The University of York Department of Computer Science

Submitted in part fulfilment for the degree of BSc.

Programming in Safety Critical Java

Ivaylo Hristakiev

2013-May-6

Supervisor: Dr. Ana Cavalcanti

Number of words = 12681, as counted by wc -w.
This includes the body of the report only.

Abstract

Safety-critical systems are systems in which a failure can cause damage to property, even loss of human life. Such systems are required by law to be certified, e. g. to DO-178B Level A. Production of certification evidence is costly and time-consuming, hence the whole project lifecycle is carefully planned. Languages for developing such systems are minimalistic with specific support. For example, Ada 2005 together with the Ravenscar profile have been used widely in industry.

The Safety-Critical Java (SCJ) Specification has recently emerged as an attempt to make Java more suitable for the development of safety-critical systems. SCJ has a Scoped Memory model and support for fixed-priority preemptive scheduling and is defined as a subset of predictable, analysable Java subset called the *Real-Time Specification for Java (RTSJ)*. Scoped Memory is memory not subject to garbage collection and is a popular topic of research.

However, there exist only a limited number of use cases illustrating common use of concurrency mechanisms in SCJ. The language mainly focuses on event handling as its support for periodic and aperiodic activities.

This project explores several concurrency techniques in SCJ. We develop them using the departmental Reference Implementation together with the public draft of the SCJ Specification. Our results allow us to draw conclusions about SCJ as a concurrency programming paradigm from the perspective of a developer new to the area of safety-critical systems.

Acknowledgements

I would like to thank my Supervisor Dr. Ana Cavalcanti for the all the invaluable guidance and support throughout the year.

I would also like to thank Prof. Andy Wellings for the insights into the rationales behind SCJ.

Last but not least, I would like to thank all my family and friends for the amazing support and source of motivation.

Contents

1 Introduction

1.1 Motivation

Concurrent programming allows multiple processes to execute in parallel and cooperate towards achieving a common goal. However, this type of programming is harder than traditional sequential programming - Bloch [1, §9] advises to always try and use a library that can save the programmer from doing low-level multithreaded programming. This is good advice because concurrency can lead to undesirable behaviour such as deadlock and corrupted data. In extreme circumstances, these problems can cause the system to be unable to progress further or to produce wrong results.

Safety-critical systems must be highly reliable and respond to input stimuli within a finite period of time. Such systems must be engineered so that the problems of concurrency never arise and the system designers need to provide evidence of that to a certification body.

Languages used for the development of safety-critical systems are usually small subsets that provide fine-grain control over the finished product to enable such certification. The Ada 2005 programming language, together with the Ravenscar profile, have been proven to be certifiable and have been used widely in industry to develop such systems.

Java is a novel programming language. Since its release in 1995, it has received widespread attention from the community of Internet businesses. The language is familiar, easy to use, object-oriented and robust. The use of inheritance and polymorphism provided programmers with the power to design applications that model the real-world. Since then, Java has spread into almost every niche in the IT industry.

Attempts have been made to extend Java into the real-time and embedded systems domains. Although Java has many advantages over its competitors, it does have some serious drawbacks with regards to the development of real-time and embedded systems, namely unpredictable garbage collection and its 'stop-the-world' paradigm [2]. This unpredictable blocking can cause concurrent processes to fail to respond within their deadline. This has led to the development of the Real-Time

Specification for Java (RTSJ). The RTSJ, although suitable for the development of soft and hard real-time systems, is too complex and big to make certifiability tractable.

This has led to a further restriction of the language and its run-time capabilities called Safety Critical Java (SCJ). The language specification is fairly recent and there only exist a limited number of use cases illustrating its use for concurrent programming.

1.2 Objectives

The aims of this project are summarised as follows:

- Identify classical problems in concurrent and real-time programming.
- Develop and test programs that solve these problems in SCJ.
- Evaluate the programs and the adequate ability of SCJ as a language and programming paradigm.
- Generalize the conclusions in relation to further work in SCJ.

1.3 Report structure

The report is organised as follows:

- Chapter 2 — Gives background information on Java concurrency, RTSJ and its Scoped Memory Model, and SCJ.
- Chapter 3 — Shows related work done in the field of SCJ.
- Chapter 4 — Presents our choice of concurrency mechanisms that we develop in SCJ along with design decisions.
- Chapter 5 — Includes the discussion of our implementations of the designed concurrency examples in SCJ. Takes into consideration any issues presented by the Reference Implementation.
- Chapter 6 — Tests the programs with regards to their concurrency aspects.

- Chapter 7 — Draws attention to the evaluation of our work. It also presents our review on the ability of SCJ to provide sufficient concurrent programming mechanisms.
- Chapter 8 — Concludes our work and discusses future work.

1.4 Ethical statement

I declare that throughout this project I have maintained highest of ethical and moral standards as prescribed by the IET Rules of Conduct of which I am a member.

I declare the work presented to be my own, unless explicitly referenced using the departmental guidelines.

None of my work required human volunteers to participate, hence there are no implications regarding their welfare.

All software used to produce this work was properly licenced and runnable on the departmental machines.

I acknowledge that parts of this work, be it code or results, may contribute or be partly used for the development of some sort of component relevant to safety-critical systems, whose integrity has implications to human and property welfare.

2 Literature Review

This chapter focuses on providing the reader with enough background information to understand the rest of this report. The main topics included are Java's concurrency, Real-Time Specification for Java and its Scoped Memory paradigm, the Safety Critical Java Specification.

2.1 Java

Since its release in 1995, Java has gained popularity unrivalled by any other programming language. Java is a simple, interpreted, object-oriented language whose syntax and semantics contributed to its spread from the telecommunications industry to a wide spectrum of application fields. The main advantages are as follows [3]:

- simple, robust, object-oriented — objects are the first-class citizens of Java; the object-oriented paradigm allows programmers to model real world systems in an intuitive way. Java's syntax is similar to C++, also a popular OO language.

- interpreted and portable —Java source code is compiled to bytecode, an intermediate version of the code that can execute in any JVM (Java Virtual Environment); this allows the programmer to abstract away from low-level architecture concerns.

- fault tolerant —the language allows for the insertion of code that handles the rise of exceptional circumstances during the program execution.

- dynamic —classes are loaded only when actually needed by the program, and new classes can be linked dynamically during execution.

- multithreaded — incorporates the notion of a Thread as a concurrent activity in a program.

- implicit memory management — Memory is handled by the JVM Garbage collector that recycles unused objects. It alleviates the burden of worrying about allocating and deallocating memory when creating/destroying objects.

2.1.1 Concurrency in Java

Concurrency is a technique for specifying a set of processes that are to be executed in parallel. It allows for a higher CPU utilization and also for structuring the software in a more readable, reliable and maintainable fashion. However, concurrency implies synchronisation and communication between some (or all) parallel processes (threads of execution). If such cooperations are not properly regulated (either by the language or by the programmer), several problems can arise that are not exhibited in ordinary sequential programs including:

- deadlock — no processes can make further progress in their execution.
- interference — the interleaving of two processes causes a shared object to be in an inconsistent state.
- starvation — a process is continuously denied a resource as a result of the execution of another.

Concurrent programming is more difficult than single-threaded programming. For a further discussion see Bloch [1, §9].

Java incorporates the notion of concurrency in two ways. First, threads of execution are represented by the Thread class. Second, different threads can synchronize their use of a shared object by accessing the object's monitor using the **synchronized** keyword.

Threads

In Java, there are two ways of creating a thread. First, you can create a class that extends the `java.lang.Thread` class and overwrite the `run()` method. Then, you can create an instance of that class and start the thread by calling the `start()` method. The second way is to implement the `java.lang.Runnable` interface and to pass that implementation to the constructor of a `java.lang.Thread` object. Either way, the thread is not started until the `start()` method is called.

Threads also have a state. When a thread is created, it enters the NEW state. When it is actually started by the application, it becomes RUNNABLE and is put on the Runnable queue. The Runnable queue is a queue of threads that can be ran by the Scheduler. When a thread is picked by the Scheduler, it starts to execute. During execution, the thread can self-suspend and become WAITING, or request a resource by entering a monitor becoming BLOCKED.

A thread can terminate in several ways:

- it finishes the execution of its `run()` method; or

- destroy() is called without giving the thread a chance to cleanup (deprecated since Java 1.5); or

- stop() is called but this time the thread has the chance to cleanup before termination. This allows the release of any locks currently being held and the execution of any finally blocks. This method of thread termination is inherently unsafe because locks are being preempted making it possible for shared data protected by said locks to become corrupt. Therefore, the `stop()` method has also been deprecated.

Threads can also have priorities associated with them. The Scheduler can preempt the currently executing thread, place it back on the Runnable queue and pick the highest priority thread to start execution. However, a JVM may use priorities merely as a guide on how and when to schedule threads. There is no guarantee that the highest priority runnable thread is always the one that is executing. This can be a problem, for example, in real-time systems.

The Scheduler determines which thread to run next and for how long it should run. To ensure no thread is starved, a reasonable JVM implementation will try to make the choice fair. However, well-written multithreaded programs should not depend on the policy implemented by the scheduler — all possible interleavings of concurrent activities should lead to correct program execution. Bloch [1, §9] warns that if not so, the program will be neither robust nor portable as thread priorities vary widely between operating systems on which the JVM is dependant.

Common concurrency mechanisms involving threads are covered in Burns and Wellings [4, §9], Wellings [5, §5].

Monitors

The monitor construct allows shared data to be accessed by multiple concurrent processes under mutual exclusion. In Java, an object's lock is used when a thread tries to execute a synchronized block or method. If the lock is already held by some other thread, the calling thread is blocked until the lock becomes available again. As long as all access to the shared resource is via a monitor using the **synchronized** keyword, then the resource will always be in a consistent state. However true, this does not tell the whole story.

Not only does synchronization prevent an object being observed in an inconsistent state, but it also makes sure the object transfers from one consistent state to another in a sequential manner [1, §9]. Every thread that executes a synchronized block or method will see the effects of all previous synchronized blocks or methods, if any. Furthermore, Bloch [1, Item 48] gives further advice on the use of monitors - as little work as possible should be done inside synchronized regions and Thread.wait() must never be called outside a loop due to spurious JVM wakeups.

2.1.2 Summary

This chapter has introduced the basic Java concurrency model. The main points were the `Thread` class for representing concurrent activities and the notion of a monitor for communication and synchronization between threads. The RTSJ and SCJ adopt those main features, so it is important for us to be familiar with their use.

2.2 Real-time systems and RTSJ

Real-time systems are information processing systems that respond to inputs within a specified amount of time [6]. Their correctness depends not only on their output but also on the time that output was produced. The languages used to program such systems must incorporate several facilities including representation of time, specification of periodic, aperiodic and sporadic activities, and control over input and output jitter.

Real-time systems are often embedded and as such are made to exhibit reliability and safety. When failure can cause damage to property or endanger the well-being of its users, the system is called *safety critical*.

Java has been a major success as a programming platform for the development of large-scale, complex systems in a variety of application areas. However, the general concurrency model and low support for real-time facilities have made it impossible to extend its applicability to the real-time and embedded systems domain. This has led to the development of the *Real-Time Specification for Java* (RTSJ). It enhances the development and analysis of real-time systems.

2.2.1 RTSJ Memory Management

Java has an implicit memory management model. The run-time system creates objects on the heap, and manages their finalization through a garbage collector, which may execute either when it detects that there is little to none free heap space available or incrementally. The existence of garbage collection may have a negative impact on analyzing the timing properties of the system (Wellings [5, p. 1]). Therefore, for hard real-time systems, the vagaries of garbage collection must be avoided.

RTSJ introduces a new type of memory called Scoped Memory ([5, §8]). A scoped memory area is where objects with a well-defined life-span are created. Scoped memory is logically outside the heap, hence not subject to garbage collection. Scoped memory can be entered explicitly by threads of control and such threads can allocate objects within the memory area. More than one thread can enter the same scoped memory and a thread can enter multiple *nested* memory areas. When a scoped memory area is no longer used, i. e. all threads of control have exited the area, all objects resident in it are finalized and the memory can be reused by the JVM.

Scoped memory comes in two flavours. The first is called `LTMemory` and requires the allocation time to be linear with respect to the size of the objects being created. The second, called `VTMemory`, does not have such a requirement and allocation can occur in variable time (variable time memory). Allocation in `VTMemory` is expected to be faster but less predictable than allocation in `LTMemory`.

A typical example of Scoped Memory use is a top-level loop that repeatedly invokes a sequential piece of code. This is the cyclic execution approach and a general pattern for scoped memory use in this case is presented in Pizlo et al. [7]:

```
    while ( true ) {
            // read some sensor data
            // compute next action
            // output commands to actuators
    }
```

Listing 2.1: Cyclic execution

If the loop body allocates objects, then the response time of each iteration will vary depending on the amount of interference it gets from the garbage collector. Therefore, it is useful to use a scoped memory area for such objects that is created and recycled at each iteration.

Listing 2.2 shows the use of Scoped Memory for the cyclic executive.

```
    memory = new LTMemory ( initSize, maxSize );
    loopLogic = new Runnable(){
            void run() {
            // read some sensor data
            // compute next action
            // output commands to actuators
            }
    };
    while ( true ) memory.enter( loopLogic );
```

Listing 2.2: Scoped Cyclic execution

This does several things. First, it defines a linear time memory area of initial size `initSize` that has the capacity to store upto `maxSize` data. Next, it extracts the logic of the original loop body into an object that implements the `Runnable` interface. At each iteration of the `while` loop, it makes a call to the RTSJ method `ScopedMemory.enter(Runnable logic)`. It associates the current thread of execution with the scoped memory area, enters into that area, executes the `run()` method of the passed logic parameter, and finally exits the scoped memory area thus

Stored In	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Permit	Permit	Forbid
Immortal	Permit	Permit	Forbid
Scoped	Permit	Permit	Permit if from same or outer scope
Local variable	Permit	Permit	Permit

Table 2.1: RTSJ Assignment rules [5, p. 145]

forcing the JVM to deallocate any objects created during the execution of the `run()` method.

RTSJ introduces an extra flavour of Scoped Memory called Immortal memory; it is for objects that are never finalized throughout the execution of the program. It is anticipated that objects will usually be allocated here at the initialization phase of a program, hence whether allocation is linear in relation to object size becomes a less relevant concern.

Assignments between different types of memory can be problematic. Since objects in Scoped Memory have a defined lifetime, a reference to such an object that lives in another memory area (e. g. Heap Memory or Immortal Memory) will become dangling after the object is reclaimed. To avoid this problem, RTSJ defines assignment rules (Table 2.1) They must be enforced at run-time, otherwise the safety of the Java program would be compromised. However, the compiler can also perform some static analysis to help the programmer in deciding if a particular assignment is possible or not.

Portals

Consider the situation when cooperating schedulable objects have no other relationship between themselves except for using a common scoped memory. If they are to use share objects between themselves and cooperate, they must obtain a reference to such objects. Due to the assignment rules, such a reference can be held only in a scope that is the same or more deeply nested.

Portals, introduced by the RTSJ, are a novel way of dealing with this situation. Each Memory Area has associated methods `setPortal(Object o)` and `getPortal()` that can save and obtain a reference to an object allocated in the specific memory area. The lifetime of the reference is the lifetime of all other objects in the scoped memory — for as long as there are active objects in it. After all schedulable objects have existed the area, the portal reference is reclaimed as well. One can create a dummy thread to keep the memory area alive if the two schedulable objects enter and leave at different times. This is the Wedge Thread pattern [7, §3.5].

2.2.2 Events and Event Handlers

It may not always be appropriate to use threads (Ousterhout [8], van Renesse [9]) because either there are many external objects with simple non-blocking control algorithms or the external objects are tightly coupled and their control is difficult in terms of synchronization and communication.

RTSJ extends the notion of a concurrent activity by supporting event handlers — stateless, short-lived objects whose execution is triggered by the occurrence of some event. Threads and event handlers come under the umbrella term Schedulable Object. Each Schedulable object has associated release requirements (when it should become runnable), memory requirements (e. g. the rate of object allocation) and scheduling requirements (at what priority level it should execute). An event can be associated with many handlers and vice versa. When events are triggered, they are placed on a queue (which is ordered in some way, e. g. by Priorities, Deadlines, FIFO, etc). Events are processed from the queue by server threads. When a thread picks an event, it executes its attached handlers. The advantages of this model are better scalability, resource usage and performance. However, it can be difficult to have event handlers with tight deadlines. Events compete for a limited number of server threads and a higher importance handler may have to wait for a server thread to be freed up by a potentially long-lived lower importance handler.

2.2.3 Summary

This chapter has introduced RTSJ and its main differences with Java with regards to memory management and concurrency support. In designing SCJ programs, it is important to understand the differences between threads and event handlers and when one is preferred over the other as SCJ level 1 only allows periodic/aperiodic event handlers. Programmers new to the language may need to pay extra attention to these differences in order to develop correct, clear and maintainable applications. Furthermore, the Scoped Memory model and the restrictions it imposes are central to Real-Time Java and Safety-Critical Java and is considered one of the major differences with Standard Java.

2.3 Safety Critical Java

This section presents the Safety-Critical Java Specification and the reasons that led to its development.

2.3.1 Safety-critical systems and SCJ

As already mentioned in section 2.2, safety-critical systems are systems in which an incorrect response or a wrongly timed response can cause harm to property or to human well-being. Such systems are required to follow rigorous development, validation and certification phases and must obtain such certification by law, e. g. the US Aviation certification DO-178B [10]. The production of evidence for such certification is expensive and time-consuming, hence the development and validation of the system and evidence of its properties are carefully planned and designed. The produced software is minimal with respect to its specification, e. g. no recursion is used and memory is carefully controlled to avoid out-of-memory conditions.

To bring Java closer to the area of safety-critical systems, the JSR-302 has developed the Safety Critical Java (SCJ) Specification [11]. It is based on the RTSJ v1.1 and addresses the areas of memory management and concurrency of RTSJ, among others. SCJ attempts to bring together the advantages of software development with Java, namely increased software productivity, modularity and readability, together with the high reliability observed in safety-critical applications.

The SCJ specification makes several constraints on the RTSJ, namely

- the usage of dynamic memory allocation — to mitigate out-of-memory conditions and simplify memory analysis during production of certification evidence.
- SCJ software will execute correctly on an RTSJ-compliant platform.
- defines new classes, which are designed to be implementable using RTSJ facilities, and redefines RTSJ classes and facilities that are too complex or confusing.
- defines annotations to enable off-line analysis of memory management thus proving the absence of specific run-time errors.
- omits and modifies some standard Java capabilities:
  - dynamic class loading is not required

  - finalizers are not required
  - the heap is not allowed
  - self-suspension of any type is not allowed
  - the procedure for starting an applications differs from other Java platforms
  - Priority Ceiling Emulation is required, also called Immediate Ceiling Priority Protocol (ICPP) in Scheduling theory ([6, §11]). Priority Inheritance, an inferior protocol, is not required.

- a JVM does not need to be used; the application can be instead compiled directly to object code

Concepts unique to the SCJ Specification such as a mission and compliance levels are considered in turn.

2.3.2 The Mission lifecycle

An SCJ application consists of one or more `Mission` objects executed in sequence. Each `Mission` is an independent computational unit that consists of several Schedulable objects which communicate via shared objects residing in the Mission memory. It has an initialization, execution and cleanup phases:

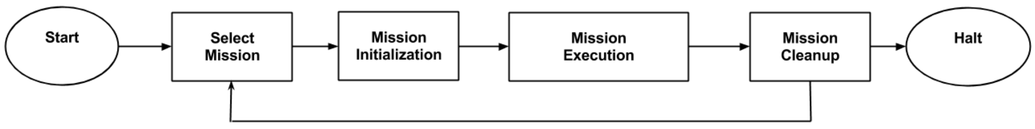

Figure 2.1: The Mission lifecycle

Each mission has an associated MissionMemory where all the mission objects are allocated during the initialization phase. When initialization is complete, the execution phase is entered. During execution, no new objects will usually be allocated, however this may not always be the case (See section 2.4.1). When a Schedulable object is released, it enters its own private scoped memory area that is not shared. The execution finishes when all Schedulable objects have finished executing. The cleanup phase allows for any resources to be reclaimed. After that, either a new mission is picked or the application halts under the command of a MissionSequencer.

2.3.3 SCJ Programming Model

Safety-critical software varies greatly in terms of complexity and requirements. The SCJ Specification defines 3 *compliance levels* that allow the developers to tailor the capabilities of the SCJ platform. The definition of each level includes the types of schedulable objects allowed, the synchronization capabilities of the infrastructure and other capabilities.

Level 0 adopts the cyclic executive model. Only Periodic Event Handlers (PEHs) are allowed. Each has a period, priority and a relative start time. The Mission computation consists of a strict ordering of their releases that is repeated throughout the lifecycle of the mission. Synchronization can be safely ignored as the PEHs execute in sequence. The methods `Object.wait()` and `Object.notify()` are not available.

Level 1 consists only of PEHs and Aperiodic Event Handlers (APEHs) that execute concurrently under the control of a fixed-priority preemptive scheduler. Access to shared objects in `MissionMemory` and `ImmortalMemory` should be synchronized in order to maintain their integrity. The methods `Object.wait()` and `Object.notify()` are not available.

Level 2 may have nested missions executing concurrently with the initial mission. PEHs, APEHs and NoHeapRealtimeThreads are allowed. Each nested mission has its own mission memory. `Object.wait()` and `Object.notify()` are allowed.

Event Handlers provide their release logic by overriding the `handleEvent()` method. They all are subclasses of the RTSJ `BoundAsyncEventHandler` class, meaning that the mapping between event handlers and server threads that execute them is a 1-to-1 static mapping. Wellings and Kim [12] provide a further insight into why event handling was adopted as the main focus of SCJ levels 0 and 1.

Sporadic Event Handlers are not supported at any level as monitoring of inter-arrival times has been deemed infeasible for safety-critical certification.

2.3.4 Concurrency and Scheduling in SCJ

The main aim of the SCJ concurrency model is to facilitate schedulability analysis of safety-critical applications and to ease the development of small and efficient SCJ applications. Furthermore, the model aims to support the transition from sequential to concurrent safety critical systems. Level 0 provides that support while Level 1 and Level 2 have more dynamic and flexible scheduling and concurrency models.

The Priority Ceiling protocol ICPP has emerged as a preferred protocol for access to shared resources on single processor systems. It has an efficient implementation and can guarantee that the application is deadlock free. SCJ only supports ICPP. The ceiling of an object can be set using the static methods in the `javax.safetycritical.Services` class.

The following apply at all compliance levels:

- Sporadic release parameters are not supported.
- Priority Ceiling is mandatory compared to RTSJ, where it is optional.
- the number of processors available shall be fixed.

- synchronized blocks are not supported.
- nested calls to synchronized methods are allowed; the ceiling priority associated with a nested synchronized method call must be greater than or equal to the ceiling priority associated with the outer call.
- synchronized code is not allowed to self-suspend while holding a monitor lock; requesting a lock (via the synchronized method) is not considered self-suspension.

Specific level restrictions:

- Level 0
  - the handlers are executed non preemptively;
  - no synchronization is required. However, it is recommended in order to facilitate portability of code between levels;
  - no deadline miss detection facility
- Level 1
  - preemptive FP Scheduling with at least 28 priority levels (hardware and software) under Priority Ceiling Emulation.
  - Deadline miss detection shall be supported; the miss shall be signalled no earlier than the deadline of the associated event handler.
- Level 2
  - Multiprocessor environment is supported; the processors are split into AffinitySets
  - Calls to the `Object.wait()` and `Object.notify()` methods are allowed. However, calling `Object.wait` from nested synchronization code is illegal.

In its public draft, the SCJ Specification still had the RTSJ concept of a Portal. However, these have fallen out of favour and are no longer included in the proceeding drafts. The reason given is that Portals require run-time support to check the allocation context of both the portal and the caller. Furthermore, since Portals are associated with every Scoped Memory, this introduces an extra overhead depending on the number of private memory areas used in the application. Sharing objects in an

SCJ application is more easily done by directly passing references to constructors during the mission initialization phase. Such references would be legal because the schedulable objects themselves live in Mission Memory. For these reasons, we do not use portals in our work although we could have.

Due to the hierarchical nature of memory areas in the SCJ, Rios et al. [13] have shown it is possible to entirely remove the need for exposing references to memory areas to the SCJ application developer. This reduces the complexity of the implementation, and this approach is welcomed from our perspective as new SCJ programmers. For a further discussion of this work, see subsection 2.4.1.

However, this means that we are no longer writing code compliant with the public SCJ Specification draft. Our implementation will have to be split — one version adhering to the public draft and being supported by the departmental Reference Implementation, and another that is more lightweight due to the more recent SCJ Specification draft.

The SCJ Reference Implementation available at the department is also a work in progress. It has several differences with the public SCJ Specification which we have to be aware of when trying to implement our work:

- The mechanism for creating events and their handlers — the original SCJ draft creates an event from a default constructor and passes a reference to the constructor of the event handlers. In our RI, this is the other way around — an event needs a reference to a handler, and the handler constructor is a default parameter-less one.

- not all Level 1 classes are supported, e. g. `AsyncLongEvent` which allows the passing of a primitive long value from event generator to event handler.

- Priority Ceiling Emulation is not supported

These all have to be taken into account during the Implementation and Testing phases.

2.4 Related Work

Given the limited number of examples in SCJ, it is worth exploring other SCJ work as it may influence our design and implementation. The related work is also useful for guiding us on other issues regarding SCJ programming.

2.4.1 SCJ Memory Patterns

Rios et al. [13] present several patterns of memory usage in SCJ. The paper evaluates the expressiveness of the Scoped Memory model used by SCJ. The main focus is on how to pass arguments and return objects between memory areas. Their approach consists of moving data between scopes through mission memory compared to mechanisms such as the Memory Tunnel pattern that are very unlikely to pass any certification such as DO-178B. We proceed with presenting the patterns that we found useful in our work.

The Loop Pattern

This pattern is concerned with the situation when during a single handler execution, a continuous block of work has to be done, e. g. by a for loop, and each of the individual cycles can benefit from temporary allocation of results. The SCJ supports this by allowing entering into a nested scope via the `enterPrivateMemory()` method.

```
class MyHandler extends PeriodicEventHandler {
  public void handleEvent() {
        Worker w = new Worker();

        for (int i = 0; i < BLOCK_SIZE; i++)
                ManagedMemory.enterPrivateMemory(256, w);
  }
}
```

Listing 2.3: The Loop patter

Two important issues are considered. One is the avoidance of illegal references. The other is the possibility of introducing memory leaks — when every iteration of the loop allocates objects inside Mission of Immortal memory areas.

Returning a Newly Allocated Object

We can improve the previous pattern by allowing upon each execution, the worker to return a result object. However, a reference to such an object living in an inner scope cannot be passed to an outer scope because the inner scope is reclaimed as soon as the method returns. Therefore, the result object needs to be allocated in the (outer) scope we plan to use the result. Creating objects in an outer scope is possible using the SCJ API:

- use `executeInArea()` — enters the referenced memory area and any new object allocation will be done there
- use `newArray()` and `newInstance()` — allocates the new objects inside the referenced memory area. Effectively the same as above but clearer. A drawback is that the object constructor called always the default constructor.

In both cases a reference to a memory area is needed. These are available using the `getCurrentMemoryArea()` method.

Runnable factory

Runnable objects reduce the readability of the code. One can fix this by using a `Factory` whose methods return `Runnable` objects.

```
class MyFactory {
  public Runnable readTemp(final int inputs, final ReturnObject rObj) {
        return new Runnable(){
                public void run() {
                  // do work here
                  // then change execution context (another runnable)
                  MemoryArea.getMemoryArea(rObj).executeInArea(
                    new Runnable(){ public void run() {
                                AuxObject auxObj = new AuxObj();
                                auxObj.temp = 50;
                                rObj.result = auxObj; }}; );
                }};
  }
}

class MyHandler extends PeriodicEventHandler {
        public void handleEvent() {

            MyFactory fact = new MyFactory();
            ReturnObjct rObj = new ReturnObject();

            ManagedMemory.enterPrivateMemory(256, fact.readTemperature(5, rObj));
            // we can now use rObj.result
        }
    }
```

Listing 2.4: The Runnable Factory pattern

This encapsulates the passing of inputs and returning of results within the call to the factory method. The factory method itself returns a `Runnable` which does the work needed, and then enters the memory area of where the result should be stored in and creates it. After the method returns, the result object can be used.

This pattern increases the expressiveness of the code, but can be a bit heavy for a new programmer to understand.

Producer/Consumer

The authors make some interesting remarks about the Producer-Consumer style of interaction between components in control systems. They note that such interaction may not always involve only primitive values, but also objects. The correct management of such objects is crucial for the integrity of the application, not only from a point of view of using shared data, but also from a memory management perspective.

Communication between SCJ application components goes through shared Mission and/or Immortal memory. Objects in those areas are not reclaimed until the end of the mission/application. The authors argue that since SCJ applications are often restricted in memory, developers must reuse objects in those memory areas, e. g. using memory pools. However, developers can safely predict how much memory their object allocations will take using the SCJ `SizeEstimator` class, which provides a conservative upper bound on the size of an object based on its class information (Section 7.3.10 of the SCJ Specification). This approach we explore further in our work.

Authors' Discussion

The authors propose a number of changes to the memory API. They note that references to memory areas are not needed at all due to the hierarchical structure of SCJ memory areas. This justifies adding a static method `executeInOuter()` and removing the non-static methods requiring a reference to a Managed Memory area together with `getCurrentManagedMemory()` and `getMemoryArea()`.

This change is semantically equivalent to the old model, but is also free from memory reference leaks and also alleviates the need for run-time checks that the current memory area is on the current thread's scope stack. The change has been accepted by the SCJ committee just before the JTRES'12 paper submission deadline, and we adopt this in our clean implementation versions.

Furthermore, libraries used by an SCJ application may indirectly allocate temporary objects, which can cause memory leaks if used in mission or immortal memory. Such libraries need to be modified in order to be scope-aware. The authors note that this is an interesting topic for future work.

Summary

The paper involves several patterns of using scoped memory, all focusing around safe and predictable memory usage in combination with extended expressiveness and quality of software code. The authors propose several API changes to the current SCJ Specification, some of which were accepted by the SCJ group and adopted in our work.

2.4.2 CDx

Zeyda et al. [14] present a refinement of a benchmark collision detection algorithm called CDx [15] to SCJ using Circus. Their implementation involves several SCJ features, namely handlers control via software events, sharing of data between handlers, and the use of control mechanisms for synchronization between handlers. The application is a SCJ Level 1.

The algorithm itself periodically reads a frame via device access to a radar, maps aircrafts to voxels (frame subdivisions), checks for collisions at each voxel and reports the number of detected collisions. Each step is represented by a handler, and control flow is regulated by events. After each handler is complete, it fires the corresponding event that releases the next handler in the chain. This means the handlers effectively execute in a sequential manner, except for the four `DetectorHandlers` that execute in parallel in response to a single `detect` event.

The control of these four concurrent handlers is of particular interest. After each has completed its release, it calls a shared `DetectorControl.notify(int id)` method, which effectively acts as a barrier. After all `Detector` handlers have completed their release, the barrier releases the next handler in the control chain. The fact that all 4 handlers respond to a single event being fired simplifies their logic.

There might be an issue with wasted computation if handlers that respond to different events need to synchronize on a barrier. This is something we explore in our work.

2.4.3 A Desktop 3D Printer

Strøm and Schoeberl [16] highlight the need of developing SCJ use cases in order to evaluate the expressiveness of SCJ, its simplicity and ease of use. They develop an SCJ-based application for controlling a robot and also provide feedback on the SCJ API from a programmer's perspective. The work presented focuses on coding the control of a desktop printer capable of creating 3D objects in plastic. In a standard setup, a computer reads a 3D drawing (e. g. made by a CAD application) and sends printing instructions to the printer's firmware, which interprets the instructions and ensures the printing head is moved to the instructed coordinates while heating and extruding the plastic (i. e. has timing requirements).

The implementation is loaded into a predictable Java processor JOP [17] that executes bytecodes as native instructions. The firmware is designed as an SCJ Level 1 application. The implementation resides on an FPGA and consists of four PeriodicEventHandlers (PEHs) as shown in Figure 2.2.

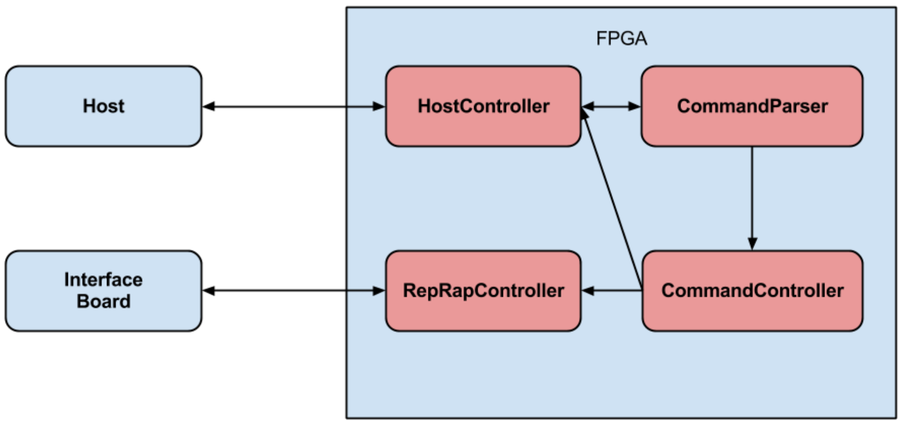

Figure 2.2: PEH communication

- `HostController` — handles serial communication with the host computer.
- `CommandParser` — polls the HostController for a command string and parses it.

- `CommandController` — takes valid commands from an object pool and enqueues them.
- `RepRapController` — handles communication with the RepRap hardware; sends commands; receives all necessary measurements like temperature and end-stop signals.

During the evaluation the authors highlight the fact that storage parameters for each handler must be known beforehand. This can be hard to obtain as size of objects is platform dependant. Furthermore, the use of library code is problematic. For example, `StringBuilder` automatically expands when an `append()` is called and the internal buffer is full; this allocation is hidden from the programmer and can cause illegal references. The feasibility analysis of the application uses a utilization test to deem all handlers schedulable. It also highlights the problems associated with analysing the timings of unbounded loops - this is circumvented by recording the loop counts as annotations to be used by the WCET Analysis tool.

The paper thoroughly follows the implementation of the RepRap firmware and its underlying implementation constraints. The authors test their implementation against an optimised C solution, showing that C has a clear advantage. The paper provides good insight into the strengths and weaknesses of SCJ for programmers that are new to the field. These can be summarized as follows:

- tools should be available to automate the process of WCET Analysis and Memory Usage Analysis
- Libraries, frameworks and platforms need to be modified so such tools can perform the analysis. This possibly includes the introduction of annotations along with the removal of unbounded loops
- programmers need to be careful when new objects are created and referenced.
- programmers need to have a deeper knowledge of library code to ensure objects are not created and referenced in an unsafe manner.

The paper also introduced the importance of using schedulability analysis at the evaluation stage because the application must be deemed feasible through static means.

2.4.4 The Cardiac Pacemaker case study

Singh et al. [18] consider the full software architecture of a cardiac pacemaker in SCJ and Ravenscar Ada. The aim is to provide a case study for developing safety-critical applications and also to provide feedback on the concurrency and timing models of SCJ as well as the API.

The algorithm for sensing and pacing the heart has complex control requirements. The authors use a verified model of a pacemaker to design the respective SCJ implementation. The implementation is a Level 1 application consisting of several missions switching between each other. There are a number of periodic and aperiodic event handlers. The authors note that SCJ has limited support for time-triggered programming compared to Ravenscar Ada because it lacks support for a timeout facility, e. g. a one-shot timer. Hence, the Ada solution requires very little run-time support, but its structure is a more complex than the SCJ implementation.

3 Design

This chapter introduces our set of concurrency examples in SCJ. Each is presented in turn along with the top-level class diagram, the concurrency mechanism involved (represented as a collaboration diagram) and our SCJ program design. We also include our design justifications which abide with the the restrictions of the SCJ Specification and also with the lessons learnt from Related Work. The actual implementation of the programs will be covered separately.

3.1 Shared Buffer

Consider the well-known Producer-Consumer problem using a shared buffer (Figure 3.1).

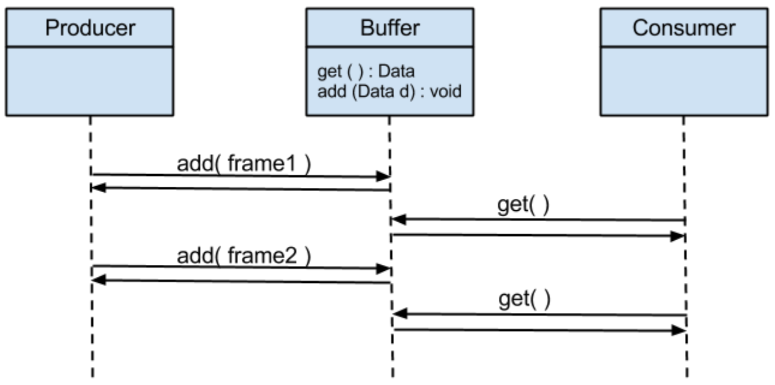

Figure 3.1: The Producer-Consumer problem

The producer writes the data in the shared buffer that is read by a consumer concurrently. Insufficient synchronization may lead to missed data, the more extensive of which can lead to security flaws. Hence, this can be modelled as a safety-critical use case.

The important issues in this scenario are:

- how to share references to shared data — the data needs to exist in a memory area where it can be accessed by all interested event handlers.
  - If the data is of primitive type, then it is placed on the stack so there is no issue here.
  - If the data is more complex, i. e. an `Object`, then it cannot be allocated in a private memory area. Rios et al. [13] discuss moving data between scopes that goes through Mission memory [1] and this is the approach we adopt. However, in section 2.4.1 it was noted that in order to avoid memory leaks, either the application must make use of object reuse via pools or to be able to make valid predictions on memory usage. The second approach does not work if the application is non-terminating or is severely constrained on memory.
- how to correctly use shared objects — correctly encapsulating shared data is a well-known issue. Some form of locking is required to ensure the data is always in consistent state. For our solution, we use **synchronized** methods for reading and writing the data in the shared buffer.

Our SCJ design is summarized in Figure 3.2. The Producer and Consumer are event handlers created during the `Mission` initialization phase. The Producer is periodically released and generates the data that needs to be shared[2]. The data is written in MissionMemory and is referenced by the `Buffer` (also residing in MissionMemory). The `Producer` then triggers an event to release the `Consumer`, which accesses the data stored in the `Buffer`.

The top-level collaboration diagram is shown in Figure 3.3. Note that the event handling paradigm ensures a *happens-before* relationship between the generation of the data by the Producer and its reading by the Consumer.

[1] The authors also note that copying from one private memory to another private memory, e. g. the RTSJ Handoff design pattern [7, §3.6], breaks the reference assignment rules.

[2] The actual mechanism for generating this data is out of the scope of this project. We envision it can involve some sort of device access depending on the actual application.

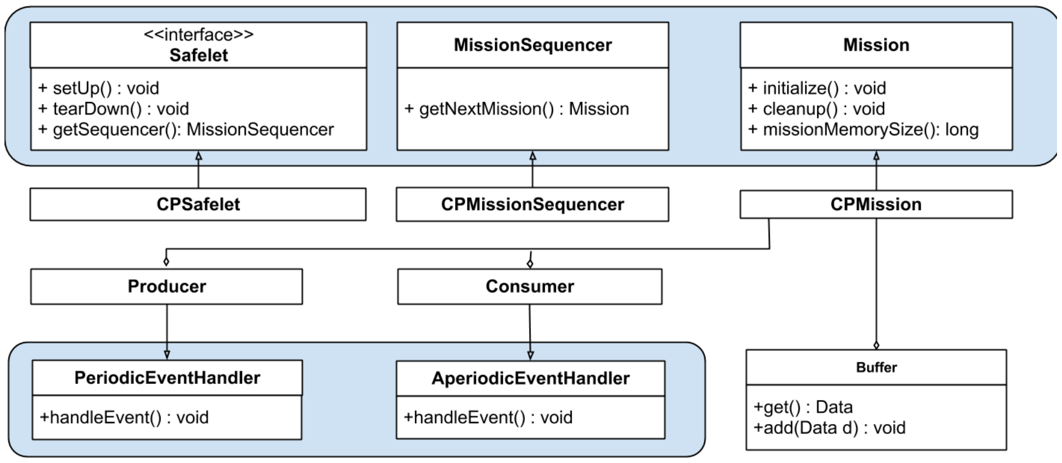

Figure 3.2: SCJ Producer-Consumer class diagram. Highlighted sections are part of the SCJ Specification

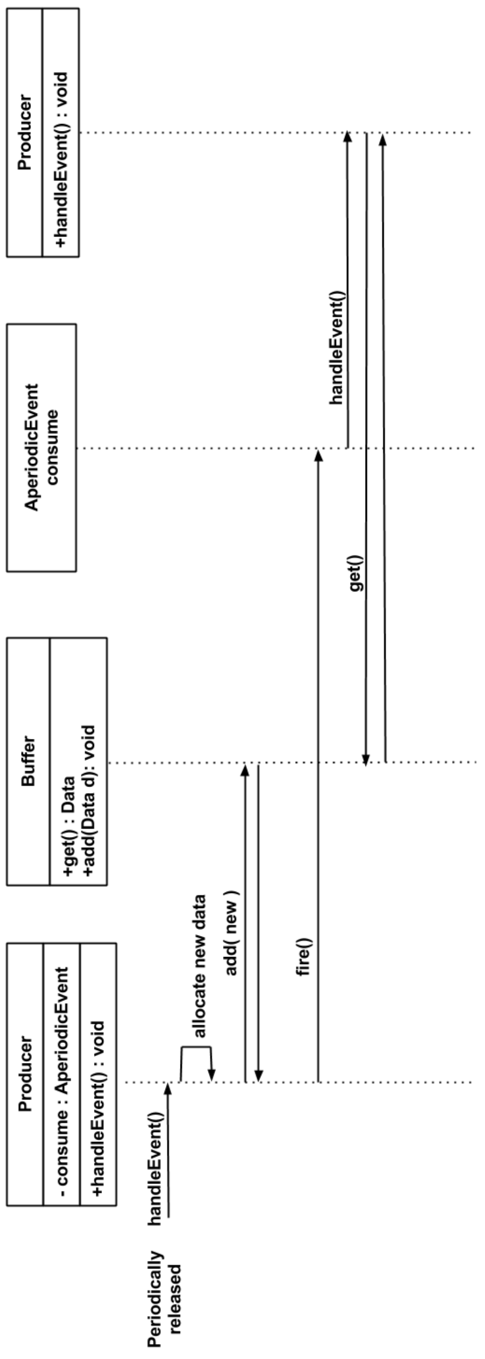

Figure 3.3: SCJ Producer-Consumer collaboration diagram

3.2 Barrier

Consider a scenario where two independent entities must wait for each other in order to progress. As a simple real-world example, take a missile launcher that controls the firing of a missile — two independent threads of control must communicate a fire action to a common controller. This can be done using a barrier, as shown in Wellings [5, §8.7] (Figure 3.4). The two entities concurrently notify the Barrier that they want to fire the

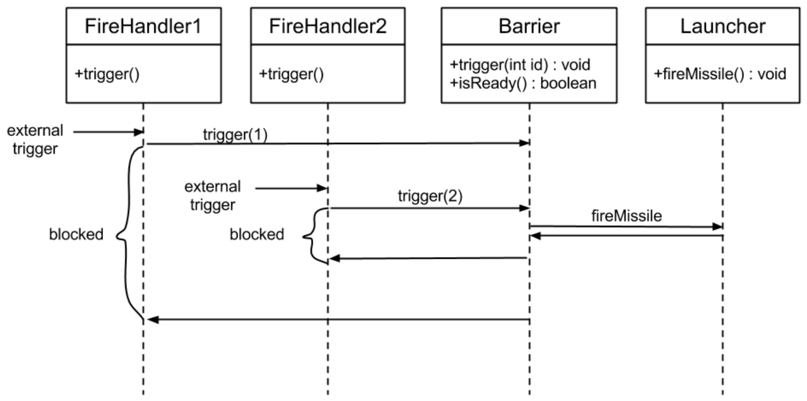

Figure 3.4: Top-level Missile Launcher class diagram

missile. The Barrier then suspends each caller until both entities have been suspended. Then the Barrier wakes them so they can proceed with the missile launch [3]

The SCJ design components are shown in Figure 3.5. It follows the event-handling model of Level 1. The two separate handlers FireHandler1 and FireHandler2 are triggered by the occurrences of external events, e. g. pressing a button[4]. During its release, each handler notifies the Barrier that it has been triggered, but the handlers are not suspended and can be fired again, e. g. if one of the buttons is pressed multiple times. When the Barrier detects that both handlers have executed, the logic for firing the missile is launched.

[3]However, this is not the focus of the RTSJ solution; it stops at the point at which both entities are woken up — where the launch logic resides has been left for the reader to decide.

[4]Since the nature of our work does not involve specific hardware, we represent the buttons as Periodic Event Handlers in the Implementation. This decision also supports the testing phase where we check the outputs produced.

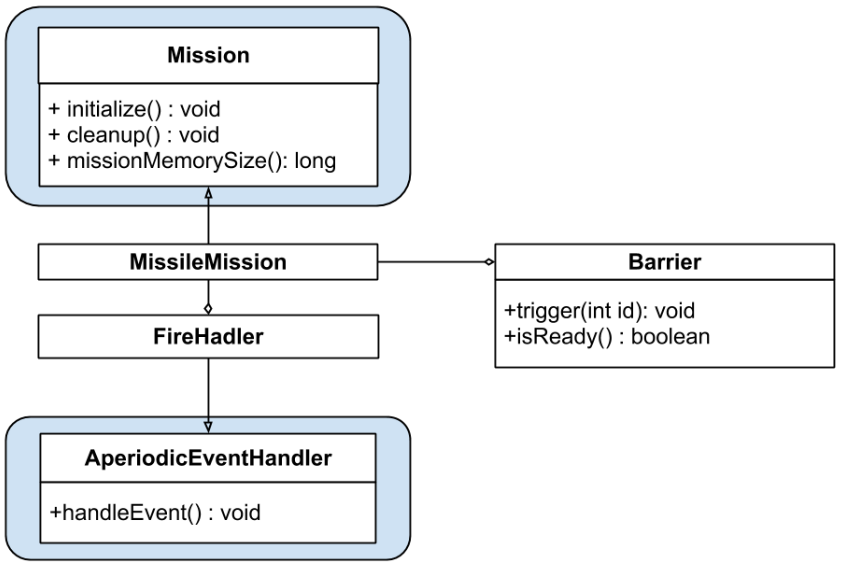

Figure 3.5: SCJ Barrier class diagram

We note that it is not needed to include an extra handler that holds the logic for firing the missile — each FireHandler can check if the barrier is ready after it has sent a notification for its current release and then proceed to reset the barrier and fire the missile. However, this adds time to the longest path through the handler code, which has implications for its worst-case execution time (WCET). Therefore, we chose the Barrier to be responsible for triggering the missile launch handler.

The collaboration between the components can be seen in Figure 3.6.

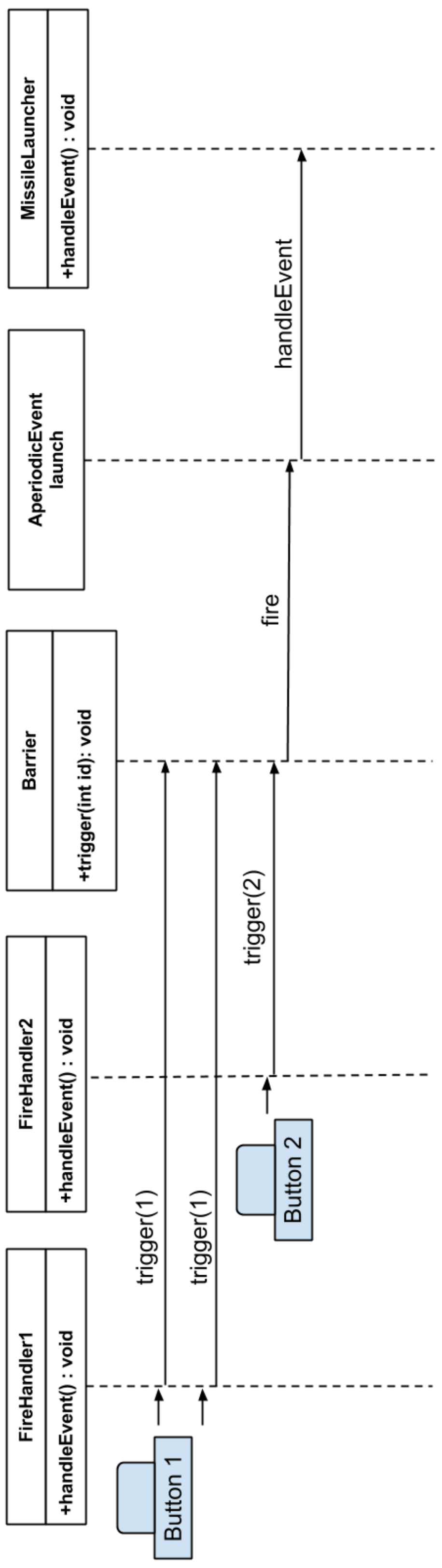

Figure 3.6: SCJ Missile Launcher collaboration diagram

3.3 Persistent Signal

Consider a scenario where a thread of control wants to delegate some work to another decoupled thread, e. g. to a Logger that provides asynchronous output [5, §5.2] (Figure 3.7). The calling thread is not blocked

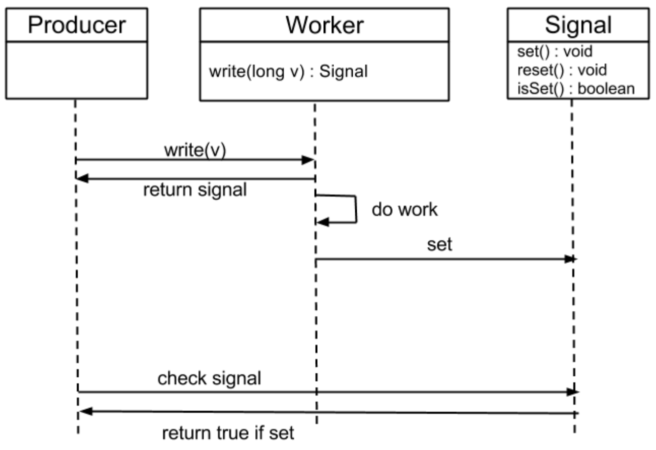

Figure 3.7: Asynchronous delegation using Persistent Signal

when it triggers the worker, but needs to check if the work has been done later on. Upon signalling, the worker queues up the work and returns a reference to a shared persistent signal. The original thread can proceed as soon as the procedure returns and later check the signal to see if the work has been completed. When the worker completes, it sets the associated signal. Inside the worker, the output may be queued and not completed for some time.

The components in our SCJ design are shown in Figure 3.8.
The application is a single Mission with Producer and Worker event handlers. The Signal is shared and can be set and reset. The Producer can pass some primitive data to the Worker instead of a String , e. g. a `long` value to be shared via the `AsyncLongEvent` hierarchy defined the SCJ Specification [5]. We change the Worker to not return a reference to the Persistent Signal because there is no SCJ mechanism for this. Instead, we can simply share the reference during object construction in the Mission

[5]A `LongEvent` is the same as an ordinary Event except that it carries a primitive long value assigned when the event is released. That value can be accessed by the `handleLongAsyncEvent(long value)` method in the handler class.

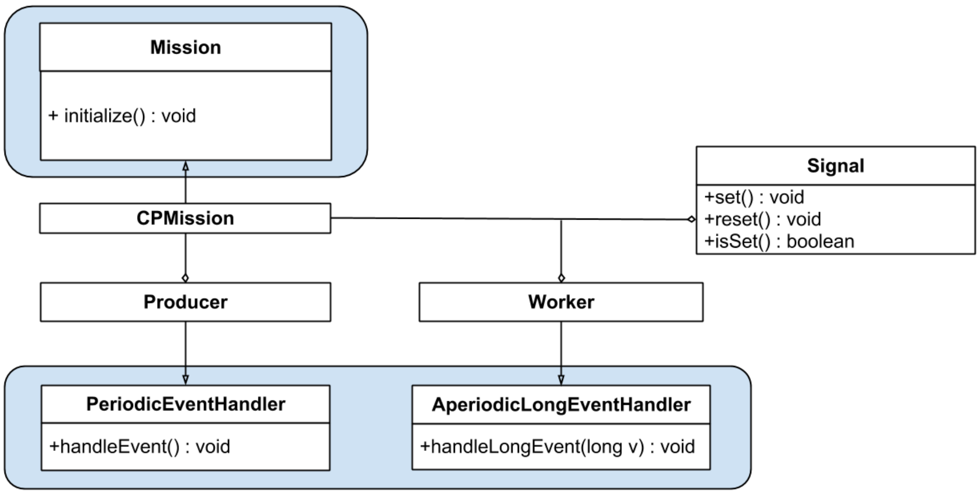

Figure 3.8: The SCJ Signal class diagram

initialization phase.

The collaboration between the components is shown in Figure 3.9.

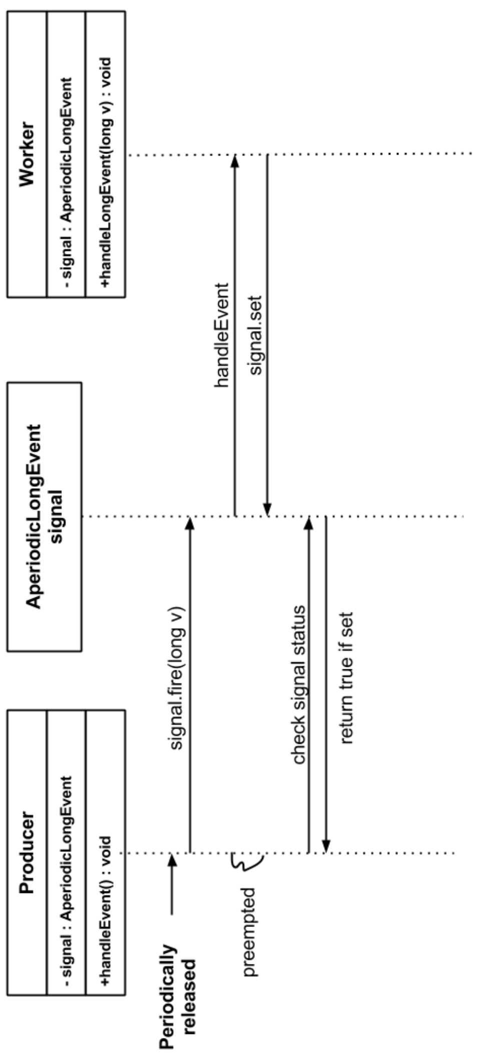

Figure 3.9: SCJ Signal collaboration diagram

4 Implementation

In this section, we present our implementation for each of the use cases.

4.1 Buffer

As discussed in section 3.1, this scenario involves two event handlers, a Producer and a Consumer, that share data via a bounded buffer. The class components of the implementation are:

- Buffer — a shared object living in MissionMemory. It buffers references to objects produced by the Producer handler.
- Producer — a Periodic Event Handler that creates objects in MissionMemory that are to be shared. Must first enter MissionMemory in order to allocate a new object.
- Consumer — reads references to shared objects via the Buffer.

The collaboration between the components was shown in Figure 3.3.

4.1.1 Mission

The application consists of a single mission. The main body of interest is the initialization phase of the mission:

```
    protected void initialize() {
            System.out.println("Initializing␣main␣mission");

            Buffer buffer = new SimpleBuffer();
            Consumer consumer = new Consumer(buffer);

            /* The event consume releases Consumer. */
            AperiodicEvent consume = new AperiodicEvent(consumer);
            new Producer(consume, buffer);
    }
```

Listing 4.1: Mission initialization

The initialize method is executed by the SCJ infrastructure within the MissionMemory for the application. Here, the shared buffer is created, the two handlers Consumer and Producer, and the aperiodic event that is triggered by the Consumer when a new data item is ready. Note that PEHs and APEHs register themselves with the Mission automatically in our Reference Implementation, so we don't need to explicitly call `register()` in our runnable version of the code[1]. A reference to the shared buffer is passed to both handlers, and such a reference is legally stored because the objects representing the handlers live in `MissionMemory` even though they execute in their own private `ScopedMemory`.

4.1.2 Buffer

The buffer implementation is very simple:

```
public class BoundedBuffer implements Buffer {

    private int first;
    private int last;
    private int stored;
    private int max = 5;
    private Object[] data;

    public BoundedBuffer() {
        this.data = new Object[ this.max ];
        this.first = 0;
        this.last = 0;
        this.stored = 0;
    }

    public synchronized void put(Object item) {
        // check if buffer is not full
        // Do nothing if we are already full
        if ( this.stored == this.max ) return;
        this.last = ( this.last + 1 ) % this.max;
        this.stored++;
```

[1]The SCJ Specification requires all `ManagedSchedulables` to be registered explicitly.

```
		this.data[last] = item;
	}

	public synchronized Object get() {
		// check if empty
		if ( this.stored == 0 ) return null;
		this.first = (this.first + 1) % this.max;
		this.stored−−;
		return this.data[first];
	}

	public synchronized boolean isFull() { return this.stored == this.max; }
	}
```

Listing 4.2: Buffer implementation

It implements the Buffer interface, which would allow us to later come back and use a different buffer implementation if it was needed. It provides a clear abstraction of the data model we use. The actual buffer is a circular bounded buffer with a predefined maximum capacity. Its methods are synchronized to ensure access under mutual exclusion. The first and last indecies keep track of the positions of the next reference to be read or written.

4.1.3 Producer

The Producer is a Periodic Event Handler. A reason for this is that the SCJ infrastructure invokes the `handleEvent()` method based on an internal real-time clock, compared to the mechanism for Aperiodic Event Handlers that involves the firing of an event (either application-defined or mapped to an external *happening*, which is the RTSJ representation of external events). The Producer stores references to the shared Buffer and also to the event that is triggered when new data is produced.

The actual logic does several things (Listing 4.3).

- check if more objects can be allocated.

If it cannot, then the current release is prematurely abandoned with a return statement [2].

- allocate the new object by using the `static ManagedMemory.executeInOuter()`. The actual logic passed as a parameter has been preallocated [3], and involves calling **new** and saving the reference.
- saves a reference to the new object in the shared buffer.
- fires the `consume` event to trigger the Consumer handler.

```
public void handleEvent() {
        /*
         * Limit the creation of new objects
         * Avoids running out of Mission Memory
         */
        if (NUM_OF_OBJECTS <= MAX_NUM_OF_OBJECTS && !buffer.isFull())
        {
                /*
                 * Allocate new data object and update count
                 */
                this.data = ManagedMemory.executeInOuter(this._switch);

                NUM_OF_OBJECTS++;

                /*
                 * Store a reference to the new object in the buffer
                 */
                this.buffer.put(data);

                /*
                 * Trigger the Consumer handler
                 */
                this.e.fire();
        }
}
```

[2]Logically, all subsequent allocations also cannot occur because objects in Mission-Memory are not deallocated before the end of the Mission. The SCJ Specification does not have a mechanism for deregistering schedulable objects.

[3]for benefits of object reuse compared to creating temporary objects during execution

Listing 4.3: Producer logic in Producer.java

Note that the buffer only stores a reference to an object in Mission Memory, which is legal according to the reference assignment rules.

4.1.4 Consumer

The Consumer is less complicated than the Producer. It only keeps a reference to the shared buffer.

```
    public void handleEvent() {
        /*
         * Get a reference to the new object
         */
        Object data = buffer.get();

        /*
         * Confirm we can use the object
         */
        System.out.println(data.toString());
    }
```

Listing 4.4: Consumer.java, handleEvent()

During execution, the Consumer obtains a reference to the next object to use. This reference is safe to use for the duration of the current release because it points to an object in MissionMemory (an outer scope) [4].

A clear advantage of this implementation approach is allowing the developer to determine how much memory is to be used dynamically to pass objects from one entity to another.

However, due to objects not being deallocated in Mission Memory, there can only be a finite number of them. Developers need to ensure this at compile-time, either by limiting the number of objects that can be created and using that limit to estimate the memory requirements of their program or to use a pool of preallocated objects which we already discussed in section 2.4.1.

[4]there is nothing stopping the reference being stored inside the Consumer if the developer wishes to do so.

4.2 Barrier

Our second use case involves the synchronization of two independent event handlers using a barrier. The components involved were shown in Figure 3.6. Each we present in turn.

4.2.1 Mission

The application consists of a single Mission. During the initialization phase, all the events and their handlers are created and registered.

```
	public void initialize() {
		System.out.println("Initializing␣main␣mission");

		/* The launch event triggers the Launch handler. */
		AperiodicEvent launch = new AperiodicEvent();

		/*
		 * Create Launch AEH
		 * Pass a reference to the shared barrier
		 * ManagedHandlers need to register themselves upon creation
		 */
		(new LaunchHandler(launch)).register();

		/* Create a barrier for 2 handlers,
		 * Triggers launch event when ready to proceed
		 */
		Barrier barrier = new Barrier(2, launch);

		/* The fire1 and fire2 events release fire1Handler and fire2Handler. */
		AperiodicEvent fire1 = new AperiodicEvent();
		AperiodicEvent fire2 = new AperiodicEvent();

		/*
		 * Create Fire1 and Fire2 AEH
		 * Pass a reference to the shared barrier
		 * ManagedHandlers need to register themselves upon creation
		 */
		(new FireHandler(fire1, barrier, 0)).register();
		(new FireHandler(fire2, barrier, 1)).register();

```

```
            /*
             * Create PEHs that generate event occurrences.
             */
            (new Button(fire1, 2000, 0)).register(); //2s
            (new Button(fire2, 9000, 9000)).register(); //9s + 9s offset
    }
```

Listing 4.5: Initialization of Missile Application

Each handler gets a reference to its triggering event as a constructor parameter because subscribing to an event is done during object creation [5].

A note should be taken on the special `Button` objects. Realistically the application would do some form of device access and receive external event occurrences from the underlying infrastructure depending on its requirements. The SCJ specification represents external events by the `Happening` class hierarchy. However, if we decide to include this in our implementation, then it would take the focus away from what this use case is meant to present, which is the concurrency aspect of using a barrier. Furthermore, device access and happenings are out of the scope for our work, so we instead use Periodic Event Handlers which are triggered by the SCJ infrastructure. This decision simplifies detection of a 'button pressed' event.

4.2.2 Barrier

The Barrier class controls the synchronization of the specified handlers. The event to be fired when all the handlers have triggered the barrier is passed as a parameter to the constructor and saved as a field. The internal state of the barrier is represented by an array of booleans, each corresponding to each unique handler integer id. The SCJ specification allows collections only at Level 2, so we have to resort to arrays instead of using for example a HashMap to store handler–flag mappings.

Each of the barrier methods are self-explanatory and include triggering the barrier and checking if the barrier is triggered. When all the handlers have triggered the barrier, it internally fires the associated event and resets its state. The **synchronized** keyword is used to protect the object state. The ceiling priority is set to the FireHandler priority in accordance with the mandatory Priority Ceiling Emulation protocol.

[5] In the Reference Implementation, this is the other way around, the event needs a reference to the handler instead

4.2.3 FireHandler

The FireHandlers are responsible for responding to a button being pressed.

```
	public void handleEvent() {
		System.out.println("** 'fire2' event **");

		/*
		 * If we have already triggered the barrier,
		 * do not re-trigger
		 */
		if (barrier.isAlreadyTriggered(this.id)) return;

		barrier.trigger(this.id);
	}
```

Listing 4.6: FireHandler.java, handleEvent

Each handler is subscribed to its own event during Mission initialization. When the event is fired, the `handleEvent()` logic first checks if it has already triggered the barrier. If it has not, it triggers it.

This check reduces the amount of computation to be done at each release if one of the events is fired multiple times. This does not reduce the WCET, but it does help reduce the average case.

4.2.4 LauncherHandler

The LaunchHandler is triggered by the barrier when all fire handlers have executed. In a real-world application, it would perform some sort of hardware access to do the actual launching. As mentioned before, we do not delve into that area of the SCJ specification.

4.3 Persistent Signal

Our persistent signal use case presents a way for a handler to offset some of its computation to another worker handler and receive feedback for when the work is complete. This can be done using a Persistent Signal that is state-aware.

4.3.1 Application

The application consists of a single Mission.

```
	protected void initialize() {
		System.out.println("Initializing␣main␣mission");

		/*
		* Signal is an AperiodicEvent with a state
		* used for backwards propagation of information
		* between the Worker and Producer
		*/
		PersistentSignal signal = new PersistentSignal();

		/*
		* Creates Worker APEH
		* Pass a reference to the triggering event
		* ManagedHandlers need to register themselves upon creation
		*/
		(new Worker(signal)).register();

		/*
		* Cr eater Producer PEH
		* Pass a reference to the event to be triggered
		* ManagedHandlers need to register themselves upon creation
		*/
		(new Producer(signal, 2000, 0)).register();
	}
```

Listing 4.7: Mission initialization phase

Both the Producer and the Worker share the signal object. The Producer triggers the Worker by firing the signal. The Worker itself will do the computation it needs to do in its handleEvent() method and *set* the signal.

The Producer will then be able to check if the work has been done by observing the state of the signal.

4.3.2 Persistent Signal

The Persistent Signal is an event with a boolean state flag. It can be either set or reset. It follows the guidelines of clear abstraction and separation of logic as outlined for the previous shared objects (subsection 4.1.2)

```
public class PersistentSignal extends AperiodicEvent {

        /*
         * Records the internal state of the signal
         */
        private boolean _set;

        public PersistentSignal(){
                super();

                /*
                 * Set the ceiling priority for this shared object
                 * used by Priority Ceiling Emulation protocol
                 * Worker is at max priority
                 */
                Services.setCeiling(this,
                                PriorityScheduler.instance().getMaxPriority());

                this._set = false;
        }

        /**
         * Resets the state of the signal
         */
        public synchronized void reset()
        {
                this._set = false;
        }

        /**
         * Sets the state of the signal
```

```
                 */
                public synchronized void set() {
                        this._set = true;
                }

                /**
                 * Observes the state of the signal
                 * @return true if the signal is set
                 */
                public synchronized boolean isSet()
                {
                        return this._set;
                }
        }
```

Listing 4.8: Persistent Signal

The class extends AperiodicEvent. This allows us to have a single object shared between both Producer and Worker. An alternative would be to have separate objects - one AperiodicEvent instance and a separate Signal instance that is basically a wrapper for a boolean. However, since the logic for the signal is so small and the original RTSJ case corresponds to the event handling paradigm, we chose to first approach and extend AperiodicEvent.

The ceiling of the object is set as the maximum of the Producer and Worker and its methods are properly synchronized . However, priority values are something we explore further in our Testing. Due to the nature of the PCE protocol, unless the Producer and Worker share objects with other handlers having higher priority, then neither of them can preempt each other as they would inherit the ceiling priority of the signal.

The **synchornized** keyword is used to protect methods that access the state of the signal. This ensures that the data is always consistent. The monitor lock is obtain with accordance with the PCE protocol as with the previous protected objects presented.

4.3.3 Producer

The Producer class involves periodically triggering the Worker and then checking if the work has been completed.

```
        public void handleEvent() {
```

```

        /* reset signal at each release */

        this._event.reset();
        this._event.fire();

        /* do some computation */
        System.out.println("+ Producer - starting  computation ");
        for (int i = 0; i < 100; i++) i+=i;
        System.out.println("+ Producer - finishing computation ");

        /* check if output is done */
        if (this._event.isSet()) System.out.println("+ Producer - output done\n");
        else System.out.println("+ Producer - output not done yet");

    }
```

Listing 4.9: Producer.java

The Producer implementation has placeholder work it performs between triggering the event and checking it. This does not strictly need to be included, but we show it for completeness. This proves helpful later in the Testing stage. As mentioned earlier, neither handlers should preempt each other due to Priority Ceiling Emulation. However, we see that to not be the case when we test the output in the Testing section; it turns out to be a problem with the Reference Implementation.

4.3.4 Worker

The Worker code seems quite short. We again cannot be sure what sort of computation a real-world application would require, so we instead insert a placeholder of incrementing a private counter. At the end, the shared signal is set to let the Producer know that the work has been completed.

```
    public void handleEvent() {
        /* do work */
        this._iteration++;
        System.out.println("@ Logger - output iteration: " + this._iteration);

        /* Work done, set signal */
        this._signal.set();
        }
```

Listing 4.10: Mission initialization phase

5 Testing

As example use cases, it is important that our implementations are robust and conform to the SCJ Specification. Due to the nature of concurrency and predictable scheduling in SCJ, the best way to test our programs is to insert print statements and observe the output over multiple runs. Our test results are shown in Appendix A.

5.1 Buffer

During Mission initialization, we first check the current memory area. This should give us the name of the Mission Memory of the mission which proves to be useful later on. To obtain this information, we use the `static javax.realtime.RealtimeThread.getCurrentMemoryArea()`[1], which returns a reference to the current area we are executing in. Since the Reference Implementation builds on top of a RTSJ JVM [19], the `toString()` method shows the type of the memory area is LTMemory, which is observed in the output.

At the beginning of each `handleEvent()` method, we output a message with the name of the handler. This ensures that we have registered the appropriate handlers with the infrastructure and that they are being triggered in accordance with event firings.

In the Producer, we check the memory area of the newly created object. This has to be different from the private memory area as the object is to be shared. The output shows this is the Mission Memory, which is consistent with the way the object was allocated.

In the Consumer, we check the name of the object being obtained through the buffer and observe that after each handler execution different names are returned. This is consistent with the fact that we create several different objects.

[1]This method is allowed at Level 1 and above.

5.2 Barrier

This test involves tracking the program output over a number of `FireHandler` executions.

As before, we add an output statement at the start of a `handleEvent()` method invocation showing that the event handlers are registered with the SCJ infrastructure. This output also confirms that the auxiliary Button periodic handlers are being released as they trigger the FireHandlers.

We also add output depending on whether the fire handler triggered the barrier or not. If the barrier has already been triggered, the output confirms this and the logic returns prematurely from the method. This directly corresponds to the original RTSJ design involving threads that are being suspended while waiting on the barrier.

The timing conditions are as specified at handler creation and are as follows:

- FireHandler1 - executes every 2 seconds
- FireHandler2 - executes every 9 seconds with a 9 seconds offset

These ensures that the `handler1` event is fired multiple times before the first firing of `handler2`. The first firing of `handler2` is offset relative to the start of the application.

5.3 Persistent Signal

For this output, we place print statements on several positions:

- at the start of each `handleEvent()` method
- Producer — between the start and end of the extra dummy computation performed between the firing of the signal event and the check for the signal flag. We inserted this extra dummy computation to try and see if timing has an effect on the output.
- Worker — before we set the signal to represent the extra work being complete.

If one can recall, the SCJ Specification makes Priority Ceiling Emulation mandatory when using shared objects. This involves setting the *ceiling* of the resource using the `static Services.setCeiling(Object o, int ceiling)`. However, our Reference Implementation does not support this and defaults to the RTSJ Priority Inheritance [2]. Therefore, handler priorities will have an impact on the program output so we devise several tests to explore this:

- both handlers having the same priority
- the Worker having a higher priority than the Producer
- the Producer having a higher priority than the Worker

The test results can be seen in section A.3.

In the first test, no priority changes should happen during execution. However, there is random interleaving between the two handlers. This has been confirmed to be a problem in the Reference Implementation — once a release starts, it should finish. This highlights one of the challenges with using incomplete or unverified implementations.

In the second test, the Worker always preempts the Producer. This is consistent with the Priority Inheritance Protocol. Furthermore, it should not happen under PCE because the ceiling of the shared object would be set to the Producer priority, and therefore, the Worker would also be executing at the higher priority level.

In the last test, the Producer should never get preempted by the Worker because it is running at a higher priority. This is not observed — there is some interference, which is again attributed to the bug in the Reference Implementation.

[2]This was discovered when trying to run code that uses the `setCeiling()` method.

6 Evaluation

In this chapter we evaluate our work with respect to the project objectives outlined in section 1.2, more specifically:

- Evaluate our work as a software product — lines of code, bytecode size, use of good software practices
- Evaluate the adequate ability of SCJ as a language and programming paradigm from our experience.

6.1 Software measurements

In this section, we evaluate our work as a software end product. We use an open-source tool called Cloc [20], which provides statistics of application source code (Table 6.1).

Package	**Blanks**	**Comments**	**Code**	**LOC**	**Bytecode size**
Buffer-clean	66	70	184	320	
Buffer-runnable	63	75	178	316	8954B
Barrier-clean	56	72	162	290	
Barrier-runnable	54	65	167	286	7823B
Signal-clean	54	53	127	234	
Signal-runnable	54	47	123	224	5935B

Table 6.1: Statistics per package.

The tool separates the code into blanks, comments and code. The Lines of Code statistic is the sum of the previous. For bytecode size, we report the memory footprint in bytes of the executables produced by the Reference Implementation's SCJ compiler. None of the statistics include the code of the Reference Implementation. This is comparable to the size of other bytecode SCJ use cases [21, §5].

6.2 SCJ evaluation

The SCJ language specification is publicly available under JSR-302 [11]. In this section, we provide insights into SCJ that we consider useful to new SCJ programmers.

Documentation

The Reference Implementation available at the department is still a work in progress. Our work has provided some feedback during the testing phase where some problems were discovered. The lack of an official SCJ Reference implementation is also a concern. Furthermore, the SCJ Specification is a work in progress itself.

This highlights several points of interest. First, different versions of the specification might not be backward compatible with each other. This implies non-trivial complications if an application is to upgraded to a later version of the language. An example of this is the use of Portals — they were supported in the original public draft but later on removed.

Second, tools used for automated analysis, e. g. WCET Analysis or Memory Safety Analysis, might also require certification [21, §7] if their evidence is to be used for certification purposes. Such tools, and their supporting evidence, may also change drastically if the underlying SCJ Specification is not backward compatible.

Furthermore, the Reference Implementation used has differences with the official SCJ specification. This has introduced different versions of the code — one compliant with the specification and one compliant with the implementation (subsection 2.3.4). This introduces severe code duplication, a bad engineering practice.

Compliance levels

The SCJ Specification introduces compliance levels to support different complexity of language features. They are supported by the SCJ meta-data annotations and allows for the restriction of SCJ capabilities available to the developer, which complies with existing safety-critical engineering practises [22, §4]. Disabling such features is useful in achieving a cheaper, easier to develop and less complex software end product and certification evidence.

Furthermore, the SCJ Specification is thoroughly annotated with respect to compliance levels, which is another useful feature of the documentation.

Memory usage

SCJ adopts the RTSJ Scoped Memory model with a modification to make it hierarchical in nature. This simplifies complexity of code and underlying implementation (section 2.4.1). However, this paradigm is quite different to standard Java, which is probably a concern for new developers.

Furthermore, this memory model makes it non-trivial and complex to use scoped memory for temporary computation. Results from the computation need to be stored in an area where they can be used by the computation caller, otherwise they get reclaimed by the JVM upon exiting the temporary scoped memory.

Lastly, code modification poses a concern. Not only do the timings of the application change as source code evolves, but so does memory usage. This has direct impact on the memory requirements when the application is to execute in a limited memory environment.

Event handling

SCJ Level 1 only supports asynchronous events and their handlers. This includes periodic and aperiodic activities, but not sporadic as monitoring of inter-arrival time is not supported. However, nowhere in our work did we encounter the need for sporadics, so commenting on their use is inappropriate.

Event-based programming attempts to solve some of the issues associated with threads [12, §2.2] — it involves simpler communication and synchronization, less complexity when it comes to implementation, shorter and more maintainable code base. However, it is not always clear how to translate between event- and thread-based programs. For example, some of our original concurrency mechanisms were presented in the context of threads, so we had to extract the their concurrent nature and turn them into event-based programs. This can be difficult for programmers new to SCJ.

Furthermore, the absence of threads relieves some of the concerns with underlying thread management, e. g. starting and stopping. Periodic computation is better represented using event handlers compared to threads that make use of self-suspension (calls to Thread.sleep()).

Scheduling

Level 1 applications use a fixed-priority preemptive scheduler with Priority Ceiling Emulation protocol.

The focus of scheduling in SCJ is predictability. Priorities are assigned during object initialization by passing arguments to the super constructors. However, PCE dictates the existence of dynamic priorities based on shared object ceilings to avoid unbounded priority inversion and deadlock. The implications of the protocol to scheduling are not always obvious to new programmers. A lower priority entity may release a higher priority one, but preemption will not happen if both share an object. That can be seen in our testing of the persistent signal in section 5.3.

As thread priorities are only a guidance to the JVM, Java SE programmers will need to adjust to the use of the protocol. However, it ensures that applications are deadlock-free which is a justifiable trade-off.

Scope-aware libraries

The use of external libraries can be problematic [13, §5.2] as they may internally allocate temporary objects (`StringBuiler`) or rely on the use of garbage collection (`java.util.Collections`). These can potentially cause memory leaks or break the reference assignment rules. This is just another issue of which SCJ programmers should be aware.

6.3 Limitations

Our work has achieved the desired level of concurrency and although our solutions are not black-and-white, there exist some limitations.

Memory and WCET Analysis

The availability of automated tools that analyze memory usage is still a work in progress. Dalsgaard et al. [21] develop a prototype implementation for memory analysis that is not based on SCJ annotations. However, their library is based on the JOP implementation. We are unaware of any similar tool for our Reference Implementation.

WCET Analysis is as equally important for an application whose timely response is also a requirement. Tools exist that automate the process on method level [23], allowing for fine-grained analysis of the application code. As above, doing this on our Reference Implementation is impossible at present, which is another limitation for our work.

6.4 Summary

SCJ Level 1 has provided a good starting point for investigating concurrent programming for safety-critical systems in Java. It has a rich enough API and comprehensive documentation. Its scoped memory model is daunting at first sight, but leads to more structured abstraction of memory than the RTSJ. However, the Specification and the Reference Implementations are still a work in progress. SCJ programmers also have to be aware of the underlying libraries and platforms used for the development of the application as they can be either not scope aware and create the possibility of memory leaks or provide an API that is different from the official public draft, which has implications for the (re-)certification of the developed product.

7 Conclusions and Future Work

Writing concurrent programs in SCJ has provided suitable insight into real-time and high-integrity programming with strict memory and timing requirements while aiming to develop code compliant with good software engineering practices. We have attempted to provide discussions on real-world problems that are non-trivial for safety-critical systems, namely sharing of non-primitive data, synchronization using a barrier and a signal mechanism for communication between handlers.

We combined the best aspects of previous work done in the field with our own ideas. Being able to provide feedback on the ability of SCJ as a concurrent language from the point of view of a junior programmer in the field has been extremely rewarding.

We hope our work will aspire further development into SCJ and its concurrency uses.

Future Work

There are a number of possible areas that can be investigated as future work:

- application of automated tools for memory and WCET analysis — such analysis is mandatory for certification, so we hope that technology evolution will prove useful in this aspect

- there can be other concurrency mechanisms that may be possible to develop in SCJ that we lacked the time to do; a broadcast and a transient signal seem like good candidates.

- SCJ Level 2 allows the mixing of event- and thread- based programming. A future development might be to extend our work to use level 2 features or even translate the implementation to a more intuitive model that is inline with the original context of the concurrency mechanisms we picked.

Bibliography

[1] J. Bloch, *Effective Java*, 2nd ed. Prentice Hall PTR, 2008.

[2] D. Bacon. (2001, Feb.) Realtime garbage collection. [Online]. Available: http://queue.acm.org/detail.cfm?id=1217268

[3] J. Gosling and H. McGilton. (1996, May) The java language environment: A white paper. [Online]. Available: http://www.oracle.com/technetwork/java/langenv-140151.html

[4] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

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

[6] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*, 4th ed. Addison-Wesley Educational Publishers Inc, 2009.

[7] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek, "Real-time java scoped memory: design patterns and semantics," in *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2004, pp. 101–110. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1300335

[8] J. Ousterhout, "Why threads are a bad idea (for most purposes)," in *USENIX Winter Technical Conference*, Jan. 1996. [Online]. Available: http://www.cs.utah.edu/~regehr/research/ouster.pdf

[9] R. van Renesse, "Goal-oriented programming, or composition using events, or threads considered harmful," in *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, ser. EW 8. ACM, 1998, pp. 82–87. [Online]. Available: http://doi.acm.org/10.1145/319195.319208

[10] RTCA, *Software considerations in airborne systems and equipment certification*. DO-178B, RTCA, 1992.

[11] 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. (2011) Safety critical java technology specification. [Online]. Available: http://jcp.org/en/jsr/detail?id=302

[12] A. Wellings and M. Kim, "Asynchronous event handling and safety critical java," in *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '10. New York, NY, USA: ACM, 2010, pp. 53–62. [Online]. Available: http://doi.acm.org/10.1145/1850771.1850778

[13] J. R. Rios, K. Nilsen, and M. Schoeberl, "Patterns for safety-critical java memory usage," in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '12. ACM, 2012, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2388936.2388938

[14] F. Zeyda, A. Cavalcanti, A. Wellings, J. Woodcock, and K. Wei, "Refinement of the Parallel CDx," University of York, Department of Computer Science, York, UK, Tech. Rep., 2012.

[15] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek, "Cdx: a family of real-time java benchmarks," in *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '09. ACM, 2009, pp. 41–50. [Online]. Available: http://doi.acm.org/10.1145/1620405.1620412

[16] T. B. Strøm and M. Schoeberl, "A desktop 3d printer in safety-critical java," in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '12. ACM, 2012, pp. 72–79. [Online]. Available: http://doi.acm.org/10.1145/2388936.2388949

[17] M. Schoeberl, "A java processor architecture for embedded real-time systems," *J. Syst. Archit.*, vol. 54, no. 1-2, pp. 265–286, Jan. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2007.06.001

[18] N. K. Singh, A. Wellings, and A. Cavalcanti, "The cardiac pacemaker case study and its implementation in safety-critical java and ravenscar ada," in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '12. ACM, 2012, pp. 62–71. [Online]. Available: http://doi.acm.org/10.1145/2388936.2388948

[19] Jamaicavm. [Online]. Available: http://www.aicas.com/jamaica.html

[20] A. Danial. Cloc - count lines of code. [Online]. Available: http://cloc.sourceforge.net/

[21] A. E. Dalsgaard, R. R. Hansen, and M. Schoeberl, "Private memory allocation analysis for safety-critical java," in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '12. ACM, 2012, pp. 9–17. [Online]. Available: http://doi.acm.org/10.1145/2388936.2388939

[22] J. F. Ruiz. Mission-critical on-board software using the ada 95 ravenscar proïňĄle. [Online]. Available: http://www.adacore.com/uploads/technical-papers/GNAT_Ravenscar_ERC32.pdf

[23] C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen, "Wcet analysis of java bytecode featuring common execution environments," in *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '11. New York, NY, USA: ACM, 2011, pp. 30–39. [Online]. Available: http://doi.acm.org/10.1145/2043910.2043916

A Test Results

A.1 Buffer test

```
$> scjvm main
[SafeletExecuter] Safelet started
Initializing main mission
1 MissionMemory : (LTMemory) Scoped memory # 3
** Producer executing in (LTMemory) Scoped memory # 5
1. New Object[1] is in : (LTMemory) Scoped memory # 3
** Consumer executing in (LTMemory) Scoped memory # 4
2. Object.toString() : java.lang.Object@8edabcc

** Producer executing in (LTMemory) Scoped memory # 5
1. New Object[2] is in : (LTMemory) Scoped memory # 3
** Consumer executing in (LTMemory) Scoped memory # 4
2. Object.toString() : java.lang.Object@11f492f5

** Producer executing in (LTMemory) Scoped memory # 5
1. New Object[3] is in : (LTMemory) Scoped memory # 3
** Consumer executing in (LTMemory) Scoped memory # 4
2. Object.toString() : java.lang.Object@1bfe187f

** Producer executing in (LTMemory) Scoped memory # 5
1. New Object[4] is in : (LTMemory) Scoped memory # 3
** Consumer executing in (LTMemory) Scoped memory # 4
2 Object.toString() : java.lang.Object@65006701

** Producer executing in (LTMemory) Scoped memory # 5
1. New Object[5] is in : (LTMemory) Scoped memory # 3
** Consumer executing in (LTMemory) Scoped memory # 4
2. Object.toString() : java.lang.Object@6f0aed8b
```

A.2 Barrier test

```
$> scjvm main
[SafeletExecuter] Safelet started
Initializing main mission

** Fire0 is executing **
0. Fire0 – triggering barrier **


** Fire0 is executing **
0. Fire0 – barrier already triggered **


** Fire0 is executing **
0. Fire0 – barrier already triggered **


** Fire0 is executing **
0. Fire0 – barrier already triggered **


** Fire0 is executing **
0. Fire0 – barrier already triggered **


** Fire1 is executing **
1. Fire1 – triggering barrier **

3. LaunchHandler – LAUNCHING MISSILE

** Fire0 is executing **
0. Fire0 – triggering barrier **


** Fire0 is executing **
0. Fire0 – barrier already triggered **
```

A.3 Signal test

A.3.1 Normal Producer, Normal Worker;

```
Norm + Norm; no clocks
$> scjvm main
[SafeletExecuter] Safelet started
Initializing main mission

1.1 Producer — starting computation
1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output not done yet
2 Worker — output iteration: 1

1.1 Producer — starting computation
1.2 Producer — starting extra computation
2 Worker — output iteration: 2
1.3 Producer — finishing computation
1.4 Producer — output done

1.1 Producer — starting computation
1.2 Producer — starting extra computation
2 Worker — output iteration: 3
1.3 Producer — finishing computation
1.4 Producer — output done

1.1 Producer — starting computation
1.2 Producer — starting extra computation
2 Worker — output iteration: 4
1.3 Producer — finishing computation
1.4 Producer — output done

1.1 Producer — starting computation
1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output not done yet
2 Worker — output iteration: 5

1.1 Producer — starting computation
```

```
        1.2 Producer — starting extra computation
        1.3 Producer — finishing computation
        1.4 Producer — output not done yet
        2 Worker — output iteration: 6
```

A.3.2 Normal Producer, Higher Worker;

```
        $> scjvm main
        [SafeletExecuter] Safelet started
        Initializing main mission

        1.1 Producer – starting computation
        2 Worker – output iteration: 1
        1.2 Producer – starting extra computation
        1.3 Producer – finishing computation
        1.4 Producer – output done

        1.1 Producer – starting computation
        2 Worker – output iteration: 2
        1.2 Producer – starting extra computation
        1.3 Producer – finishing computation
        1.4 Producer – output done

        1.1 Producer – starting computation
        2 Worker – output iteration: 3
        1.2 Producer – starting extra computation
        1.3 Producer – finishing computation
        1.4 Producer – output done

        1.1 Producer – starting computation
        2 Worker – output iteration: 4
        1.2 Producer – starting extra computation
        1.3 Producer – finishing computation
        1.4 Producer – output done

        1.1 Producer – starting computation
        2 Worker – output iteration: 5
        1.2 Producer – starting extra computation
        1.3 Producer – finishing computation
        1.4 Producer – output done

        1.1 Producer – starting computation
        2 Worker – output iteration: 6
        1.2 Producer – starting extra computation
        1.3 Producer – finishing computation
```

```
1.4 Producer — output done

1.1 Producer — starting computation
2 Worker — output iteration: 7
1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output done

1.1 Producer — starting computation
2 Worker — output iteration: 8
1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output done

1.1 Producer — starting computation
2 Worker — output iteration: 9
1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output done

1.1 Producer — starting computation
2 Worker — output iteration: 10
1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output done
```

A.3.3 Max Producer, Normal Worker;

The Producer is set to Max priority, Worker is set to Normal priority.

```
 [SafeletExecuter] Safelet started
Initializing main mission

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output not done yet
  2 Worker — output iteration: 1

1.1 Producer — starting computation
```

```
     1.2 Producer – starting extra computation
      2 Worker – output iteration: 2
    1.3 Producer – finishing computation
    1.4 Producer – output not done yet

    1.1 Producer – starting computation
     1.2 Producer – starting extra computation
      2 Worker – output iteration: 3
    1.3 Producer – finishing computation
    1.4 Producer – output not done yet

    1.1 Producer – starting computation
     1.2 Producer – starting extra computation
      2 Worker – output iteration: 4
    1.3 Producer – finishing computation
    1.4 Producer – output not done yet

    1.1 Producer – starting computation
     1.2 Producer – starting extra computation
      2 Worker – output iteration: 5
    1.3 Producer – finishing computation
    1.4 Producer – output not done yet

    1.1 Producer – starting computation
     1.2 Producer – starting extra computation
      2 Worker – output iteration: 6
    1.3 Producer – finishing computation
    1.4 Producer – output not done yet

    1.1 Producer – starting computation
     1.2 Producer – starting extra computation
      2 Worker – output iteration: 7
    1.3 Producer – finishing computation
    1.4 Producer – output not done yet

    1.1 Producer – starting computation
     1.2 Producer – starting extra computation
      2 Worker – output iteration: 8
    1.3 Producer – finishing computation
    1.4 Producer – output not done yet
```

```

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
  2 Worker — output iteration: 9
1.3 Producer — finishing computation
1.4 Producer — output not done yet

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
  2 Worker — output iteration: 10
1.3 Producer — finishing computation
1.4 Producer — output not done yet

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output not done yet
  2 Worker — output iteration: 11

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output not done yet
  2 Worker — output iteration: 12

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
1.3 Producer — finishing computation
  2 Worker — output iteration: 13
1.4 Producer — output not done yet

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output not done yet
  2 Worker — output iteration: 14

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
1.3 Producer — finishing computation
```

```
1.4 Producer — output not done yet
  2 Worker — output iteration: 15

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output not done yet
  2 Worker — output iteration: 16

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output not done yet
  2 Worker — output iteration: 17

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output not done yet
  2 Worker — output iteration: 18

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output not done yet
  2 Worker — output iteration: 19

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
1.3 Producer — finishing computation
1.4 Producer — output not done yet
  2 Worker — output iteration: 20

1.1 Producer — starting computation
 1.2 Producer — starting extra computation
  2 Worker — output iteration: 21
1.3 Producer — finishing computation
1.4 Producer — output not done yet
```

B Compiling and running the programs

First, you need to obtain the runnable version of the code which you want to run. All the source is contained in the src folder.

To compile and run, you will need to have the JamaicaVM available. This is already installed on the departmental machines.

```
> cd src
> ../../ri_rtsjBased_j4.fix/bin/nscjavac main.java
```

This should compile without problems using the supplied Reference Implementation version, the application is stared by doing:

```
> ../../ri_rtsjBased_j4.fix/bin/nscjvm main
```

If you want to recompile, you simply remove the class files and repeat the process above