divergences Refinement checker (FDR) [17] to model check the specifications. Because CSP does not have a notion of state in the same way that *Circus* does, the CSP versions of our models use state processes to model the reading of and assignments to variables that *Circus* allows, which means that the CSP models have more states than the *Circus* versions.

The results obtained are from running a check for divergence-freedom while hiding any channels relating to the state processes. The model of the current framework shows 4,539,021 states, whereas the model of the new framework shows only 249,869 states. Our proposed changes decrease the number of states in the model by 94.5% in comparison to the original model. Such a decrease in the number of states in our model shows a simplification that is useful, both for further modelling efforts and for programmer understanding of the SCJ paradigm.

6 Related Work

There have been previous efforts to provide safe language subsets for safety-critical systems, similar to the intent of Safety-Critical Java (SCJ). MISRA C [26] is a restricted subset of standard C that originated in the automotive industry, but now now provides guidelines for the use of C in other critical systems.MISRA C has gained wide popularity in aircraft, medical systems, and other critical software domains [18].

Several subsets of Ada have been developed since the language was first defined. One of the most widely used 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 programs. This is balanced by ensuring that the language has the right level of abstraction to provide the expressive power needed to hide the details of implementations. SPARK also acknowledges the desire for safety-critical programs to be verifiable and restricts the language with this objective [2]. SPARK has become one of the most popular choices for high-integrity real-time systems.

The Ravenscar [13] profile is another subset of Ada. It has a similar level of complexity to SCJ Level 1. 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 or sporadic; 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, the Ravenscar-Java profile [21] was created to improve the reliability of Java-based systems using the Real-Time Specification for Java (RTSJ). The RTSJ is the basis for SCJ Programs written in the Ravenscar-Java profile conform to the RTSJ standard, with extra restrictions to ensure

the program adheres to the Ravenscar rules. Other Java profiles have been proposed: for example that by Schoeberl et al. [33], who also considers the possibility of missions as application modes of operations [32].

As far as we are aware, other than the work by Hunt and Nilsen [20], there has been no previous work that has considered how components should be implemented cleanly in SCJ. There have been several approaches suggested for the RTSJ – see [29] 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 the RTSJ's memory areas.

There have been attempts to integrate the OSGi Java-based framework [28] 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 [31], which considers real-time admission control of components within a Real-Time OSGi framework. They recognise 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 the RTSJ's 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 replenishment period. This resets the schedulable objects to their original priorities. Effectively, this approach can be used to implement a sporadic-server. It forms the basis of the approach that we have proposed in Section 4.3 for SCJ.

Other formalisations of languages for safety-critical systems exist. Blazy and Leroy [3] present a formal semantics of the C subset, Clight. Ellison and Rosu [14] present an executable formal semantics of C, which allows model checking of the translated C program. Tews et al. [35] provide a formal semantics of a subset of C++. This semantics is embedded in the Prototype Verification System (PVS) prover and a prototype translator tool allows the translation of program code to the PVS semantics. Automatic translation from safety-critical programming languages is part of our agenda for future work.

The memory model of SCJ is based upon that of the RTSJ. Engel has produced a formalisation of a restricted version of the RTSJ memory model [15, 16] to prove the absence of runtime errors caused by misuse of the memory model. This formalisation is implemented with the KeY theorem prover [1]. The restricted memory model Engel considers is similar to, but much less restrictive than, that used in SCJ. This indicates that Engel's approach may be capable of being adapted to model the SCJ memory model. Our model does not cover the memory model of SCJ, which is formalised in [9].

Brooke et al [4] use CSP – one of the components of *Circus* – to model the semantics of the real-time extension to Eiffel, SCOOP. This follows a similar direction to our work, but since SCJ is a more restricted language, the CSP model in [4] is more complex due to the generality it supports.

Kalibera et al [22] present a technique to allow the model checking of SCJ Level 0 or 1 programs. They extend the Java PathFinder tool with a scheduling algorithm that allows it to explore the possible schedulings of an SCJ program. Their approach is concerned with scheduling errors, assertion failures, and scheduling dependant memory access errors. While that work actually focusses on SCJ, our approach is more general and allows the checking of a wider range of program properties.

7 Conclusions

SCJ Level 2 has received little public scrutiny. Most papers address SCJ either Level 0 or 1. Whilst it is clear from the SCJ language 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 scenarios in which applications cannot be easily implemented at Level 1 and, therefore, Level 2 support is required. In doing so, we have found no redundant features of Level 2. For each feature (only available at Level 2) we have presented good examples that require use of that feature.

Our studies also reveal some deficiencies in the features provided at Level 2. The lack of convenient support for terminating managed threads is not controversial and is probably just an omission in the current SCJ specification.

It could be argued that the inability to set a deadline on the transition between missions is not necessary for safety-critical systems as static analysis should have determined whether deadlines can be met. However, we note that SCJ does support detection of deadline misses on managed schedulable objects at Levels 1 and 2.

The need to enrich the mission sequencer concept to support composibility of timing constraints 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 implementation of the approach when the threads within the process can execute in parallel.

SCJ support at Level 2 needs to be more complex than at Level 0 or Level 1. The protocol that supports the termination of missions and their sequencers is more complex than is necessary and allows programs to break through the mission hierarchy in an uncontrolled fashion. We have proposed simplifications of the protocol that reinforces the hierarchical nesting of mission sequencers.

8 Acknowledgements

This research reported in this paper is funded by the UK EPSRC under grant EP/H017461/1. Wellings is a member of the Java Community Process JSR 302 Expert Group, which is tasked with developing the Safety-Critical Java Specification. We would like to thank the other members of the Expert Group for their contributions and feedback on some of the ideas expressed in this paper. No new primary data were created during this study.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. Schmitt. The Key Tool. *Software & Systems Modeling*, 4(1):32–54, 2005.

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

[3] S. Blazy and X. Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[4] P. J. Brooke, R. F. Paige, and J. L. Jacob. A csp model of eiffel's scoop. *Formal Aspects of Computing*, 19(4):487–512, 2007.

[5] 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.

[6] 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.

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

[8] A. Cavalcanti, A. Sampaio, and J. Woodcock. A refinement strategy for Circus. *Formal Aspects of Computing*, 15(2-3):146–181, 2003.

[9] A. L. C. Cavalcanti, A. Wellings, and J. C. P. Woodcock. The Safety-Critical Java memory model formalised. *Formal Aspects of Computing*, 25(1):37–57, 2013.

[10] R. Davis and A. Burns. 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.

[11] 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.

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

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

[14] C. Ellison and G. Rosu. An executable formal semantics of c with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, New York, NY, USA, 2012. ACM.

[15] C. Engel. Deductive verification of rtsj programs. In *Proceedings of the 2nd Junior Researcher Workshop on Real-Time Computing (JRWRTC 2008)*, 2008.

[16] C. Engel. *Deductive verification of safety-critical Java programs.* PhD thesis, Karlsruhe Institute of Technology, 2009.

[17] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe. *Failures Divergences Refinement (FDR) Version 3*, 2013.

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

[19] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

[20] 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.

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

[22] T. Kalibera, P. Parizek, M. Malohlava, and M. Schoeberl. Exhaustive testing of safety critical java. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 164–174, New York, NY, USA, 2010. ACM.

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

[24] 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, JSR 302, 6 December 2012.

[25] C. Morgan. *Programming from Specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[26] Motor Industry Software Reliability Association. MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems. Technical report, Motor Industry Research Association, 2013.

[27] M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for circus. *Form. Asp. Comput.*, 21(1-2):3–32, 4 Dec. 2007.

[28] OSGi Alliance. Osgi service platform core specification, 2014. http://www.osgi.org/Specifications/HomePage.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., 1992.

[35] H. Tews, T. Weber, and M. Völp. A formal model of memory peculiarities for the verification of low-level operating-system code. *Electronic Notes in Theoretical Computer Science*, 217:79–96, 2008.

[36] 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.

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

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

[39] 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.

[40] F. Zeyda, L. Lalkhumsanga, A. Cavalcanti, and A. Wellings. Circus Models for Safety-Critical Java Programs. *The Computer Journal*, page bxt060, 2013.