The Safety-Critical Java Mission Model: a formal account

Frank Zeyda, Ana Cavalcanti, and Andy Wellings

University of York, Deramore Lane, York, YO10 5GH, UK
{Frank.Zeyda,Ana.Cavalcanti,Andy.Wellings}@cs.york.ac.uk

Abstract. Safety-Critical Java (SCJ) is a restriction of the Real-Time Specification for Java to support the development and certification of safety-critical applications. It is the result of an international effort from industry and academia. Here we present the first formalisation of the SCJ execution model, covering missions and event handlers. Our formal language is part of the *Circus* family; at the core, we have Z, CSP, and Morgan's calculus, but we also use object-oriented and timed constructs from the *OhCircus* and *Circus Time* variants. Our work is a first step in the development of refinement-based reasoning techniques for SCJ.

Keywords. *Circus*, real-time systems, models, verification, RTSJ.

1 Introduction

Safety-Critical Java (SCJ) [11] restricts the Java API and execution model in such a way that programs can be effectively analysed for real-time requirements, memory safety, and concurrency issues. This facilitates certification under standards like DO-178B, for example. It also makes possible the development of automatic tools that support analysis and verification.

SCJ is realised within the Real-Time Specification for Java (RTSJ) [21]. The purpose of RTSJ itself is to define an architecture that permits the development of real-time programs, and SCJ reuses some of RTSJ's concepts and actual components, albeit restricting the programming interface. SCJ also has a specific execution model that imposes a rigid structure on how applications are executed.

The SCJ specification, as designed by the JSR 302 expert group, comprises informal descriptions and a reference implementation [8]. As a result, analysis tools have been developed to establish compliance with the SCJ restrictions [20].

In this paper, we complement the existing work on SCJ by presenting a formal model of its execution framework in a *Circus*-based language. The Open Group's informal account of SCJ [8] relies on text and UML diagrams, and our objective is to formalise the execution model. *Circus* [5] is a refinement notation for state-rich reactive systems. Its variants cover, for instance, aspects of time and mobility. We use its object-oriented variant, *OhCircus*, as our base notation.

Our formal model first elicits the conceptual behaviour of the SCJ framework, and secondly illustrates the translation of actual SCJ programs into their *OhCircus* specifications in a traceable manner. For now, we ignore certain aspects of SCJ, such as the memory model, which we discuss in a separate paper [7], and scheduling policy. Our focus is the top-level design and execution of SCJ programs, and its primary framework and application components.

The SCJ framework as designed in Java is a reflection of a general programming paradigm. It embeds a particular view of data operations, memory, and event-based versus thread-based designs [22]. Our model identifies the fundamental concepts of SCJ at a level where it can be regarded itself as a programming language. The fact that it can be realised on top of Java and the RTSJ is a bonus. It is conceivable, however, to implement specific support based on other mainstream languages, or even define an entirely new language, and formalisation is conducive to the development of such a language which is our future ambition.

What we present here is a precise semantics for core elements of SCJ. It enables formal verification of SCJ applications beyond the informal validation of statically checkable properties currently available [20]. *OhCircus* provides a notion of refinement, and our work is an essential first step to justify development and verification methods that can produce high-quality SCJ implementations.

Our work also highlights the need for a particular integration of *Circus* variants. Their Unifying Theories of Programming (UTP) [13] foundation facilitates this work. The UTP is a uniform framework in which the semantics of a variety of programming paradigms can be expressed and linked. UTP theories have already been presented for *Circus* and *Circus Time* [16,18], and also for object-orientation [17] and the SCJ memory model [7]. We thus identify the *Circus* variant necessary to formalise SCJ programs. The design of the semantic model establishes the right level of detail for reasoning about SCJ, and determines where the added expressiveness of Java should be ignored.

Finally, our work guides the construction of a platform for reasoning. Our models are free from the noise that originates from the expressiveness of Java. They allow us to reason about SCJ programs using refinement-based techniques. For verification, we can construct models of particular programs, and use the *Circus* and UTP techniques for reasoning. For development, we can start from an abstract specification, and develop implementations that follow the structure and respect the restrictions of our models.

In the next section, we introduce the SCJ framework and a case study used throughout as an example. We also provide a brief overview of our formal notation. In Section 3 we present our models and modelling approach. In Section 4, we discuss our contributions and some related work.

2 Preliminaries

In this section we present first the SCJ execution model and introduce an example: an automotive cruise controller. Afterwards, we present *Circus* and *OhCircus*.

2.1 Safety-Critical Java

SCJ recognises that safety-critical software varies considerably in complexity. Consequently, there are three compliance levels for SCJ programs and framework implementations. In this work, we are concerned with Level 1, which, roughly,

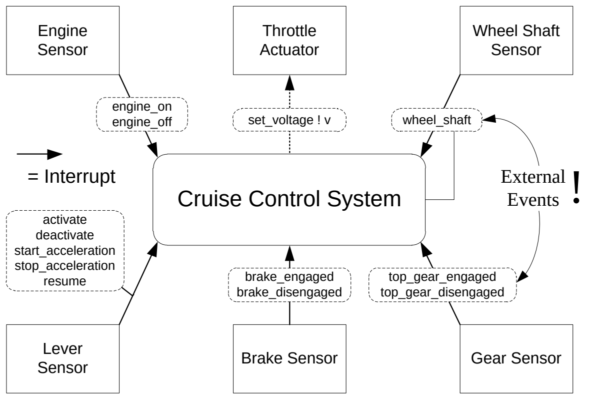

Fig. 1. ACCS interactions

corresponds in complexity to the Ravenscar profile for Ada [4]. Level 1 applications support periodic event handlers and aperiodic event handlers.

The SCJ programming model is based on the notion of missions. They are sequentially executed by an application-specific mission sequencer provided by a safelet, the top-level entity of an SCJ application. All these concepts are realised by either interfaces or abstract classes. Namely, they are the `Safelet` interface, and the abstract classes `MissionSequencer` and `Mission` (see Fig. 2).

A Level 1 mission consists of a set of asynchronous event handlers; both periodic and aperiodic handlers are supported. Each aperiodic handler is associated with a set of events: firing of one of them causes the handler method to be scheduled for execution. Periodic event handlers, on the other hand, are controlled by a timer. Event handlers are also provided through abstract classes whose handling method must be implemented by concrete subclasses (see Fig. 2).

A cruise control system As an example of an SCJ program, and to illustrate our modelling approach, we present an implementation of Wellings' automotive cruise control system (ACCS) in [21] that uses SCJ Level 1.

The goal of an ACCS is to automatically maintain the speed of a vehicle to a value set by the driver; in Fig. 1 we give an overview of its main components and commands. Explicit commands are given by a lever whose positioning corresponds to the following instructions: *activate*, to turn on the ACCS if the car is in top gear, and maintain (and remember) the current speed; *deactivate*, to turn off the ACCS; *start accelerating*, to accelerate at a comfortable rate; *stop accelerating*, to stop accelerating and maintain (and remember) the current speed; and *resume* to return to the last remembered speed and maintain it. Implicit commands are issued when the driver changes gear, operates the brake pedal, or switches on or off the engine. The ACCS is deactivated when the driver changes out of top gear, presses the brake pedal, or switches the engine off.

The speed of the vehicle is measured via the rotation of the shaft that drives the back wheels. The shaft generates an interrupt for each rotation, which causes an event being fired and an associated handler being scheduled for execution.

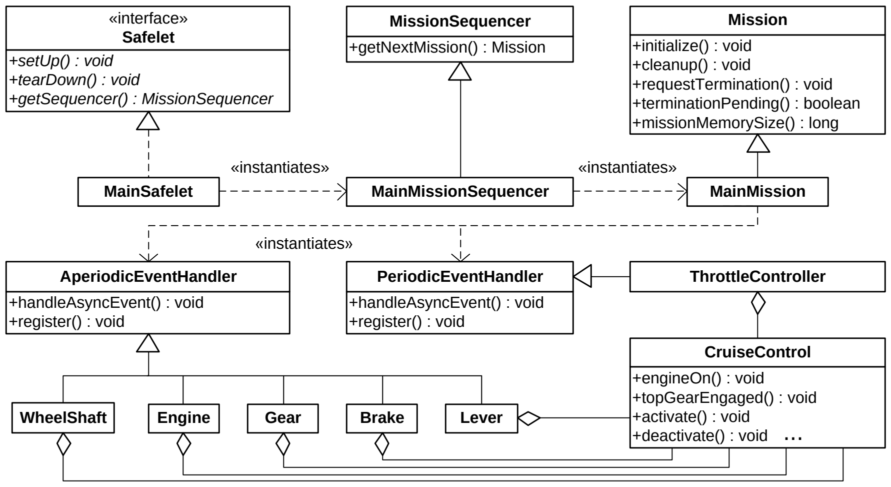

Fig. 2. UML class diagram for the cruise controller

The actual speed of the car depends on the throttle position, which is determined by the depression of the accelerator pedal and a voltage supplied by the ACCS. The combination of these values is performed outside the ACCS.

Sensors detect external happenings and generate appropriate interrupts, as illustrated in Fig. 1. These interrupts are reflected in the SCJ program by the firing of SCJ events that correspond to the possible happenings. For the setting of the throttle voltage, communication of the new voltage value to the ACCS components is realised in the program using a hardware data register.

Fig. 2 presents a UML class diagram that gives an overview of the design of the ACCS as an SCJ Level 1 safelet. As said above, `Safelet` is an interface, and the classes `MissionSequencer`, `Mission`, `AperiodicEventHandler` and `PeriodicEventHandler` are abstract. They are part of the SCJ API developed on top of the RTSJ API to capture the SCJ programming model.

`MainSafelet` is the entry point for the application. It provides the method `getSequencer()` that returns the mission sequencer. The other two methods `setUp()` and `tearDown()` are provided for initialisation and cleanup tasks. The `MainMissionSequencer` class constructs instances of the `Mission` class, by implementing `getNextMission()`. Concrete subclasses of `Mission` have to implement the `initialize()` and `missionMemorySize()` methods. The former creates the periodic and aperiodic event handlers of the mission. The handlers register themselves with the mission by way of the `register()` method.

Both periodic and aperiodic handlers implement `handleAsyncEvent()` to specify their behaviour when the handler is released. The two extra methods `requestTermination()` and `terminationPending()` cannot be overridden; they allow for the mission to be terminated by one of the handlers.

Fig. 2 does not show all components of the SCJ API. There are eight classes that realise the mission framework, twelve classes in the handler hierarchy, five classes that deal with real-time threads, seven classes concerned with scheduling, and ten classes for the memory model. The formal model that we present here abstracts from all these details of the realisation of the SCJ Level 1 programming paradigm in Java. We capture the main concepts of its novel execution model. This enables reasoning based on the core components of the SCJ paradigm.

2.2 *Circus* and *OhCircus*

The *Circus* language [5] is a hybrid formalism that includes elements from Z [19], CSP [12], and imperative commands from Morgan's calculus [15]. Several examples are provided in the next section: see Fig. 4, 5, 6, and 7, for instance.

Like in CSP, the key elements of *Circus* models are processes that interact with each other and their environment via channels. Unlike CSP, *Circus* processes may encapsulate a state. The definition of a *Circus* process hence includes a paragraph that identifies the state of the process using a Z schema.

The behaviour of a process is defined by its main action (which may reference local actions, introduced for structuring purposes). The language of actions includes all constructs from CSP, such as *Skip* and *Stop*, input and output prefixes, sequencing, parallelism, interleaving and hiding, as well as operations to modify the state. Parallelism and interleaving are parametrised in terms of the state components that each parallel action can modify to avoid potential write conflicts. State operations can be specified either by Z operation schemas or guarded commands. We explain the details of the notation as needed.

OhCircus [6] extends *Circus* with an additional notion of class. Unlike processes, objects can be used in arbitrary mathematical expressions. The permissible notation for *OhCircus* class methods includes all schema operations, guarded commands, and some additional notations used to instantiate new data objects, invoke methods, access object fields, and support inheritance.

Processes describe the active behaviour of the model (or of its components), including the whole system. Classes model passive data objects and operations performed on them. In the following section we present our model for SCJ programs. The notation we use is *OhCircus*. We, however, use a few action operators of the *Circus Time* [18] variant, and object references from our previous SCJ memory model in [7]. The latter is specified at the level of the Unifying Theories of Programming [13], the semantic framework of *Circus* and its extensions.

3 Framework and application models

Our model of SCJ factors into two dimensions: a generic framework model, and an application model that corresponds to a particular concrete SCJ program. We specify the semantics of safelets, the mission sequencer, missions, and aperiodic as well as periodic event handlers. To illustrate the application model, we make use of the cruise controller application as it was presented in the previous section.

Fig. 3 presents an overview of the structure of the model of a typical SCJ application — here the cruise controller. Each of the five top-level boxes refers to a process that realises a specific component of the SCJ programming model.

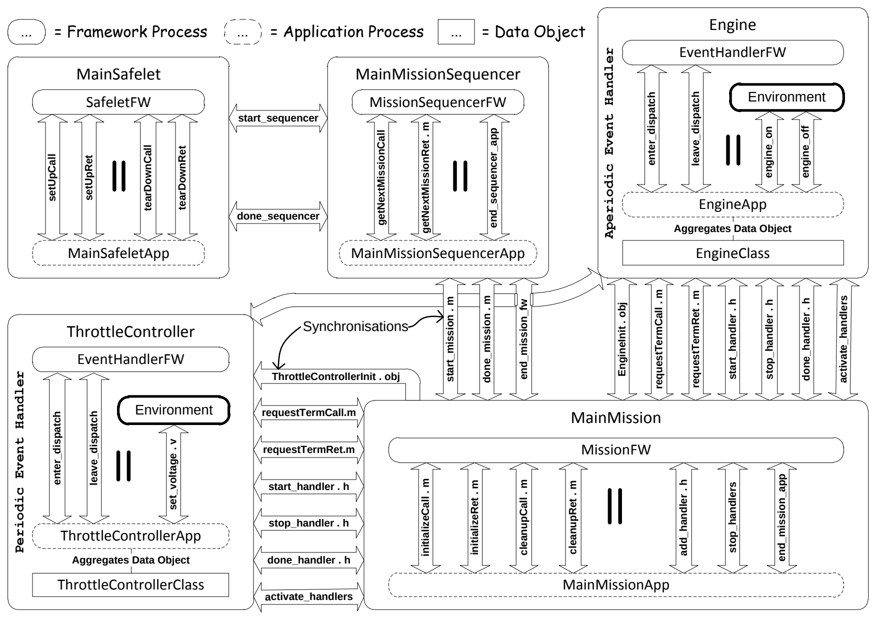

Fig. 3. Structure of the model the SCJ cruise controller

We label these boxes with the process names. Arrows indicate the channels on which the components communicate. For instance, the processes *MainSafelet* and *MainMissionSequencer* communicate on *start_sequencer* and *done_sequencer*.

The model of the application is obtained by parallel composition of the top-level processes, and by hiding all but the external channels. These define the interface of the system; for example, we define the event *engine_on* to represent the happening that occurs when the engine is switched on.

Each top-level process is itself defined by the parallel composition of a generic framework process (suffix *FW*), and a process that is in direct correspondence with the Java code (suffix *App*). We have an instance of the *EventHandlerFW* framework process for each handler. To obtain the model of an existing SCJ program, we can follow the strategy explained below to construct the *App* processes, and use the *FW* processes as defined later on; except only that, in the case of a handler *App* process, we need to be aware of the events the handler is bound to, and declare channels to represent them.

The following provides the definition of the *MainSafelet* process.

$$
\begin{array}{l}
\textbf{channelset}\ \mathit{MainSafeletChan} \mathrel{\widehat=} \\
\quad \{\!|\ \mathit{setUpCall}, \mathit{setUpRet}, \mathit{tearDownCall}, \mathit{tearDownRet}\ |\!\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{process}\ \mathit{MainSafelet} \mathrel{\widehat=} \\
\quad (\mathit{SafeletFW} \ [\![\ \mathit{MainSafeletChan}\ ]\!]\ \mathit{MainSafeletApp}) \setminus \mathit{MainSafeletChan}
\end{array}
$$

The channels on which framework and application process communicate are hidden (operator $\setminus$). Here, these are *setUpCall*, *setUpRet*, *tearDownCall*, and *tearDownRet*. Above, a channel set *MainSafeletChan* is defined to contain all these channels. In the definition of *MainSafelet*, it is used to define the synchronisation set of the parallelism (operator $[\![\dots]\!]$), and the set of channels to be hidden. The synchronisation set defines the channels over which communication requires synchronisation between the two parallel processes.

We differentiate between channels that represent framework events, and channels that represent method calls. Channels suffixed with *Call* and *Ret* encode method calls. Method calls are in some cases modelled by channel communications rather than mere *OhCircus* data operations to allow the framework processes to trigger or respond to those calls. A call to `requestTermination()`, for instance, has to interact with the mission framework process. We then require a *Call* and a *Ret* channel for this method.

In the following we specify each of the top-level processes.

3.1 Safelet model

The framework process *SafeletFW* for a safelet is given below; it has no state.

$$
\begin{array}{l}
\textbf{process}\ \mathit{SafeletFW} \mathrel{\widehat=} \textbf{begin}\\
\quad \mathit{SetUp} \mathrel{\widehat=} \mathit{setUpCall} \longrightarrow \mathit{setUpRet} \longrightarrow \mathit{Skip}\\
\quad \mathit{Execute} \mathrel{\widehat=} \mathit{start\_sequencer} \longrightarrow \mathit{done\_sequencer} \longrightarrow \mathit{Skip}\\
\quad \mathit{TearDown} \mathrel{\widehat=} \mathit{tearDownCall} \longrightarrow \mathit{tearDownRet} \longrightarrow \mathit{Skip}\\
\quad \bullet\ \mathit{SetUp}\ ;\ \mathit{Execute}\ ;\ \mathit{TearDown}\\
\textbf{end}
\end{array}
$$

The main action, which is given at the end after the $\bullet$, sequentially executes the *SetUp*, *Execute* and *TearDown* local actions. They correspond to the initialisation, execution, and cleanup phases of the safelet. *SetUp* and *TearDown* synchronise in sequence (prefixing operator $\longrightarrow$) on the $\mathit{setUp}[\mathit{Call}/\mathit{Ret}]$ and $\mathit{tearDown}[\mathit{Call}/\mathit{Ret}]$ channels, before terminating (basic action *Skip*). The synchronisations model calls to the methods `setUp()` and `tearDown()` of the Java class. Since the methods are parameterless and do not return any values, the communications through the channels are just synchronisations: there is no input or output. The methods themselves are specified in the application process as exemplified below: the framework process defines the flow of execution, and the application process defines specific program functionality. *Execute* raises two framework events: *start_sequencer* to start the mission sequencer, and *done_sequencer* to detect its termination. These channels are exposed by the *Safelet* component (that is, not hidden in its definition as shown above), and their purpose is to control the *MissionSequencer* component which we specify later on.
We now present the application process for the safelet in our example.

$$
\begin{array}{l}
\textbf{process}\ \mathit{MainSafeletApp} \mathrel{\widehat=} \textbf{begin}\\
\quad \mathit{setUpMeth} \mathrel{\widehat=} \mathit{setUpCall} \longrightarrow \mathit{Skip}\ ;\ \mathit{setUpRet} \longrightarrow \mathit{Skip}\\
\quad \mathit{tearDownMeth} \mathrel{\widehat=} \mathit{tearDownCall} \longrightarrow \mathit{Skip}\ ;\ \mathit{tearDownRet} \longrightarrow \mathit{Skip}\\
\quad \mathit{Methods} \mathrel{\widehat=} \mu X \bullet \mathit{setUpMeth}\ ;\ X\\
\quad \bullet\ \mathit{Methods} \mathbin{\triangle} \mathit{tearDownMeth}\\
\textbf{end}
\end{array}
$$

The specification is trivial here since `setUp()` and `tearDown()` in `MainSafelet`

```
**process** $MissionSequencerFW \mathrel{\widehat=}$ **begin**
    $Start \mathrel{\widehat=} start\_sequencer \longrightarrow Skip$
    $Execute \mathrel{\widehat=} \mu X \bullet getNextMissionCall \longrightarrow getNextMissionRet\,?\,next \longrightarrow$
        **if** $next \neq null \longrightarrow start\_mission\,.\,next \longrightarrow done\_mission\,.\,next \longrightarrow X$
        $[\!]\ next = null \longrightarrow Skip$
        **fi**
    $Finish \mathrel{\widehat=} end\_sequencer\_app \longrightarrow end\_mission\_fw \longrightarrow done\_sequencer \longrightarrow Skip$
    $\bullet\ Start\ ;\ Execute\ ;\ Finish$
**end**
```

Fig. 4. Mission sequencer framework process

do not contain any code in the ACCS implementation. More important is the modelling approach, which we adopt in all application processes. A local action *Methods* recursively (operator μ) offers a choice of actions that correspond to methods of the SCJ class; the choice is exercised by the associated framework process. For the safelet application process, the only action offered by *Methods* is *setUpMeth*. In the main action, we have a call to *Methods*. Termination occurs when there is a call to the `tearDown()` method. In the main action, this is captured by an interrupt (operator $\triangle$) that calls the *tearDownMeth* action.

A method action, here *setUpMeth* or *tearDownMeth*, synchronises on the channel that represents a call to it, *setUpCall* or *tearDownCall*, for instance, and then executes actions that correspond to the method implementation. Since, as already mentioned, `setUp()` and `tearDown()` in `MainSafelet` do not contain any code, in our example above, these actions are just *Skip*. At the end the method action synchronises on the channel that signals the return of the call, *setUpRet* or *tearDownRet*, for instance. If the method has parameters or returns a value, the call and return channels are used to communicate these values. Examples of our encoding of parametrised methods are shown below.

3.2 Mission sequencer model

The mission sequencer process (Fig. 4) communicates with the safelet process to determine when it has to start, and to signal its termination.

The main action executes *Start* to wait for the mission sequencer to be started, which is signalled by a synchronisation on *start_sequencer*. Afterwards, execution proceeds as specified by a recursion in the action *Execute*. In each iteration, it synchronises on the channels *getNextMissionCall* and *getNextMissionRet* to obtain the next mission via *next*. This corresponds to a call to the SCJ method `getNextMission()`. Since it returns a (mission) object, *getNextMissionRet* takes as input a value *next* of type *MissionId* (containing identifiers for the missions of an application). A special mission identifier *null* is used to cater for the case in which the method returns a Java `null` reference to signal that there are no more missions to execute. In *Execute*, a conditional checks the value of *next*. If it is not *null*, synchronisations on $start\_mission\,.\,next$ and $done\_mission\,.\,next$ are used to control the *Mission* process (defined later on) that manages execution of the particular mission *next*, and then *Execute* recurses (calls X) to handle the next mission. Otherwise, *Execute* finishes. At the end, in the *Finish* action, synchronisation on *end_sequencer_app* is used to terminate the mission sequencer application process. Next, synchronisation on *end_mission_fw* terminates the mission framework process. Finally, synchronisation on *done_sequencer* acknowledges to the safelet process that the mission sequencer has finished.

For our example, the mission sequencer application process is as follows.

```
process $MainMissionSequencerApp \mathrel{\widehat=}$ begin
  state $MainMissionSequencerState == [mission\_done : BOOL]$
  $Init \mathrel{\widehat=} mission\_done := FALSE$
  $getNextMissionMeth \mathrel{\widehat=} getNextMissionCall \longrightarrow$
    if $mission\_done = FALSE \longrightarrow$
      $mission\_done := TRUE \; ; \; getNextMissionRet \, ! \, MainMissionId \longrightarrow Skip$
    $[\!] \; \neg \; mission\_done = FALSE \longrightarrow getNextMissionRet \, ! \, null \longrightarrow Skip$
    fi
  $Methods = \mu X \bullet getNextMissionMeth \; ; \; X$
  $\bullet \; Init \; ; \; (Methods \bigtriangleup end\_sequencer\_app \longrightarrow Skip)$
end
```

This is a more complete illustration of our approach to modelling SCJ classes as *Circus* processes. The member variables of the class become state components. In the above example, we have one state component *mission_done* corresponding to a variable of the same name in the SCJ class `MainMissionSequencer`. We define a free type $BOOL ::= TRUE \mid FALSE$ to support boolean values in Z.

The action *Init* specifies the constructor. Other method actions are named after the methods of the class. In the case of the mission sequencer application class modelled above, we have just the method `getNextMission()`.

The main action of an application process is always of the above shape: a call to *Init*, if present, and a call to *Methods*, with an interrupt that allows a controlling process to terminate it via a special event (here *end_sequencer_app*). In the case of the safelet application process discussed earlier, the special termination event corresponded also to a call to its `tearDown()` method.

In *MainMissionSequencerApp*, the specification of *getNextMissionMeth* is in direct correspondence with the code of `getNextMission()`. We have a conditional that, depending on the value of *mission_done* updates its value and outputs (returns) the next mission or *null*. The difference is that, instead of representing a mission by an object, we use constants of type *MissionId*. In our example, since we have only one mission, we have just one constant *MainMissionId*.

We omit the definition of the process *MainMissionSequencer*, which is a parallel composition of *MissionSequencerFW* and *MainMissionSequencerApp*, similar to that used to define *MainSafelet* at the beginning of this section.

3.3 Mission model

The purpose of a mission process, defined by a parallelism between the mission framework process and an associated mission application process, is to create the mission's event handlers, execute the mission by synchronously starting them,

$$\begin{array}{l}
\mathbf{process}\ \mathit{MissionFW} \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\ \mathit{MissionFWState} == [\mathit{mission} : \mathit{MissionId}, \mathit{handlers} : \mathbb{F}\ \mathit{HandlerId}] \\
\quad \mathit{Init} == [\mathit{MissionFWState}\,' \mid \mathit{mission}' = \mathit{null} \land \mathit{handlers}' = \varnothing] \\
\quad \mathit{Start} \mathrel{\widehat=} \mathit{Init}\ ;\ \mathit{start\_mission}\,?\,m \longrightarrow \mathit{mission} := m \\
\quad \mathit{AddHandler} \mathrel{\widehat=} \mathbf{val}\ \mathit{handler} : \mathit{HandlerId} \bullet \mathit{handlers} := \mathit{handlers} \cup \{\mathit{handler}\} \\
\quad \mathit{Initialize} \mathrel{\widehat=} \mathit{initializeCall}\,.\,\mathit{mission} \longrightarrow \\
\qquad \left(\mu X \bullet \left(\begin{array}{l} \mathit{add\_handler}?h \longrightarrow (\mathit{AddHandler}(h);\ X) \\ \Box \\ \mathit{initializeRet}\,.\,\mathit{mission} \longrightarrow \mathit{Skip} \end{array} \right) \right) \\
\quad \mathit{StartHandlers} \mathrel{\widehat=} \interleave\ h : \mathit{handlers} \bullet \mathit{start\_handler}\,.\,h \longrightarrow \mathit{Skip} \\
\quad \mathit{StopHandlers} \mathrel{\widehat=} \interleave\ h : \mathit{handlers} \bullet \mathit{stop\_handler}\,.\,h \longrightarrow \mathit{done\_handler}\,.\,h \longrightarrow \mathit{Skip} \\
\quad \mathit{Execute} \mathrel{\widehat=} \mathit{StartHandlers}; \mathit{activate\_handlers} \longrightarrow \mathit{stop\_handlers} \longrightarrow \mathit{StopHandlers} \\
\quad \mathit{Cleanup} \mathrel{\widehat=} \mathit{cleanupCall}\,.\,\mathit{mission} \longrightarrow \mathit{cleanupRet}\,.\,\mathit{mission} \longrightarrow \mathit{Skip} \\
\quad \mathit{Finish} \mathrel{\widehat=} \mathit{end\_mission\_app}\,.\,\mathit{mission} \longrightarrow \mathit{done\_mission}\,.\,\mathit{mission} \longrightarrow \mathit{Skip} \\
\quad \bullet\ (\mu X \bullet \mathit{Start}\ ;\ \mathit{Initialize}\ ;\ \mathit{Execute}\ ;\ \mathit{Cleanup}\ ;\ \mathit{Finish}\ ;\ X) \\
\qquad\qquad \triangle\ \mathit{end\_mission\_fw} \longrightarrow \mathit{Skip} \\
\mathbf{end}
\end{array}$$

Fig. 5. Mission framework process

wait for their termination, and afterwards finish the mission. It also allows the termination of the mission by a handler at any point.

Fig. 5 presents the framework process for mission execution. Its state has two components: the identifier *mission* of the mission being executed, if any, and its finite set *handlers* of handlers. The handlers are identified by values of a type *HandlerId*. The action *Init* is a standard Z operation to initialise the state. The declaration $\mathit{MissionFWState}'$ introduces dashed versions of the state component names ($\mathit{mission}'$ and $\mathit{handlers}'$) to represent the values of the components after initialisation. *Init* defines that, initially, there is no mission executing, so that the value of *mission* is *null*, and therefore, the set of handlers is empty.

In the main action, we use again the modelling pattern where we have a sequence of actions that define the different phases of the entity life-cycle, here a mission. In the case of the mission framework process, a recursion perpetually calls this sequence of actions, because this process controls all missions in the program, and so repetitively offers its service. Termination of the service is determined by the mission sequencer process using the channel *end_mission_fw*.

The *Start* action initialises the state and waits for the mission sequencer to start a mission. Since this framework process can handle any mission, *Start* uses *start_mission* to take a mission identifier *m* as input, and records it in *mission*. *Finish* uses that mission identifier to terminate the application process for the mission with a synchronisation on $\mathit{end\_mission\_app}\,.\,\mathit{mission}$, and to signal to the mission sequencer that the mission has finished with $\mathit{done\_mission}\,.\,\mathit{mission}$.

The *Initialize* action models the initialisation phase which is initiated by the framework calling the `initialize()` method. It is specified using a recursion which continually accepts requests from the mission application process, through the channel *add_handler*, to add a handler h to the mission (this is achieved by the parametrised action *AddHandler*). Besides, the application process may use the event $initialiseRet\,.\,mission$ to terminate *Initialize* at any time.

In the action *Execute*, first of all, all handlers are started with a call to the action *StartHandlers*. It uses synchronisations $start\_handler.h$ to start in interleaving (operator $|||$) all handlers h recorded in the state. The processes corresponding to the handlers h synchronise with the mission process on *start_handler*.

The handlers do not immediately become active after they are started. For that, the action *Start* uses a channel *activate_handlers*. All handler processes synchronise on it, but only those that previously synchronised on *start_handler* proceed to execute their active behaviour. In this way, we ensure that handlers can be initialised asynchronously, but have to start execution synchronously.

Termination of the handlers is initiated by the mission application process with a synchronisation on *stop_handlers*, raised by the action corresponding to `requestTermination()`. After that, *Execute* calls the action *StopHandlers*. For each handler h of the mission, *StopHandlers* uses $stop\_handler.h$ to stop it, and then waits for the notification $done\_handler.h$ that it actually terminated.

Finally, the *Cleanup* action calls the action of the mission application process corresponding to its `cleanup()` method. In what follows we discuss the application process, using the ACCS `MainMission` class as example.

Action methods are encoded as before; the model for `initialize()` is different, though, since it not only results in the creation of data objects, but also provides information to the framework about the handlers that have been created. Below we include an extract of its specification for the ACCS model.

$$
\begin{array}{l}
initializeMeth \mathrel{\widehat=} initializeCall\,.\,MainMissionId \longrightarrow \\
\quad \mathbf{var} \ldots;\ speed : SpeedMonitorClass; \\
\qquad\quad throttle : ThrottleControllerClass; \\
\qquad\quad cruise : CruiseControlClass;\ \ldots \bullet \\
\qquad throttle := \mathbf{new}\ ThrottleControllerClass(speed, \ldots); \\
\qquad ThrottleControllerInit\,!\,throttle \longrightarrow Skip; \\
\qquad add\_handler\,.\,ThrottleControllerHandlerId \longrightarrow Skip \\
\qquad cruise := \mathbf{new}\ CruiseControlClass(throttle, speedo); \\
\qquad engine := \mathbf{new}\ EngineClass(cruise, \ldots); \\
\qquad EngineInit\,!\,engine \longrightarrow Skip; \\
\qquad add\_handler\,.\,EngineHandlerId \longrightarrow Skip\ ;\ \ldots \\
\quad initializeRet\,.\,MainMissionId \longrightarrow Skip
\end{array}
$$

This formalises the declaration of local variables *speed*, *throttle*, and so on for handler objects. These variables have a class type, and are initialised using its constructor. For instance, $throttle := \mathbf{new}\ ThrottleControllerClass(speed, \ldots)$ is a reference assignment to *throttle* of an object of class *ThrottleControllerClass* defined by its constructor, given *speed* and other parameters.

An important observation is that a handler is characterised not merely by (framework and application) processes, but also by a data object. In Fig. 3 this is indicated by boxes in the processes for handlers. Accordingly, we need to establish a connection between the data object and the process that aggregates it. This is achieved via a designated channel with suffix *Init*. The application process uses this channel to retrieve the data object it is connected to.

A pair of Java statements that create and register a handler with the current mission is, therefore, translated to one assignment and two communications. As already explained, the assignment constructs the handler's data object and assigns it to the appropriate local variable. Next, we have a communication like $ThrottleControllerInit\,!\,throttle$, which outputs a reference to the data object to the handler process. Finally, to record the handler as part of the mission, we have a communication like $add\_handler\,.\,ThrottleControllerHandlerId$. In the program this corresponds to a call to `register()` on the handler object.

We note, however, that not all data objects need to be wrapped in a process. For example, the *CruiseControlClass* object does not need to be associated with a process since the framework does not need to directly interact with it. It is used to aggregate other objects and has a direct translation as an *OhCircus* class.

Another method of a mission application class that needs special encoding is `requestTermination()`; it also needs to communicate with the framework process as it raises the $stop\_handlers$ event. All other action methods, like, for instance, the action for the `missionMemorySize()` method, and the main action are as already explained and exemplified for application processes.

3.4 Handler models

As already noted, the application process for a handler associates application events to it. On the other hand, the specification of the framework process is similar for periodic and aperiodic handlers. In Fig. 6, we sketch the generic framework process for an event handler. It is parametrised by an identifier that must be provided when the framework process is instantiated for a particular handler. For the engine handler, for example, we use $EventHandlerFW(EngineHandlerId)$.

The state component *active* of the *EventHandlerFW* records if the handler is active in the current mission or not. The main action defines an iterative behaviour that is interrupted and terminated by the event $end\_mission\_fw$, which, as mentioned before, indicates the end of the mission execution.

Each iteration defines the behaviour of the handler during a mission. First, the state is initialised using *Init*. Afterwards, the handler waits to be started using the *StartHandler* action in external choice (operator $\Box$) with a synchronisation on $activate\_handlers$, offered by *ActivateHandlers*. The action *StartHandler* synchronises on a particular $start\_handler$ event determined by the handler identifier. Afterwards, it also offers a synchronisation on $activate\_handlers$ (calling *ActivateHandlers*), which always occurs prior to entering the execution phase.

If the $start\_handler$ event occurs before $activate\_handlers$, the value of *active* is *TRUE*. In this case, the handler calls the action *DispatchHandler*. It raises the $enter\_dispatch$ event to notify the application process that it has to enter the handler's dispatch loop in which it starts responding to the external events associated with it. The dispatch loop is interrupted after the $stop\_handler\,.\,handler$

$$
\begin{array}{l}
\textbf{process}\ \mathit{EventHandlerFW} \mathrel{\widehat=} \mathit{handler} : \mathit{HandlerId} \bullet \textbf{begin}\\
\quad \textbf{state}\ \mathit{EventHandlerFWState} == [\mathit{active} : \mathit{BOOL}]\\
\quad \mathit{Init} == [\mathit{EventHandlerFWState}' \mid \mathit{active}' = \mathit{FALSE}]\\
\quad \mathit{StartHandler} \mathrel{\widehat=} \mathit{start\_handler}\,.\,\mathit{handler} \longrightarrow \mathit{active} := \mathit{TRUE}\\
\quad \mathit{ActivateHandlers} \mathrel{\widehat=} \mathit{activate\_handlers} \longrightarrow \mathit{Skip}\\
\quad \mathit{DispatchHandler} \mathrel{\widehat=} \mathit{enter\_dispatch} \longrightarrow\\
\qquad \mathit{stop\_handler}\,.\,\mathit{handler} \longrightarrow \mathit{leave\_dispatch} \longrightarrow \mathit{Skip}\\
\quad \bullet \left(\begin{array}{l}
\mu X \bullet \mathit{Init};\\
\left(\begin{array}{l}
((\mathit{StartHandler}\ ;\ \mathit{ActivateHandlers}) \mathrel{\Box} \mathit{ActivateHandlers});\\
\textbf{if}\ \mathit{active} = \mathit{TRUE} \longrightarrow \mathit{DispatchHandler}\\
[\!]\ \mathit{active} = \mathit{FALSE} \longrightarrow \mathit{Skip}\\
\textbf{fi}
\end{array}\right) ;\ X
\end{array}\right)\\
\qquad\qquad \triangle\ \mathit{end\_mission\_fw} \longrightarrow \mathit{Skip}\\
\textbf{end}
\end{array}
$$

Fig. 6. Framework process for event handlers

event, by synchronising on *leave_dispatch*. If *active* is *FALSE*, the handler process skips, as in this case the handler is not part of the current mission.

As already said, the application processes for handlers are factored into a data object modelled by an *OhCircus* class, and a process that aggregates it and releases the handler. Fig. 7 presents the *OhCircus* class for the `Engine` Java class. The correspondence is direct, with member variables defined as state components, and the constructor defined in the **initial** paragraph. For methods, the only difference is that events are not treated as objects: we use event identifiers. So, *handleAsyncEvent* takes an event identifier as a value parameter, and compares it to the identifiers of the events that are handled in the class.

The application process for a handler lifts its data objects to a process that can interact with the other components of the model. We present in Fig. 8 the process for the engine handler. The object for the handler is recorded in its state component *obj*. The *Init* action initialises it with the object input through the constructor call channel: here, the channel *EngineInit* of type *EngineClass*.

The *handleAsyncEventMeth* action simply executes the corresponding data operation. We cannot adopt exactly this model when `handleAsyncEvent()` handles an output event. For instance, the throttle controller handler process has to carry out communications $set\_voltage\,!\,v$. In such cases, we cannot represent the method by just a call to a data operation like in Fig. 8, but have to encode it by an action. The *handleAsyncEventMeth* of the application process, in this case, reflects directly the Java code, but outputs a value in the correct external channel where in Java we have a device access to achieve the hardware interaction.

Since a handler the used by several missions, the application process repeatedly initialises (*Init*), executes (*Execute*), and terminates (*Terminate*) it. Execution waits for the *enter_dispatch* event, and then enters a loop that repeatedly waits for the occurrence of one of the external events associated with

```
$\mathbf{class}\ \mathit{EngineClass} \mathrel{\widehat=} \mathbf{begin}$
  $\mathbf{state}\ \mathit{EngineState} == [\mathbf{private}\ \mathit{cruise} : \mathit{CruiseControlClass}]$
  $\mathbf{initial}\ \mathit{EngineInit} \mathrel{\widehat=} \mathbf{val}\ \mathit{cruise?} : \mathit{CruiseControlClass} \bullet \mathit{cruise} := \mathit{cruise?}$
  $\mathbf{public}\ \mathit{handleAsyncEvent} \mathrel{\widehat=} \mathbf{val}\ \mathit{event} : \mathit{EventId} \bullet$
    $\mathbf{if}\ \mathit{event} = \mathit{EOnEvtId} \longrightarrow \mathit{cruise.engineOn}()$
    $[\!]\ \mathit{event} = \mathit{EOffEvtId} \longrightarrow \mathit{cruise.engineOff}()$
    $\mathbf{fi}$
$\mathbf{end}$
```

Fig. 7. *OhCircus* class for the `Engine` handler

```
$\mathbf{process}\ \mathit{EngineApp} \mathrel{\widehat=} \mathbf{begin}$
  $\mathbf{state}\ \mathit{EngineState} == [\mathit{obj} : \mathit{EngineClass}]$
  $\mathit{Init} \mathrel{\widehat=} \mathit{EngineInit}\,?\,o \longrightarrow \mathit{obj} := o$
  $\mathit{handleAsyncEventMeth} \mathrel{\widehat=} \mathbf{val}\ e : \mathit{EventId} \bullet \mathit{obj.handleAsyncEvent}(e)$
  $\mathit{Execute} \mathrel{\widehat=} \mathit{enter\_dispatch} \longrightarrow \mathit{Dispatch}$
  $\mathit{Dispatch} \mathrel{\widehat=}$
```

$$\left(\begin{array}{l}\mu X \bullet \\ \quad \left(\begin{array}{l} \mathit{leave\_dispatch} \longrightarrow \mathit{Skip} \\ \Box \\ \left(\left(\begin{array}{l} \mathit{engine\_on} \longrightarrow \mathit{handleAsyncEventMeth}(\mathit{EOnEvtId}) \\ \Box \\ \mathit{engine\_off} \longrightarrow \mathit{handleAsyncEventMeth}(\mathit{EOffEvtId}) \end{array}\right) ; \\ \bigsqcap t : 0..\mathit{EngineDeadline} \bullet \mathbf{wait}\ t \end{array}\right)\right) \end{array}\right) ; X \end{array}\right)$$

```
  $\mathit{Terminate} \mathrel{\widehat=} \mathit{done\_handler} \,.\, \mathit{EngineHandlerId} \longrightarrow \mathit{Skip}$
  $\bullet\ (\mu X \bullet \mathit{Init} \,;\, \mathit{Execute} \,;\, \mathit{Terminate} \,;\, X) \mathbin{\triangle} \mathit{end\_mission\_fw} \longrightarrow \mathit{Skip}$
$\mathbf{end}$
```

Fig. 8. Application process for the `Engine` handler

the handler. In our example, these are *engine_on* and *engine_off*. When such an event occurs, *Dispatch* calls the *handleAsyncEventMeth* action. The subsequent nondeterministic **wait** captures the permissible amount of time the program may take to execute it. The dispatch loop is abandoned when the *leave_dispatch* event occurs. Termination that follows raises a particular $\mathit{done\_handler} \,.\, h$ event to notify the mission framework process that the handler has terminated.

In the case of an application process for a periodic handler, the only difference is in *Dispatch*. It does not wait for external events and calls `handleAsyncEvent()` when an internal timer event *release* occurs. An additional parallel action *Release* generates the timer events. It is given below for `ThrottleController`.

```
$\mathit{Release} \mathrel{\widehat=}$
  $(\mu X \bullet (\mathit{release} \longrightarrow \mathit{Skip} \blacktriangleright 0) \,;\, \mathbf{wait}\ \mathit{ThrottleControllerPeriod} \,;\, X)$
    $\mathbin{\triangle} \mathit{leave\_dispatch} \longrightarrow \mathit{Skip}$
```

The *Circus Time* **wait** t action waits for the end of the period before terminating, and the $\blacktriangleright$ operator specifies that the *release* event happens immediately afterwards. *ThrottleControllerPeriod* is a constant that specifies the period of the handler. (We have one such constant for each periodic handler.)

4 Conclusions

As far as we know, what we presented here is the first formalisation of the SCJ paradigm. Our models capture the essence of its design, and are an essential asset for analysis and development techniques for SCJ programs based on refinement.

To validate the models, we have translated them for FDR. The CSP translation encapsulates all *Circus* state into process parameters. Timing aspects are ignored, and so is the detailed application-level behaviour of handlers. We ensured that simple interaction scenarios do not result in a deadlock, and also that the mechanism for starting and terminating missions works as expected.

The direct correspondence between SCJ programs and our models enables automation in both directions. The framework processes are the same for all programs. The application processes use a fixed modelling pattern. The formalisation of the model-generation strategy discussed here, and the development of an associated tool, is work in progress. What remains to be done is to formalise the translation rules, and we believe this can be done in a compositional manner to facilitate their implementation using visitors. The tool will allow us to tackle larger industrial examples like those in Kalibera et al.'s benchmark [14].

The SCJ also incorporates a region-based memory model with restrictions on access to support safe dynamic memory management, and associated static verification techniques. We have abstracted from this here, but refined versions of our model will incorporate the language features we have formalised elsewhere [7]. For this we will further introduce constructs into the language that make explicit the memory areas in which objects are allocated. Importantly, this does not impact on any of the models presented earlier: they remain valid.

There are many approaches and tools to reason about object-oriented programs and Java [3,1], but they do not cater for the specificities of concurrency in SCJ. Brooke et al. present a CSP specification for a concurrency model for Eiffel (SCOOP) [2]. Their CSP specification shares some basic ideas with our *Circus* models, but is necessarily more complex due to its generality.

Kalibera et al.'s work in [14] is concerned with scheduling analysis and race conditions in SCJ programs, but it does not use proof-based techniques. Instead, exhaustive testing and model-checking is applied. Annotation-based techniques for SCJ can be found in [20,9]. In [20] annotations are used to check for compliance with a particular level of SCJ, and for safe use of memory. Haddad et al. define SafeJML [9], which extends JML [3] to cover functionality and timing properties; it reuses existing technology for worst-case execution-time analysis in the context of SCJ. Our model is a conceivable candidate to justify the soundness of checks supported by the annotations and carried out by the tools.

Our long term goal is the definition of refinement-based techniques for developing SCJ programs. Like in the *Circus* standard technique, we will devise a refinement strategy to transform centralised abstract *Circus Time* models into an SCJ model as described here. The development of this strategy, and the proof of the refinement rules that it will require are a challenging aspect of this endeavour. This involves the identification of refinement and modelling patterns. All this shall also provide further practical validation of our model.

Acknowledgements This work is funded by the EPSRC grant EP/H017461/1. We have discussed our models with Chris Marriott, Kun Wei, and Jim Woodcock.

References

1. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
2. P. Brooke, R. Paige, and J. Jacob. A CSP model of Eiffel's SCOOP. *Formal Aspects of Computing*, 19(4):487–512, 2007.
3. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212 – 232, 2005.
4. A. Burns. The Ravenscar Profile. *ACM SIGAda Ada Letters*, XIX:49–52, 1999.
5. A. Cavalcanti, A. Sampaio, and J. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2-3):146–181, 2003.
6. A. Cavalcanti, A. Sampaio, and J. Woodcock. Unifying classes and processes. *Software Systems and Modeling*, 4(3):277–296, 2005.
7. A. Cavalcanti, A. Wellings, and J. Woodcock. The Safety-critical Java Memory Model: a formal account. In *Formal Methods*, LNCS, 2011.
8. The Open Group. Safety Critical Java Technology Specification. Technical Report JSR-302, Java Community Process, January 2011.
9. G. Haddad, F. Hussain, and G. T. Leavens. The Design of SafeJML, A Specification Language for SCJ with Support for WCET Specification. In *JTRES*. ACM, 2010.
10. W. Harwood, A. Cavalcanti, and J. Woodcock. A Theory of Pointers for the UTP. In *ICTAC*, volume 5160 of *LNCS*, pages 141–155. Springer, 2008.
11. T. Henties, J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for Safety-Critical Applications. In *SafeCert*, 2009.
12. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
13. C.A.R. Hoare and H.Jifeng. *Unifying Theories of Programming*. Prentice Hall,1998.
14. T. Kalibera, P. Parizek, and M. Malohlava. Exhaustive Testing of Safety Critical Java. In *JTRES*. ACM, 2010.
15. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
16. M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1-2):3–32, 2009.
17. T. Santos, A. Cavalcanti, and A. Sampaio. Object-Orientation in the UTP. In *UTP*, volume 4010 of *LNCS*, pages 18–37. Springer, 2006.
18. A. Sherif, A. Cavalcanti, H. Jifeng, and A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, 2009.
19. J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.
20. D. Tang, A. Plsek, and J. Vitek. Static Checking of Safety Critical Java Annotations. In *JTRES*, pages 148–154. ACM, 2010.
21. A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004.
22. A. Wellings and M. Kim. Asynchronous event handling and safety critical Java. In *JTRES*. ACM, 2010.