A Formal Model of the Safety-Critical Java Level 2 Paradigm

Matt Luckcuck, Ana Cavalcanti, and Andy Wellings

Department of Computer Science, University of York,
York, YO10 5GH, UK

ml881@york.ac.uk

Abstract

Safety-Critical Java (SCJ) introduces a new programming paradigm for applications that must be certified. The SCJ specification (JSR 302) is an Open Group Standard, but it does not include verification techniques. Previous work has addressed verification for SCJ Level 1 programs. We support the much more complex SCJ Level 2 programs, which allows the programming of highly concurrent multi-processor applications with Java threads, and wait and notify mechanisms. We present a formal model of SCJ Level 2 that captures the state and behaviour of both SCJ programs and the SCJ API. This is the first formal semantics of the SCJ Level 2 paradigm and is an essential ingredient in the development of refinement-based reasoning techniques for SCJ Level 2 programs. We show how our models can be used to prove properties of the SCJ API and applications.

1 Introduction

Safety-Critical Java (SCJ) [20] is a version of Java that embeds a new programming paradigm for applications that must be certified for example, using the highest level of the avionics standard ED-12/DO-178 [4]. To aid certification, SCJ is organised into three compliance levels. Level 0 applications are simple single-processor programs executed by a cyclic executive. Level 1 applications introduce concurrency and less-restricted release patterns. By contrast, Level 2 applications are highly concurrent, potentially multi-processor, and make use of suspension and a variety of release patterns.

The verification of SCJ programs requires specific techniques, but these are not covered by the SCJ specification. Verification has been addressed for Level 1, but not Level 2. SCJ, and its Level 2 profile in particular, present several challenges for verification. The new programming paradigm of SCJ restricts the program structure and provides a predictable memory model. The unique features of Level 2 allow programming applications that may contain multiple modes of operation or independently developed subsystems, and computations that require non-standard release patterns or suspension [23].

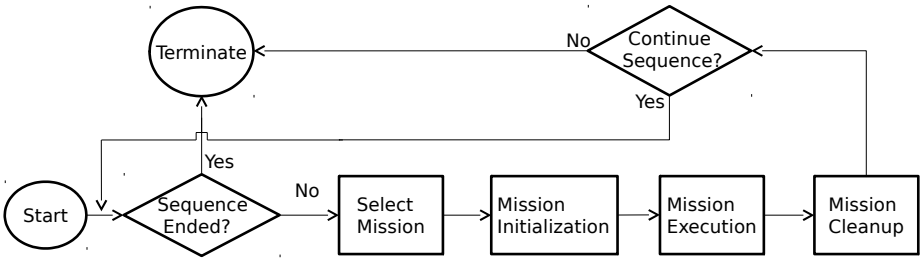

Figure 1: Mission Phases

In this paper, we provide support for verification of SCJ Level 2 programs by modelling its programming paradigm using the state-rich process algebra *Circus* [24]. This is a combination of Z [18] for modelling state, CSP [8] for modelling behaviour, and Morgan's refinement calculus [14]. A *Circus* program is organised around processes, which contain variables and actions, to describe a data model and reactive behaviours. Each process has a main action that defines its behaviour, possibly using a combination of other actions in the process. Communication between processes is achieved via channels. In our work we use the *Circus* extensions *OhCircus* [2], which introduces object orientation and inheritance, and *Circus* Time [16] to specify timers and deadlines.

Circus has already been used to model SCJ Level 1 [25]. *Circus* has also been used to produce a refinement strategy [3] to derive SCJ programs that are correct by construction. Our models provide the the possibility of extending the refinement strategy to target SCJ Level 2 programs.

What we present in this paper is the first formalisation of SCJ Level 2. The SCJ API covers approximatively 112 pages of the specification [20] as a collection of approximately 36 classes and interfaces. Our work characterises a semantics for SCJ Level 2 programs. To support its use, we have developed a tool that generates *Circus* models from SCJ programs. We have used the models to prove, via model checking, properties of both the SCJ API and of specific programs.

In Sect. 2 we describe the unique features of the SCJ Level 2 paradigm. Section 3 describes our modelling approach, model structure, and how we model Java synchronisation and suspension behaviour. Section 4 describes the direct applications of our models for verification, including a brief account of our tool. Section 5 presents related work. Finally, Sect. 6 concludes this paper with a summary of our contribution and a discussion of future work.

2 Safety-Critical Java Level 2 Paradigm

Safety-Critical Java (SCJ) is a version of Java that adopts a new programming paradigm. SCJ programs have a specific concurrent design and use region-based memory management (instead of garbage collection); specialised virtual machines [15, 17] are available to execute SCJ programs. SCJ also uses the real-time constructs introduced in the Real-Time Specification for Java [21], but enforces a more structured programming paradigm.

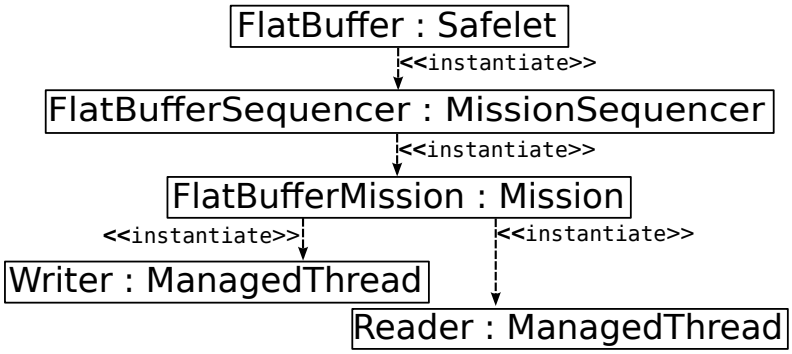

Figure 2: Object Diagram of the Flatbuffer

An SCJ program is controlled by a *safelet* object, which manages the top-level *mission sequencer*. This is used to activate an application-defined sequence of *missions*. A mission encapsulates a particular function or phase of operation as a set of *schedulable objects* to perform a particular task. An SCJ API supports the programming of these components.

Each mission progresses through an initialisation, execution, and cleanup phase, as shown in Fig. 1. During initialisation, a mission's schedulable objects are created and registered. These schedulables are activated simultaneously at the start of the execution phase. A mission's schedulables execute until one of them requests termination, or they all terminate, when a cleanup phase is performed. At the end of the cleanup phase, the mission may indicate that no further missions should execute, in which case the sequence will terminate. If not, and there are more missions to run, the next mission is prepared.

At Level 2, schedulable objects may adopt one of four release patterns. Periodic event handlers execute once in a given time period, aperiodic event handlers execute when triggered by a method call, one-shot event handlers execute once after a time offset, and managed threads simply run to completion. Level 2 supports the execution of concurrent missions by allowing missions to manage schedulable mission sequencers. Level 2 can also use Java suspension methods, `wait()` and `notify()`, but they may only be called on `this`.

To illustrate some of the features of SCJ Level 2 programs we introduce FlatBuffer, which is a simple solution to the Producer-Consumer Problem, using a one-place buffer. FlatBuffer is structurally simple, only containing one mission and two schedulables, but uses two of Level 2's unique features: managed threads and suspension. Larger examples of applications that use the unique features of Level 2 can be found in [23].

Figure 2 shows an object diagram of the FlatBuffer program at the end of its mission's initialise phase. It is controlled by the safelet `FlatBuffer`, which starts the top-level mission sequencer `FlatBufferMissionSequencer`. This mission sequencer starts the mission, `FlatBufferMission`, which starts the two managed threads. The `Writer` is the producer and the `Reader` is the consumer.

The `FlatBufferMission` holds the buffer and controls access to it. The mission has a `bufferEmpty()` method to indicate if the buffer if empty or full, a `read()` method to control reading from and resetting the buffer, and a `write()` method to control updating the buffer. The `read()` and `write()` methods both

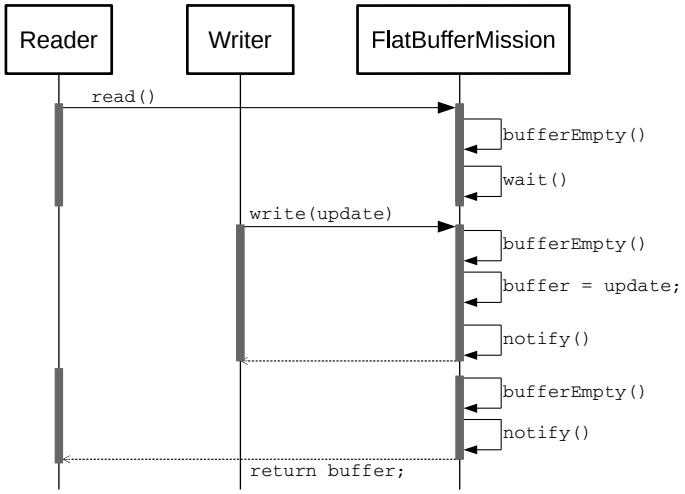

Figure 3: Sequence Diagram of an Example Execution of FlatBuffer

use synchronisation to control access to the buffer.

In an example execution of FlatBuffer, illustrated in Fig. 3, the `Reader` runs first, and calls the mission's `read()` method. The method calls `bufferEmpty()` on the mission, which returns a boolean indicating that the buffer is empty. Because there is nothing to read, the method calls `wait()` to suspend the `Reader`.

Next, the `Writer` runs, calling the mission's `write()` method. This method calls `bufferEmpty()` on the mission, which still indicates that the buffer is empty, prompting the `Writer` to update the buffer. Then, the method calls `notify()` on the mission – which resumes the `Reader`. When the `Reader` resumes, it is still inside the `read()` method. The method calls `bufferEmpty()`, which indicates that the buffer is full, so the value is read and the buffer is reset. Since this is a simple test program, the `Writer` terminates the mission after 5 writes.

Despite SCJ's restricted infrastructure, the unique features of Level 2 mean that its programs can become very complex. Providing the first semantics for this paradigm and devising a model for Level 2 programs is, therefore, a challenging task. We need to deal with a variety of schedulable objects, a preemptive scheduler that guarantees absence of priority inversion, a complex protocol for termination of missions, and suspension in the context of all of these features. We discuss our approach to modelling SCJ Level 2 in the next section.

3 Modelling Approach

We view the programming paradigm of SCJ separately from its realisation in Java. We capture this paradigm, abstracting away from most of the details of its Java implementation. Our modelling approach is agnostic of Java.

We model the state and behaviour of application objects in the program and the use of suspension. We also capture exceptions, but not the Java exception handling mechanism. We only capture exceptions where they indicate a misuse of the paradigm. Specifically we capture exceptions when: a thread

Action	Syntax	Description
Skip	**Skip**	A simple operator that terminates
Simple Prefix	$c \longrightarrow A$	Simple synchronisation with no data
Input Prefix	$c?x \longrightarrow A$	Synchronisation with a value bound to x
Output Prefix	$c!x \longrightarrow A$	Synchronisation outputting the value of x
Parameter Prefix	$c.x \longrightarrow A$	Synchronisation with some data x
Sequence	$A \; ; \; B$	Executes A then B in sequence
External Choice	$A \Box B$	Offers a choice between two actions A and B
Interrupt	$A \triangle c \longrightarrow \mathbf{Skip}$	Executes A unless c occurs, which terminates A
Recursion	$\mu X \bullet A \; ; \; X$	A process X that executes A then X
Wait	**wait** t	Waits for t time units and then terminates
Chaos	**Chaos**	The action that immediately diverges

Table 1: Summary of *Circus* operators

is interrupted, a thread attempts to use suspension without holding the lock, a thread attempts to lock an object with a priority lower than the thread's, a method receives an inappropriate argument, or a mission attempts to register a schedulable that is already registered to another or the same mission.

Our models consist of two parallel components, following the approach in [25]. The framework component captures the behaviour of the library supporting the SCJ API and is reused for all programs. The application component captures the specific behaviour of a particular program. Each framework process has a counterpart application process. The complete specification of the framework model [12] comprises approximately 3700 lines of *Circus* over 11 processes.

Table 1 summarises the *Circus* action operators that we use in this paper. Most of them are familiar to users of CSP. We describe them to support the discussion of our model; a comprehensive account of *Circus* is in [24]. We note that *Circus* processes can also be combined using most CSP operators.

We describe our models in Sect. 3.1 and present our approach in more detail in Sect. 3.2 using the mission models as an example. Finally, in Sect. 3.3, we discuss how we model synchronisation and suspension.

3.1 Model Overview

Each SCJ library class and application object is represented by a *Circus* process. Each process retains the name of the class it models, suffixed with 'FW' for framework processes or 'App' for application processes. Methods are represented by an action in the relevant process. Method calls and returns are represented by (usually pairs of) events; this allows method calls between processes.

Figure 4 shows the framework processes in our model and the channels that they use to communicate. The channels with underscores in their names are control signals (for example, *start_mission*) and those in camel case represent method calls (for example, *initializeCall* and *initializeRet*). Some of the channels have been omitted for brevity, indicated by three dots. The layering indicates potentially multiple instances in one model. Each of these framework processes communicate with an application process; these are not shown in Fig. 4.

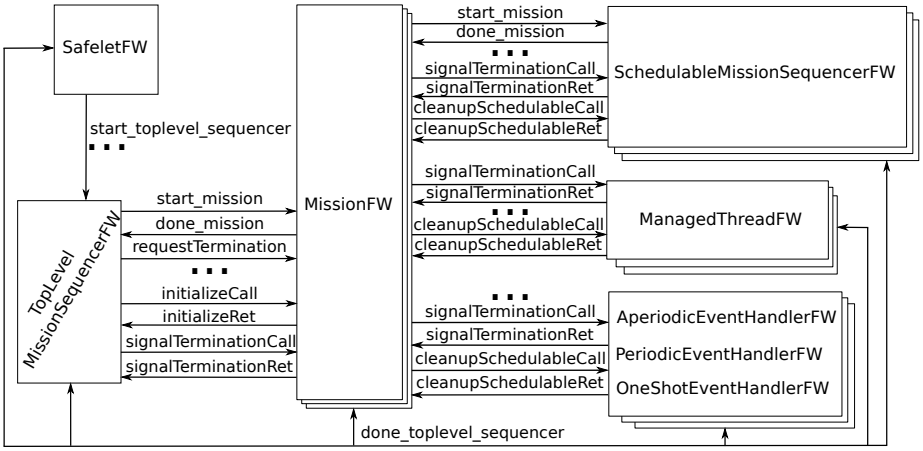

Figure 4: Level 2 Model Structure

When a framework process encounters application-specific behaviour, it signals its application counterpart to take control and perform the behaviour. Control is returned to the framework with another signal. These signals are call-return event pairs that retain the method name, suffixed with '*Call*', for the event modelling the method call, or '*Ret*', for the event modelling its return.

Each application process is assigned a unique identifier, allowing framework processes to communicate with their application counterparts. An exception is the *SafeletFW* process, which only has one instance because there is only one safelet in an SCJ program. If a program class has multiple instances in the program, then each instance has its own *Circus* process identifier.

Our model uses *OhCircus* classes to capture non-reactive behaviour, such as methods that are purely data operations. *OhCircus* classes are similar to Java classes: they may hold variables, specify constructors, make use of inheritance, and must be instantiated before use. Specifically, data operations are captured in methods, which may be called from processes. In contrast to *Circus* processes, *OhCircus* classes can be related by inheritance.

Instead of simply adding Level 2 features to the Level 1 model [25], we also capture Level 1 features not found in the previous model. Namely, we consider that a period or deadline may be overrun and capture exceptions and synchronisation. While Level 1 programs may not use suspension, they are allowed to use synchronisation. In addition, in contrast to the Level 1 model, we provide separate framework processes for each of the three kinds of event handlers, each encapsulating their particular release pattern. This simplifies the application models considerably and lessens the burden on translation. Further, as already mentioned, our model raises an exception if a schedulable is registered twice.

3.1.1 Safelet

The framework process *SafeletFW* handles the operations of the safelet. *SafeletFW* gets the identifier of the top-level mission sequencer from its application counter-

$$
\begin{array}{l}
\mathit{InitializePhase} \mathrel{\widehat{=}} \\
\quad \left(\begin{array}{l}
\mathit{initializeCall} \,.\, \mathit{FlatBufferMissionMID} \longrightarrow \\
\mathit{register} \,!\, \mathit{ReaderSID} \,!\, \mathit{FlatBufferMissionMID} \longrightarrow \\
\mathit{register} \,!\, \mathit{WriterSID} \,!\, \mathit{FlatBufferMissionMID} \longrightarrow \\
\mathit{initializeRet} \,.\, \mathit{FlatBufferMissionMID} \longrightarrow \\
\mathbf{Skip}
\end{array} \right)
\end{array}
$$

Figure 5: The *FlatBufferMissionApp*'s *InitializePhase* action

part and starts it. Additionally it raises an exception if the program attempts to register a schedulable that is already registered. This is the process that defines the main execution flow of the program.

3.1.2 Mission Sequencers

Two framework processes model mission sequencers. The *TopLevelMissionSequencerFW* process models the top-level mission sequencer and the *SchedulableMissionSequencerFW* models a mission sequencer used as a schedulable. This simplifies both processes because they each only have to be involved in events relevant to their context.

Both flavours of mission sequencer fetch the identifier of the next mission from their application counterpart and start that mission. However, *SafeletFW* starts *TopLevelMissionSequencerFW*, which signals to the entire model when it is terminating, to indicate that the program is done.

SchedulableMissionSequencerFW is started by a mission and signals to that mission once terminated. Since it is a schedulable, it must respond to termination requests from either its controlling mission or the mission it is executing.

3.1.3 Mission

The *MissionFW* process is started by a mission sequencer process. It then allows its application counterpart to register schedulables. It starts each schedulable and deals with their termination and cleanup. If requested, it terminates itself and its active schedulables, and signals to its controlling mission sequencer that it has done. In Sect. 3.2 we describe the *MissionFW* process in more detail and present the model of one of its actions.

3.1.4 Schedulables

Schedulables are modelled by *PeriodicEventHandlerFW*, for periodic event handlers; *AperiodicEventHandlerFW*, for aperiodic event handlers; *OneShotEventHandlerFW*, for one-shot event handlers; *ManagedThreadFW*, for managed threads; and *SchedulableMissionSequencerFW*, for mission sequencers used as schedulables. Each is started by a mission, performs its behaviour, accepts termination requests from its mission, and cleans up after it terminates.

Each event handler has actions that control its specific release pattern. Event handlers may have deadlines associated with them, and periodic event handlers have an associated period. Our models consider that periods may be overrun and deadlines may be missed, and captures the response if this happens. This

$$
\begin{array}{l}
Register \mathrel{\widehat=} \\
\quad register\,?\,s\,!\,mission \longrightarrow \\
\left(\begin{array}{l}
\left(\begin{array}{l} checkSchedulable\,.\,mission\,?\,check : (check = \mathbf{True}) \longrightarrow \\ AddSchedulable \end{array}\right) \\
\Box \\
\left(\begin{array}{l} checkSchedulable\,.\,mission\,?\,check : (check = \mathbf{False}) \longrightarrow \\ throw.illegalStateException \longrightarrow \\ \mathbf{Chaos} \end{array}\right)
\end{array}\right)
\end{array}
$$

Figure 6: The *MissionFW*'s *Register* action

allows our models to be used to check if, for example, an event handler may overrun its deadline. Managed threads are simpler and begin their release as soon as they are started. Mission sequencers used as schedulables are described above.

3.2 Mission Example

The *Circus* model of a mission is ideal to illustrate our modelling approach. The FlatBuffer application in Sect. 2 contains one mission, `FlatBufferMission`, which we model using three components described next.

As previously indicated, like every mission, an instance of *MissionFW* represents the behaviour of the mission prescribed by the SCJ paradigm. It is outlined above. The non-reactive application-specific behaviour is captured in the *OhCircus* class *FlatBufferMissionClass*. It contains the *buffer* variable, corresponding to the buffer field of the `FlatBufferMission`, and the `bufferEmpty()` method, because it is purely a data operation without any reactive behaviour.

The *FlatBufferMissionApp* process captures the reactive application-specific behaviour of the mission. It has actions modelling the behaviour of the API methods `initialize()` and `cleanup()` and actions modelling the application-defined methods: *writeSyncMeth*, *readSyncMeth*, and *bufferEmptyMeth*. It stores a reference to an instance of *FlatBufferMissionClass*, which contains the method `bufferEmpty()`. The *bufferEmptyMeth* action wraps this method, so that it can be called by other processes.

Channels on which the instance of *MissionFW* and *FlatBufferMissionApp* communicate are parametrised by the mission identifier *FlatBufferMission*; this ensures that the *FlatBufferMissionApp* communicates with the right framework process. The *FlatBufferMissionApp* instantiates and communicates with the *FlatBufferMissionClass* to call its `bufferEmpty()` method.

In an SCJ program, the `Mission`'s `initialize()` method is overridden to register the schedulables that this particular mission manages. In Fig. 5 we show the *InitializePhase* action of the *FlatBufferMissionApp* process, which models the `initialize()` method in `FlatBufferMission`. The events *initializeCall* and *initializeRet* model the call to and return from `initialize()`.

The registration of a schedulable is modelled by the event $register.s.m$, where m is the identifier of the mission registering the schedulable and s is the identifier of the schedulable being registered. The order of registration shown in

Fig. 5 corresponds to the order in the program. After registration, all registered schedulables are started simultaneously.

In *MissionFW*, *initializeCall* triggers the *Register* action (Fig. 6), which accepts a *register* event, with any schedulable identifier as long as the mission identifier is the same as this mission's. The *checkSchedulable* event indicates, via the variable *check*, if the schedulable may be registered.

If *check* is **True**, then *Register* can add the schedulable. If *check* is **False**, then the schedulable is already registered and we use the *throw* channel to model an exception being thrown and then diverge (**Chaos**). This allows the detection of an attempt to register a schedulable more than once.

3.3 Synchronisation and Suspension

The synchronisation model of SCJ constrains that of Java. First, SCJ programs cannot use `synchronized` blocks, only `synchronized` methods. Second, threads queue for a lock in order of eligibility. In SCJ, the most eligible thread is the thread at the highest priority level that has been waiting for the longest time. We model this using the type *PriorityQueue*, which is a total function from *PriorityLevel* to injective sequences of *ThreadID*. *PriorityLevel* is a free type containing the priorities available to the system and *ThreadID* is the set of thread identifiers.

Our models use extra processes to control synchronisation and suspension. In SCJ, each schedulable is executed by a thread. In our model, schedulables that call a synchronised method are associated with an instance of the *ThreadFW* process. *ThreadFW* holds the thread identifier and keeps track of its priority and interrupted status. Overall, the framework model of a schedulable that calls a synchronised method is the parallel composition of its associated *ThreadFW* process with the appropriate framework process, which depends on the type of schedulable (event handler, managed thread, and so on).

Additionally, each object used as a lock is associated with an instance of the *ObjectFW* process, which stores the threads waiting on this object and controls the threads trying to lock this object. In the FlatBuffer, the mission is used as a lock, so it has an associated instance of *ObjectFW*. Again, the overall framework model of each object that represents a paradigm component and is used as a lock is its framework process in parallel with an instance of *ObjectFW*. Non-paradigm objects used as locks are modelled in the framework by just an instance of *ObjectFW*.

The FlatBuffer program uses synchronisation and suspension to control access to the buffer in its mission. The synchronised `read()` method suspends the calling thread (by calling `wait()`) if the buffer is empty. This is wrapped in a loop that checks if the buffer is empty, to deal with spurious wake ups.

The `FlatBufferMission`'s `read()` method is modelled by the *readSyncMeth* action in the *FlatBufferMissionApp* process (Fig. 7), which shows the pattern we use for modelling all synchronised methods. The action begins and ends with the familiar call-return event pair, *readCall* and *readRet*, which correspond to the call to and return from the method. In this case, however, because this is a synchronised method, these events take an extra parameter *thread*, which is the identifier of the thread that is calling the method.

The *ObjectFW* process associated with the *FlatBufferMissionApp* process controls the synchronisation and suspension behaviour using the *startSyncMeth*,

$$
\begin{array}{l}
readSyncMeth \mathrel{\widehat=} \textbf{var}\ ret : \mathbb{Z} \bullet \\
\left(\begin{array}{l}
readCall\ .\ FlatBufferMissionMID\ ?\ caller\ ?\ thread \longrightarrow \\
\left(\begin{array}{l}
startSyncMeth\ .\ FlatBufferMissionOID\ .\ thread \longrightarrow \\
lockAcquired\ .\ FlatBufferMissionOID\ .\ thread \longrightarrow \\
\left(\left(\begin{array}{ll}
\mu X \bullet & \left(\begin{array}{l}
\textbf{var}\ loopVar : \mathbb{B} \bullet loopVar := bufferEmpty(); \\
\textbf{if}\ (loopVar = \textbf{True}) \longrightarrow \\
\quad \left(\begin{array}{l} waitCall\ .\ FlatBufferMissionOID\ .\ thread \longrightarrow \\ waitRet\ .\ FlatBufferMissionOID\ .\ thread \longrightarrow \end{array}\right) ;\ X \\
\quad \textbf{Skip} \\
[\!]\ (loopVar = \textbf{False}) \longrightarrow \textbf{Skip} \\
\textbf{fi}
\end{array}\right)
\end{array}\right) ; \right) \\
\textbf{var}\ out : \mathbb{Z} \bullet out := this\ .\ buffer; \\
this\ .\ buffer := 0; \\
notify\ .\ FlatBufferMissionOID\ !\ thread \longrightarrow \\
ret := out
\end{array}\right) ; \\
endSyncMeth\ .\ FlatBufferMissionOID\ .\ thread \longrightarrow \\
readRet\ .\ FlatBufferMissionMID\ .\ caller\ .\ thread\ !\ ret \longrightarrow \textbf{Skip}
\end{array}\right)
\end{array}
$$

Figure 7: The *FlatBufferMission* proceess's *readSyncMeth* action

lockAcquired, and *endSyncMeth* events. The *startSyncMeth* event models the beginning of a synchronized method and triggers the *ObjectFW* process to request a lock on this object by the thread calling this action.

Because the lock may already be held by another thread, the *readSyncMeth* action waits for the *lockAcquired* event (from the *ObjectFW* process) to signal that it has the lock and can proceed. After the body of the method, the *endSync* event signals that the synchronised method is complete, to trigger *ObjectFW* to release the lock on the mission currently held by the calling thread. We note that SCJ does not support Java's `ReentrantLock`, however, SCJ does support reentrant locking by allowing synchronised methods to call other synchronised methods in the same object. The *ObjectFW* process provides this behaviour; to unlock the object, after the first *lockAquired* event, each subsequent *startSyncMeth* event (which must be from the same thread) must be matched by a *endSyncMeth* event from the locking thread.

We model the call to `wait()` using the call-return event pair *waitCall* and *waitRet*. These events take the identifier of the associated *ObjectFW* instance (*FlatBufferMissionOID*, in Fig. 7) and the identifier of the *thread* calling this action. The instance of *ObjectFW* associated with the mission adds *thread* to its queue of waiting threads. The process calling *waitCall* waits for *waitRet* to communicate its identifier.

We model the call to `notify()` using the event *notify*. Like *waitCall* and *waitRet*, this event also takes the identifier of the associated *ObjectFW* process and the identifier of the *thread* calling this action. The *notify* event triggers the *ObjectFW* process to resume the most eligible thread. If there are no waiting threads, then *ObjectFW* allows the call to *notify*, but does nothing. To resume a thread, *ObjectFW* calls *waitRet* with the identifier of the thread to be resumed. SCJ Level 2 can also use `notifyAll()`, which resumes all the waiting

threads. We model a call to `notifyAll()` with the event *notifyAll*. It triggers the *ObjectFW* to call *waitRet* with the identifier of each waiting thread in eligibility order.

The complete *Circus* models of the framework processes can be found in [12], and the application processes of the FlatBuffer in [11]. In the next section, we discuss the validation and application of our models.

4 Initial Evaluation

Our *Circus* model is written to closely correspond with the SCJ API. We have frozen development of our model at version 0.100 of the SCJ language specification. One of the authors is a member of the SCJ Expert Group, which helped in clarifying ambiguities in the language specification.

Our model of Level 2 is based on the *Circus* model of Level 1 presented in [25], which has been validated against the SCJ language specification. Our model adds the features of Level 2 and updates the model to reflect recent changes in the language specification.

Our modelling effort has influenced the development of SCJ. In [13], which is under review, we present a model of the SCJ termination protocol and a proposed simplified termination protocol. The comparison of these models shows that our proposed protocol reduces the number of states in the system. This simplified protocol is useful for improving programmer understanding and further modelling efforts. Our simplified termination protocol was adopted by the SCJ expert group from version 0.96.

We have, by hand, translated 10 SCJ programs to *Circus* using our approach; the examples are summarised in Table 2. The programs are constructed to cover the features of SCJ. They range from simple tests of SCJ's features, such as different release patterns or synchronisation and suspension, to more complex programs that use nested mission sequencers to provide concurrent missions.

Further, we have developed a prototype tool[1] to automatically generate the *Circus* application models of a given SCJ application, called T$^{\text{ight}}$R$^{\text{ope}}$. We have used this prototype to produce the application models of the FlatBuffer application presented in this paper and a more complex example, both summarised in Table 3. The 10 hand-translated examples, and more realistic programs, will be considered for automatic translation as T$^{\text{ight}}$R$^{\text{ope}}$ matures.

T$^{\text{ight}}$R$^{\text{ope}}$ is a small Java program that compiles an SCJ application and explores the resulting abstract syntax trees to extract the information required for the translation. T$^{\text{ight}}$R$^{\text{ope}}$ generates the *Circus* processes, *OhCircus* classes, and *Circus* channels required to model the application-specific behaviour of the input program. These are combined with the existing fixed models of the framework previously described, to form a specification of the whole program.

To facilitate model checking and animation using FDR3 [5], we have translated our models of the framework and of full programs into `CSPm`. This translation has been optimised so that FDR3 can check specifications of even complex programs in an acceptable amount of time. We have proved that the `CSPm` version of the framework model is deadlock- and divergence-free, which lends extra validation to the framework. We have also proved that the models of the full programs that we translated do not deadlock or diverge.

[1] T$^{\text{ight}}$R$^{\text{ope}}$ can be found at `www.cs.york.ac.uk/circus/hijac/tools.html`.

Name	Description	№ Classes
Mission1	A single mission with periodic event handler that releases an aperiodic event handler	5
Mission2	A single mission with a managed thread and a one-shot event handler	5
ThreeOneShots	A single mission with three one-shot event handlers	6
ThreeThreads	A single mission with three managed threads	6
SequentialMissions	Two sequential missions, each with two managed threads	8
NestedSequencer1	A single mission with a single nested mission sequencer	7
NestedSequencer2	A mission, with three nested mission sequencers. Each has one mission controlling a periodic event handler	14
NestedSequencer3	A mission, with a nested mission sequencer that has two sequential nested missions, each with a managed thread.	8
NestedSequencer4	A complicated example using two levels of nesting. It contains 4 missions and 3 managed threads	12
NestedSequencer5	Extends NestedSequencer4, combines complex nesting, all schedulable types, and sequential missions	12

Table 2: Summary of SCJ programs translated by hand

Using the version of the CSP animator ProBE that is included in FDR3, we have animated the `CSPm` versions of the framework model and compared their behaviour with that prescribed in the SCJ language specification. This gives us confidence that the models capture the behaviour of the SCJ API. We have also used ProBE to examine the behaviour of these full models, to compare them to the running programs. We have compared the execution of our example SCJ Level 2 programs, using the IceLab [9] implementation, to animations of our models of these programs. These comparisons examined the behaviour and output from the executing programs with the corresponding events in the animated model to ensure that they have the same behaviour.

Future work in the analysis of our models includes extending the checks we make to cover more SCJ-specific criteria. We intend to check that the program does not attempt to register its top-level mission sequencer or throw any of the exceptions that we model. Because we model exceptions using an event followed by divergence, they are flagged by a divergence-freedom check. However, the counter examples provided by a specific check would be more useful during SCJ development. These SCJ-specific checks will be standardised for easy reuse.

In summary, because our framework model captures the behaviour of the SCJ paradigm separately from the program-specific behaviour, we can reason about it in isolation. We have used FDR3 to prove that the framework model does not deadlock or diverge. Models of full SCJ Level 2 programs can be model checked and animated in FDR3. Our formal semantics of the Level 2 paradigm enables further areas of study for SCJ Level 2, such as theorem proving.

Name	Description	№ Classes	Translation Time
FlatBuffer	Small program using managed threads and synchronisation	6	1.2 seconds
Aircraft	Program using a schedulable mission sequencer to represent phases of aircraft flight	25	2.3 seconds

Table 3: Summary of SCJ programs translated by T$^{\text{ight}}$R$^{\text{ope}}$

5 Related Work

This is the first work supporting verification for SCJ Level 2 programs. K-Java [1] models a subset of SE Java 1.4 and produces executable specifications for model checking. However, SCJ programs have features not included in SE Java. The authors of [22] present a technique for translating SCJ programs into timed automata models. However, their technique appears to only be aimed at Levels 0 and 1. Further, neither of these techniques provide support for top-down refinement of SCJ Level 2 programs or refinement-based reasoning.

RSJ [10] is a adaptation of the Java PathFinder [7] that explores all possible schedulings of the threads within an SCJ program to check for scheduling-dependent errors. It, however, does not cater for SCJ Level 2 programs.

Older versions of the SCJ specification define annotations for specifying compliance level, behavioural, or memory restrictions. Previous approaches to ensuring the safety of SCJ programs have used these annotations to provide run-time checks [19] or to specify checkable program constraints [6]. However, the memory annotations have been moved to an appendix of the standard as they were judged not ready for standardisation.

Our modelling approach is similar to that of [25] in capturing the paradigm of SCJ Level 1. The underlying structure of programs written in Level 2 and Level 1 is the same, however, Level 2 allows much more complicated program hierarchies and provides more complicated features (such as suspension).

6 Conclusion

We have presented the first formal semantics of SCJ Level 2, using the *Circus* family of specification languages. It is an essential ingredient to enable customised top-down development of SCJ Level 2 programs that are correct by construction. Our models provide this development process with a target for SCJ Level 2.

The features *Circus* provides make it a good fit for modelling object-orientated languages, such as SCJ. A *Circus* process provides similar encapsulation to classes and the language can capture variables and methods. This means that our models correspond very closely to the programs they model.

We have validated our model of the SCJ API and Level 2 programs by translating them into `CSPm` and model checking it using FDR3 to show that it does not deadlock or diverge. Our prototype tool, called T$^{\text{ight}}$R$^{\text{ope}}$, has produced *Circus* models of SCJ applications. Work is ongoing to update the tool, so that it can generate the models for all of our example applications.

In addition to the further areas of study that our work enables, future work includes the formalisation of the translation strategy that we use to derive the application models from the SCJ programs. The translation strategy also needs to be evaluated on more applications to further test our modelling approach.

Acknowledgements

This research reported in this paper is funded by the UK EPSRC under grant EP/H017461/1. No new primary data was produced during this work. One of the authors is a member of the SCJ Expert Group; we would like to thank the other members of the Expert Group. We would also like to thank Frank Zeyda, Alan Burns, and Thomas Gibson-Robinson for their very helpful suggestions.

References

[1] Bogdanas, D., Roşu, G.: K-Java: A Complete Semantics of Java. SIGPLAN Not. 50(1), 445–456 (Jan 2015)

[2] Cavalcanti, A., Sampaio, A., Woodcock, J.: Unifying Classes and Processes. Software & Systems Modeling 4(3), 277–296 (2005), dx.doi.org/10.1007/s10270-005-0085-2

[3] Cavalcanti, A., Wellings, A., Woodcock, J., Wei, K., Zeyda, F.: Safety-Critical Java in Circus. In: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 20–29. JTRES '11, ACM, New York, NY, USA (2011), doi.acm.org/10.1145/2043910.2043915

[4] EUROCAE and RTCA: Software Considerations in Airborne Systems and Equipment Certification. Norm ED-12C, EUROCAE (2012)

[5] Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: Failures Divergences Refinement (FDR) Version 3 (2013), www.cs.ox.ac.uk/projects/fdr/

[6] Haddad, G., Hussain, F., Leavens, G.T.: The Design of SafeJML, a Specification Language for SCJ with Support for WCET Specification. In: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 155–163. JTRES '10, ACM, New York, NY, USA (2010), doi.acm.org/10.1145/1850771.1850793

[7] Havelund, K., Pressburger, T.: Model Checking JAVA Programs using JAVA PathFinder. International Journal on Software Tools for Technology Transfer 2(4), 366–381 (2000), dx.doi.org/10.1007/s100090050043

[8] Hoare, C.A.R.: Communicating Sequential Processes. www.usingcsp.com/cspbook.pdf (2004)

[9] IceLab: IceLab. www.icelab.dk/index.html, www.icelab.dk/index.html

[10] Kalibera, T., Parizek, P., Malohlava, M., Schoeberl, M.: Exhaustive Testing of Safety-Critical Java. In: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 164–174. JTRES '10, ACM, New York, NY, USA (2010), doi.acm.org/10.1145/1850771.1850794

[11] Luckcuck, M.: hiJaC Case Studies (2016), www.cs.york.ac.uk/circus/hijac/case.html, [Online; accessed 14-January-2016]

[12] Luckcuck, M.: Safety-Critical Java Level 2 Framework Model (2016), www.cs.york.ac.uk/circus/publications/techreports/reports/SCJLevel2Framework.pdf

[13] Luckcuck, M., Wellings, A., Cavalcanti, A.: Safety-Critical Java: Level 2 in Practice (Submitted)

[14] Morgan, C.: Programming from Specifications. Prentice-Hall, Inc. (1990)

[15] Schoeberl, M.: A Java Processor Architecture for Embedded Real-Time Systems. Journal of Systems Architecture 54(1–2), 265 – 286 (2008), /www.sciencedirect.com/science/article/pii/S1383762107000963

[16] Sherif, A., Cavalcanti, A., Jifeng, H., Sampaio, A.: A process algebraic framework for specification and validation of real-time systems. Form. Asp. Comput. 22(2), 153–191 (15 Jul 2009)

[17] Søndergaard, H., Korsholm, S.E., Ravn, A.P.: Safety-critical Java for Low-end Embedded Platforms. In: Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems. pp. 44–53. JTRES '12, ACM, New York, NY, USA (2012)

[18] Spivey, J.M.: The Z Notation: A Reference Manual. International Series in Computer Science (1992)

[19] Tang, D., Plsek, A., Vitek, J.: Static Checking of Safety Critical Java Annotations. In: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 148–154. ACM, Prague, Czech Republic (2010)

[20] The Open Group: Safety-Critical Java Technology Specification V0.100. Tech. rep., The Open Group (27 December 2014), jcp.org/en/jsr/detail?id=302

[21] The Real-Time for Java Expert Group: Real-Time Specification for Java Language Specification. www.timesys.com/java/ (2005)

[22] Thomsen, B., Luckow, K.S., Leth, L., Bøgholm, T.: From Safety Critical Java Programs to Timed Process Models. In: Programming Languages with Applications to Biology and Security, pp. 319–338. Lecture Notes in Computer Science, Springer International Publishing (2015)

[23] Wellings, A., Luckcuck, M., Cavalcanti, A.: Safety-Critical Java Level 2: Motivations, Example Applications and Issues. In: Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems. pp. 48–57. JTRES '13, ACM, New York, NY, USA (2013), doi.acm.org/10.1145/2512989.2512991

[24] Woodcock, J., Cavalcanti, A.: The Semantics of Circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002: Formal Specification and Development in Z and B, Lecture Notes in Computer Science, vol. 2272, pp. 184–203. Springer Berlin Heidelberg (2002)

[25] Zeyda, F., Lalkhumsanga, L., Cavalcanti, A., Wellings, A.: Circus Models for Safety-Critical Java Programs. The Computer Journal (2013), comjnl.oxfordjournals.org/content/early/2013/07/02/comjnl.bxt060.abstract