SCJ-Circus: a refinement-oriented formal notation for Safety-Critical Java

Alvaro Miyazawa Ana Cavalcanti
Department of Computer Science, University of York, York, YO10 5GH, UK
alvaro.miyazawa@york.ac.uk ana.cavalcanti@york.ac.uk

Safety-Critical Java (SCJ) is a version of Java whose goal is to support the development of real-time, embedded, safety-critical software. In particular, SCJ supports certification of such software by introducing abstractions that enforce a simpler architecture, and simpler concurrency and memory models. In this paper, we present *SCJ-Circus*, a refinement-oriented formal notation that supports the specification and verification of low-level programming models that include the new abstractions introduced by SCJ. *SCJ-Circus* is part of the family of state-rich process algebra *Circus*, as such, *SCJ-Circus* includes the *Circus* constructs for modelling sequential and concurrent behaviour, real-time and object orientation. We present here the syntax and semantics of *SCJ-Circus*, which is defined by mapping *SCJ-Circus* constructs to those of standard *Circus*. This is based on an existing approach for modelling SCJ programs. We also extend an existing *Circus*-based refinement strategy that targets SCJ programs to account for the generation of *SCJ-Circus* models close to implementations in SCJ.

1 Introduction

Safety-Critical Java (SCJ) [9] is a subset of the Real-Time Specification for Java (RTSJ) [19]. This is a version of Java that targets the development of real-time software. It avoids the issue of unpredictable timing associated with garbage collection by introducing memory areas.

SCJ restricts the RTSJ to facilitate certification; it imposes a particular structure for programs embedding simplified memory and concurrency models. The structure of an SCJ application is composed of a safelet (the main program), a mission sequencer that provides missions in a particular order, and a number of missions that are composed by concurrent handlers. SCJ supports different types of handlers, such as periodic and aperiodic handlers.

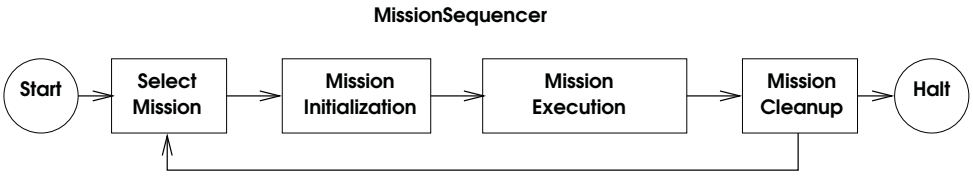

Figure 1: SCJ programming model

Figure 1 depicts the programming model of SCJ, which essentially consists of a cycle where at each step a new mission is selected, initialised, executed and terminated. During execution, a number of handlers run in parallel. When there are no missions left, the program terminates. The memory model is based on scoped memory regions, rather than garbage collection. The safelet, the missions, and the handlers have associated memory regions, which are cleared at predictable points of the program control flow.

In [5], the SCJ standard is complemented with a design technique based on the *Circus* family of languages for refinement. *Circus* [4] is a state-rich process algebra for refinement that has been applied

Eerke Boiten, John Derrick & Steve Reeves (Eds.):
Refinement Workshop 2015 (Refine '15)
EPTCS 209, 2016, pp. 71–86, doi:10.4204/EPTCS.209.6

to the verification of a variety of models including Simulink and Stateflow diagrams [3, 13]. The semantics of *Circus* is based on Hoare and He's Unifying Theories of Programming [7], which is a semantic framework that supports the formalisation of paradigms in an independent fashion and their combination through specific techniques. As refinement is a central concept in UTP, it is also an important aspect of *Circus* as evidenced by its rich refinement calculus [15]. *Circus* has been extended to support a number of different programming paradigms: for example, *OhCircus* [2] supports the specification of object-oriented programs, and *Circus Time* [18] supports modelling real-time programs.

We introduce here a new member of the set of *Circus* languages: *SCJ-Circus* combines *OhCircus* and *Circus Time*, and extends them with the abstractions introduced by SCJ. It supports either verification or full development of SCJ programs from an abstract timed-model to an object-oriented timed model that explicitly uses the SCJ abstractions. *SCJ-Circus* models define a safelet, a mission sequencer, missions and handlers. Additionally, *SCJ-Circus* introduces object creation statements (**new** in *OhCircus*) tailored to the hierarchical memory model adopted in Safety-Critical Java. Abstraction can still be achieved using the constructs of *Circus* for data and behavioural modelling. Yet, the architecture of the models is in direct correspondence with that of SCJ programs, although platform specific aspects of an application, such as memory and thread availability, are not covered.

The refinement strategy proposed in [5] is based on the notion of anchors, which are models written in different subsets of *Circus* following specific architectural patterns. There are four anchors related by refinement: A, O, E and S. The first anchor (A anchor) defines an abstract model and the last anchor (S anchor) describes a refinement of the A anchor that follows the programming paradigm of SCJ. The O anchor introduces the object-oriented model, and the E anchor introduces the notions of missions, handlers and memory areas. Whilst [5] details the refinement strategy between the three first anchors (A, O and E), it only briefly indicates how to proceed from the E to the S anchor.

In this work, we extend [5] exploring the use of *SCJ-Circus* to define the S anchor. We specify the syntax and semantics of *SCJ-Circus*, and describe the last phase of the refinement strategy to use *SCJ-Circus* as target models. To define the semantics of *SCJ-Circus*, we build on a *Circus* semantics of SCJ programs defined in [21]. To that end, we update that semantics to reflect fundamental changes to the mode of interaction between handlers and mission termination. We also propose a different structure for the *Circus* models to enable compositional refinement with respect to the *SCJ-Circus* components.

In Section 2, we introduce the *Circus* family of languages and Safety-Critical Java. In Section 3, we discuss *SCJ-Circus*, its syntax and semantics, and Section 4 discusses the extension of the refinement strategy proposed in [5] to reach *SCJ-Circus* programs. Finally, Section 5 concludes by relating our work to the existing literature and discussing future work.

2 Preliminaries

In this section, we briefly describe the base notations relevant to our work. Section 2.1 introduces the *Circus* family of languages, and Section 2.2 describes SCJ.

2.1 *Circus*

In this section, we use the *Circus Time* process *PEHFW* (periodic event handler framework) in Figure 2 that models the general behaviour of a periodic event handler to describe *Circus* and its timed variant. The main modelling element of a *Circus* specification is a process (indicated by the keyword **process**) that declares state components (identified by the keyword **state**), a number of auxiliary actions, and

$$
\begin{array}{l}
\textbf{process}\, PEHFW \mathrel{\widehat=} id : ID \bullet \textbf{begin} \\
\quad \textbf{state}\, PEHFWState == [start, period : \mathbb{N}] \\
\quad Execute \mathrel{\widehat=} \textbf{wait}\, start; \\
\quad \left(\begin{array}{l}
\mu X \bullet \left(\begin{array}{l}
(handleAsyncEventCall!id!id \longrightarrow handleAsyncEventRet!id!id \longrightarrow Skip) \blacktriangleright period;\ X \\
\Box \\
done\_handler!id \longrightarrow \textbf{Skip}
\end{array} \right) \\
[\![\{\} \mid \{\!| handleAsyncEventCall.id, done\_handler.id |\!\} \mid \{\}]\!] \\
(\mu Y \bullet ((handleAsyncEventCall!id!id \longrightarrow \textbf{wait}\, period) \blacktriangleleft 0);\ Y) \triangle done\_handler!id \longrightarrow \textbf{Skip} \\
\quad \bullet\ \mu X \bullet (start\_peh?o!id?s?p \longrightarrow (start := s;\ period := p);\ Execute;\ X)
\end{array} \right) \\
\textbf{end}
\end{array}
$$

Figure 2: Framework process of the periodic event handler.

a main action (prefixed by $\bullet$) that describes the overall behaviour of the process. In the case of our example, the process *PEHFW* is parametrised by an identifier *id* of a given type *ID*, and declares two state components, *start* and *period* both of type $\mathbb{N}$.

PEHFW only declares one auxiliary action *Execute*. Actions are specified using a combination of Z [20] for data modelling and CSP [16] for behavioural descriptions. The main action is defined by a recursion ($\mu X \bullet \ldots$) that at each step starts an instance of the event handler via a communication through channel *start_peh*. In this communication, the identifier *o* of the mission that requested the instantiation is input, the handler identifier *id* is output, and its start time *s* and period *p* are input. Whilst the input *o* is not needed for the execution, it is necessary to allow missions to reuse periodic event handlers in the same application. The values of *s* and *p* are then assigned to the state components *start* and *period*. The execution of a newly created handler is defined by the action *Execute*. It first waits for *start* time units (**wait** *start*), and then starts two recursive actions in parallel ($A_1 \,[\![ns_1 \mid cs \mid ns_2]\!]\, A_2$) synchronising on channels *handleAsyncEventCall* and *done_handler*.

The first action specifies that at each step of the recursion there is an external choice ($\Box$) for communication on channels *done_handler* or *handleAsyncEventCall* in this way, the handler can be terminated with a choice of *done_handler* or the method `handleAsyncEvent` is called through the channel *handleAsyncEventCall*. If `handleAsyncEvent` is called, the it must return, as indicated with a communication via *handleAsyncEventRet* within *period* time units. This is specified by the *Circus Time* action $A \blacktriangleright e$ that defines that the action A must terminate within e time units.

The recursive parallel action in *Execute* adds a requirement that a call to `handleAsyncEvent` must be started as soon as it is available, and should only be made available again after *period* time units. This is achieved by imposing a restriction on the communication *handleAsyncEventCall* using the start by operator ($\blacktriangleleft$) that specifies that an action must start within a certain number of time units. The requirement that *handleAsyncEventCall* is only offered after *period* time units is enforced by the action **wait** *period* after the communication on *handleAsyncEventCall*. Since the first recursion can be terminated by a synchronisation on the channel *done_handler*, the second recursion must also be terminated. This is achieved by allowing its interruption of the recursion by a synchronisation on *done_handler* using the interrupt operator ($\triangle$).

In general, a *Circus* or *Circus Time* specification consists of a sequence of paragraphs that define processes (as well as channels, constants, and other constructs that support the definition of processes). Processes are used to define the system and its components: state is encapsulated and interaction is via

```
public class Checker extends AperiodicEventHandler {
  Buffer buffer;
  public Checker(Buffer b) {
    super(new PriorityParameters(Priorities.PR98),
          new AperiodicParameters(),
          storageParameters_Handlers);
    buffer = b;
  }
  public void handleAsyncEvent() {
    if (buffer.theSame()) devices.Console.println("true");
    else devices.Console.println("false");
  }
}
```

Figure 3: SCJ Level 1 example: Aperiodic Event Handler

channels. Processes can be composed, via CSP operators, to define other processes. In *Circus Time*, wait and deadline operators can be used to define time restrictions. In *OhCircus* models, we can in addition define paragraphs that declare classes used to define types. More information about these languages can be found in [15, 18, 2]. In the sequel, we further explain the notation as needed.

2.2 Safety Critical Java

As previously mentioned, an SCJ application is formed by a safelet, mission sequencer, a number of missions, and periodic and aperiodic event handlers. Each of these is characterised by an interface or abstract class of an API that supports the development of SCJ programs via implementation and extension of these components. A safelet instantiates a mission sequencer, and iteratively obtains a mission from the mission sequencer, executes it and waits for it to terminate. The execution of a mission consists of the parallel execution of all its periodic and aperiodic event handlers. Most of the actual behaviour of the application is concentrated in the handlers, which are the focus of this section.

Our running example is a simple SCJ application: a communication medium that checks whether the three copies of a message received are the same (and, therefore, reliable). It has a single mission containing two handlers: one periodic event handler and one aperiodic event handler. The periodic event handler reads an input every at every cycle, stores it in a buffer, and releases the aperiodic event handler. Upon release, the aperiodic event handler examines the last three elements and outputs “true” or “false” depending on whether the last three values of the buffer are all the same or not.

Figure 3 shows the code for the aperiodic handler in our example. It extends the SCJ API class `AperiodicEventHandler`, and declares a local variable `buffer`, a constructor that receives an instance of the class `Buffer` and assigns it to `buffer`, and a `handleAsyncEvent` method that defines the main behaviour of the handler. The constructor of `Checker` calls the constructor of the superclass with priority 98, a new aperiodic parameter, and storage parameters that specify the amount of memory used by the handler. The method `handleAsyncEvent` checks whether the last three elements of `buffer` are the same using the method `theSame`; if they are, it prints “true”, otherwise it prints “false”. For simplicity, we print the output of the checker, which in practice needs to be sent to another component of the system.

The complete program contains classes to implement the safelet, the mission sequencer, the mission and the periodic handler. It can be found in `http://www.cs.york.ac.uk/~alvarohm/er2015.zip`.

$$
\begin{array}{l}
\textbf{safelet } \mathit{Safelet} \mathrel{\widehat=} \ldots \\
\textbf{sequencer } \mathit{Sequencer} \mathrel{\widehat=} \ldots \\
\textbf{mission } \mathit{Mission} \mathrel{\widehat=} \ldots \\
\textbf{periodic handler } \mathit{PeriodicHandler} \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{start } 0 \textbf{ period } P \\
\quad \textbf{state } [ah : ID] \\
\quad \textbf{initial} \mathrel{\widehat=} ah : ID \bullet this.ah := ah \\
\quad \textbf{handleAsyncEvent} \mathrel{\widehat=} \\
\qquad ((input?x \longrightarrow \textbf{Skip}) \blacktriangleleft ID;\ setBuffer!(buffer \frown \langle x \rangle) \longrightarrow release();\ (\textbf{wait } 0 \,..\, PTB)) \blacktriangleright PD \\
\textbf{end} \\
\textbf{aperiodic handler } \mathit{AperiodicHandler} \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{handleAsyncEvent} \mathrel{\widehat=} \\
\qquad \left(\begin{array}{l} getBuffer?buffer \longrightarrow \\ \left(\begin{array}{l} \textbf{if } buffer \in theSame \longrightarrow (output!true \longrightarrow \textbf{Skip}) \blacktriangleleft OD \\ [\!] buffer \notin theSame \longrightarrow (output!false \longrightarrow \textbf{Skip}) \blacktriangleleft OD \\ \textbf{fi} \end{array} \right) ;\ \textbf{wait } 0..ATB \end{array} \right) \blacktriangleright AD \\
\textbf{end}
\end{array}
$$

Figure 4: SCJ Level 1 example: S-anchor

3 *SCJ-Circus*

As previously mentioned, *SCJ-Circus* extends *OhCircus* and *Circus Time* with abstractions that are specific to Safety-Critical Java. Below, Section 3.1 briefly discusses the syntax of *SCJ-Circus*, Section 3.2 presents the semantic models of the SCJ framework, that is, its API and programming model, and Section 3.3 describes the semantics of the language based on the *Circus* models of Section 3.2.

3.1 Syntax

SCJ-Circus extends the syntax of *OhCircus* and *Circus Time* with paragraphs that allow the specification of safelets, mission sequencers, missions and handlers. Figure 4 presents the specification of our running example in *SCJ-Circus*. It matches the structure of our example, but further specifies timing requirements. The periodic event handler reads an input every P time units, with an input deadline of ID time units. Each cycle of the periodic event handler takes any time between 0 and PTB time units, and must terminate within PD time units. The aperiodic event handler outputs values within OD time units, and each release takes at most ATB time units, and must terminate within AD time units.

The constants PTB, ATB, ID, OD, PD, AD and P need to satisfy a number of conditions to ensure that the two handlers run in lockstep. For the periodic event handler, these conditions require that the sum of periodic time budget (PTB) and the input deadline (ID) does not exceed the periodic deadline (PD). Additionally the sum of the periodic deadline (PD) and the aperiodic deadline AD must not exceed the period P of the periodic event handler.

In general, as shown in Figure 3.1, an *SCJ-Circus* program is a sequence of SCJParagraphs, which can be a *Circus* paragraph, or the declaration of a safelet, mission sequencer, mission or handler. The structure of each of the SCJ-specific abstractions is determined by the values and behaviours that must be specified for an application according to the SCJ standard [9]. For instance, a safelet must implement

SCJProgram ::= SCJParagraph*
SCJParagraph ::= Safelet | MissionSequencer | Mission | Handler | CircusParagraph

Safelet ::= **safelet** N $\widehat{=}$ **begin**
SCJSSafeletProcessParagraph*
state Schema-Expression
SCJSafeletProcessParagraph*
initialize $\widehat{=}$ SCJSafeletAction
SCJSafeletProcessParagraph*
getSequencer $\widehat{=}$ **res return** : **sequencer** • SCJSafeletAction
SCJSafeletProcessParagraph*
end

Figure 5: Syntax of *SCJ-Circus* (sketch)

the `initialize` method that allows the allocation of global objects, and the `getSequencer` method that provides a mission sequencer.

Accordingly, the *SCJ-Circus* construct corresponding to a safelet in Figure 3.1 has a name taken from the set of valid *Circus* names N, and allows the specification of state components (**state**), the initialisation (**initialize**) and **getSequencer** methods, as well as auxiliary actions (SCJSafeletProcessParagraph). The state components model the fields of the safelet class. An SCJSafeletProcessParagraph allows the specification of an action whose body is a SCJSafeletAction, which restricts the constructs that can be used in an action of a safelet, in particular, the type of allocation constructs as discussed next.

SCJ enforces a hierarchical memory-model in which different components (safelets, missions and so on) may only instantiate new objects in their memory areas or parent memory areas. We reflect this discipline in *SCJ-Circus* by restricting syntactically which paragraphs may include allocations, through different **new** keywords, to particular memory areas. For instance, a safelet may only instantiate objects in the immortal memory, and therefore may only use the keyword **newI** for instantiation of new objects. A handler, on the other hand, may allocate objects in the immortal memory area, mission memory area (**newM**), per-release memory area (**newPR**) and private memory area (**newPM**).

These restrictions are reflected in the use of different syntactic categories for the actions and paragraphs of the different constructs. For example, the `getSequencer` method of a safelet must be an SCJSafeletAction and the `handleAsyncEvent` method of a handler must be as SCJHandlerAction. The first only allows instantiation via **newI**, whilst the other allows all possible instantiation keywords.

The syntax of the *SCJ-Circus* paragraphs for the mission sequencer, missions and handlers are similar, providing means for the specification of state components (**state**), constructors (**initial**), and the methods of the corresponding element that must be provided by the developer. For further details about the syntax of *SCJ-Circus* refer to [12].

3.2 Semantic model

In [21], an approach to modelling SCJ programs has been proposed; it is a translation strategy defined as a semantic function that maps SCJ programs to *Circus* specifications. We adopt a similar approach here to give semantics to *SCJ-Circus*. Our *Circus* models, however, are updated to consider recent significant changes to SCJ and to cater for compositional reasoning about SCJ constructs.

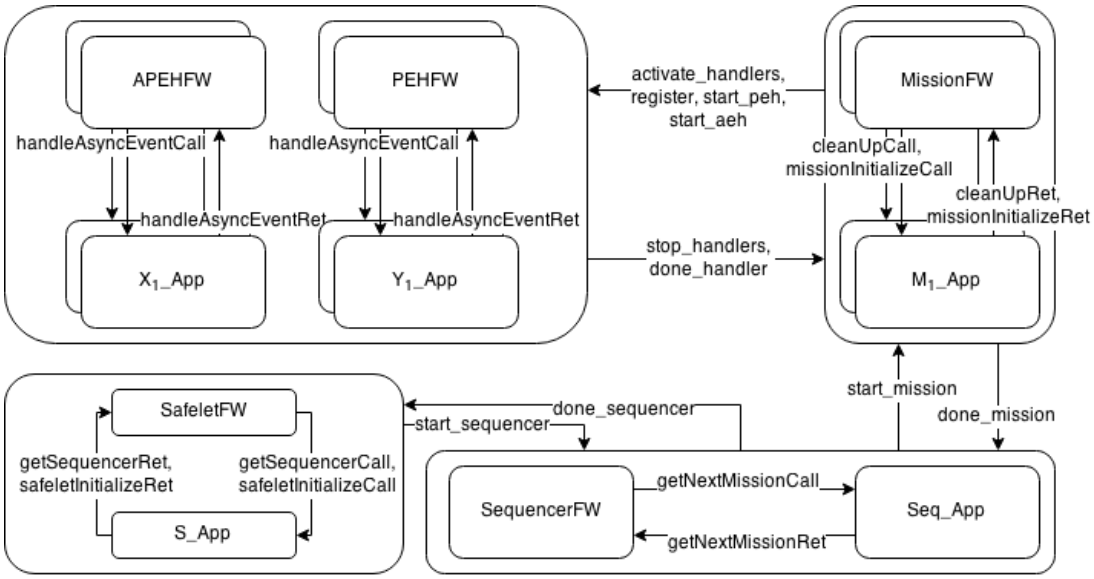

Figure 6: Structure of semantic models

Each *Circus Time* process is defined as the parallel composition of two processes: a framework process that captures the behaviour of the corresponding SCJ component as an element of the SCJ programming model, and a process that captures the behaviour of that component as defined in a particular application. For example, the process *PEHFW* in Figure 2 presents the framework process for a periodic handler. It defines the general flow of execution of such a handler without giving the details of a particular handler implementation.

The framework and application processes of each SCJ element interact through a number of channels that correspond to method calls in the implementation. For example, the channels *safeletInitializeCall*, *safeletInitializeRet*, *getSequencerCall* and *getSequencerRet* in Figure 6 are used by the safelet framework process *SafeletFW* to communicate with the application specific process *S_App* and correspond to calls to the methods `initialize` and `getSequencer` of the application.

In the models of SCJ programs presented in [21], the application processes are combined together in interleaving, framework processes are grouped together in parallel, and both groups are then combined in parallel to yield the semantic model of the whole application. This structure proved not ideal for the compositional analysis of *SCJ-Circus* programs because the aspects relevant to a specific *SCJ-Circus* construct, such as a handler, are spread through the complete model and cannot be isolated for reasoning purposes. Figure 6 depicts the structure of the updated semantic model. *Circus* specifications model each of the *SCJ-Circus* paragraphs as standard *Circus Time* processes.

It is worth mentioning that the model where application and framework processes are composed on a per-element basis is a refinement of the model structured as in [21]. This fact is established in our refinement strategy described in Section 4 because the framework is first introduced as a monolithic process, and then distributed through the application processes.

The framework process that specifies the generic behaviour of a safelet is shown in Figure 7. It is a process parametrised by the safelet identifier, and its behaviour consists of requesting to the application process the initialisation of the safelet using the channels *safeletInitializeCall* and *safeletInitializeRet*, obtaining a mission sequencer via the channels *getSequencerCall* and *getSequencerRet*, and if the se-

$$
\begin{array}{l}
\textbf{process}\ SafeletFW \mathrel{\widehat=} id : ID \bullet \textbf{begin} \\
\quad Execute \mathrel{\widehat=} getSequencerCall!id!id \longrightarrow getSequencerRet!id!id?s \longrightarrow \\
\qquad \left(\begin{array}{l} \textbf{if}\ s \neq null \longrightarrow start\_sequencer \longrightarrow done\_sequencer \longrightarrow \textbf{Skip} \\ [\!]\ s = null \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array} \right) \\
\quad \bullet\ safeletInitializeCall!id!id \longrightarrow safeletInitializeRet!id!id \longrightarrow Execute; \\
\qquad end\_safelet\_app \longrightarrow \textbf{Skip} \\
\textbf{end}
\end{array}
$$

Figure 7: Framework process for Safelet.

quencer is different than *null*, starting it (using the channel *start_sequencer*). At this point the safelet framework process waits for the mission sequencer to complete its execution and signal on the channel *done_sequencer*, in which case the safelet indicates to the application process that it is terminating through the channel *end_safelet_app*, and terminates (**Skip**).

The complete definition of the model can be found in [12].

3.3 Semantics

The semantics of *SCJ-Circus* is formalised as a function from well-formed models written in accordance with the abstract syntax of *SCJ-Circus* to *Circus* models, that is, elements of the category *CircusProgram*, as defined in [15]. In order to improve readability, the semantics is presented in terms of translation rules that output *Circus* concrete syntax. In essence, the semantic function composes the behaviours specified in *SCJ-Circus* with the model of the SCJ framework discussed in Section 3.2 in a compositional way.

Formally, the semantics of an *SCJ-Circus* program p is given by the *Circus* program formed by the *Circus* paragraphs that are obtained by applying specific semantic functions to the paragraphs of p. This is specified below by the function $[\![\_]\!]_{SCJProgram}$ that takes a well-formed *SCJ-Circus* program and outputs a *Circus* program composed of the paragraphs produced by the semantic functions $[\![\_]\!]_{SCJParagraphs}$ and $[\![\_]\!]_{Application}$. The first takes a sequence of *SCJ-Circus* paragraphs and outputs a sequence of *Circus* paragraphs, and the second takes a program and outputs the definition of a process that composes the processes defined in the previous paragraphs to specify the overall meaning of the application.

$$
\begin{array}{|l}
[\![\_]\!]_{SCJProgram} : SCJProgram \pfun Program \\
\hline
\forall p : WF\_SCJProgram \bullet [\![p]\!]_{SCJProgram} = [\![p.paragraphs]\!]_{SCJParagraphs} \frown [\![p]\!]_{Application}
\end{array}
$$

We use the mathematical notation of Z [20] to specify our semantic functions, and explain any non-standard use of notation as needed. In what follows, we focus on the semantic function for the safelet, which is used by $[\![\_]\!]_{SCJParagraphs}$ to give semantics to a safelet paragraph.

As explained in the previous section, the semantics of a safelet is given by the parallel composition of a *Circus* process that characterises the application-specific behaviours and a *Circus* process that models the generic behaviour of the framework. It is given by the function $[\![\_]\!]_{Safelet}$ below, which takes a safelet s and outputs a sequence of two processes: the application process *s_app* and the process that models the complete behaviour of s as the parallel composition of *s_app* and the framework process *SafeletFW* instantiated by the identifier of s. In the definition of the semantic function, guillemots («») are used to distinguish the *Circus* syntax from the meta-language used to specify the rules. For instance,

$$
\begin{array}{l}
safelet\_app : Safelet \pfun BasicProcess \\
\hline
\forall s : WF\_Safelet \bullet \\
safelet\_app(s) = \\
\left(\begin{array}{l}
\textbf{begin} \\
\quad \textbf{state}\ «s.state» \\
\quad «for\ each\ p : s.paragraphs\ of\ (N \mathrel{\widehat=} SCJSafeletParametrisedAction)\ do» \\
\quad\quad «N»Meth \mathrel{\widehat=} «translate\_method(name(s)ID, N, p.body)» \\
\quad «end» \\
\quad getSequencerMeth \mathrel{\widehat=} «translate\_method(name(s)ID, getSequencer, s.getSequencer)» \\
\quad initializeApplicationMeth \mathrel{\widehat=} initializeApplicationCall?x!«name(s)»ID \fun \\
\quad\quad\quad «s.initialize»; initializeApplicationRet!x!«name(s)»ID \fun \textbf{Skip} \\
\quad Methods \mathrel{\widehat=} \mu X \bullet \\
\quad\quad getSequencerMeth; X \extchoice initializeApplicationMeth; X \\
\quad «for\ each\ p : s.paragraphs\ of\ (N \mathrel{\widehat=} A)\ do» \extchoice «N»Meth; X«end» \\
\quad\quad \extchoice end\_safelet\_app \fun \textbf{Skip} \\
\quad \bullet\ Methods \\
\textbf{end}
\end{array}\right)
\end{array}
$$

Figure 8: Semantic function *safelet_app*.

«*safelet_app*(*s*)», indicates that the function *safelet_app* must be evaluated on the parameter *s* and the resulting syntax tree must be substituted in place of «*safelet_app*(*s*)».

$$
\begin{array}{l}
[\![\_]\!]_{Safelet} : Safelet \pfun \mathrm{seq}\ CircusParagraph \\
\hline
\quad \forall s : WF\_Safelet \bullet \\
[\![s]\!]_{Safelet} = \left(\begin{array}{l}
\textbf{process}\ «name(s)»\_App \mathrel{\widehat=} «safelet\_app(s)» \\
\textbf{process}\ «name(s)» \mathrel{\widehat=} \\
\quad (SafeletFW(«name(s)»ID) \lpar «SafeletCS(s)» \rpar «name(s)»\_App) \\
\quad\quad \hide «SafeletCS(s)»
\end{array}\right)
\end{array}
$$

As shown above, the definition of $[\![\_]\!]_{Safelet}$ relies on the function *safelet_app* that produces the application specific process, and a function *SafeletCS* that calculates the channels on which the application and the framework must communicate. These channels are internal to the safelet and therefore hidden ($\hide$).

The *safelet_app* function shown in Figure 8 takes a safelet *s* and constructs a process named after *s* using the function *name* concatenated with *_App*, and with the same state as *s*. Each auxiliary method of the safelet is translated into an *Circus* action using a pair of channels to model the call and return of the method. Similarly, the methods `getSequencer` and `initialize` are translated into the actions *getSequencerMeth* and *initializeApplicationMeth*. All these actions are used to construct the action *Methods* that recursively offers a choice between each of those actions, and the possibility to terminate the recursion via a synchronisation on the channel *end_safelet_app*.

The overall behaviour of the process is the action *Methods*. The parallel composition of the process obtained from the safelet and the framework process synchronises on the call and return channels used to encode method calling, as well as on the channel *end_safelet_app*, and these channels are made internal

using the hiding operator ($\setminus$).

The functions *sequencer_app*, *mission_app*, *PEH_app* and *AEH_app* that define the application processes for mission sequencers, mission, periodic event handler and aperiodic event handlers are defined similarly and are omitted. The complete semantics is defined in [12].

4 Refinement Strategy

The refinement strategy proposed in [5] covers the refinement of abstract *Circus Time* models into a process written following a pattern in which some of the structure of an SCJ application is identified but not explicitly described in terms of independent SCJ components, as can be done using *SCJ-Circus*. Here, we further elaborate the original strategy to obtain an S-Anchor like that shown in Figure 4.

Our refinement strategy starts from an E-Anchor in the form shown in Figure 9, which is a single *Circus* process in which each action models a component of the desired SCJ implementation, but the different elements (e.g., safelet, mission sequencer, and so on) are not yet isolated. The only parallelism is between the two handlers. The E-Anchor of our running example obtained through the application of the refinement strategy in [5] to the abstract model is shown in Figure 10. It is a single *Circus* process whose main action calls the safelet, which then calls the mission sequencer. The mission sequencer calls the single mission of our example, which calls in parallel the periodic and aperiodic handlers as well as an action that models the mission memory shared by both handlers.

In order to obtain the S-anchor, we propose a refinement strategy based on four phases: (1) introduction of SCJ control flow, (2) introduction of application process, (3) introduction of framework processes, (4) conversion to *SCJ-Circus*. The resulting S-anchor for our example is shown in Figure 4.

The first phase introduces the patterns of control observed in *SCJ-Circus* models, such as call-return channels, which model method calls, *start* and *done* channels that model the execution and termination of *SCJ-Circus* abstractions (e.g., Safelet), and release mechanisms. The second phase separates application-specific behaviours (e.g., reading of input) from framework behaviours (e.g., request of mission sequencer in the safelet). The third phase takes the incomplete model of framework behaviour isolated in the second phase and expands it by completing them with all possible behaviours of the *SCJ-Circus* framework processes. This is necessary because the E-anchor does not cover aspects of the framework that are not used by the application. For instance, our running example does not model termination and, therefore, the framework-specific behaviour isolated in the second phase does not cover the termination mechanisms of the SCJ framework. These are introduced in the third phase. The fourth phase introduces the paragraphs of the S-anchor, where the *SCJ-Circus* abstractions are explicitly declared.

4.1 E-anchor: starting point

We identify four main patterns of E-anchors with respect to the synchronisation between a number of periodic and aperiodic event handlers. The first has both types of handlers executing cyclically in lockstep and terminating within the period of the periodic event handler. In this pattern, all handlers are executed at every cycle and must terminate before the next cycle.

The second pattern is similar, except that not all aperiodic handlers are executed at each cycle. The handlers are executed cyclically, but not in lockstep. The common property to the first two patterns is that the execution of both periodic and aperiodic event handlers finishes with the period of the application. The third and fourth patterns are version of the first two where the deadline of the aperiodic event handlers cannot be guaranteed. That is, the execution of an aperiodic event handler may not terminate before the

$$
\begin{array}{l}
\textbf{process}\, P \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\, S \\
\quad Handler_i \mathrel{\widehat=} \ldots \\
\quad MArea_j \mathrel{\widehat=} \ldots \\
\quad Mission_j \mathrel{\widehat=} (MArea_j \parallel (\parallel k : handlers_j \bullet Handler_k)) \\
\quad MissionSequencer \mathrel{\widehat=} ;i : 1\,..\,n \bullet Mission_i \\
\quad Safelet \mathrel{\widehat=} MissionSequencer \\
\quad Application \mathrel{\widehat=} Safelet \\
\quad \bullet\, Application \\
\textbf{end}
\end{array}
$$

Figure 9: Refinement strategy: starting point of first phase (E-anchor)

$$
\begin{array}{l}
\textbf{process}\, System1 \mathrel{\widehat=} \textbf{begin} \\
\quad MArea \mathrel{\widehat=} \textbf{var}\, buffer : \mathrm{seq}\,\mathbb{N} \bullet \mu\, X \bullet \\
\qquad (setBuffer?x \longrightarrow buffer := x;\ X \,\Box\, getBuffer!buffer \longrightarrow X \,\Box\, stop \longrightarrow \textbf{Skip}) \\
\quad PeriodicHandler \mathrel{\widehat=} \\
\qquad \mu\, X \bullet \left(\left(\begin{array}{l} (input?x \longrightarrow \textbf{Skip}) \blacktriangleleft ID; \\ setBuffer!(buffer \frown \langle x \rangle) \longrightarrow release \longrightarrow (\textbf{wait}\, 0..PTB) \end{array} \right) \blacktriangleright PD \right) ;\ X \\
\qquad \parallel\!\!\parallel \textbf{wait}\, P \\
\quad AperiodicHandler \mathrel{\widehat=} \\
\qquad \mu\, X \bullet \left(\begin{array}{l} release \longrightarrow \\ \quad \left(\begin{array}{l} getBuffer?buffer \longrightarrow \textbf{wait}\, 0..ATB; \\ \quad \left(\begin{array}{l} \textbf{if}\, buffer \in three_0 \longrightarrow (output!true \longrightarrow \textbf{Skip}) \blacktriangleleft OD \\ [\!]\, buffer \notin three_0 \longrightarrow (output!false \longrightarrow \textbf{Skip}) \blacktriangleleft OD \\ \textbf{fi} \end{array} \right) \end{array} \right) \blacktriangleright AD;\ X \end{array} \right) \\
\quad Mission = \left(\left(\begin{array}{l} PeriodicHandler \\ [\![\{\} \mid \{\!| stop, release |\!\} \mid \{\}]\!] \\ AperiodicHandler \end{array} \right) \setminus \{\!| release |\!\} \right) \setminus \{\!| setBuffer, getBuffer |\!\} \\
\qquad [\![\{\} \mid \{\!| \ldots |\!\} \mid \{\}]\!]\, MArea \\
\quad MissionSequencer \mathrel{\widehat=} Mission \\
\quad Safelet \mathrel{\widehat=} MissionSequencer \\
\quad Application \mathrel{\widehat=} Safelet \\
\quad \bullet\, Application \\
\textbf{end}
\end{array}
$$

Figure 10: SCJ Level 1 example: E-anchor

next cycle starts. In this paper, we focus on E-anchor of the first type: cyclic in lockstep. The model in Figure 4 follows this pattern. Examples of the remaining patterns can be found in [12].

4.2 (CF) Introduction of SCJ control flow

This phase introduces some of the parallel structure observed in *SCJ-Circus* programs. Figure 11 shows the structure of the process obtained by applying the first phase.

$$
\begin{array}{l}
\textbf{process}\ CF\_P \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ S \\
\quad CF\_Safelet \mathrel{\widehat=} getSequencerCall \longrightarrow geSequencerRet?x \longrightarrow start\_sequencer!x \longrightarrow \\
\qquad done\_sequencer!x \longrightarrow \textbf{Skip} \\
\quad CF\_MissionSequencer \mathrel{\widehat=} \ldots \\
\quad CF\_Mission_j \mathrel{\widehat=} \ldots \\
\quad CF\_Handler_i \mathrel{\widehat=} \ldots \\
\quad CF\_Application \mathrel{\widehat=} \left(\begin{array}{l} CF\_Safelet \parallel CF\_MissionSequencer \\ \parallel ((\parallel i : 1 \mathinner{..} n \bullet CF\_Mission_i) \parallel (|||i : 1 \mathinner{..} m \bullet CF\_handler_i)) \end{array} \right) \\
\quad \bullet\ CF\_Application \\
\textbf{end}
\end{array}
$$

Figure 11: Refinement strategy: target of CF phase

We recall that, as the first phase of the refinement, its starting point is an E-anchor described in Figure 9, and illustrated in Figure 10 for our example. The target is shown in Figure 11. This is a model is still a single process, but its main action composes a number of auxiliary actions in parallel, each of which specifies the behaviours of an SCJ abstraction.

In this phase, parallelism introduction laws such as Law 1 are used to refine an action $F(A)$ into a parallelism where the subaction A is replaced by two communications on channels c_1 and c_2, and the parallel action is formed by the first communication on c_1, followed by the subaction A, followed by the second communication on c_2.

Law 1. Parallelism Introduction.

$$F(A) \sqsubseteq (F(c_1 \longrightarrow c_2 \longrightarrow \textbf{Skip}) \llbracket usedV(F) \mid \{\!|\, c_1, c_2 \,|\!\} \mid userV(A) \rrbracket\ c_1 \longrightarrow A;\ c_2 \longrightarrow \textbf{Skip}) \setminus \{\!|\, c_1, c_2 \,|\!\}$$

provided $usedV(F) \cap usedV(A) = \varnothing \wedge \{\!|\, c_1, c_2 \,|\!\} \cap usedC(F(A)) = \varnothing$

This law can be proved by structural induction over the structure of the action F using distribution and step laws such as the ones found in [15]. The provisos guarantee that c_1 and c_2 are not used in A, and that the variables used in the action F and the subaction A form a partition of the state, so that they can be put in parallel without creating race conditions. As shown, the *Circus* parallel operator for actions defines partitions of the state for use of each of the parallel actions.

In general, Law 1 must be applied to the actions that model the safelet, the mission sequencer, the missions, and the handlers. In our example, this law is applied to the action *Safelet* in Figure 10 to separate it from *MissionSequencer*, and then to the action *MissionSequencer* to separate it from *Mission*, and finally to the action *Mission* to separate it from *Handlers*. At this point, we obtain the action *CF_Application*. The resulting structure is depicted in Figure 11, where the actions prefixed by *CF_* are the actions in Figure 10 modified by the application of the refinement laws.

4.3 (AP) Introduction of application processes

The target of this phase is shown in Figure 12: it defines a number of application processes, and refines the process *CF_P* into the parallel composition of the interleaved application processes and a modified version of *CF_P* (*CF_P_FW*), where application-specific behaviours have been replaced by calls to actions of the application processes via channel communications using *Call* and *Ret* channels.

$$\begin{array}{l}
\textbf{process}\ Handler_i_app \mathrel{\widehat{=}} \ldots \\
\textbf{process}\ Mission_j_app \mathrel{\widehat{=}} \ldots \\
\textbf{process}\ MissionSequencer_app \mathrel{\widehat{=}} \ldots \\
\textbf{process}\ Safelet_app \mathrel{\widehat{=}} \ldots \\
\textbf{process}\ AP_P \mathrel{\widehat{=}} CF_P_FW \parallel \left(\begin{array}{l} Safelet_app \mid\mid\mid MissionSequencer_app \mid\mid\mid \\ (\mid\mid\mid i : 1 \mathinner{..} m \bullet Handler_i_app) \mid\mid\mid (\mid\mid\mid i : 1 \mathinner{..} n \bullet Mission_i_app) \end{array} \right)
\end{array}$$

Figure 12: Refinement strategy: target of phase **AP**

$$\begin{array}{l}
CF_System \left[\left(\begin{array}{l} (input?x \longrightarrow \textbf{Skip}) \blacktriangleleft ID; \\ setBuffer!(buffer \frown \langle x \rangle) \longrightarrow release \longrightarrow (\textbf{wait}\, 0 \mathinner{..} PTB) \end{array} \right) \blacktriangleright PD \right] \\
\sqsubseteq \\
\left(\begin{array}{l} CF_System \left[\begin{array}{l} handleAsyncEventCall?x!PHID \longrightarrow \\ handleAsyncEventRet!x!PHID \longrightarrow \textbf{Skip} \end{array} \right] \\ [\![\{\} \mid \{\!| handleAsyncEventCall, handleAsyncEventRet \mid\} \}]\!] \\ PeriodicHandler_App \end{array} \right) \setminus \{\!| \ldots |\!\}
\end{array}$$

Figure 13: Introduction of application process for *PeriodicHandler* in our example.

In this phase, we use the process obtained in phase **CF** to identify the behaviours that are application specific and construct application processes. Next, each application process is introduced in parallel with the original process and the behaviour provided by the application process is replaced in the original process by calls via the appropriate channels. This is achieved using refinement laws similar to Law `server-intro` in [11], which supports the introduction of a server-client architecture.

Figure 13 illustrates the application of this phase to the *CF_PeriodicHandler* action obtained after the application of the first phase to the example in Figure 10. First, the process *PeriodicHandler_App* is generated as a recursion that, at each step, offers the event *handleAsyncEventCall*, executes the behaviour of the original periodic handler, and synchronises on *handleAsyncEventRet*. Next, the process *CF_System* obtained by the first phase containing the behaviour of the periodic handler (made explicit in Figure 13 by the square brackets after *CF_System*) is refined into the parallel composition of the generated application process *PeriodicHandler_App* and *CF_System* with the behaviour of the periodic event handler in brackets replaced by the synchronisations on *handleAsyncEventCall* and *handleAsyncEventRet* (*PHID* is the identifier of the periodic handler and is necessary to support multiple handlers).

4.4 (FW) Introduction of framework processes

The target of this phase is shown in Figure 14; it consists of the interleaved application processes in parallel with the parallel composition of the framework processes discussed in Section 3.2.

This phase acts on the process of *CF_P_FW* in Figure 12, from which all application-specific behaviours have been removed (and distributed to the application processes). What remains in *CF_P_FW* after the second phase are the framework behaviours that are relevant to the particular application. In this phase, we complement these framework behaviours to account for the behaviours that are part of the framework, but not used in *CF_P_FW*. This is achieved by the application of refinement laws such as

$$\textbf{process}\ FW\_P \mathrel{\widehat=} \left(\begin{array}{l} \left(\begin{array}{l} SafeletFW \parallel SequencerFW \parallel (|||i : 1..n \bullet MissionFW(mission_i)) \parallel \\ (|||i : 1..m \bullet HandlerFW(handler_i)) \end{array} \right) \\ \parallel \left(\begin{array}{l} Safelet\_app \mid\mid\mid MissionSequencer\_app \mid\mid\mid \\ (|||i : 1..m \bullet Handler_i\_app) \mid\mid\mid (|||i : 1..n \bullet Mission_i\_app) \end{array} \right) \end{array} \right)$$

Figure 14: Refinement strategy: target of phase **FW**

handler $S\_Handler_i \mathrel{\widehat=} \ldots$
mission $S\_Mission_j \mathrel{\widehat=} \ldots$
sequencer $S\_MissionSequencer \mathrel{\widehat=} \ldots$
safelet $S\_Safelet \mathrel{\widehat=} \ldots$

Figure 15: Refinement strategy: target of phase **Conv**

Law 2 to introduce the actions that correspond to the control flow present in the framework processes but not used by the application processes.

Law 2. Unused behaviour introduction.

$$(a \longrightarrow A \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, a \longrightarrow B) \sqsubseteq (a \longrightarrow A \,[\![\, ns_1 \mid cs \cup \{\!|\, b \,|\!\} \mid ns_2 \,]\!]\, (a \longrightarrow B \,\Box\, b \longrightarrow C))$$

provided $a \in cs \land b \notin usedC(A, B)$

Law 2 allows the introduction of actions in a parallelism that are never used; it relies on the fact that the channel b is not used anywhere else in the left hand side. Since b is in the synchronisation set of the refined action, the action $b \longrightarrow C$ can never be started.

Finally, process parallelism introduction laws are used to refine the process CF_P into the process FW_P defined as a parallel composition of processes whose main actions are FW_A. The structure of the refined process follows the structure of the main action of CF_P. Figure 14 shows the structure of process FW_P obtained in this phase, where $mission_i$ is the identifier of the i-th mission, $handler_i$ is the identifier of the i-th handler, and $HandlerFW$ is either $PEHFW$ or $APEHFW$ depending on whether the i-th handler is periodic or aperiodic.

4.5 (Conv) Conversion to *SCJ-Circus*.

The target of this phase is shown in Figure 15; it explicitly refers to the SCJ abstractions that have been incorporated in *SCJ-Circus*. In this final phase of our refinement strategy, the top-level parallel actions of FW_P in Figure 14 are merged into a parallel composition of pairs of application and framework processes. This is achieved by the application of a procedure similar to the one used in a refinement strategy described in [14]. It relies on the syntactic structure of the parallel actions and the use of refinement laws to eliminate or distribute the parallel composition over other *Circus* constructs such as external choice, recursion and interleaving.

Next, each parallel composition of application and framework processes ($A\_app \parallel A\_FW$) is used to define a new process A, and the process FW_P is refined by replacing the parallelisms of the form $A\_app \parallel A\_FW$ by a call to the newly defined processes. The resulting processes are shown in Figure 16. At this point, each SCJ abstraction is defined by its own process that includes the application and

$$\textbf{process}\ Safelet \mathrel{\widehat=} SafeletFW \parallel Safelet\_app$$
$$\textbf{process}\ MissionSequencer \mathrel{\widehat=} SequencerFW \parallel MissionSequencer\_app$$
$$\textbf{process}\ Mission_j \mathrel{\widehat=} MissionFW(mission_j) \parallel Mission_j\_app$$
$$\textbf{process}\ Handler_i \mathrel{\widehat=} HandlerFW(handler_i) \parallel Handler_i\_app$$
$$\textbf{process}\ FW\_P \mathrel{\widehat=} (Safelet \parallel Sequencer \parallel (|||i : 1 \mathinner{..} n \bullet Mission_i) \parallel (|||i : 1 \mathinner{..} m \bullet Handler_i))$$

Figure 16: Refinement strategy: parallelism elimination in phase **FW**

framework-specific behaviours in different parallel processes.

Finally, the semantics of *SCJ-Circus* is used to refine each newly defined process A into the corresponding *SCJ-Circus* abstraction, and the sequence of *SCJ-Circus* abstractions into a complete *SCJ-Circus* program. Figure 15 shows the general structure of the program resulting from this phase, and Figure 4 shows the result of applying our refinement strategy to the E-Anchor of our running example.

5 Conclusions

In this paper, we extend previous work [21, 5] on both the semantics of SCJ and refinement strategies for SCJ programs. We propose a variant of *Circus* suitable for modelling SCJ concepts, update existing models of SCJ to reflect changes to the SCJ specification and better suit the goal of compositional verification, formalise the semantics of *SCJ-Circus* in terms of these updated models, and extend a previously proposed refinement strategy to account for the refinement to *SCJ-Circus* specifications.

Other significant differences between our model of SCJ and that in [21] include: (1) the shift from the use of events to trigger the execution of aperiodic event handlers in previous version of the SCJ specification, to the direct use of the asynchronous method `release` of the aperiodic event handler, and (2) modelling of handlers using two processes *PEHFW* (periodic event handler) and *APEHFW* (aperiodic event handler) so that the distinction between periodic and aperiodic event handlers are made at the framework level, instead of the application level.

The SCJ standard specifies the new constructs, the API, and the SCJ VM, but says nothing about verification and design of programs. Our effort complements those in [8, 17, 6, 10]. Kalibera *et al.* [8] apply model checking and exhaustive testing to perform scheduling and race-condition analysis in SCJ programs. Haddad *et al.* [6] extend the Java Modeling Language [1] with timing properties to support worst-case execution analysis of SCJ programs, whilst Tang *et al.* [17] use annotations to analyse SCJ programs for memory safety and compliance to SCJ levels. Marriott *et al.* [10], on the other hand, perform automatic verification of memory-safety without requiring the user to annotate the program.

We identify four main application patterns with respect to the timing properties of the periodic and aperiodic event handlers. Whilst we focus our effort here on the refinement of one particular pattern (cyclic in lockstep), the refinement strategy is general enough to be applied to the other patterns with localized changes. We will address this issue in the context of SCJ in future work. We will also detail the refinement strategy and mechanise it in a theorem prover in order to further validate it.

Acklowledgements. This work is funded by the EPSRC grant EP/H017461/1. No new primary data was created during this study.

References

[1] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino & E. Poll (2005): *An Overview of JML Tools and Applications*. *Int. J. Softw. Tools Technol. Transf.* 7(3), pp. 212–232, doi:10.1007/s10009-004-0167-4.

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

[3] A. L. C. Cavalcanti, P. Clayton & C. O'Halloran (2005): *Control Law Diagrams in* Circus. In J. Fitzgerald, I. J. Hayes & A. Tarlecki, editors: *FM 2005: Formal Methods*, *LNCS* 3582, Springer-Verlag, pp. 253–268, doi:10.1007/11526841_18.

[4] A. L. C. Cavalcanti, A. C. A. Sampaio & J. C. P. Woodcock (2003): *A Refinement Strategy for* Circus. *Formal Aspects of Computing* 15(2 - 3), pp. 146–181, doi:10.1007/s00165-003-0006-5.

[5] A. L. C. Cavalcanti, F. Zeyda, A. Wellings, J. C. P. Woodcock & K. Wei (2013): *Safety-critical Java programs from* Circus *models*. *Real-Time Systems* 49(5), pp. 614–667, doi:10.1007/s11241-013-9182-4.

[6] G. Haddad, F. Hussain & G. T. Leavens (2010): *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*, JTRES '10, ACM, pp. 155–163, doi:10.1145/1850771.1850793.

[7] C. A. R. Hoare & J. He (1998): *Unifying Theories of Programming*. Prentice-Hall.

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

[9] D. Locke, B. S. Andersen, M. Fulton B. Brosgol, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nielsen, M. Schoeberl, J. Vitek & A. Wellings: *Safety-Critical Java Technology Specification*. Technical Report.

[10] C. Marriott & A. L. C. Cavalcanti (2014): *SCJ: Memory-safety checking without annotations*. In: *Formal Methods*, *LNCS* 8442, Springer, pp. 465–480, doi:10.1007/978-3-319-06410-9_32.

[11] A. Miyazawa (2012): *Formal verification of implementations of Stateflow charts*. Ph.D. thesis, Department of Computer Scinece, The University of York, York, UK.

[12] A. Miyazawa & A. Cavalcanti (2015): *Refinement of* Circus *models into SCJ-Circus*. `http://www-users.cs.york.ac.uk/~alvarohm/report2015a.pdf`.

[13] A. Miyazawa & A. L. C. Cavalcanti (2012): *Refinement-oriented models of Stateflow charts*. *Science of Computer Programming* 77(10-11), pp. 1151–1177, doi:10.1016/j.scico.2011.07.007.

[14] A. Miyazawa & A. L. C. Cavalcanti (2013): *Refinement-based verification of implementations of Stateflow charts*. *Formal Aspects of Computing* 26(2), pp. 367–405, doi:10.1007/s00165-013-0291-6.

[15] M. V. M. Oliveira (2006): *Formal Derivation of State-Rich Reactive Programs Using Circus*. Ph.D. thesis, University of York.

[16] A. W. Roscoe (2011): *Understanding Concurrent Systems*. Texts in Computer Science, Springer.

[17] D. Tang, A. Plsek & J. Vitek (2010): *Static Checking of Safety Critical Java Annotations*. In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, ACM, pp. 148–154, doi:10.1145/1850771.1850792.

[18] K. Wei, J. C. P. Woodcock & A. L. C. Cavalcanti (2012): Circus Time *with Reactive Designs*. In: *4th International Symposium on Unifying Theories of Programming*, LNCS, doi:10.1007/978-3-642-35705-3_3.

[19] Andrew Wellings (2004): *Concurrent and Real-Time Programming in Java*. John Wiley & Sons.

[20] J. C. P. Woodcock & J. Davies (1996): *Using Z—Specification, Refinement, and Proof*. Prentice-Hall.

[21] F. Zeyda, L. Lalkhumsanga, A. L. C. Cavalcanti & A. Wellings (2013): Circus *Models for Safety-Critical Java Programs*. *The Computer Journal*, doi:10.1093/comjnl/bxt060.