The University of York Department of Computer Science

Submitted in part fulfilment for the degree of BEng.

Programming in Safety-Critical Java

Author:
Jonathan Co

Supervisor:
Prof. Ana Cavalcanti

2014-May-6

Number of words: 20,272, as counted by detex wc -w
This includes the body of the report only

Abstract

Increasingly, software is used to control systems where failure can lead to loss of human life. Such systems are classified as safety-critical. By law these systems must be certified to ensure the correctness of the system. Certification is both a time consuming and costly process. Consequently details relating to project development are carefully planned. One such detail is the choice of implementation language. To aid in certification, these languages are minimalistic with specific features for developing such systems.

The Safety-Critical Java (SCJ) specification is an attempt to modify the Java language for use in the development of safety-critical systems. However, the specification is relatively new and there are a number of limited use cases illustrating its use.

This projects explores and evaluates the SCJ specification. We redevelop an existing system using the departmental reference implementation and the public draft of the SCJ specification. This redeveloped system is then compared against the original system in order to evaluate the use of SCJ for safety-critical systems.

Acknowledgements

I would like to thank my supervisor, Prof. Ana Cavalcanti, for all the invaluable advice and support given throughout this year.

A big thank you to my parents and friends for all the support they have given me. Their constant encouragements were very much appreciated and helped me through many late nights.

Last but not least, I would like to thank the York Hornets Cheerleading Club. Without them I would have definitely finished this project earlier.

Contents

List of Tables

List of Figures

Listings

1 | Introduction

In the modern world software is increasingly being used to control systems where failure could cause loss of human life, severe damage to equipment, significant financial loss or environmental damage. Such systems are classified as ‘Safety-Critical’ or ‘High-Integrity’ systems.

Due to the nature of such systems certification is required before such a system can be deployed in the real world. For instance, in the avionics industry software is certified in accordance with DO-178B [1]. In order to aid certification, languages used to develop safety-critical systems are often a small subset (also known as profiles) of an established language. The SPARK Ada profile in particular has been proven to be certifiable in a wide range of safety-critical applications [2].

Java is an object orientated language used in a range of applications from Enterprise level software to mobile devices. Its abstraction of lower level features such as memory management and threading have contributed to the popularity of Java. Additionally, its use of inheritance and polymorphism can arguably lead to more maintainable systems by adding further layers of abstraction.

However, the same features that make Java successful for general applications make it unsuitable for real-time systems [3]. One example of this relates to Java’s memory management feature which causes temporal unpredictability in time critical tasks [4]. To address these concerns the Real-Time Specification for Java (RTSJ) was developed [5, 6]. However extensions introduced by RTSJ to make Java suitable for real-time systems have also resulted in increased complexity, which is unsuitable for safety-critical applications.

Safety Critical Java (SCJ) is a profile of RTSJ that aims to provide a smaller set of core libraries for developing safety-critical applications.

1.1 Objectives

The aims of this project are summarised as follows:

- Identify a suitable system for implementation in SCJ
- Develop and test the aforementioned system
- Evaluate the developed implementation and the capabilities of SCJ
- Present possible avenues of further work with SCJ

1.2 Report Structure

The report is organised as follows:

- Chapter 2 - Presents background information on Java, RTSJ, SCJ and use case studies relating to SCJ.
- Chapter 3 - Presents a high-integrity system previously developed in Ada, to be used as the basis for an SCJ port. This also outlines the requirements of the system.

- Chapter 4 - Discusses our design and implementation of the identified high-integrity system.
- Chapter 5 - Tests the developed system with regards to its functional aspects.
- Chapter 6 - Presents an evaluation of the developed system and the SCJ specification.
- Chapter 7 - Concludes our work and discusses future work.

1.3 Ethical Statement

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

No use of human participants was required, hence there are no implications regarding their welfare. As this project features the use of a simulated real world, there is no endangerment of human life.

This work contributes to the development of safety-critical systems. Due diligence must therefore be performed before using this work to further assess safety-critical systems outside an academic environment.

2 | Literature Review

The aim of this chapter is to provide the reader with the background information needed to understand the contents of this report. This includes an overview of Java, the Real-Time Specification for Java and the Safety Critical Java Specification.

2.1 Java

Java is an object oriented language first released in 1995 by Sun Microsystems as an alternative to C/C++. Originally designed for digital entertainment services, Java has since been used in a wide array of applications. Notable examples include its use in the Android Operating System [7] and the Twitter social network [8].

The main features offered by Java (in no particular order) include [9]:

Object Orientated - Java is an Object-Orientated Programming (OOP) language, using objects to encapsulate both data and the methods acting on the data together. Java uses a `Class` based approach, defining objects as classes. Inheritance is achieved via the implementation of subclasses.

Architecture Neutral - Java is compiled into 'bytecode', which is interpreted into hardware instructions by a Java Virtual Machine (JVM). This allows Java programs to be compiled once and run anywhere (as long as a JVM is available for that hardware configuration).

Automatic Memory Management - Memory is implicitly managed by an automatic garbage collector. When an object in the program is no longer in use, its memory is automatically reclaimed by the garbage collector for reuse for other object allocations.

Integrated Thread Synchronisation - Multi-threading support is provided via the `Thread` class. The `synchronized` keyword can be used to define methods and variables that must only be accessed by a single thread at a time, giving mutual exclusion. To avoid using potentially cached values, the `volatile` keyword can be used to force a thread to read the current value of a variable whenever it is accessed [10].

Statically and Strongly Typed - Variables are bound to a specific data type (strong typing) and are checked at compile time (static typing). Some types, however, can only be resolved at runtime.

Dynamic Class Loading and Linking [10] - The JVM dynamically loads and links classes as they are needed during execution of a program. When a class is referenced in the program, it is first loaded into the JVM by finding the compiled binary representation of the class. The class is then recreated from this representation. Linking is the process of adding this loaded class to the current run-time.

Extensive Base Libraries - Java contains an extensive and mature set of libraries designed to ease programming. An example of this is the *Java Collections Framework*, which contains many general implementations of common data structures in Java [11].

2.1.1 Concurrency Model

Concurrent programs represent a set of processes that are run in parallel, offering significant advantages compared to their sequential counterparts. This comes in the form of higher CPU utilisation and the use of multiple CPUs at once, affording in faster execution speeds [12, Chapter 1].

Concurrency is required to model domains which are non-deterministic and concurrent, as noted by Burn [12, 1.5]. Safety-Critical applications naturally fall into this area as they are typically used to monitor and control physical entities.

However, concurrent programs suffer from additional complexity not present in single threaded programs. Additional overhead in the form of communication and synchronisation between each thread is required in order to coordinate their activities. For instance, monitor locks must be placed on shared resources (a resource used by multiple threads) whenever they are used to enforce data consistency.

This gives rise to the following problems [12, Chapter 3]:

Deadlock - Occurs when all threads are waiting for another thread to complete some operation. In this state the program cannot continue without external influence.

Starvation - Occurs when at least one thread is continually denied access to a shared resource due to the actions of other threads. Consider the situation where thread A requires the use of resource X. If Thread B continually locks and uses resource X, then thread A would enter a starvation state as it would be unable to run without access to X.

Interference - Occurs when at least two threads attempt to modify the same shared resource at once. Without placing some sort of lock on the resource this leads to corrupted data.

Concurrency is provided in Java via the `Thread` class and supported by the `synchronized` keyword, `volatile` keyword and the concurrency libraries part of the core Java Development Kit (JDK).

The Thread Class

Java supports concurrent activities via the `Thread` class, which represents a single thread of execution in a program. In its simplest form a `Thread` is created by subclassing `Thread`, overriding the `run()` method, creating a new instance of the subclass and calling the `start()` method. The subclass can be created either by extending the `Thread` class or creating an anonymous inner class as seen in Listing 2.1.

```
/**
 * Thread creation via extending Thread
 */
class CustomThread extends Thread {

        // Constructor and field declarations omitted.

        @Override
        public void run() {
                // Perform custom logic here
        }
}
```

```
// Create new instance of Thread and start it
CustomThread customThread = new CustomThread();
customThread.start();
```

Listing 2.1: Java Thread Creation

Runnable and Callable

Alternatively a class implementing the `Runnable` interface can be passed into the constructor of a new instance of `Thread`. This is preferred as it allows code reuse by enforcing composition over inheritance. A similar interface, the `Callable` interface, can be used in the same way. Unlike `Runnable`, `Callable` classes are able to throw checked exceptions and return a result on completion. The same example shown in Listing 2.1 but using the `Runnable` style is shown in Listing 2.2.

```
class CustomRunnable implements Runnable {

        // Constructor and field declarations omitted.

        @Override
        public void run() {
                // Perform custom logic here
        }
}

// Create instance of runnable and start a thread with it
CustomRunnable runnable = new CustomRunnable();
new Thread(runnable).start();
```

Listing 2.2: Java Thread Creation using `Runnable`

Executors

As of Java 1.5 it is generally advised to not work directly with the `Thread` class where possible, with preference given to the use of the *Executor Framework* [13, Item 69]. `Runnable`s can be submitted to an instance of `Executor` for execution. The `Executor` will automatically manage threads required for the execution of the `Runnable`. Typical executor usage can be seen in Listing 2.3.

```
Runnable runnable = new CustomRunnable();

// Create executor with 5 threads
ExecutorService executor =
        new Executors.newFixedThreadPool(5);

// Execute Runnable using Executor
executor.execute(runnable);
```

Listing 2.3: Executor Usage

Thread State

An instance of `Thread` can be in one of several states. The state of a thread can be retrieved using its `getState()`, which returns a value from the `Thread.State` enumerate type. The possible states and their meanings are as follows:

`NEW` - The initial state of a thread once it has been constructed. In this state the thread has not yet started.

`RUNNABLE` - The thread is currently running and executing its run logic in the JVM.

`BLOCKED` - This state is entered when the thread attempts to acquire a lock (see 2.1.1) of an object currently in use by another thread.

`WAITING` - Self-suspending threads are placed in this state. The thread is placed back into the `RUNNABLE` state when the `notify()` or `notifyAll()` methods are called.

`TIMED_ WAITING` - Similar to the `WAITING` state with the addition that the thread can also resume after the timeout has been reached.

`TERMINATED` - Once a thread has finished execution of its `run()` method, it is placed in this state.

Thread Termination

A thread will naturally terminate (switch to a `TERMINATED` state) upon successful completion of its `run()` method. The standard way to terminate a thread before the `run()` method completes, is to set an external flag specifying if the thread should exit early. The logic of the `run()` method should check this flag regularly.

The `destroy()` and `stop()` methods can also be used to terminate a thread but are inherently unsafe and have since been deprecated [14].

Thread Priority and Scheduling

When a thread is ready to begin execution, it switches from the `NEW` state to the `RUNNABLE` state and is placed onto a Runnable queue. The thread Scheduler is used to control the control and execution of these runnable threads. To aid in the determination of which thread to execute, every thread is assigned a priority.

However, the scheduling policy used by the Scheduler may not necessarily choose a thread with the highest priority to execute [10, §17.12]. As noted by Bloch [13, Item 72], applications should not rely on the Scheduler or thread priorities to ensure any sort of program correctness. Such programs are not portable as they depend on implementation specific to the operating system and JVM they run on.

The Synchronized Keyword

Java provides a monitor-like mechanism for acquiring locks on shared resources. The `synchronized` keyword can be added to method declarations in order to ensure only a single thread can access the method at once. Methods declared using this keyword therefore enforce mutual exclusion, one of the conditions required to prevent deadlock. Synchronised methods also serve to ensure all threads entering a synchronised method or block see all modifications previously performed by other threads, as noted by Bloch [13, Item 66].

2.1.2 Summary

In this section a brief overview of the features in the Java language was given, with particular focus on its concurrency support. Both RTSJ and SCJ expand upon the functionality introduced by the Java Thread model.

2.2 Real-Time Specification for Java

A real-time system is defined as any information-processing system which has to respond to externally generated input stimuli within a finite and specified period [15]. Correctness of the system depends not only on the logical result but also on the time it is delivered.

This caveat has led to real-time systems being distinguished as either **hard** or **soft** real-time systems. In hard real-time systems, failure to respond within a specified deadline would lead to system failure. Soft real-time systems however, can tolerate the occasional missed deadline.

Real-time systems are usually part of a larger engineering system and known as embedded systems. They must be engineered to be extremely reliable and safe as they are often used to control and monitor physical equipment. Concurrency support is required in order to model the parallelism of multiple real world components [3, Chapter 1]. Such systems are often required to be both reliable and safe.

This section gives an overview of the RTSJ including the reasons for its development. Particular focus is given to the memory management and the schedulable objects framework of RTSJ. A more complete discussion on all aspects of RTSJ can be found in the official specification [5] and in Wellings [3, Chapter 7].

2.2.1 Suitability of Java for Real-time Systems

The popularity of Java has led to its use in a wide variety of applications. However, the lack of support for real-time facilities renders it unsuitable for the development real-time systems.

For instance, in order to prove a real-time system executes within a specified deadline, timing analysis must be performed. Due to the use of a garbage collector, tight timing analysis is hard to perform against Java applications [16]. All garbage collections performed are known as 'stop-the-world' events. This halts program execution completely until the operation has completed. A typical collection lasts for hundreds of milliseconds [4].

In the Oracle HotSpot JVM, the Generational Collection garbage collection mechanism is utilised [17]. Though this minimises the number of major (long running) garbage collection operations needed, they will still inevitably occur at some point [4].

When these collections occur cannot be easily predicted. As a result tight timing analysis is difficult to perform. Similarly the use of dynamic class loading also adds additional complexity to timing analysis [18]. As such Java cannot be said to reliably execute programs in a finite time period, rendering it unsuitable for real-time systems.

Various other limitations in the Java concurrency model have been shown to make Java unsuitable for real-time systems [3, §4.7]. This includes a lack of support for conditional variables, lack of absolute time delays, difficult identification of nested thread-safe method invocation and a weak thread priority mechanism (see subsection 2.1.1). Wellings notes that although utilities available since Java 1.5 in the `java.util.concurrent` package solves some of these problems, fundamental changes are required to support real-time applications reliably [3, §4.4].

To address these issues, the RTSJ was developed to enhance support for real-time systems in Java. RTSJ provides these enhancements via the `javax.realtime` package and through modifications to the JVM.

2.2.2 Memory Management

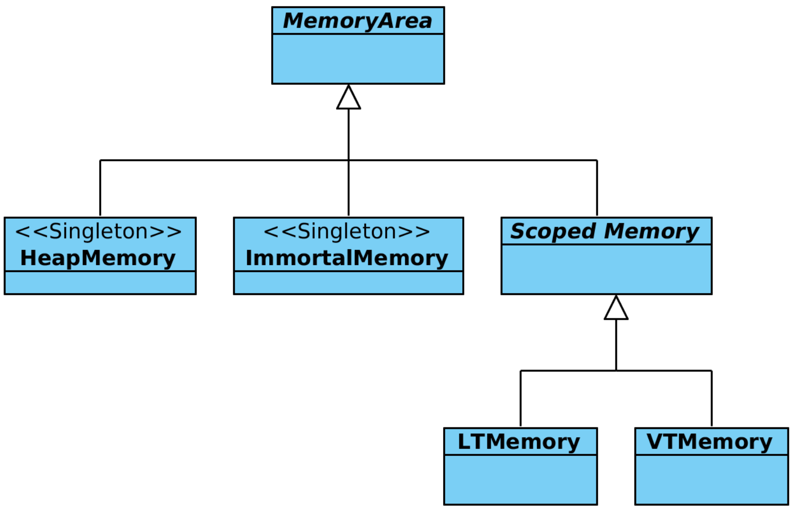

Figure 2.1: javax.realtime Memory Classes

Java divides memory into two structures: the **stack** and the **heap**. The stack is used to store primitive values which only exist in the scope of the current method they are created in. Once the method is exited, these values are discarded and the memory is reclaimed.

The heap is used to store created objects. Heap memory is reclaimed by the garbage collection using various mechanisms such as incremental collection [4]. Whilst garbage collection simplifies development, this leads to issues when performing timing analysis as discussed in subsection 2.2.1.

To remedy this RTSJ defines a new memory management model based on memory areas [19,20]. There are multiple implementations of memory areas, all of which are subclasses of the abstract base class `MemoryArea`. The implementations of `MemoryArea` are logically separate form the heap and are thus not subject to garbage collection. The two primary memory areas introduced are Immortal and Scoped memory. Their package structure can be seen in Figure 2.1.

Immortal Memory

Objects allocated in immortal memory are never reclaimed by the garbage collector during the lifetime of the application; i.e. once space has been allocated, it can never be reclaimed automatically by the garbage collector [3, §8].

Immortal memory is accessed using the `ImmortalMemory` singleton class. An instance can be retrieved using the `ImmortalMemory.instance()` method instead. `ImmortalMemory` can be accessed by all threads of execution unexceptionally to allocate objects.

Scoped Memory

Unlike immortal memory, scoped memories have a finite lifetime. A thread is able to enter a scoped-memory area and allocate objects in that area whilst executing. Each scoped-memory area has an internal reference count associated with it that keeps track of the number of threads currently active within it [3, §7.3]. Once this count reaches 0, all objects within are deallocated and the memory is reclaimed for reuse. This count can be retrieved using the `getReferenceCount()` method. By default two implementations of scoped memory are offered: `LTMemory` and `VTMemory`.

For the purposes of timing analysis, it is important to take into account the allocation time. This is the time required to allocate space for an object excluding the execution of its constructor. `LTMemory` (Linear Time Memory) requires that allocation time be proportional to the size of the object. `VTMemory` (Variable Time Memory) is similar to `LTMemory` but does not have the time restriction imposed upon it. It is expected that `VTMemory` will perform faster but less predictably than `LTMemory`.

Usage

As noted by Pizlo et al. [21], real-time applications are often structured around one top-level loop, which repeatedly executes application code. In a traditional loop that performs object allocations, the execution time would vary due to interference from the garbage collector. Such a loop can be seen in Listing 2.4.

```
void runLoop() {
        while(true) {
                // Execute loop logic
        }
}
```

Listing 2.4: Traditional Application Loop

Using memory areas, the vagaries associated with garbage collection can be eliminated by introducing an instance of `LTMemory`. Listing 2.5 demonstrates how the loop given in Listing 2.4 can be re-implemented in this style. Here the instance of `LTMemory` is defined as `memory`. At each iteration of the `while` loop, `loopRunnable` enters `memory`, executing its `run()` method within the context of `memory`. All object allocations (calls to `new`) are performed in this memory area rather than the heap or immortal areas. Once the runnable has finished executing, it leaves `memory` allowing any objects created to be deallocated immediately.

This type of loop is known as a *Scoped Run Loop* and represents a simple design pattern used in RTSJ. More complex examples of RTSJ memory-management design patterns are discussed by Pizlo et al [21].

```
void runLoop() {
        // New instance of scoped memory with size limits
        LTMemory memory = new LTMemory(initialSize, maxSize);

        // Runnable containing loop logic
        Runnable loopRunnable = new Runnable() {
                public void run() {
                        // Execute loop logic
                }
        }

        while(true) {
                memory.enter(loopRunnable);
        }
}
```

Listing 2.5: Application Loop using Memory Areas

2.2.3 Schedulable Objects

RTSJ introduces the concept of a **Schedulable Object** to encapsulate all concurrent activities. This is to abstract away further from standard Java `Threads` and to provide alternatives to threads. All schedulable objects must implement the `Schedulable` interface and allow the specification of the following parameters:

Release Parameters - Specifies when a schedulable object should be released (run). This is usually defined as periodic (invoked at set intervals), aperiodic (invoked randomly) or sporadic (invoked randomly but with a set interval between each release).

Memory Parameters - As real-time systems usually have strictly defined resource limits, this parameter allows the amount of memory required by the schedulable object to be specified.

Scheduling Parameters - Allows specification of various scheduling parameters such as priority and importance.

As a result, the `Thread` subclasses, `RealtimeThread` and `NoHeapRealtimeThread`, have been implemented. `NoHeapRealtimeThread` is a specialisation of `RealtimeThread` that guarantees that objects are not created or referenced from the heap during the threads execution.

Events and Event Handling

Real-time systems are usually required to respond to events that occur asynchronously in the environment [3, §7.7]. One way to deal with events is to have threads wait until an event is triggered. Unfortunately, this is wasteful and can introduce unnecessary complexities [22, 23].

RTSJ provides an alternative to threads by implementing an events based system. In such a system, events are fired and subsequently handled by event handlers. These are modelled by the `AsyncEvent` and `AsyncEventHandler` classes respectively. A single event can be associated with many different handlers.

Fired events are first placed onto an ordered queue. From here real-time server threads take the events and execute their associated handler. This process is repeated until the queue is empty. As binding a handler to a thread carries some overhead, `BoundAsyncEventHandler` instances can be used, which are permanently bound to their own real-time thread.

There are many benefits associated with an events-based concurrency model when compared to a thread-based model [23]. These include easier scheduling mechanics, simpler logic in the handlers and better resource usage. There are some disadvantages as well unfortunately. Tight deadlines are difficult to achieve when long-running handlers are used. These can also cause deadlines to be missed. Also, higher priority handlers can be delayed when all server threads are occupied with executing lower priority handlers.

Wellings and Kim [23] note that both thread-based and event-based models are available in RTSJ to allow programmers flexibility and to reap benefits from both models.

2.2.4 Summary

In this section RTSJ has been introduced, with focus given to the memory management and concurrency concepts. As SCJ adopts these concepts, it is important to understand their capabilities and usage.

2.3 Safety-Critical Java

Safety-Critical systems, also known as High-Integrity Systems, are systems where failure can cause loss of life, environmental harm or significant financial loss [3, Chapter 17].

To demonstrate their suitability, such systems are required to conform to strict standards. For instance, avionics components are certified under the DO-178B standard in the US [1]. Due to these requirements, development and certification of Safety-Critical systems are expensive and time consuming.

To aid in certification, safety critical systems are developed using small subsets of languages such as SPARK Ada [24]. Their smaller profiles aid in reducing the complexity and time cost of certifying these systems.

Despite the memory management and concurrency improvements introduced in RTSJ, it is still not suitable for safety-critical applications [24–26]. To rectify these problems the SCJ specification has been introduced [27]. This specification builds upon the features introduced by RTSJ to bring Java to the safety-critical domain.

This section presents an overview of the Safety-Critical Java Specification.

2.3.1 The SCJ Mission Model

SCJ introduces the concept of missions, with each application being made up of one or more missions. Each Mission consists of a number of predefined schedulable objects residing in mission memory. Missions allow modular components to be built, with each mission representing an operational phase or activity in the application. For instance control software for an aircraft could consist of four missions representing the taxi, take-off, cruise and landing phases of flight [27, Chapter 3]. Advanced SCJ applications allow the nesting of missions in missions.

A mission is organised into three distinct phases:

Initialization - In this phase all schedulable objects are registered to the mission for later execution. Schedulable objects are usually allocated to the mission's MissionMemory. MissionMemory is an indirect extension of LTMemory and is not reclaimed until the mission is terminated. During this phase any shared data objects required by the schedulable objects should be allocated.

Execution - During execution each schedulable object is allocated its own PrivateMemory area to use during its release. Once all schedulable objects have finished, this phase is terminated.

Cleanup - This phase is used to reclaim resources used and reset application state.

The execution order of the missions is controlled by a MissionSequencer, the lifecycle of which is illustrated in Figure 2.2.

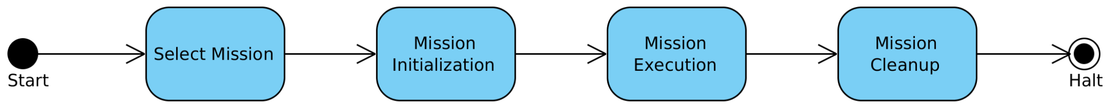

Figure 2.2: Safety Critical Mission Phases (figure redrawn from [27])

The top-level sequencer resides in a Safelet, which is allocated in immortal memory. A Safelet implementation is used to represent the SCJ application and to house the outermost MissionSequencer.

2.3.2 Concurrency Model

The SCJ concurrency model consists almost entirely of asynchronous event handlers. The notion of a Schedulable has been extended to form ManagedSchedulable, in order to denote their control by missions. Further specialisations are provided by ManagedEventHandler and ManagedThread.

All ManagedEventHandler must specify their release logic, priority, storage parameters (such as required private memory allocation) and deadline miss handlers. The PeriodicEventHandler (PEH) is one such implementation of ManagedEventHandler. In addition to the previously mentioned parameters, it must also specify a relative start time and period between each subsequent release. The AperiodicEventHandler (APEH) on the other hand requires no timing parameters but will not release unless bound to an event.

Upon each release, a handler will enter and exit its own instance of PrivateMemory. Additionally, handlers are able to access the applications immortal memory and any memory instances declared inside the handler. No sharing of memory instances can be done across handlers however.

In practice all event handlers are bound handlers. This is to eliminate the overhead of thread binding on handler release and for predictability purposes [27].

2.3.3 Compliance Levels

The SCJ specification notes that safety-critical applications vary considerably in complexity [27, §2.2]. To support this notion, SCJ introduces the notion of different compliance levels. Each complicance level defines the types of schedulable objects allowed and synchronization infrastructure in addition to other capabilities. It is expected that lower level applications are forwards compatible with higher levels.

Level 0 - Level 0 applications follow a cyclic executive programming model. Applications consist of a single mission sequence of PEHs. Each handler is released sequentially according to their timing parameters. The entire schedule of releases is known as a major cycle. It is expected that the developer constructs this schedule and timings independently.

At this level the only schedulable object supported is the PEHs. Furthermore, the Object.wait() method, Object.notify() method and self-suspending handlers are not supported.

Level 1 - Applications can consist of both PEHs and APEHs. Handlers are executed in priority order by a fixed-priority preemptible scheduler. As preemption can occur, access to immortal or mission memory should be controlled using synchronized methods to maintain integrity. Similar to Level 0 threads, `Object.wait()` method, `Object.notify()` method and self-suspending handlers are not supported.

Level 2 - Level 2 applications represent the most sophisticated SCJ applications. All previous restrictions in Level 0 and 1 are lifted. Consequently in addition to PEH and APEH, `NoHeapRealtimeThread` is included in the list of allowed schedulable objects.

Nested missions are allowed at this level to model multiple concurrent sets of activities. Here schedulables may preempt other schedulables in a mission even if they do not belong to the same mission.

2.3.4 Summary

In this section the SCJ specification has been introduced. It is important to understand the event based model the specification uses as it influences the design of SCJ applications.

2.4 SCJ Use Cases

This section presents various SCJ case studies, which may be considered when undertaking the design and implementation of our own work. As SCJ is still an emergent technology, there exists a limited amount of published examples.

2.4.1 Cardiac Pacemaker

Singh et al. [28] consider the use of SCJ to control a cardiac pacemaker. A previously verified model of a pacemaker was used to inform the design of the software. The resulting implementation demonstrated how SCJ could be used to support multiple complex operating modes with strict timing requirements in a full software architecture. Previous studies had only considered individual components with little consideration given on how components would interact with each other or be scheduled in the whole system.

For comparison, an implementation in Ravenscar Ada was also provided. It was found that whilst Ada provided a more efficient solution, the timing properties were more difficult to calculate. In contrast SCJ was less efficient but the scheduling characteristics were easier to determine.

The study criticised SCJ for lacking explicit support for one-shot timed events. As a result the most recent specification has added support for this in the form of `OneShotEventHandler`.

The work from this study can be used to inform the design of any SCJ application that requires the use of watchdog timers and multi-mode applications. Of particular note is its use of polling to determine the current system state, which is often required in real-time systems. It can also be incorporated into any implementation that controls real world devices.

2.4.2 CDx

CD$_x$ is a aircraft collision detection system, originally designed as a benchmarking suite for RTSJ [29]. The original implementation used a cyclic executive. At each iteration the system would:

1. From a device read in a *frame* containing the positions and IDs of all aircraft.
2. Perform a *reduction* step to map aircraft to voxels.
3. In each voxel *detect* if a collision had occurred between any aircraft.
4. *Record and report* any collision that had occurred.

Using SCJ, Zeyda et al. [30] implemented a concurrent version of CD_x. This was to both demonstrate the capabilities of SCJ and to improve performance of CD_x by introducing parallelism. Each step of the system was mapped to an event handler. These consisted of InputEventHandler, ReducerHandler, DetectorHandler and OutputCollisionsHandler respectively. As the detection step was most computationally expensive, four instances of DetectorHandler were used in order to simulate the use of four processors.

Control flow between the handlers was performed using events. As each handler completes their execution, an event would be fired that would release the next handler. Releases were sequential in nature until the detection phase was reached. Here all four DetectorHandlers would have to finish execution before the next event was fired. A barrier was introduced in the form of DetectorControl to coordinate this. Each DetectorHandler would need to notify the DetectorControl of completion before the next event was fired.

This study highlights possible ways of synchronising communication across different handlers in the system using a barrier construct. Additionally, this study can help in devising a way of simulating external input into an SCJ application for testing.

2.4.3 A Desktop 3D Printer

In order to evaluate the SCJ specification and introduce a new use case using SCJ, Strøm and Schoeberl developed an application to control a desktop 3D printer [31].

This case study provides useful insight into the strengths and weaknesses of SCJ. Especially useful is its discussion on the location of created objects and referencing such objects. Shared data between handlers needs to be created within a scope of memory accessible to all handlers, such as within immortal or mission memory scopes.

The design of our implementation should therefore consider the location of shared object creation and their synchronisation amongst different handlers.

2.5 Summary

In this chapter an overview of SCJ and its parent languages, Java and RTSJ, has been given. Various use cases have also been presented, which can be used to inform and guide our own design and implementation.

3 | Problem Analysis

The TIS project is a research study jointly undertaken by Praxis High Integrity Systems (now Altran Praxis) and the National Security Agency (NSA) [32]. The end product of this project is the TIS, a component in a larger security system. The TIS is used to provide security functions to a physically secure enclave and its contents. This involves the monitoring and manipulation of several external peripheral devices.

The public release of the TIS consists of approximately 10,000 lines of code accompanied by approximately 900 pages of documentation and reports. Due to the size and complexity of the TIS, this report restricts its presentation of the following aspects:

Security - Only the control of the physical peripherals relating to security, such as the door, are considered in by this report. Similarly, aspects relating to the identification and verification of the security requirements against formal standards are also ignored. Further discussion on these topics can be found in [33–37].

The functionality and features that have been extrapolated from these aspects have already been consolidated into the System Requirements Specification (SRS) and the formal specification by the original developers [38, 39]. Furthermore functionality relating to cryptographic functions such as biometric verification are ignored as this is outside the scope of this project.

Verification - Verification of the produced code-base against the formal specification is not considered due to time constraints. Instead, user acceptance testing is performed using the Use Cases outlined in the SRS [38, §5]. The verification process used in the original TIS project are presented in [40, 41].

This chapter begins by presenting an overview of the original Praxis Tokeneer project. The original Tokeneer design documents are presented, along with an introduction to the Z specification language used to write these documents. From these documents, features are identified for implementation in SCJ. These features are then used to draw up a list of system requirements.

3.1 Praxis Tokeneer

The development of theTIS demonstrates the development and verification of a high-quality, low-defect system. Its original purpose was to show that a system could be developed to conform to Evaluation Assurance Level 5 (EAL5) of the Common Criteria framework [38, §2] in a cost-effective manner.

The fifth of seven assurance levels, EAL5 compliance states that the system has been “semiformally designed and tested” [42, §8.5]. EAL5 conformance is sought when a high level of security assurance is required without incurring unreasonable costs during development and verification. It is expected that assurance levels higher than EAL5 will incur higher costs and be performed on smaller systems [42, §8.9].

Development of the system followed the ‘Correctness by Construction’ process pioneered by Praxis [32]. This process consists of the following phases [43]:

1. Requirements Analysis (using the REVEAL® process pioneered by Praxis [44])
2. Formal Specification (using the formal language Z)
3. Design (using the INformation Flow ORiented MEthod of (object) Design (INFORMED) process [45] and refinement of the formal specification)
4. Implementation in SPARK Ada [2]
5. Verification (using the SPARK Examiner toolset)

The Praxis developers note that this development process produces reliable products whilst still being cost-effective [32]. Indeed, since its public release in 2008, the TIS has proven to have a low defect rate [46–48]. This is remarkable due to the fact the entire system was developed in a period of nine months by a development team consisting of three people [49].

The TIS itself is comprised of several distinct components, each responsible for a specific function in the system.

3.1.1 System Description

An overview of the various components that comprise the TIS can be seen in Figure 3.1.

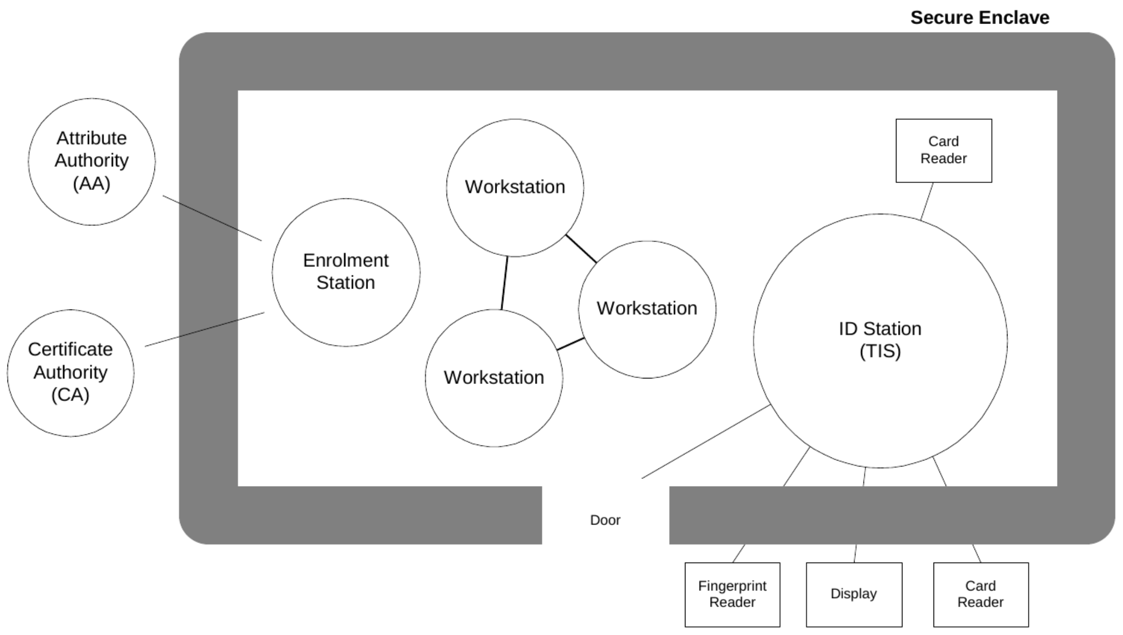

Figure 3.1: Tokeneer System Overview [50]

As already mentioned, the TIS itself is a part of the much larger Tokeneer system. Its main function is to provide security for the components inside the **Secure Enclave** (represented by the grey border). This is achieved by providing the following core functionality [50, §2]:

Physical Security - The TIS is responsible for controlling the only physical entry point into the enclave, the **Door**. This involves locking and unlocking the **Latch** of the door. Additionally the system is in control of an audible **Alarm**. This Alarm is sounded in the event that the security of the system is compromised.

The three aforementioned components are represented by the Door component shown in Figure 3.1. These are all controlled by the **ID Station** component.

User Enrolment and Management - The **Enrolment Station** can be used by administrators to enrol other users into the system. During enrolment, the **Attribute Authority** and **Certificate Authority** components are used to generate certificates that hold various metadata such as the user's clearance level. Further discussion on the topic of certificates is outside the scope of this report but can be found in [38, 39, 50].

These certificates are combined into a **token** which is stored onto a smart card.

User Entry - Using the physical security functions of the system, the TIS is responsible for allowing a user access into the enclave. In order to gain access, a user has first to present a card with a valid token to the **Card Reader** followed by a valid fingerprint to the **Fingerprint Reader**. The latter step is also known as **biometric** validation.

These two peripherals are controlled by the ID Station as shown in Figure 3.1.

Workstation Access - Upon successful entry into the enclave, the token is appended with an **Authorisation Certificate**. This is used by a **Workstation** to determine if the user is allowed access to its functions.

Auditing - The TIS is also responsible for keeping an audit log of every event that occurs in the system for later analysis. This forms a part of the ID Station.

From Figure 3.1 it can be seen that the TIS is comprised of three main components: the Enrolment Station, Workstations and the ID Station. Only the ID Station and its required peripherals are considered for implementation in SCJ. This component is chosen as it performs the core functions of the TIS system and is the main component implemented by Praxis themselves. The functions of the Workstations were provided by a third-party. Similarly the Enrolment Station uses third-party cryptographic libraries to perform its main functions [50, §2.2].

3.2 Formal Specification

A formal specification uses mathematical notation to define the functionality of a system unambiguously, without associating it with any specific implementation details [51]. Clearly this benefits the development of safety-critical systems, which must undergo rigorous verification before they can be deployed.

Using a formal specification allows all parties involved in the development of a system, including the customer, to have a clear overview of the entire system without any code written [32]. This allows design flaws to be discovered earlier in development, leading to an overall reduction in development costs [52].

During the development of the original TIS, a formal functional specification has been produced by Praxis using the Z modelling language outlining the expected behaviour of the system [39]. This comprehensive document specifies details ranging from what configuration data the system requires, to the operations that result in an entry being added to the audit log. During the later stages of development, this document is referenced during testing and reviews to verify correctness of the produced code base and other deliverables.

The formal specification is also notable for its inclusion of informal commentary by the original developers. This allows readers who are unfamiliar with Z to read and understand the specification. However, knowledge of Z notation gives context to this informal commentary [39, §2.3.2]. As such, a brief introduction to the Z notation is given.

3.2.1 The Z Modelling Language

As mentioned above, the TIS formal specification is written using the Z notation, a formal specification language based on standard mathematical notation used in set theory and first-order predicate logic. Though the developers give informal commentary about each aspect of the specification, understanding the basics of Z will give context to the commentary and aid in understanding.

For illustration, consider the ‘Birthday Book’ example originally outlined by Spivey [51]. This example describes a system used to record people’s birthdays using the Z notation.

Firstly, the basic types of the system are declared. In Listing 3.1 the set of all names and dates are defined for the system. We note that this declaration does not say anything about how these types may be represented or the information they hold.

$[NAME, DATE]$

Listing 3.1: Z Basic Type Declaration

Using these type declarations, the schema shown in Listing 3.2 can be defined.

```
── Map ──────────────────────────────
  known : ℙ NAME
  birthday : NAME ⇸ DATE
├──────────
  known = dom STRING
└────────────────────────────────────
```

Listing 3.2: Birthday Book Z Schema

A Z schema consists of a number of declarations followed by a number of constraints if applicable. In the above schema, there are two declarations: $known$ and $birthday$. The $known$ parameter represents the set of all names with which birthdays are recorded.

$birthday$ represents a function, which outputs the birthday associated with a name. Note the constraint dictates that the set of $known$ names is the same as the domain of $birthday$. Therefore all values in the set of $known$ can applied to the $birthday$ function to define a birthday.

A full overview of the Z notation and applications in formal methods is outside the scope of this report. Further discussion on the use of Z for formal specification can be found in [51, 52]. We only explain the notation as needed.

3.3 The TIS Formal Specification

As stated earlier in the chapter, due to the size of the TIS system it is not possible to present all aspects of the original system in detail. Indeed the formal specification itself is comprised of approximately 100 pages. Instead this report will present two core features of the TIS that will be developed in SCJ. These features are:

- The user entry process, and
- The control of physical security (the door mechanism)

Whilst both features can be implemented independently of each other, they both depend on each other to a degree. For instance, we consider the case where the door control mechanism has not been implemented. The user entry process can still determine if a user is authorised to enter the secure enclave via token and biometric validation. However after authorisation has occurred, there could be no further progression as the functionality to unlock the enclave is not present. Therefore it is logical to implement both these features.

The following section presents these features, along with an overview of their operation and their formal specification. This forms the Inception phase of the Rational Unified Process (RUP) software development methodology. This phase identifies the core requirements and functionality required by the project. Subsequently, the information gathered in this phase will be used to inform the Elaboration and Construction phases, where design and implementation of the system takes place.

3.3.1 Control of Physical Security

One of the critical functions handled by TIS is the control of the peripherals that affect the security of the system. As previously noted in subsection 3.1.1, these peripherals are the **Door**, the **Latch** and the **Alarm**. Each peripheral can be in one of two states. These states are formally defined in Listing 3.3.

$$\begin{array}{l} DOOR ::= open \mid closed \\ LATCH ::= unlocked \mid locked \\ ALARM ::= silent \mid alarming \end{array}$$

Listing 3.3: Peripheral States [39, §2.7.1]

Additionally, the system specifies a **latch and an alarm timeout**. These timeouts dictate when the latch relocks and when the alarm sounds.

The combination of these states represent the current security status of the TIS. This aggregation of states along with the timeouts are defined by the Z schema shown in Listing 3.4.

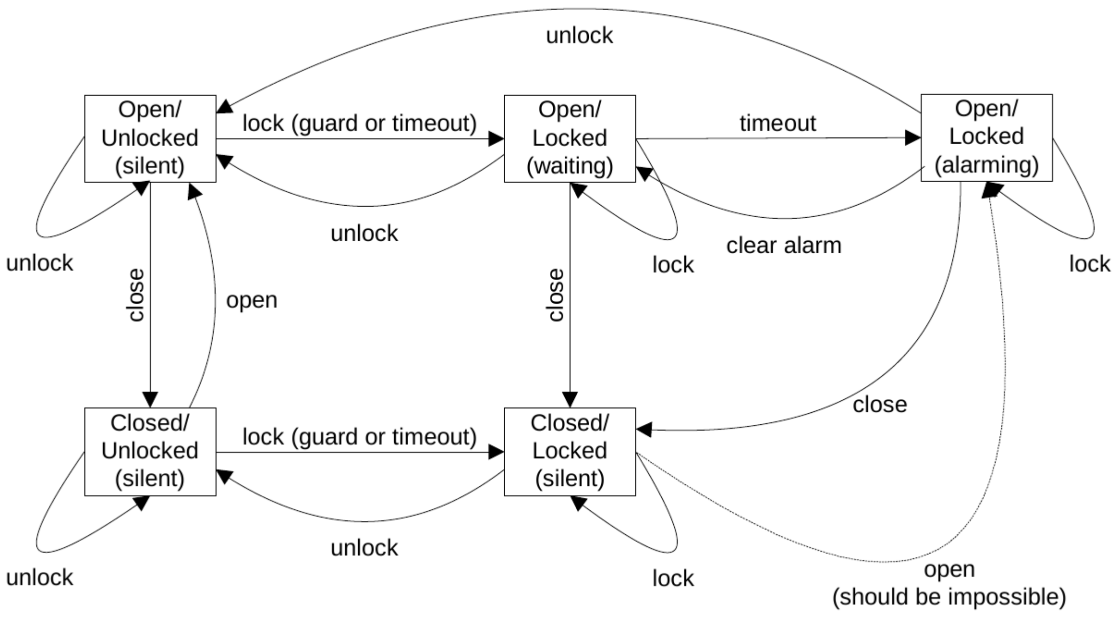

Figure 3.2: System Security State Transition [38, §7.2]

$$
\begin{array}{|l}
\hline
\textit{DoorLatchAlarm} \\
\textit{currentTime} : \textit{Time} \\
\textit{currentDoor} : \textit{DOOR} \\
\textit{currentLatch} : \textit{LATCH} \\
\textit{doorAlarm} : \textit{ALARM} \\
\textit{latchTimeout} : \textit{TIME} \\
\textit{alarmTimeout} : \textit{TIME} \\
\hline
\textit{currentLatch} = \textit{locked} \Leftrightarrow \textit{currentTime} \geq \textit{latchTimeout} \\
\textit{doorAlarm} = \textit{alarming} \Leftrightarrow \\
\quad (\textit{currentDoor} = \textit{open} \\
\qquad \wedge \textit{currentLatch} = \textit{locked} \\
\qquad \wedge \textit{currentTime} \geq \textit{alarmTimeout} \\
\hline
\end{array}
$$

Listing 3.4: State of Door Mechanism [39, §3.6]

The two constraints of this schema show that the state of the Latch and Alarm can be derived from whether the latch timeout has been fired. The first conjunct defines that the latch must be locked if the latch timeout has elapsed. The second conjunct defines that the alarm must be active if the door is open, the latch is locked and the alarm timeout has elapsed.

The possible state combinations can be visualised in the state transition diagram seen in Figure 3.2.

The combination of the individual states of each peripheral corresponds to a single state in the diagram. For instance, the state **Open/Locked (alarming)** represents the door being open, the latch being locked and the alarm sounding. All states, with the exception of the **Closed/Locked (silent)** state, denote that the system is currently in an insecure state.

Closed/Locked (silent) - When the system is in this state, it is secure. Generally, the system only changes to this state upon successful completion of a user entry operation. This will cause the system to transition to a **Closed/Unlocked (silent)**. When this transition occurs, the latch and alarm timeouts are started.

Note that an implementation should enforce the inability to open the door when the system is in this state.

Closed/Unlocked (silent) - In this state, the door is unlocked and able to be opened. If the door does not open before the latch timeout is exceeded, the latch will lock itself, moving the system back into the secure **Closed/Locked (silent)** state.

Once the door has been opened, the system moves to the **Open/Unlocked (silent)** state.

Open/Unlocked (silent) - Physical entry into the secure enclave is possible in this state. The system moves back to the **Closed/Unlocked (silent)** state when the door is closed. If the door is left open for long enough, the latch timeout will expire and the system will transition to the **Open/Locked (waiting)**.

Open/Locked (waiting) - In this state the alarm is still silent but, is waiting for the alarm timeout to expire. If the door is closed, then the alarm timeout is reset and the system moves to the **Closed/Locked (silent)**.

When the alarm timeout is exceeded, then the alarm is activated and the system transitions to the **Open/Locked (alarming)** state.

Open/Locked (alarming) - If the system reaches this state, then security is compromised and action needs to be taken to re-secure the system. This can be done in two ways. The first is to simply close the door, sending the system back to the **Closed/Locked (silent)** state. The second is for the alarm timeout to be reset which transitions the system back to the **Open/Locked (waiting)** state.

From Figure 3.2 and Listing 3.4, it can be seen that the system needs to be able to perform a number of functions in order for a state transition to successfully occur. For instance, to perform the transition from the **Closed/Locked (silent)** state to **Closed/Unlocked (silent)** state, the system must be able to:

1. Unlock the door
2. Start the latch timeout
3. State the alarm timeout

By identifying the functions needed to perform each state transition, the set of requirements in Table 3.1 can be derived. This is used during the design phase to ensure the SCJ implementation will have the required functionality to successfully control the physical security of the system.

Req Key	**Req Description**
PS-1	The system MUST be able to monitor the state of the Latch.

Req Key	Req Description
PS-2	The system MUST be able to monitor the state of the Door.
PS-3	The system MUST be able to monitor the state of the Alarm.
PS-4	The system MUST be able to control the Latch by Locking and Unlocking the Latch.
PS-5	The system MUST be able to control the Alarm by turning the Alarm On and Off.
PS-6	The system MUST start a latch timeout when the Latch is unlocked. *Depends on:* PS-1, PS-4
PS-7	The system MUST start an alarm timeout when the Latch is unlocked. *Depends on:* PS-1, PS-4
PS-8	Once the latch timeout expires, the Latch MUST lock. *Depends on:* PS-4, PS-6
PS-9	Once the alarm timeout expires, the Alarm MUST activate if the Door is open and the Latch is locked. *Depends on:* PS-1, PS-2, PS-5, PS-7
PS-10	The Alarm MUST deactivate if the system detects that the Door is closed and the Latch is locked. *Depends on:* PS-1, PS-2, PS-5

Table 3.1: Physical security control requirements

The functionality outlined by these requirements are depended upon by the User Entry feature of the system.

3.3.2 The User Entry Process

The second core function the TIS to be implemented in SCJ is the authentication of users for entry into the secure enclave. If authentication is successful, the latch is unlocked. This triggers the state transitions as described in subsection 3.3.1.

The user entry process is a multi-staged operation, which can be visualised using the state transition diagram seen in Figure 3.3. This diagram has been constructed based on Z definitions and schemas that describe the user entry process. Each state represents a distinct stage in the user entry operation:

quiescent - Any user entry operation can only begin when the TIS is in an **quiescent** state. Therefore another user entry operation must not be taking place. Other operations, such as administration tasks, could also place the system in an non-idle, state but will not be considered in this report.

The user entry operation begins when a user presents a card to the card reader. The token is read off the card (the **ReadUserToken** transition), moving the operation into the **gotUserToken** state. Regardless of whether the user entry operation succeeds or fail, all possible paths will eventually lead back to this quiescent state.

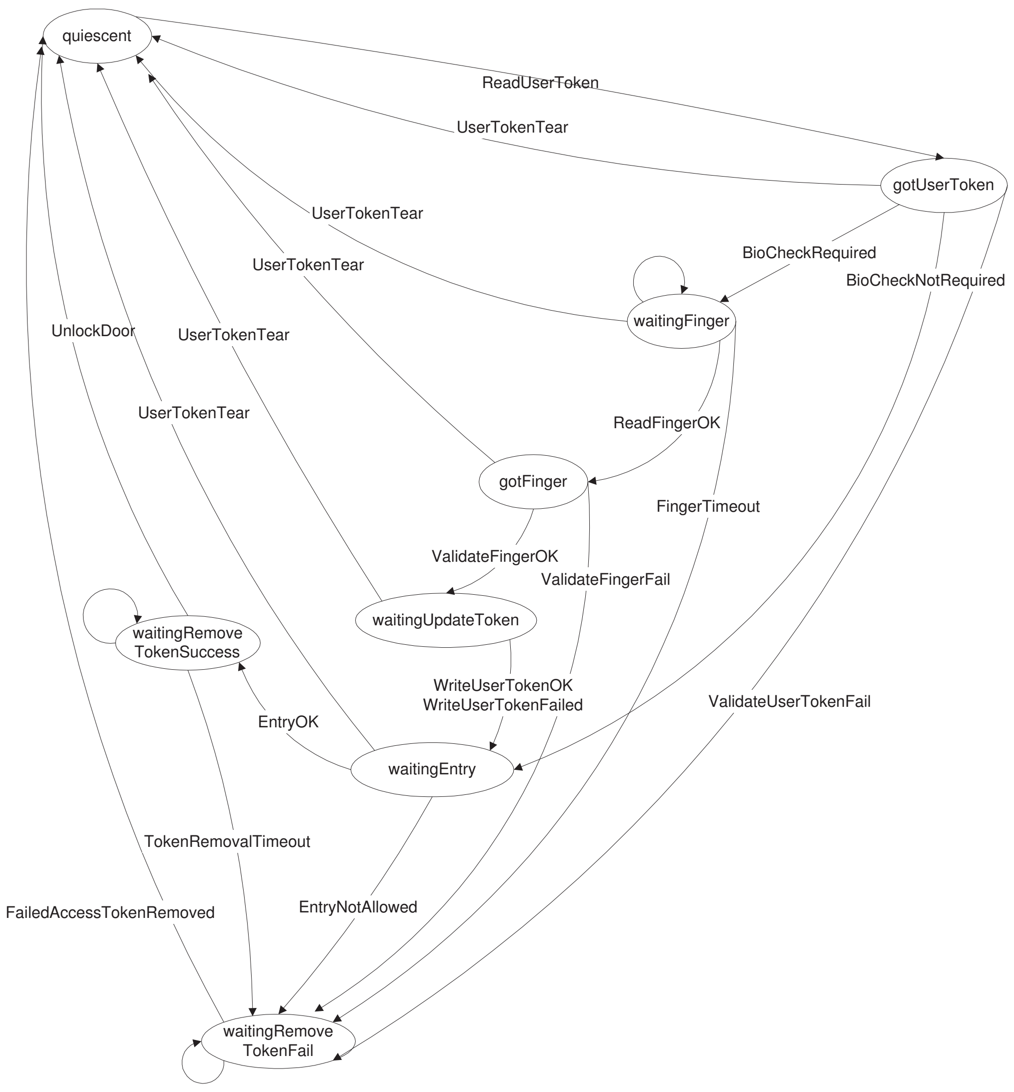

Figure 3.3: User Authentication and Entry State Transitions [39, §6]

gotUserToken - Here the read token is checked for validity. If the token fails the validity check, then the user entry operation has failed and moves to the **waitingRemoveTokenSuccess** state.

When the validity checks succeeds, there are two possible outcomes. The first, is that the user requires no further validity checks and is authorised to enter the enclave. In this case the operation moves to the **waitingEntry** state. However, a further fingerprint check will be required in most cases, moving the operation to the **waitingFinger** state.

waitingFinger - Upon transitioning to this state, the **finger timeout** is started. It is only when this timeout is active that the fingerprint reader accepts inputs. This is similar to the alarm and latch timeouts described earlier in subsection 3.3.1.

If the timeout elapses without a fingerprint being given, the user entry operation fails and moves to the **waitingRemoveTokenFail** state. Once a fingerprint is given however, the operation moves to the **gotFinger** state. After a fingerprint is given, no other fingerprints can be given for the current user entry operation.

gotFinger - The fingerprint read from the previous state, **waitingFinger**, is validated here. If the fingerprint is validated successfully, then the user has been successfully authenticated against all checks. The operation then moves to the **waitingUpdateToken** state. In the event validation was unsuccessful the operation instead moves to the **waitingRemoveTokenFail** state.

waitingUpdateToken - During this state, the token is waiting for an Authorisation Certificate to be written. As described earlier in subsection 3.1.1, this certificate is used to determine the access privileges of the user. These privileges include if the user is allowed entry to the enclave or if they are allowed access to the workstations.

Though it is possible for the certificate writing process to fail, the process always moves to the **waitingEntry** state.

waitingEntry - It is at this point the Authorisation Certificate written during the **waitingUpdateToken** state is evaluated. This determines whether the user has sufficient access privileges to enter the enclave.

If the user is allowed entry, then the operation moves to the **waitingRemoveTokenSuccess** state. Failure results in a transition to the **waitingRemoveTokenFail** state.

waitingRemoveTokenSuccess - Here the system is awaiting the user to remove the card from the card reader. Similar to the **waitingFinger** state, a timeout is started upon transitioning to this state. If the timeout expires before the token is removed, the user entry operation fails and the operation moves to the **waitingRemoveTokenFail** state.

However, if the the card is removed before the timeout has elapsed, the latch is unlocked allowing the user to open the door. The effects of unlocking the latch have already been described in subsection 3.3.1. From here the system resets back to the **quiescent** state, awaiting the next user entry attempt.

waitingRemoveTokenFail - If any stage of the user entry operation fails then the operation transitions to this state. Here the user is required to remove the card from the card reader in order to reset the system back to the **quiescent** state.

The **waitingRemoveTokenFail** state can be reached in one of three ways:

1. A user fails to give a fingerprint or remove their card before a timeout elapses
2. A user gives a fingerprint or token that is not valid
3. A user does not have sufficient privileges to enter the enclave

The case of the card being removed prematurely has also been considered. This is represented by the **UserTokenTear** transition. In this case, any currently active user entry operation is terminated regardless of its stage and the system is reset back to the **quiescent** state.

When developing the SCJ implementation, the process of updating the token with an Authorisation Certificate will be omitted. Instead it is assumed that any successfully authenticated user will be allowed entry into the enclave. This is due to the fact that cryptographic functions will not be implemented which are key to the generation of the Authorisation Certificate. Similarly token and fingerprint validation will be spoofed in the SCJ implementation.

Finally the situation where a user does not require a biometric check is also omitted. This is justified as the entry operation can reach the same state with or without the biometric check.

A set of requirements can be constructed using a similar methodology described in subsection 3.3.1. These requirements correspond to the functionality required in the system in order to perform the state transitions, as shown in Figure 3.3.

Req Key	**Req Description**
UE-1	The system MUST be able to detect a Token being placed into the Token Reader.
UE-2	The system MUST be able to detect a finger being placed into the Fingerprint Reader.
UE-3	The system MUST be able to detect a Token being removed from the Token Reader.
UE-4	The system MUST be able to read a Token from the Token Reader. *Depends on:* UE-1
UE-5	The system MUST be able to read a Fingerprint from the Fingerprint Reader. *Depends on:* UE-2
UE-6	The system MUST be able to validate a read Token. *Depends on:* UE-4
UE-7	The system MUST be able to validate a read Fingerprint. *Depends on:* UE-5
UE-8	When a Token validation fails, the token MUST be removed from the reader. *Depends on:* UE-3, UE-6
UE-9	When a Fingerprint validation fails, the token MUST be removed from the reader. *Depends on:* UE-3, UE-7
UE-10	When a Token validation succeeds, a fingerprint input timeout MUST be started. *Depends on:* UE-6
UE-11	When a Fingerprint validation succeeds, a token removal timeout MUST be started. *Depends on:* UE-7

Req Key	Req Description
UE-12	Token validation MUST only occur when the fingerprint input or token removal timeouts are not active.
UE-13	Fingerprint validation MUST only occur when the fingerprint input timeout is active. *Depends on:* UE-10
UE-14	The Latch MUST unlock if a Token is removed whilst the token removal timeout is active. *Depends on:* UE-3, UE-11, PS-1
UE-15	The Latch MUST not unlock if a Token is removed whilst the token removal timeout is inactive. *Depends on:* UE-3

Table 3.2: User entry requirements

The requirements listed in Table 3.1 and Table 3.2 form the core functionality of our implementation.

3.4 Summary

This chapter has presented an overview of the Praxis Tokeneer project including its history, the original goals of the project and an overview of the implemented system. Using the original SRS and formal specification, a subset of the functionality from the original TIS has been identified for implementation in SCJ. Finally, a set of requirements has been constructed for the chosen functionality, which will need to be fulfilled in the implementation.

4 | Design and Implementation

In this chapter details relating to the design and implementation of the SCJ TIS are given. An overview of the original TIS design is given along with a walk-through of its execution. Next an overview of the application as a whole is given, with each component being discussed in turn. This includes discussion on the final design, alternative designs considered and any implementation specific details. Where necessary code examples are given from the implementation though these will be abbreviated for brevity. Also, where applicable, any requirements fulfilled by a particular component are also given.

4.1 SCJ Reference Implementation

Our implementation is developed using the University of York's Reference Implementation (RI) of the SCJ specification. This RI is not fully compliant with the SCJ specification. As such only support for the following exists:

- Support for upto Level 1 SCJ applications
- The Mission programming model as described in subsection 2.3.1
- The memory model
- `AperiodicEvent`, `AperiodicEventHandler` and `PeriodicEventHandler`

Notably, support for forming sockets and connections to perform input/output operations has yet to be implemented. The SCJ TIS implementation will therefore bypass this limitation by providing dummy interfaces to the real world peripherals.

One other issue of concern is the lack of support for `OneShotEventHandler`. Without these handlers, the range of designs is constrained. Fortunately this was rectifiable.

4.1.1 OneShotEventHandler Implementation

Although the RI lacks support for `OneShotEventHandlers`, access to the RI source was available, support for these handlers could be implemented. This implementation conforms with the SCJ specification [27, §4.4.11]. However no formal verification has been performed as improvement of the RI is outside the scope of this report. An abbreviated version of this implementation is shown in Listing 4.1.

```
package javax.safetycritical;

import javax.realtime.AbsoluteTime;
import javax.realtime.AperiodicParameters;
import javax.realtime.HighResolutionTime;
import javax.realtime.OneShotTimer;
import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;
```

```java
public abstract class OneShotEventHandler extends
  ManagedEventHandler {

  // Timer for one shot event
  private OneShotTimer timer;

  public OneShotEventHandler(PriorityParameters priority,
    HighResolutionTime time,
    AperiodicParameters release,
    StorageConfigurationParameters storage,
    long size,
    String name) {

    super(priority, release, storage, size, name);

    if (time == null) {
      time = new RelativeTime(0, 0);
    }
    this.timer = new OneShotTimer(time, this);
  }

  public boolean deschedule() {
    final boolean isRunning = this.timer.isRunning();
    this.timer.stop();
        return isRunning;
  }

  public AbsoluteTime getNextReleaseTime(AbsoluteTime dest)
    throws IllegalStateException {
    return this.timer.getFireTime(dest);
  }

  public void scheduleNextReleaseTime(HighResolutionTime time) {
    this.timer.reschedule(time);
        this.timer.start();
  }

}
```

Listing 4.1: OneShotEventHandler implementation

This implementation uses the adapter pattern to wrap an instance of the OneShotTimer. Calls to the scheduleNextReleaseTime() and deschedule() actually delegate to the OneShotTimer to perform their respective functions.

A OneShotEventHandler will not fire until it has been scheduled for release using the scheduleNextReleaseTime() method. When this method is invoked, the internal timer is started. Once the timer ends, an AsyncEvent is fired that triggers the handlers handleEvent() method.

4.2 Original SPARK Ada Design

The original TIS is implemented using SPARK, a high-integrity subset of the Ada language [2]. SPARK is notable for its inclusion of a specific design process for applications, known as the INFORMED design process [45]. This process consists of the following steps:

Identification of System Boundary - SPARK applications are typically responsible for controlling interactions with the physical world. This steps serves to identify the components that form the real world.

Identification of SPARK Boundary - SPARK applications are usually a part of a larger system. This step serves to identify the functionality of the system that will need to be implemented using SPARK.

Identification and Localization of System State - In this step, states of data are identified. Appropriate locations for the storage and management of this state is also decided upon in this step.

Handling Initialization of State - This step serves to identify how to initialise the system.

Handling Secondary Requirements - Any additional functionality not integral to the core functionality of the system is identified.

Implementation - Full implementation of the system is performed in this step.

Although Ada does not explicitly define class constructs like Java, it still allows the use of object-oriented principles using its package and type systems. As such, the structure of the original TIS can be modelled using a class diagram, as shown in Figure 4.1. This figure has been adapted from [53], omitting components outside the scope of this project such as administration tokens.

This structure notably consists of two distinct sub-systems. The first, coloured in red, relates to the core functionality of the TIS. The second, coloured in blue, provides abstraction to the card reader interface. This is necessary due to technical limitations in the original implementation.

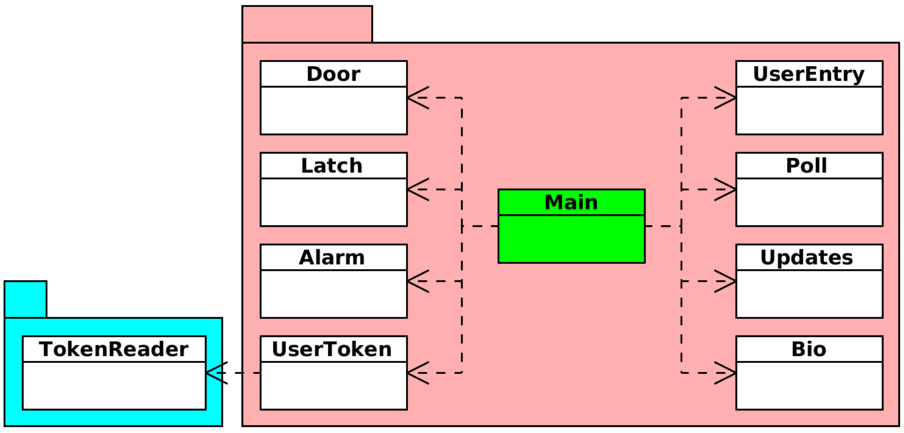

Figure 4.1: High-level class diagram of Ada TIS implementation adapted from [53]

As part of the SPARK INFORMED design process, the objects shown in Figure 4.1 can be categorised into the following package types [45]:

Main Program is the top-level entry point of the application, represented by `Main`. It is the main control point of the application.

Type Packages introduce types, such as enum definitions, and possible operations on those types without introducing a persistent state into the system. For brevity and clarity, these packages are omitted from Figure 4.1.

Variable Packages are for most purposes equivalent to objects. They introduce a persistent state to the system as well as operations to manipulate that state.

The only variable package considered in this report is the `UserEntry` package. This package acts as a state-machine, controlling the user entry operation in the system.

Boundary Variables are specific kinds of variable packages, used to provide an interface between the internal system and external real world.

Within the original design the `Door`, `Latch`, `Alarm`, `UserToken` and `Bio` are boundary variables. They are responsible for holding the state of their respective peripheral and provide methods for manipulating this state. As such they also act as a facade to lower level hardware APIs used to control the peripherals.

Utility Packages are used to provide shared services to type and variable packages. They enable the definition of methods that affects multiple type or variable packages, but would result in lowered cohesion were these methods be defined directly in a type or variable package.

Within the original design utility packages consist of:

- `Poll` - This package aggregates the various polling methods of the boundary variables into one method. This is used by the main program to update the internal state representations of the various peripherals in unison.
- `Updates` - This package aggregates various methods involved in updating the state of the peripherals in the real world. An update is equivalent to manipulating a physical peripheral, such as locking the latch.

This design models the execution of the system sequentially using a single main loop in Figure 4.2. This loop consists of four distinct phases:

Polling - This phase serves to poll all the peripherals for new data. This may change the internal state of the peripherals.

Early Updates - Using the data obtained from the **Polling** phase, updates to the critical peripherals are performed. These are the door, latch and alarm.

Processing - Any other operations that occur as part of the TIS is performed in this phase. For the purposes of this report, the only operation considered is the user entry operation. The full TIS implementation also consists of operations relating to administration and auditing.

Like the **Early Updates** phase, this phase uses the internal state obtained during the **Polling** phase to perform the operations. These operations can possibly alter the internal state of the system.

Updates - Finally, any updates to the peripherals are performed, as a result of any changes from the **Processing** phase.

From the main loop it can be seen that the original TIS is designed to be a sequential system. Therefore a direct port conforming to Level 0 of the SCJ specification is possible, operating in

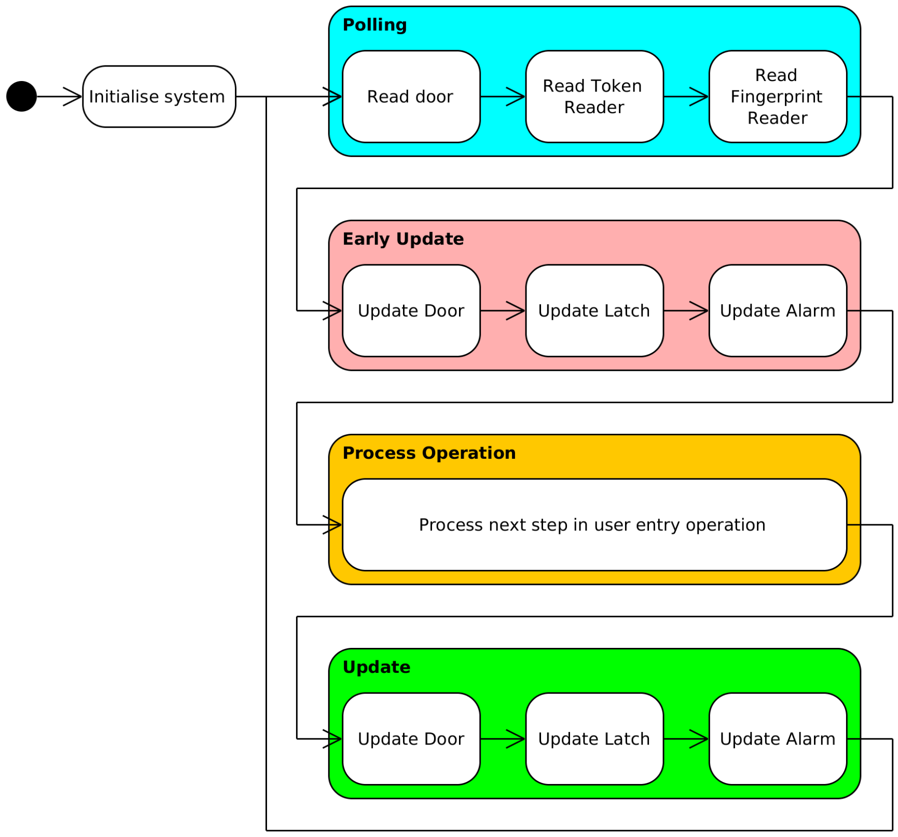

Figure 4.2: Execution flow of Ada TIS implementation

much the same way as the orginal TIS. In this model, the system utilizes a status-driven mechanism to interface with the peripherals. This involves periodically polling the status of the peripherals and then acting upon that data.

The alternative to this is to use an interrupt mechanism. In this mechanism a device notifies the system, via an interrupt, that a change in its real world state has occurred. As noted by Burns and Wellings, this type of mechanism is becoming more common than status-driven mechanisms [15]. Interrupt mechanisms are also deemed more efficient as no work is performed by the system unless a change occurs. In order to support this mechanism, aperiodic event support is required. This is feature is only supported by Level 1 and above in the SCJ specification, as mentioned previously in chapter 2.

4.3 Application Overview

The process of designing the overall system forms part of the Elaboration phase of the RUP software development methodology. The subsequent Construction phase is then undertaken to implement the design.

Like all SCJ applications, a Safelet is defined that contains all MissionSequencers and Missions. Figure 4.3 shows the structure of these components. Not shown is the structure of the handlers which is elaborated upon later in section 4.5.

Since the functionality that will be implemented occurs during only one mode of operation, only a single Mission has been defined: the TokeneerMission. In the future if other modes of operation are required, new Missions could be defined and ordered in the TokeneerMissionSequencer. For instance, a peripheral initialisation Mission could be defined to occur before TokeneerMission.

The main purpose of a Mission is to manage the handlers and other schedulable objects contained within it. In order to maintain the cohesion of the TokeneerMission, classes have been created that contain functionality required by the mission, but do not necessarily relate to the managing of handlers. These utility classes perform functions such as the management of application settings. Similarly, classes managing the interfaces to the real world peripherals are created. These are represented by the realworld package in Figure 4.3.

The structure of the application shows some similarity to the original SPARK Ada design, presented previously in section 4.2. Indeed elements from the original design and the INFORMED design process has been used to guide our own design and implementation.

For instance when combined, the TokeneerSafelet, TokeneerMissionSequencer and TokeneerMission are comparable to the Main Program component in a SPARK system. Similarly, the aforementioned utility classes offer enumeration types and shared services. As such they could be thought of as Type and Utility Packages. Following on from this analogy, the peripheral interfaces are analogous to Boundary Variables since they are used to manipulate the state of the real world and cache the state of the real world.

The classification of the event handlers in SPARK terms is more difficult. To maintain low coupling and high cohesion, the release logic of the handlers has been kept small. Generally a handler will be responsible for checking the state of the system, before delegating to more complex actions within the peripheral interfaces. As such they can be classified as Utility Packages. However, as they also dictate the execution flow of the system, they can also be considered to be part of the Main Program.

Therefore to ease design the release logic is isolated and identified as a pseudo Utility Package responsible for a single function in the system. For instance, the release logic of the AlarmTimeout handler class is responsible for checking and manipulating the state of the door, latch and alarm. Using this methodology, release logic that performs a distinct function in the system can be created. Once this release logic is created, the handlers can then be considered in terms of the Main Program. It is at this stage that the scheduling mechanics are considered.

In the subsequent sections the utility classes shown in Figure 4.3 are presented. The functions of these classes are important as they directly effect the operation of the event handlers, which are presented in section 4.5.

4.4 Utility Classes

Utility classes are used to provide shared functions required by multiple components [45, §3.4]. For instance, the functionality to manipulate the latch is required by both the physical security and user entry features.

Each of the components described in this section must be instance controlled as they are shared, directly or indirectly, by the event handlers in the application. For instance, consider the case

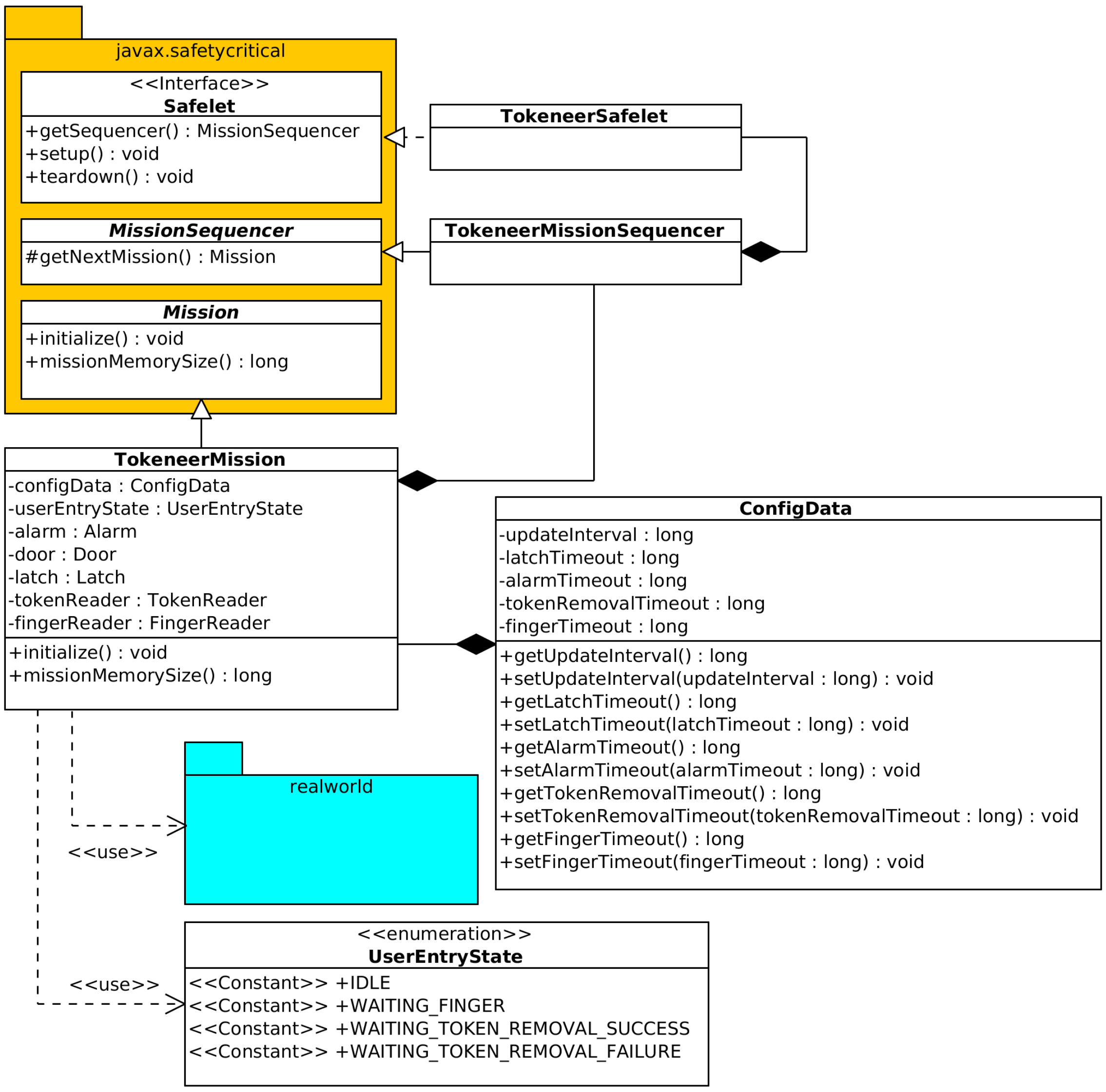
javax.safetycritical
<<Interface>>
Safelet
+getSequencer() : MissionSequencer
+setup() : void
+teardown() : void
MissionSequencer
#getNextMission() : Mission
Mission
+initialize() : void
+missionMemorySize() : long
TokeneerSafelet
TokeneerMissionSequencer
TokeneerMission
-configData : ConfigData
-userEntryState : UserEntryState
-alarm : Alarm
-door : Door
-latch : Latch
-tokenReader : TokenReader
-fingerReader : FingerReader
+initialize() : void
+missionMemorySize() : long
ConfigData
-updateInterval : long
-latchTimeout : long
-alarmTimeout : long
-tokenRemovalTimeout : long
-fingerTimeout : long
+getUpdateInterval() : long
+setUpdateInterval(updateInterval : long) : void
+getLatchTimeout() : long
+setLatchTimeout(latchTimeout : long) : void
+getAlarmTimeout() : long
+setAlarmTimeout(alarmTimeout : long) : void
+getTokenRemovalTimeout() : long
+setTokenRemovalTimeout(tokenRemovalTimeout : long) : void
+getFingerTimeout() : long
+setFingerTimeout(fingerTimeout : long) : void
realworld
<<use>>
<<use>>
<<enumeration>>
UserEntryState
<<Constant>> +IDLE
<<Constant>> +WAITING_FINGER
<<Constant>> +WAITING_TOKEN_REMOVAL_SUCCESS
<<Constant>> +WAITING_TOKEN_REMOVAL_FAILURE

Figure 4.3: Top Level Class Diagram of the SCJ TIS

where two handlers use two different instances of Latch. The application may behave incorrectly as they could control two different latches.

One method of achieving instance control is by using the Singleton design pattern [54]. This pattern is generally seen as bad design practice, often resulting in difficult to test code [13, Item 3] amongst other problems.

To prevent these types of problems from occurring, an instance of each utility class is initialised by TokeneerMission. These instances are then passed to any handler or other object during construction. This technique is know as dependency injection.

From Figure 4.3, it can be seen that the application design features the ConfigData and UserEntryState utility classes. Additionally, classes to represent the peripheral devices are provided by the realworld package.

4.4.1 ConfigData

Various values are required by the application for proper execution. For instance, a value must be defined to specify how long the alarm should wait before activating when the system is placed in an insecure state.

One design considered placing all these values as private fields in TokeneerMission. Though TokeneerMission does need to access these values, it lowers the cohesion of the class by giving it the additional responsibility of managing configuration values.

Instead, the ConfigData class was created to aggregate these configuration values. Access to these values is provided via the accessor methods of the class (as seen in Figure 4.3. There are two advantages of the chosen design. Firstly, better cohesion is achieved as the sole purpose of this class is to aggregate configuration values. Secondly, changes to the configuration values affect the entire application assuming all components depending on these values access them via the ConfigData class.

The implementation of ConfigData is a simple Plain Old Java Object (POJO) class shown in Listing 4.2.

```
public class ConfigData {
  // Default values
  private long updateInterval = 1000;
  private long latchTimeout = 5000;
  private long alarmTimeout = latchTimeout + 5000;
  private long tokenRemovalTimeout = 5000;
  private long fingerTimeout = 5000;

  /**
   * Default no argument constructor
   */
  public ConfigData() {
    ;
  }

  // Getters and Setters for each internal field
  public long getUpdateInterval() {
    return updateInterval;
  }

  public void setUpdateInterval(long updateInterval) {
    this.updateInterval = updateInterval;
  }

  // remainder of class omitted
}
```

Listing 4.2: ConfigData.java

It is comprised of internal fields and getters and setters for those fields. Synchronized access to the encapsulated fields are not required as this class is only used during the initialisation phase of the mission. During this phase, only `TokeneerMission` will access this class, using its encapsulated values to aid in the construction of the various handlers. Once constructed, the internal values of the handlers cannot be changed.

4.4.2 UserEntryState

The enum `UserEntryState` is used to represent each stage of the user entry operation. Each possible value listed in the enum corresponds to a state listed earlier in subsection 3.3.2:

IDLE - No operation currently in progress, ready to accept a token to begin user entry process

WAITING_FINGER - Token has been successfully validated, waiting for a fingerprint from the user to continue

WAITING_TOKEN_REMOVAL_SUCCESS - Token and fingerprint successfully validated, waiting for user to remove token before unlocking the latch

WAITING_TOKEN_REMOVAL_FAILURE - User entry failed, waiting for user to remove token to reset the state of the system.

A notable advantage is that enums provide compile-time type safety. It is always guaranteed that a parameter of type `UserEntyState` will be one of the constant values defined in `UserEntryState`. By having a well-defined set of possible values, programming errors are less likely to be introduced [16].

An alternative to using Java enums was to use `int` constants to represent each state. Known as the *int enum pattern*, this pattern has many disadvantages as noted by Bloch [13, Item 30]. These include the lack of compile-time safety and the possibility of using invalid values, resulting in possible programming errors.

Within the application, a single reference to `UserEntryState` is held by `TokeneerMission` as seen in Figure 4.3. This parameter represents the current stage of the user entry operation, being passed to any handler that requires the knowledge of this value.

In order to track the current stage of user entry in the entire application, a single reference to `UserEntryState` is held by `TokeneerMission`. To achieve this, the `AtomicReference` class is used from the `java.util.concurrent.atomic` package. This is acts as a wrapper for the state, providing access and mutator methods that ensure thread safety. Usage of this technique in the SCJ TIS can be seen in Listing 4.3.

```java
public class TokenInputHandler extends AperiodicEvent {

    // other parameters omitted
    private final AtomicReference<UserEntryState> state;

    // other construction parameters omitted
    public TokenInputHandler(
        final AtomicReference<UserEntryState> state) {

        this.state = state;
    }

    @Override
    public void handleEvent() {
        if (this.state.get() == UserEntryState.IDLE) {
            // Process token only if current state IDLE
            final String token = this.token.pollToken();
            if (this.isTokenValid(token)) {
                ...
                this.state.set(UserEntryState.WAITING_FINGER);
                ...
            } else {
                ...
                this.state.set(
                    UserEntryState.WAITING_TOKEN_REMOVAL_FAILURE);
                ...
            }
        }
    }

    ...
}
```

Listing 4.3: Usage of AtomicReference

The underlying implementation of the `get()` and `set()` methods of `AtomicReference` has the same effect as reading or setting a normal `volatile` variable. These Atomic classes guarantee the wrapped variable is accessed under mutual exclusion, without using locking. Additionally, the `AtomicReference` class offers convenience methods for performing atomic Compare-and-Swap (CAS) operations conveniently. The use of the atomic classes is considered good practice in Java when dealing with variables which must be used in a thread-safe manner [13]

4.4.3 Peripherals

The package `realworld` and the classes contained within represent the interfaces to the peripherals controlled by the application in the real world. The class structure of the package is shown in Figure 4.4.

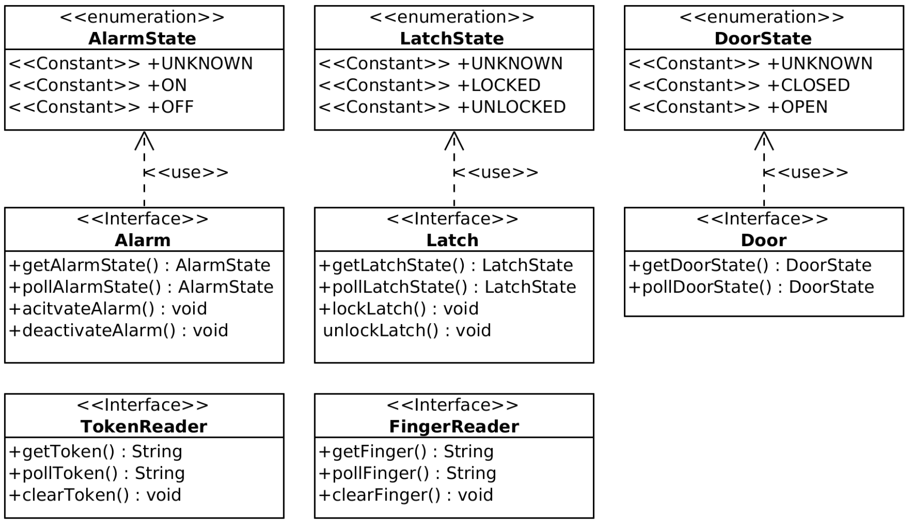

Figure 4.4: Class structure of the peripheral interfaces located in the `realworld` package

For flexibility, no concrete classes are used to represent the peripherals. Instead their functionality is defined using standard Java `interface`s. This design is advantageous as it is conceivable that the exact hardware used across application deployments could vary. As such flexibility to accommodate different implementations of the peripheral interfaces is needed, which this design provides [13, Item 18].

Each peripheral has a `poll*()` method that reads data from the peripheral itself, caches it internally and returns it. When using `poll*()`, time costs are incurred due to latency from accessing the real peripheral. Therefore a `get*()` method is provided to return a cached value from the last invocation of `poll*()`. Figure 4.5 shows the polling process.

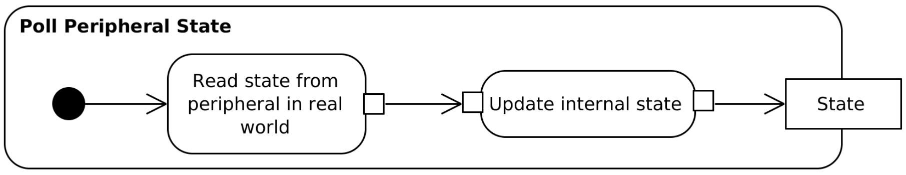

Figure 4.5: Activity diagram showing the peripheral status polling process

Where necessary the interfaces have been given the ability to manipulate the peripherals they represent. These types of control operations could potentially involve multiple complex sub-operations. As such the design is in accordance with the Facade design pattern, where a single, unified high-level interface is provided to hide a potentially complex subsystem [54].

The peripherals can be split into two distinct groups. The first group accepts no user input and consists of the `Alarm`, `Latch` and `Door`. The second group, consisting of `TokenReader` and `FingerReader`, do accept user inputs.

Alarm, Latch and Door

As the `Alarm`, `Latch` and `Door` do not accept user input there are a finite number of states they can be in. These states are represented using the `AlarmState`, `LatchState` and `DoorState` enums. The advantage of using enums to represent state is discussed earlier in subsection 4.4.2. In addition to the possible states defined in subsection 3.3.1, the `UNKNOWN` state is also defined. This state is used when the system cannot make connection to the peripheral device.

Additionally the `Alarm` and `Latch` interfaces feature methods that manipulate the state of their respective peripherals.

Requirements Fulfilled: PS-1, PS-2, PS-3, PS-4, PS-5

TokenReader and FingerReader

The `TokenReader` and `FingerReader` differ from the other peripheral interfaces as they are able to read data from the real world. As full token and fingerprint validation will not be implemented, their `poll()` methods return the `String` type.

For simplicity, the token and fingerprint data is represented using a `String`. In the future, specific objects should be made for these data types.

Requirements Fulfilled: UE-4, UE-5

4.5 Handlers

Since the application is designed to comply with level 1 of the SCJ specification, only asynchronous event handlers are permitted. These handlers are used to encapsulate the main execution logic of the application.

As SCJ primarily uses an event based programming paradigm, as discussed in subsection 2.3.2 and [27, §4.4], each handler is designed to handle a specific event or function. These events and functions are derived using the requirements listed in Table 3.1 and Table 3.2. For instance, the requirements UE-3, UE-14 and UE-15 all relate to functionality required when a token is removed. Therefore it is logical to group this functionality into one handler.

An overview of all handlers and related components used in the SCJ TIS can be seen in Figure 4.6. These are all constructed and initialised during the **Initialization** phase of `TokeneerMission`.

Handlers, and their related components, can be grouped by feature. However, there is some overlap between these components. As mentioned earlier in chapter 3, the user entry operation depends on the physical security control feature. This can be seen in the usage dependencies between `TokenRemovalHandler`, `AlarmTimeout` and `LatchTimeout`.

The remainder of this section will present the design of the components shown in Figure 4.6.

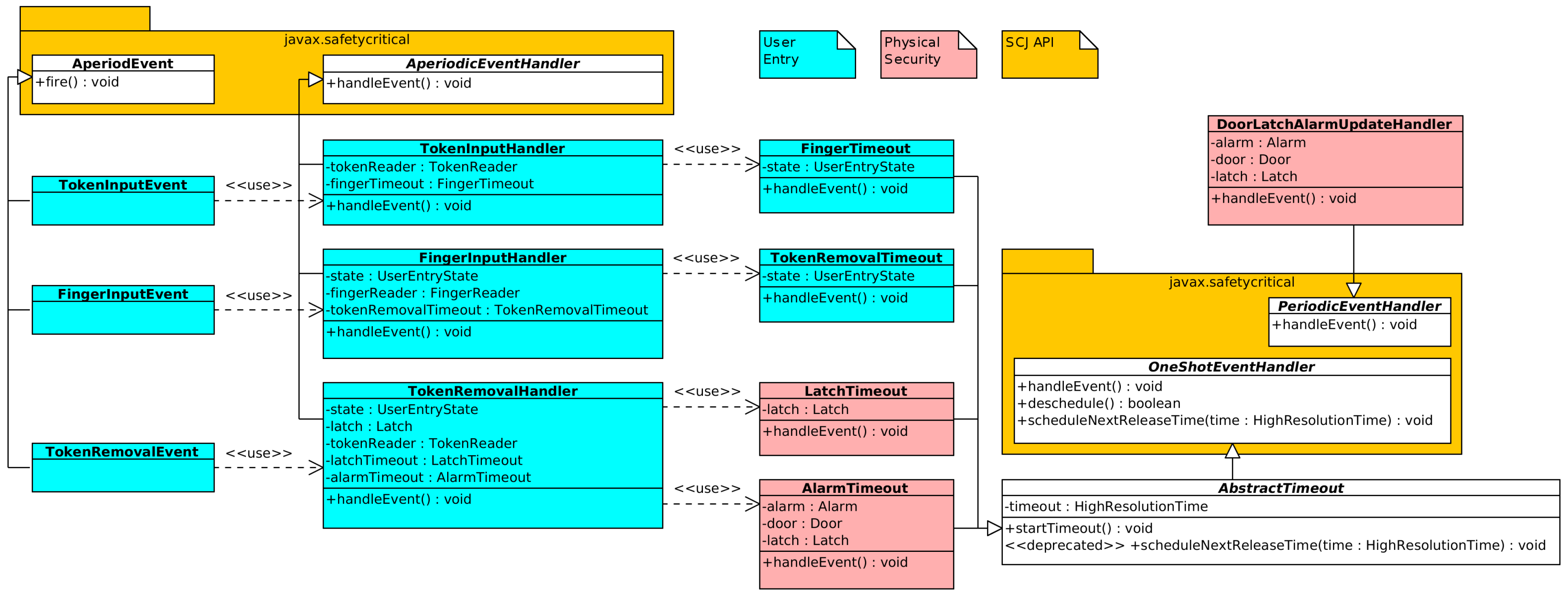

Figure 4.6: Class diagram of event handlers and related components

4.5.1 Timeouts

Both the security control and user entry features require the use of timeouts, as described earlier in subsection 3.3.1 and subsection 3.3.2. These are represented by a subclass of `OneShotEvent-Handler`, as shown in Figure 4.6.

As previously discussed in subsection 2.4.1, the `OneShotEventHandler` has been introduced to the SCJ specification specifically to allow the use of watchdog timers. A timeout can therefore be started by scheduling a release of the handler for some time in the future. A handler reaching its release denotes the timeout expiring. An alternative is to have a handler periodically poll the current time, executing its release logic when a specific time is reached. As noted by Singh et al., this has several disadvantages [28].

A skeletal implementation of a generic timeout exists in the form of `AbstractTimeout`, a subclass of `OneShotEventHandler`. The general execution of a timeout can be seen in Figure 4.7.

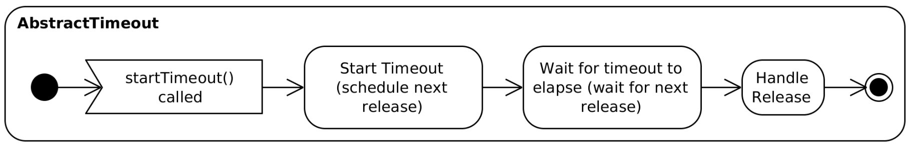
AbstractTimeout
startTimeout() called
Start Timeout (schedule next release)
Wait for timeout to elapse (wait for next release)
Handle Release

Figure 4.7: Activity diagram showing the general execution of an `AbstractTimeout` implementation

`scheduleNextReleaseTime()` has been deprecated in favour of a new method, `startTimeout()`. This is because invocation of `scheduleNextReleaseTime()` allows the specification of a new time-out value, which could affect execution unexpectedly.

To prevent these errors, `startTimeout` has been defined. This method delegates to the original `scheduleNextReleaseTime()` present in `OneShotEventHandler`, using a time interval defined during construction of the timeout. This parameter is stored in the `timeout` field. The implementation areas of interest is shown in Listing 4.4.

```
// Timeout value set during construction
private final HighResolutionTime timeout;

/**
 * Start this Timeout. Once the timeout has elapsed,
 * the {@link #handleEvent()} method will be called.
 */
public void startTimeout() {
  super.scheduleNextReleaseTime(timeout);
}

/**
 * @deprecated Use {@link #startTimeout()}.
 */
@Override
public void scheduleNextReleaseTime(
  HighResolutionTime time) {

  throw new UnsupportedOperationException(
        "Use␣startTimeout()␣method");
}
```

Listing 4.4: AbstractTimeout.java

In addition to being deprecated, `scheduleNextReleaseTime()` is overridden to throw an `Unsupported-OperationException` to forbid its use. This prevents any other classes from accidentally calling this method, without affecting the `scheduleNextReleaseTime()` method present in the super class OneShotEventHandler.

Four generalizations of `AbstractTimeout` exist, each representing a timeout used in the system. These are `LatchTimeout`, `AlarmTimeout`, `FingerTimeout` and `TokenRemovalTimeout`.

4.5.2 Physical Security

The physical security control feature consists of controlling the door, latch and alarm peripherals, as discussed previously in subsection 3.3.1. Functionality to control and monitor these peripherals are already managed by the `Door`, `Latch` and `Alarm` objects respectively. Consequently, the handlers are comprised of these objects, delegating the actual control of physical security to these objects.

The components of this feature are comprised of `AlarmTimeout`, `LatchTimeout` and `Door-LatchAlarmUpdateHandler`, highlighted in red in Figure 4.6.

LatchTimeout

The release logic of `LatchTimeout` is straightforward. Its sole function is to lock the latch on expiration of the timeout. An activity diagram of the `LatchTimeout`s control flow is shown in Figure 4.8.

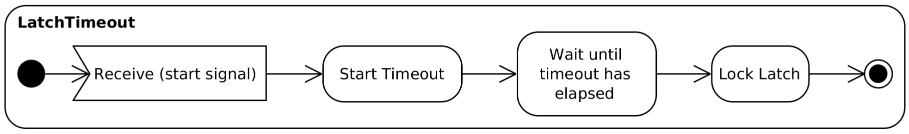

Figure 4.8: Activity diagram showing control flow of `LatchTimeout`

This timeout is started whenever the `Latch` component is unlocked. Its release logic, executed on expiration of the timeout is shown in Listing 4.5.

```
@Override
public void handleEvent() {
  this.latch.lock();
}
```

Listing 4.5: LatchTimeout.java, handleEvent()

Requirements Fulfilled: PS-6, PS-8

AlarmTimeout

Upon expiration of the AlarmTimeout, the security of the system is checked. If it is found to be insecure, then the alarm is triggered. As defined in subsection 3.3.1, the system is insecure if the door is open or the latch is unlocked. An activity diagram of the AlarmTimeouts control flow is shown in Figure 4.9.

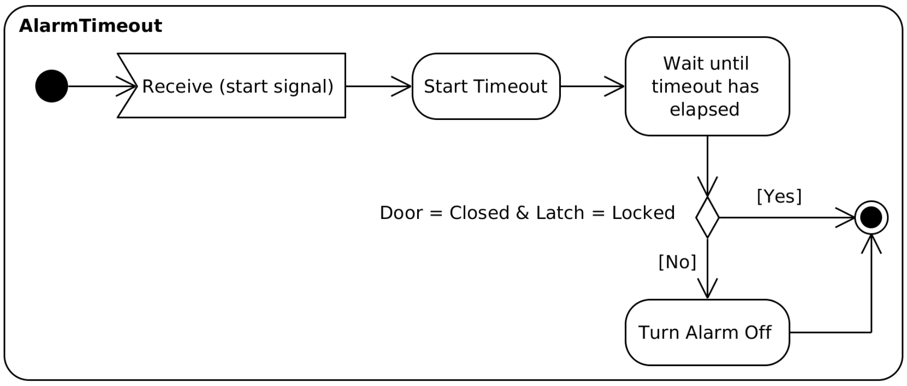

Figure 4.9: Activity diagram showing control flow of AlarmTimeout

This timeout is started whenever the Latch component is unlocked.

```
@Override
public void handleEvent() {
  if ( !(this.door.getState() == DoorState.CLOSED
          && this.latch.getState() == LatchState.LOCKED)) {
    this.alarm.activate();
  }
}
```

Listing 4.6: AlarmTimeout.java, handleEvent()

The implementation of the release logic of this timeout is shown in Listing 4.6. A test is performed to determine the security status of the system. If this test fails, then the alarm is activated. The test uses cached states which is updated at regular intervals by DoorLatchAlarmUpdate-Handler.

Requirements Fulfilled: PS-7, PS-9

Monitoring Physical Security

A PeriodicEventHandler, DoorLatchAlarmUpdateHandler, is used periodically poll states of the Door, Latch and Alarm in the real world. This serves two purposes.

The first is to turn off any active alarm if the system is re-secured from an insecure state. The second is to perform poll() methods of the Door, Latch, Alarm peripherals. This serves to refresh the cached internal state of these peripherals, for use by other components. As previously mentioned, this reduces the time costs incurred by doing a real world read. An activity diagram modelling the execution of the handler is shown in Figure 4.10.

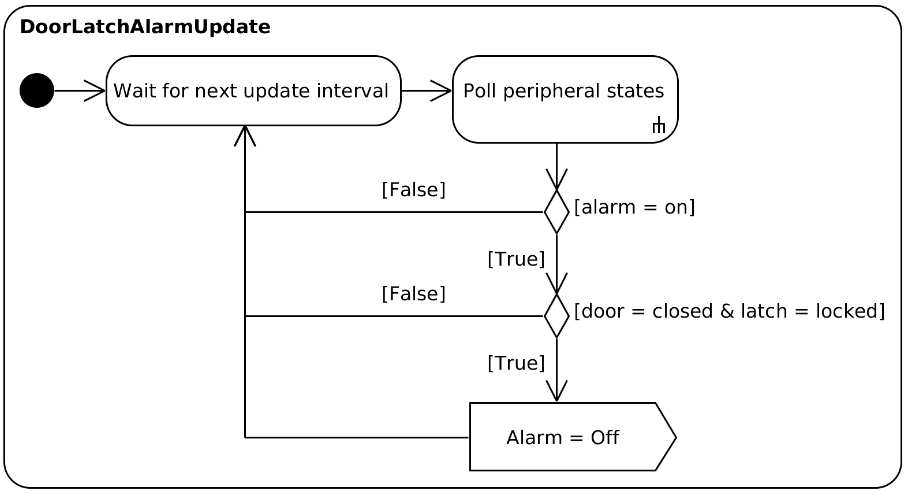

Figure 4.10: Activity diagram showing control flow of DoorLatchAlarmUpdate

All other handlers that require knowledge of the state of the Door, Latch or Alarm in the real world relies on DoorLatchAlarmUpdateHandler executing regularly. As such the update interval must be sufficiently small so that cached states that do not accurately reflect the real world are used.

The release logic of this handler is shown in Listing 4.7. This implementation acts as described previously.

```
// Set during construction
private final Alarm alarm;
private final Door door;
private final Latch latch;

@Override
public void handleEvent() {
  // Update internal state representation
    final AlarmState alarmState = this.alarm.pollState();
        final DoorState doorState = this.door.pollState();
        final LatchState latchState = this.latch.pollState();

  // Deactivate alarm if system is secured
  if (alarmState == AlarmState.ON
      && doorState == DoorState.CLOSED
      && latchState == LatchState.LOCKED) {
    this.alarm.deactivate();
  }
}
```

Listing 4.7: DoorLatchUpdateHandler.java, handleEvent()

An alternative design was considered where interrupts were used to detect changes in the states of the Door, Latch and Alarm in the real world. However, as the status of these peripherals is critical to the security of the system, a polling mechanism is used to ensure the state of the real world is always up-to date.

Requirements Fulfilled: PS-1, PS-2, PS-3, PS-10

4.5.3 User Entry

Like the physical security feature, the user entry feature uses handlers to control the flow of execution for the user entry process. Figure 4.11 shows a collaboration diagram between user input events, the handlers and the peripheral interfaces.

External events are represented in the system using `AperiodicEvents`. Each event is registered to a `AperiodicEventHandler` which handles the triggering of the event. From Figure 4.6 and Figure 4.11 it can be seen that there are three distinct external events that will be handled by the system. Each event corresponds to an user input into the system as described in subsection 3.3.2.

Additionally timeouts are defined, specifying a window for when certain inputs can be accepted by the system. These are started as the user entry process is advanced by the handlers, as seen in Figure 4.11.

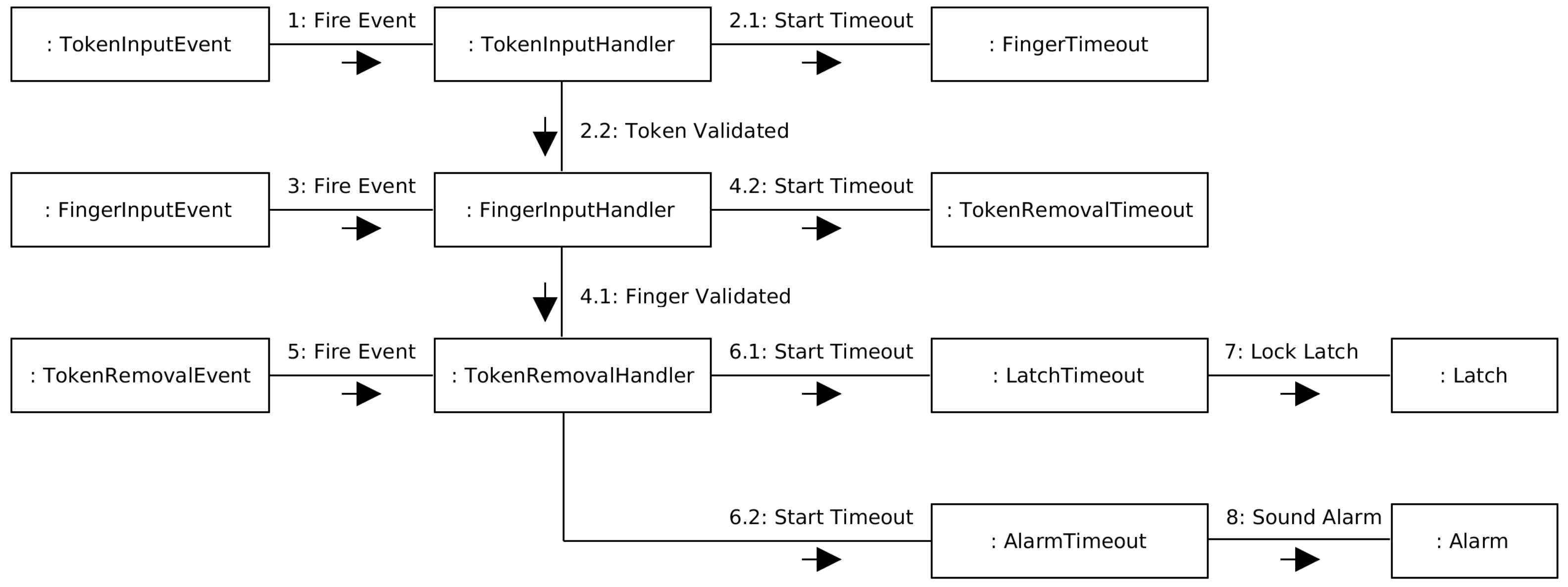

Figure 4.11: Communication diagram showing interaction between handlers

FingerTimeout and TokenRemovalTimeout

Both the `FingerTimeout` and `TokenRemovalTimeout` serve to prevent the system from idling during a user entry operation. They both define a window where the user can provide certain inputs before the system considers the entry operation as failed.

As its name suggests, `FingerTimeout` defines the window for when a fingerprint can be given. Similarly `TokenRemovalHandler` defines the window in which the token must be removed after a successful authentication before the latch unlocks.

Activity diagrams for both timeouts are shown in Figure 4.12 and Figure 4.13. It can be seen that both timeouts are nearly identical in operation.

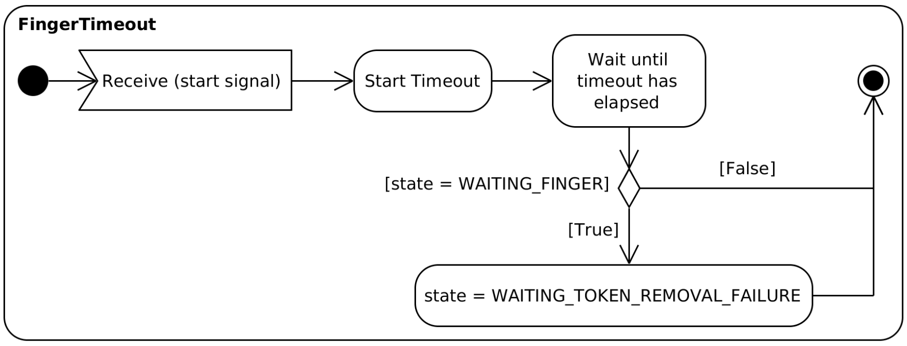

Figure 4.12: Activity diagram showing control flow of `FingerTimeout`

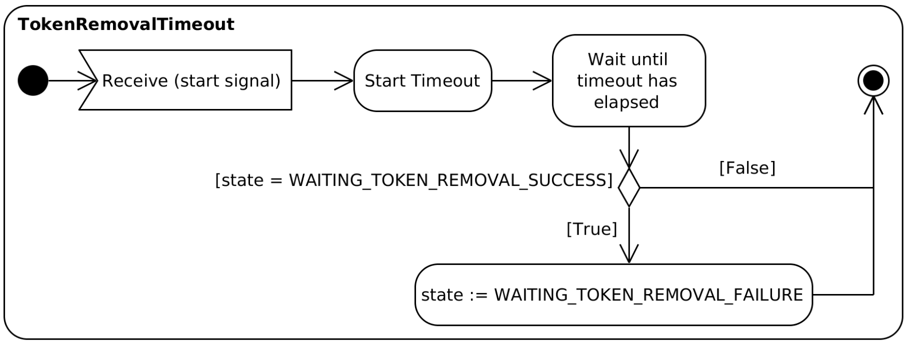
TokenRemovalTimeout
Receive (start signal)
Start Timeout
Wait until timeout has elapsed
[state = WAITING_TOKEN_REMOVAL_SUCCESS]
[False]
[True]
state := WAITING_TOKEN_REMOVAL_FAILURE

Figure 4.13: Activity diagram showing control flow of `TokenRemovalTimeout`

Upon release, both timeouts check the current state of the user entry operation. If the state has not changed since the starting of the timeout, then the entry operation has failed. The system is then able to proceed to the failure state of the entry operation, requiring the user to remove their token in order to reset the system.

However, if the state has changed since the start of the timeout, then no action needs to be taken. This path of execution denotes that the required user input had been given before the expiration of the timeout.

The implementation of the release logic for these timeout handlers can be seen in Listing 4.8 and Listing 4.9.

Upon release of these timeout handlers, the current state is checked to see if the relevant user

```
// Current state of user entry
// Set during construction
private final AtomicReference<UserEntryState> state;

public void handleEvent() {
  this.state.compareAndSet(UserEntryState.WAITING_FINGER,
    UserEntryState.WAITING_TOKEN_REMOVAL_FAILURE);
}
```

Listing 4.8: FingerTimeout.java, handleEvent()

```
// Current state of user entry
// Set during construction
private final AtomicReference<UserEntryState> state;

public void handleEvent() {
  this.state.compareAndSet(
    UserEntryState.WAITING_TOKEN_REMOVAL_SUCCESS,
    UserEntryState.WAITING_TOKEN_REMOVAL_FAILURE);
}
```

Listing 4.9: TokenRemovalTimeout.java, handleEvent()

input event has occurred. If not, then the state is changed to `WAITING_TOKEN_REMOVAL_FAILURE`.

As noted earlier, the current state of the user entry operation is tracked using an `AtomicReference`. A CAS operation is used to determine if the current state of the user entry needs to be changed. If the current state is not equal to the first parameter of the CAS operation, then no change to state occurs. This is equivalent to the user having given a fingerprint or removed the token in good time.

Requirements Fulfilled: UE-10, UE-11, UE-12, UE-13

Token Input

A user inserting a token into the `TokenReader` is represented by the `TokenInputEvent`. The firing of this event is handled by the `TokenInputHandler`. The control flow of this handler is shown in Figure 4.14. Though the handler will always be notified of a user inserting a token, the event will only be processed if there is no user entry operation currently taking place.

This handler is also responsible for calling the necessary functions to validate a given token. If the token is valid it will move the system to the next stage of the user entry operation: fingerprint validation. This causes the `FingerTimeout` to start and readies the system for finger input. The user entry operation is progressed further when a fingerprint input is received or if the timeout elapses.

The implementation of the release logic of this handler is shown in Listing 4.10. This implementa-

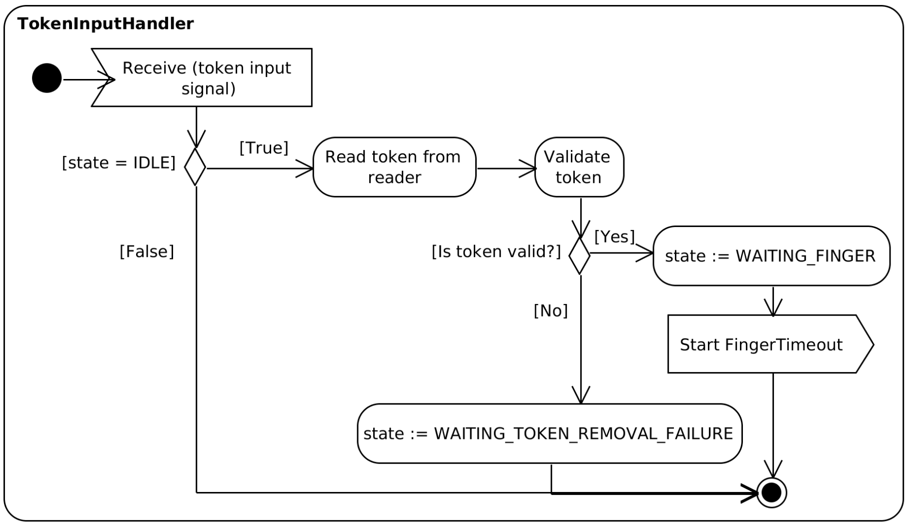

Figure 4.14: Activity diagram showing control flow of TokenInputHandler

tion follows the same execution flow outlined in Figure 4.14.

```java
// Set during construction
// Current state of user entry
private final AtomicReference<UserEntryState> state;
private final TokenReader tokenReader;
private final FingerTimeout fingerTimeout;

@Override
public void handleEvent() {
  if (this.state.get() == UserEntryState.IDLE) {
    final String token = this.token.pollToken();
    if (this.isTokenValid(token)) {
      this.state.set(UserEntryState.WAITING_FINGER);
      this.fingerTimeout.startTimeout();
    } else {
      this.state.set(UserEntryState.WAITING_TOKEN_REMOVAL_FAILURE);
    }
  }
}

private boolean isTokenValid(String token) {
  // In real scenario calls to Token Verification API
  // For simulation purposes, token is valid if True
  return Boolean.valueOf(token);
}
```

Listing 4.10: TokenInputHandler.java, handleEvent()

Since the cryptographic functions of the TIS are not considered, a token is said to be valid if it equal to the "true" string. All other strings are considered to be false. In a realistic implementation calls to the cryptography API would be performed to validate the token. Consequently, the time cost incurred from doing this would need to be analysed.

Requirements Fulfilled: UE-1, UE-4, UE-6, UE-8, UE-10, UE-12, UE-13

Fingerprint Input

A user inserting a token into the FingerReader is represented by the FingerInputEvent. The firing of this event is handled by the FingerInputHandler. The control flow of this handler is shown in Figure 4.15.

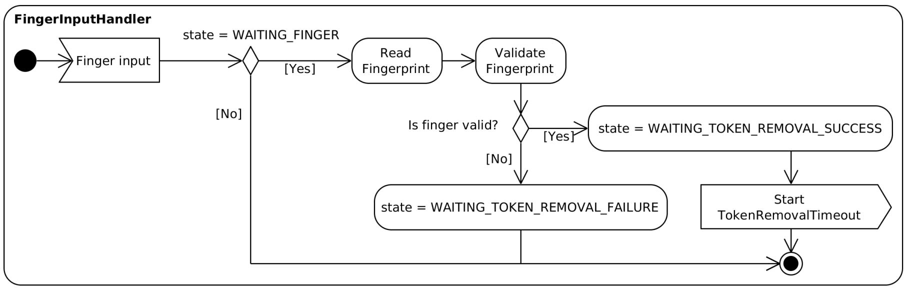

Figure 4.15: Activity diagram showing control flow of FingerInputHandler

The FingerInputHandler is similar in operation to the TokenInputHandler. Firings of FingerInputEvent will only be fully handled when the current state of the user entry operation is WAITING_FINGER. This state can only be reached by having TokenInputHandler successfully validate a token as shown in Figure 4.14.

Upon being notified of a finger input in the correct state, the handler will call the necessary methods to validate the fingerprint. If validation is successful the TokenRemovalTimeout will start and the system will wait for token removal to occur. Unsuccessful validation will result in the system to waiting for token removal only.

Additionally this handler is indirectly effected by the FingerTimeout. Once this timeout has elapsed, the state is changed to a failure state. This handler is then prevented from handling events even if a successful token validation occurred previously.

The implementation of this handler is similar to that of TokenInputHandler and is shown in Listing 4.11.

```
// Set during construction
// Current state of user entry
private final AtomicReference<UserEntryState> state;
private final TokenReader tokenReader;
private final FingerTimeout fingerTimeout;

@Override
public void handleEvent() {
  // Process the fingerprint only if in the correct step of entry
  if (this.state.get() == UserEntryState.WAITING_FINGER) {
    final String finger = this.fingerReader.pollFinger();
```

```
        if (this.isFingerValid(finger)) {
            this.state.set(UserEntryState.WAITING_TOKEN_REMOVAL_SUCCESS);
            this.tokenRemovalTimeout.startTimeout();
        } else {
            this.state.set(UserEntryState.WAITING_TOKEN_REMOVAL_FAILURE);
        }
    }
}

private boolean isFingerValid(String finger) {
    return Boolean.valueOf(finger);
}
```

Listing 4.11: FingerInputHandler.java, handleEvent()

Like the validation of tokens, a fingerprint is deemed valid if it is equal to the "true" string. As previously mentioned, were a full cryptographic library available then this handler would delegate validation of fingerprints to the relevant library methods.

Requirements Fulfilled: UE-2, UE-5, UE-7, UE-9, UE-11

Token Removal

Whenever a token is removed from the reader, the `TokenRemovalEvent` is fired. This is handled by the `TokenRemovalHandler`. As seen in Figure 4.16, the effect of a token removal is determined by the current state of the user entry operation.

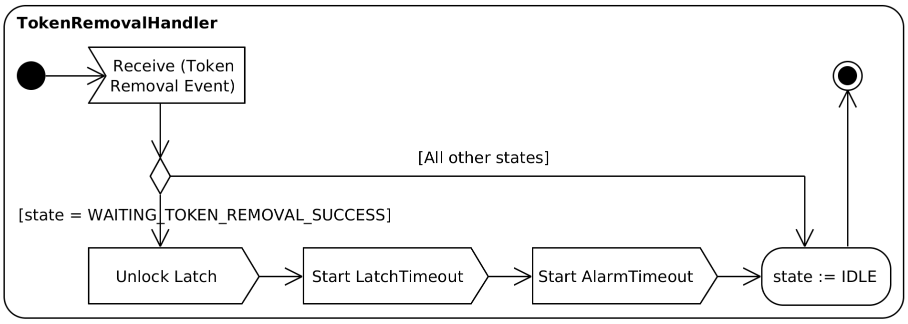

Figure 4.16: Activity diagram showing control flow of `TokenRemovalHandler`

In all but one case, a token removal will cancel the current user entry operation. This will reset the system and ready it for another user entry attempt.

The other path of execution can only be reached by having the system successfully validate a token and fingerprint. In this case, authentication has been successful and the user is required to remove their token to unlock the door latch.

Note that this path of execution is also responsible for starting the `AlarmTimeout` and `LatchTimeout`. An alternative design was considered whereby the timeouts whenever the `unlockLatch()`

method was called in the `Latch` interface (see Figure 4.4). This was deemed to lower the cohesion of the interface as its sole purpose is to manipulate the latch.

The release logic implementation of this handler is shown in Listing 4.12. This release logic acts as previously described.

```java
// Set during construction
private final AtomicReference<UserEntryState> state;
private final Latch latch;
private final TokenReader tokenReader;

private final LatchTimeout latchTimeout;
private final AlarmTimeout alarmTimeout;

@Override
public void handleEvent() {
  final UserEntryState currentState = this.state.get();

  if (currentState == UserEntryState.WAITING_TOKEN_REMOVAL_SUCCESS) {
    // User successfully authed
    this.latch.unlock();
    this.latchTimeout.startTimeout();
    this.alarmTimeout.startTimeout();
        this.state.set(UserEntryState.IDLE);
  } else {
    // Premature token tear
    this.state.set(UserEntryState.IDLE);
  }
}
```

Listing 4.12: TokenRemovalHandler.java, handleEvent()

Requirements Fulfilled: UE-3, UE-14, UE-15

4.6 Real World Simulation

In a realistic implementation, the application would need to incorporate device drivers in order to interface with the real world peripherals. However, this issue is outside the scope of this report. Therefore input and output from the real world peripherals will be simulated using ‘dummy’ interfaces. In the future, new implementations incorporating device access can be developed.

Our simulation can be split into two parts. The first is a simulated representation of the real world. The second are the implementations of the interfaces, described in subsection 4.4.3, used to interact with this simulated real world.

In the following section an overview of the implementation of these components is given.

4.6.1 Real World Representation

The global singleton `DummyRealWorld` is used to represent the real world. The implementation of this singleton can be seen in Listing 4.13.

```java
public class DummyRealWorld {
  private static final DummyRealWorld instance = new DummyRealWorld();

  // Door System
  private volatile AlarmState alarmState = AlarmState.OFF;
  private volatile DoorState doorState = DoorState.CLOSED;
  private volatile LatchState latchState = LatchState.LOCKED;

  // Identification Peripherals
  private volatile String token;
  private volatile String finger;

  private DummyRealWorld() {
    ;
  }

  /**
   * @return Instance of the Real World
   */
  public static DummyRealWorld getInstance() {
    return instance;
  }

  public synchronized AlarmState getAlarm() {
    return this.alarmState;
  }

  public synchronized void setAlarm(AlarmState state) {
    this.alarmState = state;
  }

  // Other getters and setters omitted
```

Listing 4.13: DummyRealWorld.java

A singleton is used to ensure that there is only ever one representation of the real world in the system at any one point. This is enforced by making the constructor `private`, only allowing access to an instance of the class using `getInstance()` method.

The class contains fields to represent the state of the peripherals in the real world. From Listing 4.13, it can be seen that the initial state of the system is in a secure state. These fields can be accessed in a thread-safe way using their respective getters and setters. In Listing 4.13 an example of a getter and setter pair used to access the state of the real world alarm is shown. Using a setter to modify the value of its associated field is equivalent to a change in the real world.

4.6.2 Dummy Peripherals

Each of the interfaces described in subsection 4.4.3 has an associated concrete implementation that interacts with the DummyRealWorld. These concrete implementations are used to read and modify the state of the real world. One such implementation is shown in Listing 4.14.

```java
public class DummyLatch implements Latch {
  // Internal representation of Latch
  private volatile LatchState state = LatchState.UNKNOWN;

  /**
   * Default Constructor
   */
  public DummyLatch() {
    ;
  }

  @Override
  public synchronized LatchState getState() {
    return this.state;
  }

  @Override
  public synchronized LatchState pollState() {
    this.state = DummyRealWorld.getInstance().getLatch();
    return this.state;
  }

  @Override
  public synchronized void lock() {
    DummyRealWorld.getInstance().setLatch(LatchState.LOCKED);
        this.state = LatchState.LOCKED;
  }

  @Override
  public synchronized void unlock() {
    DummyRealWorld.getInstance().setLatch(LatchState.UNLOCKED);
        this.state = LatchState.UNLOCKED;
  }

}
```

Listing 4.14: DummyLatch.java

An internal representation of the state of the peripheral is held that is updated whenever the poll() method is called. As noted earlier, this is due to the time costs incurred when accessing the peripheral. Cached values should be used where appropriate to save on these costs.

When a poll() operation does occur, the state of the peripheral is updated using the value retrieved from the instance of DummyRealWorld. In this case, DummyLatch retrieves the real world state of the latch by calling the getLatch() method present in DummyRealWorld. Similarly, control of the real world peripheral is simulated by calling setLatch().

The implementations of the other dummy peripherals are similar to DummyLatch. As such they will not be presented here.

4.6.3 Simulating User Input

The user entry operation requires input from the user to function correctly. As previously mentioned, these inputs are represented by AperiodicEvents in the system. Originally it was assumed that inputs could be mapped to keys on the keyboard, with a key press firing one of the events. Unfortunately due to the limitations of the RI this is not possible.

Instead the implemented solution makes use of DummyInputHandler, a PeriodicEventHandler. This handler is also used to perform user acceptance testing. Its implementation in abbreviated form is shown in Listing 4.15.

```
public class DummyInputHandler extends PeriodicEventHandler {
  private final AperiodicEvent tokenRemovalEvent;
  private final AperiodicEvent tokenInputEvent;
  private final AperiodicEvent fingerInputEvent;

  private int stepCounter = 0;
  private int testCounter = 0;

  // constructor omitted

  @Override
  public synchronized void handleEvent() {
    this.stepCounter++;

    switch (this.testCount) {
      case 0:
        this.testValidEntry();
        break;
      // other test cases omitted
    }
  }

  private void testValidEntry() {
    switch (this.stepCounter) {
      case 2:
        this.insertGoodToken();
                break;
          case 4:
                this.insertGoodFinger();
                break;
          case 6:
                this.removeToken();
                break;
          case 8:
                this.openDoor();
                break;
```

```
            case 10:
              this.closeDoor();
              break;
            case 12:
              this.testReset();
              break;
            default:
              break;
          }
        }

        // other test cases omitted.
        // methods for modifying real world state omitted.

        private void testReset() {
          this.c = 0;
          this.testCounter++;
        }

      }
```

Listing 4.15: Partial implementation of DummyInputHandler.java

This handler is responsible for managing the execution of a number of test cases. A test case is an ordered sequence of user inputs, based on scenarios outlined in the original SRS [38]. For instance in Listing 4.15, the test case `testValidEntry` groups the series of inputs that should result in a successful user entry. The current test case to be executed is tracked by the `testCounter` field.

Each step in a test case is associated with a number, dictating when that step occurs. Within Listing 4.15, `stepCounter` is used to keep track of which step is to be executed.

The execution flow of `DummyInputHandler` is shown in Figure 4.17.

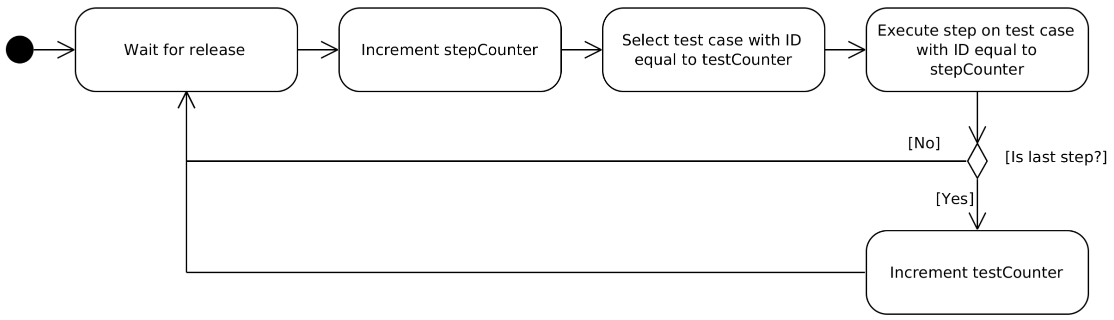
Wait for release
Increment stepCounter
Select test case with ID equal to testCounter
Execute step on test case with ID equal to stepCounter
[No]
[Is last step?]
[Yes]
Increment testCounter

Figure 4.17: Execution flow of `DummyInputHandler`

A release of the handler is as follows:

- Increment the `stepCounter`.

- Select the test case with an ID equal to `testCounter`. For instance in Listing 4.15, the test case `testValidEntry` is assigned the ID 0. As such it is the first test case executed.
- Execute the step associated with the current `stepCounter`. For instance in the test case `testValidEntry`, the door will open when the `stepCounter` reaches 8.
- If this is the last step increase the `testCounter` and reset the `stepCounter` to 0. This ensures the next test case is executed properly.
- Wait for next release.

This implementation is deemed sufficient for the needs of this project, though we note that there are obvious maintainability issues. In the future, a more robust testing framework for firing `AperiodicEvent`s will need to be developed.

Inspiration for simulating the real world and simulating user input using a handler has been gathered from the work presented by the CD_x study [30].

4.7 Summary

In this chapter the design and implementation of the SCJ TIS has been given. The various limitations of the SCJ RI are also discussed, with workarounds for these issues given. Of particular note is the inclusion of a real world simulation for testing.

5 | Testing

In order to evaluate our implementation functions correctly, acceptance testing is performed. Using the use case scenarios from the original SRS [38], a number of test cases have been constructed. These consist of an ordered set of user inputs and an expected result or output from the system. Print statements have been inserted into the release logic of the handlers so that this output can be observed.

In this chapter the use case scenarios are presented along with the result of the acceptance test. The full output of the test runs can be seen in Appendix A.

5.1 Scenarios

Each use case scenario presented consists of the following:

- A brief description of the scenario
- The success condition of the scenario
- Expected interactions performed by a user to achieve this condition

Additionally, whether or not the our implementation fulfilled the use case is also presented.

5.1.1 Successful User Entry

Description: A user who should be allowed enter the enclave is granted access.

Success Condition: The door is unlocked for the user to open.

Expected Interactions:

- User inserts valid token
- User inserts valid fingerprint
- User removes token
- User opens the door

Result: Our implementation fulfilled this use case.

5.1.2 Failed User Entry - Invalid Token

Description: A user who gives an invalid token should not be allowed into the enclave.

Success Condition: The door should be kept locked. The user should be required to remove their token.

Expected Interactions:

- User inserts invalid token

- User removes token

Result: Our implementation fulfilled this use case.

5.1.3 Failed User Entry - Invalid Fingerprint

Description: A user who gives an invalid fingerprint should be denied entry into the enclave.

Success Condition: The door should be kept locked. The user should be required to remove their token.

Expected Interactions:

- User inserts valid token
- User inserts invalid fingerprint
- User removes token

Result: Our implementation fulfilled this use case.

5.1.4 Failed User Entry - Fingerprint Timeout Exceeded

Description: A user does not present a fingerprint in good time should be denied entry into the enclave.

Success Condition: The door should be kept locked. The user should be required to remove their token.

Expected Interactions:

- User inserts valid token
- User waits for timeout to expire
- User removes token

Result: Our implementation fulfilled this use case.

5.1.5 Failed User Entry - Token Removal Timeout Exceeded

Description: A user does not remove their token in good time after passing the previous checks should be denied entry into the enclave.

Success Condition: The door should be kept locked. The user should be required to remove their token.

Expected Interactions:

- User inserts valid token
- User inserts valid fingerprint
- User waits for timeout to expire
- User removes token

Result: Our implementation fulfilled this use case.

5.1.6 Physical Security - Latch is Relocked

Description: Once a user has been granted entry, the latch should relock after a period of time.

Success Condition: The latch should be locked.

Expected Interactions:

- User inserts valid token
- User inserts valid fingerprint
- User removes token
- Wait for latch to lock

Result: Our implementation fulfilled this use case.

5.1.7 Physical Security - Alarm is Activated

Description: Once a user has been granted entry, the alarm should sound if the door is left open for too long.

Success Condition: The alarm should be activated.

Expected Interactions:

- User inserts valid token
- User inserts valid fingerprint
- User removes token
- User opens door (does not close it)
- Wait for alarm to sound

Result: Our implementation fulfilled this use case.

5.2 Summary

In this chapter the use cases used to evaluate the functionality of our implementation have been presented. In all considered cases, the use cases were fulfilled.

6 | Evaluation

In this section we evaluate our work with respect to the project objectives. This includes a comparison of SCJ against elements of SPARK Ada.

6.1 Implementation

In this section, an evaluation of our implementation as a software product is given.

Based on the results given in chapter 5, our implementation meets the original use cases of the TIS. However, as this is not a complete implementation a direct comparison to the original TIS cannot be performed. Additionally, although the system fulfils the use cases, only verification of the implementation against the original formal specification will concretely prove the correctness of the software product.

6.2 SCJ Evaluation

In this section, an evaluation of the SCJ specification as a whole is given.

6.2.1 Reference Implementation

As previously mentioned, the RI available at the department is still incomplete and in some cases, is not compliant with the SCJ specification. Consequently during the implementation of the SCJ TIS, workarounds were developed to address the lack of support for certain features such as I/O or keyboard support. In one case a missing part of the specification was implemented, specifically the `OneShotEventHandler` class.

The lack of a full reference implementation leads to the development of less than optimal implementations. An example of this is the need of the `DummyInputHandler` class and the implementation of the `DummyTokenRead` and `DummyFingerReader` classes as discussed in chapter 4. The implementation of these classes was directly effected by the lack of support for SCJ's I/O features. Were these features available, keyboard support could have been implemented, allowing the development of an interactive system.

Until the RI is developed further, the SCJ specification cannot be fully evaluated.

6.2.2 Event Handlers

By complying to the Level 1 specification of SCJ, our implementation was able to use the full range of event handlers available which included `PeriodicEventHandlers`, `AperiodicEventHandlers` and `OneShotEventHandlers`.

The event based model attempts to simplify some aspects associated with thread based models, as discussed previously in subsection 2.2.3. One other advantage of the event based model is that each event handler can be associated with handling a specific event in the system. This results in small, highly cohesive classes which leads to better maintainability in the long term. However it is not without is drawbacks.

One issue encountered during the design phase involved communication between handlers. Whilst it is easy to visualise the execution of each individual event handler, considering all the handlers at once proves difficult especially when taking into consideration the concurrent aspects of the system. For instance, the `FingerTimeout` handler is executed alongside a periodic handler. Additionally, the release logic of this handler changes based on the system state, which can be affected by an aperiodic event (user inserts a fingerprint).

6.2.3 JDK API Usage

Our implementation shows that it is possible to use elements from the standard JDK. Of particular intereset are the `java.util.concurrent`, `java.util.concurrent.atomic` and `java.util.concurrent.lock` packages.

These packages provide useful utilities that are potentially tedious or difficult to implement. For instance, the `java.util.concurrent.atomic` package provides classes to support lock-free thread-safe variables. These atomic wrapper classes are equivalent to `volatile` variables, but feature convenience methods such as `compareAndSet()`.

However, the SCJ specification forbids the usage of the majority of the standard Java API, including `java.util.concurrent` package. This highlights several issues.

Firstly, the RI does not properly forbid the usage of these classes correctly. This is especially problematic as it is possible to produce an application that is not compliant with the SCJ specification. In the case of the SCJ TIS implementation, there is only one usage of an offending class, specifically `AtomicReference`. Fortunately, this can be rectified by re-factoring the code-base to use the `volatile` keyword.

Secondly, it highlights the fact that SCJ excludes commonly used utilities of the JDK, such as the Collections framework. Such utilities may be expected to be available by veteran Java developers. Indeed the usage of the utilities present in the `java.util.concurrent` is considered best practice in Java [13].

6.3 Comparison to SPARK Ada

The original implementation of the TIS was produced using SPARK Ada, as noted earlier in chapter 4. Though the produced SCJ implementation does not replicate the exact functionality of the original, some important insights can be inferred.

6.3.1 The INFORMED Process

Key to SPARK Ada is the INFORMED design process as discussed earlier in chapter 4. As proven by the original TIS, it has shown that INFORMED results in reliable, low-defect systems. It provides guidance on how to establish the structure of the application using a well defined set of components. As previously discussed in section 4.2, these components are Type, Variable and Utility packages.

SCJ does not currently have a equivalent process established. During this study parts of the RUP development methodology and INFORMED process was used to guide implementation, as discussed in chapter 4. This does not address design issues that are specific to SCJ however. In the future, the specification would benefit from a unified design process that addresses details including, but not limited to:

Compliance Level Identification - Difficulty was encountered when identifying the compliance level of the SCJ TIS. Both a level 0 and 1 application could have been designed. As such, guidance should be provided on identifying the compliance level of SCJ applications.

Assignment of Handler Functionality - During the design of SCJ TIS, difficulty was encountered when constructing and assigning functionality to the event handlers in the system.

Identification and Localisation of System State - Realistically, some sort of system state will need to be stored and manipulated by an application. For instance, this could be the state of some real world device. Therefore, where and how this state is represented is crucial to SCJ applications.

We note that elements of the INFORMED process are general enough such that they can be used as a basis for forming a SCJ specific design process.

6.3.2 SPARK Examiner

SPARK features the heavy use of annotations, directly inserted in the source code of an application. Annotations are used to specify the expected function of a specific section of code. Using the SPARK Examiner tool, an annotation and accompanying code can be compared and analysed for correctness. This has the benefit of highlighting any flaws in the design. The Examiner tool is also capable of other operations such as lexical analysis.

No such analysis tool equivalent to SPARK Examiner currently exists for SCJ. Such a tool would be beneficial, if not outright necessary, for use in the verification process of a safety-critical application. One issue hindering the development of such a tool is the limited annotation support of SCJ, which is currently limited to checking compliance levels. One possible solution is to adapt a specification language, such as the Java Modelling Language (JML), for this purpose.

The novel memory management system of the SCJ specification would also need to be verified. Such tools are currently in development with a prototype implementation having been developed by Dalsgaard et al. [55]. Additionally, due to the concurrent nature of SCJ applications, tools are required to perform analysis of the Worst Case Execution Time (WCET) for verification of hard real-time applications. One such tool has been developed by Frost et al. [56]. However, due to technical limitations, these tools are incompatible with the RI.

6.4 Summary

SCJ shows promise for use in the development of high-integrity systems. Its novel event based programming model and API eases software development, leading to cohesive, maintainable code.

However, compared to the more established SPARK Ada language, SCJ is shown to be lacking in some areas. Of particular note is the lack of design guidance for applications and the lack of static analysis tools. As the SCJ specification evolves, these issues should be addressed.

7 | Conclusions

This project produced a Level 1 implementation of an existing high-integrity system. It has provided insight into the issues associated with the development of such high-integrity systems, specifically the handling of user input and interaction with real world devices. We have attempted to evaluate the use of SCJ in the development of a non-trivial real world problem and compared it to the more established SPARK Ada specification.

We hope that the work presented in this report will provide valuable feedback on the future development of SCJ.

7.1 Future Work

A number of avenues can be investigated for future work:

- Further development of the implementation presented in this report would provide additional insight into the advantages and disadvantages of SCJ. This would also provide a more comprehensive use case for evaluation and enable a more detailed comparison to be performed against the SPARK Ada implementation.
- The development of a unified software development methodology specifically for SCJ. Current design processes do not take into account the novel aspects of SCJ such as memory management.
- The further development of annotation usage in SCJ would enable work to progress on analysis tools for SCJ. These can be used to automate the verification of SCJ applications against formal specifications.

8 | Bibliography

[1] Faa advisory circular 20-115b. Federal Aviation Administration. [Online]. Available: http://www.airweb.faa.gov/Regulatory_and_Guidance_Library/rgAdvisoryCircular.nsf/0/dcdb1d2031b19791862569ae007833e7/$FILE/AC20-115B.pdf

[2] Adacore : Spark pro. [Online]. Available: http://www.adacore.com/sparkpro/

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

[4] D. F. Bacon, "Realtime garbage collection," *Queue*, vol. 5, no. 1, pp. 40–49, Feb. 2007. [Online]. Available: http://doi.acm.org/10.1145/1217256.1217268

[5] Real-time specification for java. [Online]. Available: http://jcp.org/en/jsr/detail?id=1

[6] RTSJ version 1.1. [Online]. Available: http://jcp.org/en/jsr/detail?id=282

[7] Android. [Online]. Available: http://www.android.com

[8] (2011, April) Twitter search is now 3x faster. [Online]. Available: https://blog.twitter.com/2011/twitter-search-now-3x-faster

[9] J. Gosling and H. McGilton, "The java language environment: A white paper," White Paper, Sun Microsystems Inc., 1997. [Online]. Available: http://www.oracle.com/technetwork/java/index-136113.html

[10] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. (2013, feb) The java®language specification. Oracle Corporation. [Online]. Available: http://docs.oracle.com/javase/specs/jls/se7/html/index.html

[11] (2013) Java se 7 collections-related apis and developer guides. Oracle. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html

[12] A. Burns and G. Daviews, *Concurrent Programming*, ser. International computer science series. Addison-Wesley Publishing Co., 1993.

[13] J. Bloch, *Effective Java (2nd Edition) (The Java Series)*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[14] (2013) Java thread primitive deprecation. Oracle. [Online]. Available: http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

[15] A. Burns and A. J. Wellings, *Real-Time Systems and Programming Languages: ADA, Real-Time Java, and C/Real-Time POSIX*, 4th ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2009.

[16] J. Kwon, A. Wellings, and S. King, "Assessment of the java programming language for use in high integrity systems," *SIGPLAN Notices*, vol. 38, no. 4, pp. 34–46, Apr. 2003. [Online]. Available: http://doi.acm.org/10.1145/844091.844099

[17] "Memory management in the java hotspot^{TM}virtual machine," White Paper, Sun Microsystems Inc., apr 2006. [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf

[18] J. Kwon, A. Wellings, and S. King, “Ravenscar-java: a high-integrity profile for real-time java,” *Concurrency and Computation: Practice and Experience*, vol. 17, no. 5-6, pp. 681–713, 2005. [Online]. Available: http://dx.doi.org/10.1002/cpe.843

[19] M. Tofte and J.-P. Talpin, “Region-based memory management,” *Information and Computation*, vol. 132, no. 2, pp. 109 – 176, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0890540196926139

[20] P. Dibble and A. Wellings, “The real-time specification for java: Current status and future work,” in *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2004*, Computer Society, IEEE. IEEE, May 2004, pp. 71–77.

[21] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek, “Real-time java scoped memory: design patterns and semantics,” in *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, 2004, pp. 101–110.

[22] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, “Event-driven programming for robust software,” in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, ser. EW 10. New York, NY, USA: ACM, 2002, pp. 186–189. [Online]. Available: http://doi.acm.org/10.1145/1133373.1133410

[23] A. Wellings and M. Kim, “Asynchronous event handling and safety critical java,” *Concurrency and Computation: Practice and Experience*, vol. 24, no. 8, pp. 813–832, 2012. [Online]. Available: http://dx.doi.org/10.1002/cpe.1756

[24] B. M. Brosgol and A. Wellings, “A comparison of ada and real-time javatm for safety-critical applications,” in *Reliable Software Technologies – Ada-Europe 2006*, ser. Lecture Notes in Computer Science, L. Pinho and M. González Harbour, Eds. Springer Berlin Heidelberg, 2006, vol. 4006, pp. 13–26. [Online]. Available: http://dx.doi.org/10.1007/11767077_2

[25] D. Sharp, E. Pla, and K. Luecke, “Evaluating mission critical large-scale embedded system performance in real-time java,” in *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, Dec 2003, pp. 362–365.

[26] M. Schoeberl, B. Thomsen, B. Thomsen, and A. Ravn, “A profile for safety critical java,” in *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC ’07. 10th IEEE International Symposium on*, 2007, pp. 94–101. [Online]. Available: http://dx.doi.org/10.1109/ISORC.2007.9

[27] D. Locke, B. S. Anderson, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. Nielsen, M. Schoeberl, J. Vitek, and A. Wellings. (2013) Safety critical java^{TM}technology specfication. [Online]. Available: http://jcp.org/en/jsr/detail?id=302

[28] 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. New York, NY, USA: ACM, 2012, pp. 62–71. [Online]. Available: http://doi.acm.org/10.1145/2388936.2388948

[29] 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. New York, NY, USA: ACM, 2009, pp. 41–50. [Online]. Available: http://doi.acm.org/10.1145/1620405.1620412

[30] F. Zeyda, A. Cavalcanti, A. Wellings, J. Woodcook, and K. Wei, “Refinement of the parallel cdx” University of York, Department of Computer Science, Tech. Rep., 2013.

[31] 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. New York, NY, USA: ACM, 2012, pp. 72–79. [Online]. Available: http://doi.acm.org/10.1145/2388936.2388949

[32] J. Barnes, R. Chapman, R. Johnson, J. Widmaier, D. Cooper, and B. Everett, “Engineering the tokeneer enclave protection software,” in *1st IEEE International Symposium on Secure Software Engineering (March 2006)*, 2006.

[33] W. W. Everett, “Token id station (tis) protection profile,” SPRE Inc., Tech. Rep., August 2002. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[34] D. Cooper, “Tokeneer id station: Security target,” Altran Praxis Limited, Tech. Rep., August 2008. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[35] ——, “Tokeneer id station: Approaching the common criteria,” Altran Praxis Limited, Tech. Rep., August 2008. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[36] ——, “Tokeneer id station: Annotated summary of security target exclusions,” Altran Praxis Limited, Tech. Rep., August 2008. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[37] ——, “Tokeneer id station: Annotated summary of security target exclusions,” Altran Praxis Limited, Tech. Rep., 2008. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[38] D. Cooper and J. Barnes, “Tokeneer id station: System requirements specification,” Altran Praxis Limited, Tech. Rep., August 2008. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[39] J. Barnes and D. Cooper, “Tokeneer id station: Formal specification,” Altran Praxis Limited, Tech. Rep., August 2008. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[40] J. Barnes, “Tokeneer id station: Code verification summary,” Altran Praxis Limited, Tech. Rep., August 2008. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[41] ——, “Tokeneer id station: System test specification,” Altran Praxis Limited, Tech. Rep., August 2008. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[42] *Common Criteria for Information Technology Security Evaluation v3.1 R4 Part 3: Security assurance components*, September 2012. [Online]. Available: https://www.commoncriteriaportal.org/cc/

[43] J. Barnes and D. Stokes, “Tokeneer id station: Project plan,” Altran Praxis Limited, Tech. Rep., Auguqst 2008. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[44] Reveal®. Altran. [Online]. Available: http://intelligent-systems.altran.com//technologies/systems-engineering/revealtm.html#.U0rbZHVdWrM

[45] S. Team, “INFORMED design method for spark,” September 2011. [Online]. Available: http://docs.adacore.com/sparkdocs-docs/Informed.htm

[46] J. Woodcock, E. Aydal, and R. Chapman, “The tokeneer experiments,” in *Reflections on the Work of C.A.R. Hoare*, A. Roscoe, C. B. Jones, and K. R. Wood, Eds. Springer London, 2010, pp. 405–430. [Online]. Available: http://dx.doi.org/10.1007/978-1-84882-912-1_17

[47] D. Spinellis. A look at zero-defect code. [Online]. Available: http://www.spinellis.gr/blog/20081018/index.html

[48] Y. Moy and A. Wallenburg, “Tokeneer: Beyond formal program verification,” *Embedded Real Time Software and Systems*, 2010.

[49] R. Chapman, “Tokeneer id station: Overview and reader’s guide,” Altran Praxis Limited, Tech. Rep., October 2011. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[50] D. Cooper and J. Barnes, “Tokeneer id station: Eal5 demonstrator: Summary report,” Altran Praxis Limited, Tech. Rep., Augugst 2008. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[51] J. M. Spivey, *The Z Notation: A Reference Manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[52] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[53] J. Barnes, “Tokeneer id station INFORMED design,” Altran Praxis Limited, Tech. Rep., August 2008. [Online]. Available: http://www.adacore.com/sparkpro/tokeneer/

[54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[55] 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. New York, NY, USA: ACM, 2012, pp. 9–17. [Online]. Available: http://doi.acm.org/10.1145/2388936.2388939

[56] 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 Successful User Entry

```
TokenInput              -   Received token input
FingerTimeout           -   Finger Timeout Started
TokenInput              -   Token valid
DoorLatchAlarmUpdate    -   Reading Door Security...
DoorLatchAlarmUpdate    -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate    -   Reading Door Security...
DoorLatchAlarmUpdate    -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate    -   Reading Door Security...
FingerInput             -   Received Finger Input
DoorLatchAlarmUpdate    -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenRemovalTimeout     -   Token Removal Timeout Started
FingerInput             -   Fingerprint valid
DoorLatchAlarmUpdate    -   Reading Door Security...
DoorLatchAlarmUpdate    -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate    -   Reading Door Security...
TokenRemoval            -   Authentication successful, user allowed entry
DoorLatchAlarmUpdate    -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DummyLatch              -   Latch Unlocked
LatchTimeout            -   Latch Timeout Started
AlarmTimeout            -   Alarm Timeout Started
DoorLatchAlarmUpdate    -   Reading Door Security...
DoorLatchAlarmUpdate    -   Alarm: OFF, Door: CLOSED, Latch: UNLOCKED
FingerTimeout           -   Finger Timeout Expired
DoorLatchAlarmUpdate    -   Reading Door Security...
DoorLatchAlarmUpdate    -   Alarm: OFF, Door: CLOSED, Latch: UNLOCKED
DoorLatchAlarmUpdate    -   Reading Door Security...
DoorLatchAlarmUpdate    -   Alarm: OFF, Door: CLOSED, Latch: UNLOCKED
TokenRemovalTimeout     -   Token Removal Timeout Expired
DoorLatchAlarmUpdate    -   Reading Door Security...
DoorLatchAlarmUpdate    -   Alarm: OFF, Door: CLOSED, Latch: UNLOCKED
DoorLatchAlarmUpdate    -   Reading Door Security...
DoorLatchAlarmUpdate    -   Alarm: OFF, Door: CLOSED, Latch: UNLOCKED
```

```
DummyLatch             -   Latch Locked
LatchTimeout           -   Latch Timeout Expired
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
```

A.2 Failed User Entry - Invalid Token

```
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenInput             -   Received token input
TokenInput             -   Token invalid
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenRemoval           -   Authentication unsuccessful, user denined entry
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
```

A.3 Failed User Entry - Invalid Fingerprint

```
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenInput             -   Received token input
FingerTimeout          -   Finger Timeout Started
```

```
TokenInput            -   Token valid
DoorLatchAlarmUpdate  -   Reading Door Security...
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate  -   Reading Door Security...
FingerInput           -   Received Finger Input
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
FingerInput           -   Fingerprint invalid
DoorLatchAlarmUpdate  -   Reading Door Security...
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate  -   Reading Door Security...
TokenRemoval          -   Authentication unsuccessful, user denined entry
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate  -   Reading Door Security...
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
FingerTimeout         -   Finger Timeout Expired
DoorLatchAlarmUpdate  -   Reading Door Security...
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate  -   Reading Door Security...
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
```

A.4 Failed User Entry - Fingerprint Timeout Exceeded

```
DoorLatchAlarmUpdate  -   Reading Door Security...
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenInput            -   Received token input
FingerTimeout         -   Finger Timeout Started
TokenInput            -   Token valid
DoorLatchAlarmUpdate  -   Reading Door Security...
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate  -   Reading Door Security...
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate  -   Reading Door Security...
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate  -   Reading Door Security...
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate  -   Reading Door Security...
DoorLatchAlarmUpdate  -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
FingerTimeout         -   Finger Timeout Expired
FingerTimeout         -   Failed to give finger in time
```

```
DoorLatchAlarmUpdate   -   Reading Door Security...
TokenRemoval           -   Authentication unsuccessful, user denined entry
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
```

A.5 Failed User Entry - Token Removal Timeout Exceeded

```
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenInput             -   Received token input
FingerTimeout          -   Finger Timeout Started
TokenInput             -   Token valid
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
FingerInput            -   Received Finger Input
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenRemovalTimeout    -   Token Removal Timeout Started
FingerInput            -   Fingerprint valid
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
FingerTimeout          -   Finger Timeout Expired
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenRemovalTimeout    -   Token Removal Timeout Expired
TokenRemovalTimeout    -   Failed to remove token in time, entry revoked
DoorLatchAlarmUpdate   -   Reading Door Security...
TokenRemoval           -   Authentication unsuccessful, user denined entry
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
```

A.6 Physical Security - Latch is Relocked

```
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenInput             -   Received token input
FingerTimeout          -   Finger Timeout Started
TokenInput             -   Token valid
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
FingerInput            -   Received Finger Input
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenRemovalTimeout    -   Token Removal Timeout Started
FingerInput            -   Fingerprint valid
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenRemoval           -   Authentication successful, user allowed entry
DummyLatch             -   Latch Unlocked
LatchTimeout           -   Latch Timeout Started
AlarmTimeout           -   Alarm Timeout Started
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: UNLOCKED
FingerTimeout          -   Finger Timeout Expired
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: UNLOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: UNLOCKED
TokenRemovalTimeout    -   Token Removal Timeout Expired
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: UNLOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: UNLOCKED
DummyLatch             -   Latch Locked
LatchTimeout           -   Latch Timeout Expired
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: CLOSED, Latch: LOCKED
```

A.7 Physical Security - Alarm is Activated

```
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenInput            -  Received token input
FingerTimeout         -  Finger Timeout Started
TokenInput            -  Token valid
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: CLOSED, Latch: LOCKED
FingerInput           -  Received Finger Input
TokenRemovalTimeout   -  Token Removal Timeout Started
FingerInput           -  Fingerprint valid
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: CLOSED, Latch: LOCKED
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: CLOSED, Latch: LOCKED
TokenRemoval          -  Authentication successful, user allowed entry
DummyLatch            -  Latch Unlocked
LatchTimeout          -  Latch Timeout Started
AlarmTimeout          -  Alarm Timeout Started
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: CLOSED, Latch: UNLOCKED
FingerTimeout         -  Finger Timeout Expired
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: OPEN, Latch: UNLOCKED
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: OPEN, Latch: UNLOCKED
TokenRemovalTimeout   -  Token Removal Timeout Expired
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: OPEN, Latch: UNLOCKED
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: OPEN, Latch: UNLOCKED
DummyLatch            -  Latch Locked
LatchTimeout          -  Latch Timeout Expired
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: OPEN, Latch: LOCKED
DoorLatchAlarmUpdate  -  Reading Door Security...
DoorLatchAlarmUpdate  -  Alarm: OFF, Door: OPEN, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: OPEN, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: OPEN, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: OFF, Door: OPEN, Latch: LOCKED
AlarmTimeout           -   Alarm Timeout Expired
DummyAlarm             -   Alarm Activated
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: ON, Door: OPEN, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: ON, Door: OPEN, Latch: LOCKED
DoorLatchAlarmUpdate   -   Reading Door Security...
DoorLatchAlarmUpdate   -   Alarm: ON, Door: OPEN, Latch: LOCKED
```

B | Compiling and Running

Accompanying this report is a copy of the source code along with Eclipse project files. To compile and run the program JamaicaVM and Eclipse is required. These are available on the departmental machines. Additionally the JamaicaVM Eclipse Plugin is required, available from https://www.aicas.com/cms/en/eclipse-plugin. Once the plugin is installed, the source can be imported into Eclipse and ran in the standard way.

Alternatively, the source can be compiled by hand using the the `scjvmc` and `scjvm` commands in a terminal.

Acronyms

APEH	`AperiodicEventHandler`. 16, 17
CAS	Compare-and-Swap. 40, 51
EAL5	Evaluation Assurance Level 5. 19
INFORMED	INformation Flow ORiented MEthod of (object) Design. 20, 33, 36, 65
JDK	Java Development Kit. 8, 65
JML	Java Modelling Language. 66
JVM	Java Virtual Machine. 7, 10–12
NSA	National Security Agency. 19
PEH	`PeriodicEventHandler`. 16, 17
POJO	Plain Old Java Object. 38
RI	Reference Implementation. 31, 58, 60, 64–66
RTSJ	Real-Time Specification for Java. 5, 11–15, 17, 18
RUP	Rational Unified Process. 23, 35, 65
SCJ	Safety Critical Java. 5, 6, 11, 15–19, 21, 23, 25, 26, 29–31, 34, 35, 39, 42, 44, 60, 64–67
SRS	System Requirements Specification. 19, 30, 59, 61
TIS	Tokeneer ID Station. 3, 19–23, 26, 30, 31, 33–35, 39, 42, 52, 60, 64–66
WCET	Worst Case Execution Time. 66