Refinement strategies for Safety-Critical Java

Alvaro Miyazawa[1] and Ana Cavalcanti[2]

[1] alvaro.miyazawa@york.ac.uk,
[2] ana.cavalcanti@york.ac.uk,
Department of Computer Science
University of York, York, UK

Abstract. Safety-Critical Java (SCJ) is a version of Java that supports the development of real-time, embedded, safety-critical software. SCJ introduces abstractions that enforce a simpler architecture, and simpler concurrency and memory models, to support easier certification. In this paper, we detail a refinement strategy that takes a state-rich process algebraic design specification that adheres to a cyclic executive pattern and produces an SCJ design that can be automatically translated to code. We then show how this refinement strategy can be extended to support more complex patterns that include non-terminating and multiple missions.

1 Introduction

Safety-Critical Java (SCJ) [7] is a version of Java suitable for the development of verifiable real-time software. It incorporates part of the Real-Time Specification for Java (RTSJ) [17], introduces new abstractions such as Safelets and Missions, and removes garbage collection by enforcing the use of scoped memory regions. All this supports predictable timing behaviours.

SCJ enforces particular programming patterns via simplified memory and concurrency models. SCJ programs can adopt one of three profiles, called levels, which support an increasing number of abstractions. In this paper, we focus on the intermediate level of SCJ programs (level 1), which enforces a structure where a safelet (the main program) defines a mission sequencer, which in turn provides a number of missions that are run in sequence as shown in Figure 1. This level is comparable in complexity to the Ravenscar profile for Ada. While adequate to a wide range of applications, it is amenable to formal reasoning.

An SCJ application is formed by a safelet, a mission sequencer, a number of missions, and periodic and aperiodic event handlers. A safelet instantiates a mission sequencer, and iteratively obtains a mission from the mission sequencer, executes it and waits for it to terminate. At level 1, each mission is formed by a collection of periodic and aperiodic event handlers that are run concurrently.

A safelet terminates when there are no missions left to be executed, and a mission terminates when one of its handlers requests termination. In the SCJ memory model, which is based on scoped memory regions, safelets, missions, and handlers each have associated memory regions, which are cleared at specific points of the program execution. This ensures predictable time properties.

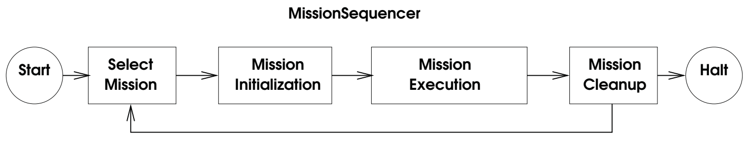

Fig. 1. SCJ programming model

Cavalcanti et al. [5] proposes a design technique for SCJ based on the *Circus* family of languages for refinement, which are state-rich process algebras for refinement that include aspects such as object-orientation and timing; it has been used to support verification of a number of different models such as Simulink/Stateflow diagrams [4,11] and SysML [13,9].

In [12], we describe the syntax and semantics of a version of *Circus* called *SCJ-Circus*, which includes the SCJ abstractions. In that work, we identify four patterns of *Circus* specifications used as a basis for the development of SCJ programs defined by *SCJ-Circus* models. We also describe the main phases of a refinement strategy that takes a *Circus* specification based on one of the patterns for a non-terminating cyclic design with one mission whose handlers are in lockstep. The strategy produces an *SCJ-Circus* program that can be directly converted into an SCJ program. Here, we show that a refinement strategy for the terminating versions of the patterns identified in [12] can be extended and composed to support the refinement of a wider variety of patterns, in particular, non-terminating and multi-mission patterns.

We first focus on the terminating cyclic in lockstep pattern, and detail the refinement strategy for it. We then show how this strategy can be extended to support the refinement of the non-terminating cyclic in lockstep pattern of [12] and a multi-mission pattern, in which the missions are provided sequentially and each mission follows the terminating cyclic in lockstep pattern. Besides considering refinement for a collection of patterns that is significantly larger than that in [12], we also provide a detailed account of the strategy, instead of just an overview. Whilst most of the laws used in our strategies have only syntactic provisos, a few require more complex provisos, such as deadlock freedom.

Section 2 introduces *Circus* and Section 3 discusses our patterns, focusing on the terminating cyclic in lockstep pattern. Section 4 details our refinement strategy for our target pattern. Section 5 and 6 extend the refinement strategy of Section 4 to cover non-terminating and multi-mission patterns. Finally, Section 7 concludes by reviewing our results and discussing future work.

2 *Circus*

In this section, we use the *Circus* process *ThreeEqual* in Figure 2, which models an SCJ level 1 application that contains two event handlers, to give an overview of the notation. It defines a program that takes integers as inputs, and outputs booleans indicating whether the last three inputs are equal or not.

$$
\begin{array}{l}
\mathbf{process}\ \mathit{ThreeEqual} \mathrel{\widehat=} \mathbf{begin}\\
\quad \mathit{InputHandler} \mathrel{\widehat=}\\
\qquad \mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} (\mathit{input}?x \longrightarrow \mathbf{Skip}) \blacktriangleleft \mathit{ID};\ \mathit{setBuffer}!x \longrightarrow \\ (\mathbf{wait}\ 0..\mathit{PTB});\ \mathit{checkRepeats} \longrightarrow \mathbf{Skip} \end{array} \right) \blacktriangleright \mathit{PD} \\ ||| \ \mathbf{wait}\ P \end{array} \right);\ X\\
\quad \mathit{OutputHandler} \mathrel{\widehat=}\\
\qquad \mu X \bullet \left(\begin{array}{l} \mathit{checkRepeats} \longrightarrow \\ \quad \left(\begin{array}{l} \mathit{getBuffer}?\mathit{buffer} \longrightarrow \mathbf{wait}\ 0..\mathit{ATB}; \\ \left(\begin{array}{l} \mathbf{if}\ \mathit{check}(\mathit{buffer}) = \mathbf{True} \longrightarrow \\ \quad (\mathit{output}!\mathit{true} \longrightarrow \mathbf{Skip}) \blacktriangleleft \mathit{OD} \\ [\!]\, \mathit{check}(\mathit{buffer}) = \mathbf{False} \longrightarrow \\ \quad (\mathit{output}!\mathit{false} \longrightarrow \mathbf{Skip}) \blacktriangleleft \mathit{OD} \\ \mathbf{fi} \end{array} \right) \end{array} \right) \blacktriangleright \mathit{AD};\ X \end{array} \right)\\
\quad \mathit{MArea} \mathrel{\widehat=} \mathbf{var}\ \mathit{buffer} : \operatorname{seq} \mathbb{N} \bullet \mathit{buffer} := \langle 0,0,0 \rangle;\ \mu X \bullet\\
\qquad \left(\begin{array}{l} \mathit{setBuffer}?x \longrightarrow \mathit{buffer} := (\mathit{tail}\ \mathit{buffer} \frown \langle x \rangle);\ X \\ \Box \mathit{getBuffer}!\mathit{buffer} \longrightarrow X \mathrel{\Box} \mathit{stop} \longrightarrow \mathbf{Skip} \end{array} \right)\\
\quad \mathit{TEMission} =\\
\qquad \left(\begin{array}{l} \left(\begin{array}{l} \mathit{InputHandler} \\ [\![\{\} \mid \{\!| \mathit{stop}, \mathit{checkRepeats} |\!\} \mid \{\}]\!] \\ \mathit{OutputHandler} \end{array} \right) \setminus \{\!| \mathit{checkRepeats} |\!\} \\ [\![\{\} \mid \{\!| \mathit{setBuffer}, \mathit{getBuffer} |\!\} \mid \{\}]\!]\ \mathit{MArea} \end{array} \right) \setminus \{\!| \ldots |\!\}\\
\quad \mathit{TEMissionSequencer} \mathrel{\widehat=} \mathit{TEMission}\\
\quad \mathit{TESafelet} \mathrel{\widehat=} \mathit{TEMissionSequencer}\\
\quad \bullet\ \mathit{TESafelet}\\
\mathbf{end}
\end{array}
$$

Fig. 2. SCJ Level 1 example: ThreeEqual

The main modelling element of a *Circus* specification is a process (indicated by the keyword **process**). A basic process declares state components (identified by the keyword **state**), a number of auxiliary actions, and a main action (at the end prefixed by •) that describes the overall behaviour of the process. Processes can also be combined using CSP operators to define other processes.

Processes communicate with each other and with the environment via channels. In the case of our example, the process *ThreeEqual* does not declare any state components; its interface is characterised by the channels *input* and *output*.

ThreeEqual declares six auxiliary actions: *InputHandler*, *OutputHandler*, *MArea*, *TEMission*, *TEMissionSequencer* and *TESafelet*. Actions are specified using a combination of Z [18] for data modelling and CSP [15] for behavioural descriptions. The main action is defined by a direct call to the action *TESafelet*.

In general, a safelet may have an initialisation, but in our example, its behaviour is just that of the action *TEMissionSequencer*. In general, a mission sequencer defines a sequence of missions, but here *TEMissionSequencer* defines just the mission *TEMission*. The action *MArea* represents a memory area that holds a buffer. It has a block (**var** ... • ...) that declares a local variable *buffer* of type $\operatorname{seq} \mathbb{N}$, and whose body is defined by a recursion ($\mu X \bullet \ldots$) that at each step offers a choice of reading a value on the (internal) channel *setBuffer*, storing it in *buffer* and recursing, or outputting the value of *buffer* on the channel *getBuffer* and recursing, or synchronising on *stop* and terminating. The action *Mission* composes in parallel ($[\![\ldots \mid \ldots \mid \ldots]\!]$) the two event handlers synchronising on *stop* and *checkRepeats*, with *checkRepeats* hidden ($\setminus$), and the action *MArea* synchronising on the channels *setBuffer* and *getBuffer*.

The action *InputHandler* represents a task with period P, that must get its input within ID time units, can take up to PTB time units to complete, but no more than PD. It is also defined by a recursion, where at each step two actions are started in interleaving. The first action must take at most PD time units as indicated by the end-by deadline operator $\blacktriangleright$. It reads an input with a deadline of ID time units as indicated by the start-by deadline operator $\blacktriangleleft$, appends the value to the end of the tail of *buffer*, waits (**wait**) between 0 and PTB time units, and then synchronises on *checkRepeats*, before terminating (**Skip**). The **wait** $0 \mathrel{..} PTB$ models a budget of time for the update of the buffer of PTB time units. We observe that data operations take no time, unless explicitly specified otherwise. The communication of *checkRepeats* triggers the *OutputHandler*. The second interleaved action in *InputHandler* waits for exactly P time units and guarantees that a new iteration of the recursion does not start before the end of the cycle, whose duration is P time units.

The action *OutputHandler* represents a task triggered by a synchronisation on the channel *checkRepeats*; the task can take up to ATB time units to complete, but must terminate in less than AD time units. It is defined by a recursion, where each step synchronises on *checkRepeats*, reads the value of the buffer on the channel *getBuffer*, waits for up to ATB time units, and checks whether the value read is in the set *check* or not. In the first case it outputs the value *true* on the channel *output* within OD time units, in the second case it outputs *false* on the same channel under the same time restriction. The whole step must terminate within AD time units as indicated by the deadline operator.

In general, a *Circus* 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 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 [14,16,3]. In the sequel, we further explain the notation as needed.

3 Patterns

The cyclic executive pattern that has been identified in our previous work is shown in Figure 3. It requires that the application has a single mission with a fixed cycle, and all periodic and aperiodic event handlers execute at each cycle. This requirement radically simplifies the refinement strategy since it allows the

$$
\begin{array}{l}
P \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\, S \\
\quad PHandler_i \mathrel{\widehat=} \mu\, X \bullet (F_i \blacktriangleright PD \interleave \mathbf{wait}\, PERIOD);\ X \Box t \longrightarrow \mathbf{Skip} \\
\quad AHandler_j \mathrel{\widehat=} \mu\, X \bullet (c_j \longrightarrow G_j) \blacktriangleright AD;\ X \Box t \longrightarrow \mathbf{Skip} \\
\quad MArea \mathrel{\widehat=} \ldots \\
\quad Termination \mathrel{\widehat=} rt \longrightarrow \mu\, X \bullet (rt \longrightarrow X \Box t \longrightarrow \mathbf{Skip}) \\
\quad Mission \mathrel{\widehat=} \left(\begin{array}{l} (MArea \parallel (\parallel i : I \bullet PHandler_i) \parallel (\parallel j : J \bullet AHandler_j)) \\ \quad [\![\alpha S \mid \{\!|\, t, rt \,|\!\} \mid \{\}]\!] \\ Termination \end{array}\right) \\
\quad MissionSequencer \mathrel{\widehat=} Mission \\
\quad Safelet \mathrel{\widehat=} MissionSequencer \\
\quad \bullet\ Safelet \\
\mathbf{end}
\end{array}
$$

$$\mathbf{where}\ PD \leq PERIOD \wedge AD \leq PERIOD$$

Fig. 3. Pattern: cyclic in lockstep.

transformation of synchronous releases of aperiodic event handlers in the models into the asynchronous releases that occur in *SCJ-Circus*.

In this pattern, parallelism occurs between the actions that model the mission and the mission memory, and the termination management action, and within the mission action between the different event handlers. The mission memory action $MArea$ declares the variables shared by the handlers and offers communications over $get\_$ and $set\_$ channels that support reading and writing to the shared variables. The termination management action accepts a synchronisation on a channel rt; this corresponds to a request from some handler to terminate. Afterwards, the action starts a recursion where at each step, it either accepts another communication on rt (corresponding to a further request to terminate) and recurses, or it synchronises on t and terminates. The communication on t has the effect of terminating the handlers of the mission.

A parallelism of actions in *Circus* needs to identify the partition of variables that each parallel action can modify to avoid race conditions. So far, these sets of variables have been empty. In the case of the $Mission$ action, all variables αS in scope can be modified by the parallelism of handlers, while the termination management action modifies no variables.

Periodic event handlers $PHandler_i$ are defined as recursive actions, where each step takes a fixed amount of time ($\mathbf{wait}\, PERIOD$) whilst executing (in interleave) some behaviour that takes at most PD time units (indicated by the $\blacktriangleright$ operator). The execution of an aperiodic event handler $AHandler_j$, on the other hand, is triggered by a synchronous event c_j, which may occur at any time during the cycle of the mission that lasts $PERIOD$ time units, but must terminate within AD time units. In order to guarantee the cyclic behaviour, the pattern imposes timing restrictions over the behaviours of both periodic and aperiodic handlers, which guarantee that they terminate within the cycle of the mission.

The second pattern that we consider allows for an aperiodic event handler not to be called in a cycle. For this to be possible, the deadlines of the aperiodic event handler must be adapted so that they can be missed as long as it has not

$$AHandler_j \mathrel{\widehat=} \mu\, X \bullet (c_j \longrightarrow G_j) \blacktriangleright AD;\ X \Box t \longrightarrow \mathbf{Skip} \Box \mathbf{wait}\ PERIOD;\ X$$

$$\mathbf{where}\ AD \leq PERIOD$$

Fig. 4. Pattern: cyclic not in lockstep.

$$\begin{array}{l}
AHandler_j \mathrel{\widehat=} \\
\left(\begin{array}{l}
\mathbf{var}\ pr : boolean \bullet pr := false; \\
\mu\, X \bullet \left(\begin{array}{l} c_j \longrightarrow pr := true;\ X \Box \left(pr\right) \mathbin{\&} hr \longrightarrow pr := false;\ X \\ \Box t \longrightarrow \mathbf{Skip} \end{array}\right) \\
[\![\{\} \mid \{\!|\, hr, t \,|\!\} \mid \alpha(G_j)]\!] \\
\mu\, X \bullet (hr \longrightarrow G_j;\ X \Box t \longrightarrow \mathbf{Skip})
\end{array}\right) \setminus \{\!|\, hr \,|\!\}
\end{array}$$

Fig. 5. Pattern: non-cyclic.

started. This pattern differs from that in Figure 3 just in the characterisation of the aperiodic handlers, which is presented in Figure 4. It requires the release via the channel c_j and the associated handling action G_j to terminate within AD time units, or termination through the channel t. Furthermore, if release or termination does not take place within $PERIOD$ time units, the choice terminates and a new cycle of the recursion is started. The timing restrictions of the first pattern over the periodic event handlers also apply to the second pattern.

Finally, the third pattern imposes no restriction on the timing of the handlers, which means that the behaviour of a periodic handler can take longer than its period, and the behaviour of the aperiodic handler can be called multiple times even if its action is not yet completed. In this case, the pattern for aperiodic handler models is as shown in Figure 5. In SCJ, release requests are asynchronous. So, if requests for further releases occur before a handler is ready to execute again, one, and just one, pending release is recorded.

Accordingly, the actions of the form described in Figure 5 accumulate an asynchronous release. It is used to support accumulation of at most one asynchronous release and consists of two parallel actions. The second parallel action models the behaviour of the handler itself in the usual way, but it is triggered by the internal event hr. The actual triggering is managed by the first parallel action. It receives release requests through the channel c_j and records them in the local variable pr (pending release), or requests for termination on t. When there is a pending release (pr), then it can release the handler (hr), but it can never have more than one pending release. This is, no doubt, a very specific pattern, in spite of the absence of time restrictions. We emphasise that the starting point of our refinement strategy is not an abstract model, but an SCJ design. Such a design can be obtained using the refinement strategy in [5], for example.

The purpose of the refinement strategy that we propose here and detail in the next few sections is threefold. First, it guarantees that the design embedded by the patterns can indeed be realised in SCJ. Not every model that conforms to our patterns can be correctly implemented in SCJ with the suggested structure of missions and actions. For example, in the design model, there may be a possibility that an aperiodic handler is not released in a particular cycle. Such a model

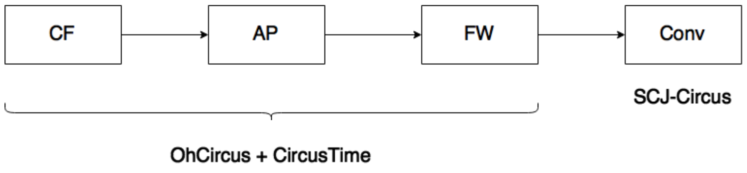

Fig. 6. Phases of the refinement strategy

$\mathbf{safelet}\, \mathit{Safelet} \mathrel{\widehat=} \mathbf{begin} \ldots \mathbf{end}$
$\mathbf{sequencer}\, \mathit{Sequencer} \mathrel{\widehat=} \mathbf{begin} \ldots \mathbf{end}$
$\mathbf{mission}\, \mathit{Mission} \mathrel{\widehat=} \mathbf{begin} \ldots \mathbf{end}$
$\mathbf{periodic\ handler}\, \mathit{PHandler}_i \mathrel{\widehat=} \mathbf{begin} \ldots \mathbf{handleAsyncEvent} \mathrel{\widehat=} F_i \blacktriangleright PD\, \mathbf{end}$
$\mathbf{aperiodic\ handler}\, \mathit{AHandler}_j \mathrel{\widehat=} \mathbf{begin} \ldots \mathbf{handleAsyncEvent} \mathrel{\widehat=} G_j \blacktriangleright AD\, \mathbf{end}$

Fig. 7. Target

cannot be realised as a cyclic execution in lockstep. Its rendering in SCJ may lead to visible inputs and outputs not allowed in the model.

Second, by deriving via refinement an SCJ-Circus model, we enable automatic translation to SCJ code via trivial transformations whose soundness is not a concern. Finally, we obtain a model whose abstractions are in direct correspondence with those of the SCJ paradigm. In this model, reasoning about use of the memory model, for example, is much simpler.

4 Refinement strategy

In this section, we present the refinement strategy for the terminating cyclic lockstep pattern. It consists of the same phases as the refinement strategy in [12] shown in Figure 6. The target is an *SCJ-Circus* program of the form defined in Figure 7. Each periodic action in the starting design model has a corresponding periodic event handler paragraph in the target specification, where the **handleAsyncEvent** method is defined as $F_i \blacktriangleright PD$. Similarly, each aperiodic, mission, mission sequencer and safelet action has a corresponding paragraph.

The four phases of our refinement strategy are as follows: (CF) introduction of SCJ control flow, (AP) introduction of application processes, (FW) introduction of framework processes, (Conv) conversion to *SCJ-Circus*. CF makes the control flow of the SCJ paradigm, which is implicitly defined in the patterns via sequential and parallel compositions, explicitly captured by channel synchronisations. For example, we introduce, a channel *activate_handlers* that models the synchronised start of the handlers in parallel. AP separates application-specific behaviours (for example, handler behaviours) from behaviours such as starting a mission, which are implemented by an SCJ runtime environment (framework). FW takes the incomplete model of framework behaviour, representing a slice of the SCJ framework actually used by the application, and replaces it by the full-fledged framework model. Finally, Conv refines the specification into a sequence of *SCJ-Circus* paragraphs. As illustrated in Figure 6, the first three phases act only on constructs of the time and object-oriented languages, *Circus Time* and *OhCircus*, whilst the last phase manipulates *SCJ-Circus* specifications, which complement those two notations with SCJ specific constructs.

4.1 (CF) Introduction of SCJ control flow

This phase extracts each of the SCJ abstractions from the starting design model into parallel actions. It derives, from a design like that in Figure 3, a process structured as shown in Figure 8. Its main action is the parallel composition of actions corresponding to specific SCJ abstractions. The order of execution imposed by the original specification is maintained through the use of communication channels such as $start\_mission$ and $start\_sequencer$.

Figure 9 presents the steps necessary to reach its target. The laws named there can be found in [1]. Due to space restrictions, only some are presented and explained here. Overall, this phase identifies the actions of the starting process that model specific abstractions and applies specialised laws to parallelise the safelet, mission sequencer and mission actions. Next, handler laws replace synchronous communications between handlers with asynchronous communications, and separate the handlers from the mission action. Finally, *Circus* laws are used to merge parallel actions associated with mission execution.

The law call-intro is used to split an action $F(A)$ into the parallel composition of two actions, one of which executes the behaviour of the subaction A of $F(A)$. To retain the control flow of $F(A)$, internal channels cs and ce are used to synchronise the parallel actions. In $F(A)$, the action A is replaced with a call to the parallel action using the internal channels.

Law [call-intro]

$$F(A) \sqsubseteq \left(\begin{array}{l} F(cs \longrightarrow ce \longrightarrow \mathbf{Skip}) \\ [\![\overline{wrtV(A)} \mid \{\!| cs, ce |\!\} \mid wrtV(A)]\!] \\ cs \longrightarrow A;\ ce \longrightarrow \mathbf{Skip} \end{array}\right) \setminus \{\!| cs, ce |\!\}$$

provided

- $\{\!| cs, ce |\!\} \cap usedC(A) = \varnothing$
- $wrtV(A) \cap usedV(F(\mathbf{Skip})) = \varnothing$
- $wrtV(F(\mathbf{Skip}) \cap usedV(A) = \varnothing$

This law is proved by structural induction over the structure of the action F using distribution and step laws such as the ones found in [14]. The provisos guarantee that the internal channels are fresh and that the state is appropriately partitioned to avoid racing conditions in the parallelism. We use $usedC(A)$ to refer to the set of channels used in an action A, and $usedV(A)$ and $wrtV(A)$ to refer to the variables used and modified by A. The law call-intro is applied to parallelise the safelet, mission sequencer and mission.

$$
\begin{array}{l}
\mathit{CF\_P} \mathrel{\widehat=} \mathbf{begin} \\
\mathbf{state}\ S \\
\mathit{PHandler}_i \mathrel{\widehat=} \mu X \bullet (F_i \blacktriangleright \mathit{PD} \interleave \mathbf{wait}\ \mathit{PERIOD});\ X \mathrel{\Box} t \longrightarrow \mathbf{Skip} \\
\mathit{AHandler}_j \mathrel{\widehat=} \mu X \bullet (c_j \longrightarrow G_j) \blacktriangleright \mathit{AD};\ X \mathrel{\Box} t \longrightarrow \mathbf{Skip} \\
\mathit{MArea} \mathrel{\widehat=} \ldots \\
\mathit{Termination} \mathrel{\widehat=} rt \longrightarrow \mu X \bullet (rt \longrightarrow X \mathrel{\Box} t \longrightarrow \mathbf{Skip}) \\
\mathit{CF\_Mission} \mathrel{\widehat=} \mathit{start\_mission} \longrightarrow \\
\quad \left(\begin{array}{l}
\mathit{MArea} \parallel \mathit{Termination} \parallel \\
\left(\begin{array}{l}
\parallel i : I \bullet \mathit{SH}_i \longrightarrow \mathit{register}.i \longrightarrow \mathit{start\_peh}.i \longrightarrow \mathit{activate\_handlers} \longrightarrow \\
\quad \mathit{done\_handler}.i \longrightarrow \mathbf{Skip} \\
\parallel \\
\parallel j : J \bullet \mathit{SH}_j \longrightarrow \mathit{register}.j \longrightarrow \mathit{start\_aeh}.j \longrightarrow \mathit{activate\_handlers} \longrightarrow \\
\ \mathit{quaddone\_handler}.j \longrightarrow \mathbf{Skip}
\end{array} \right)
\end{array} \right); \\
\quad \mathit{done\_mission} \longrightarrow \mathbf{Skip} \\
\mathit{Safelet} \mathrel{\widehat=} \\
\quad \left(\begin{array}{l}
\parallel i : I \bullet \mathit{SH}_i \longrightarrow \mathit{start\_peh}.i \longrightarrow \mathit{activate\_handlers} \longrightarrow \mathit{PHandler}_i; \\
\quad \mathit{done\_handler}.i \longrightarrow \mathbf{Skip} \\
\interleave j : J \bullet \mathit{SH}_j \longrightarrow \mathit{start\_aeh}.j \longrightarrow \mathit{activate\_handlers} \longrightarrow \\
\quad (\mathit{AHandler}_j \llbracket \{\ldots\} \mid \{\!| c_{ji} |\!\} \mid \{\} \rrbracket\ \mathit{Buffer}_j) \setminus \{\!| c_{ji} |\!\}; \\
\quad \mathit{done\_handler}.j \longrightarrow \mathbf{Skip} \\
\parallel \mathit{CF\_Mission} \\
\parallel \mathit{start\_sequencer} \longrightarrow \mathit{start\_mission} \longrightarrow \mathit{done\_mission} \longrightarrow \\
\quad \mathit{done\_sequencer} \longrightarrow \mathbf{Skip} \\
\parallel \mathit{start\_sequencer} \longrightarrow \mathit{done\_sequencer} \longrightarrow \mathbf{Skip}
\end{array} \right) \\
\bullet\ \mathit{Safelet} \\
\mathbf{end}
\end{array}
$$

Fig. 8. Refinement strategy (CF) – Target

1. Apply Law call-intro to the action *Safelet* with channels *cs* and *ce* replaced by *start_sequencer* and *done_sequencer*;
2. Apply Law call-intro to the action *MissionSequencer* with channels *cs* and *ce* replaced by *start_mission* and *done_mission*;
3. Apply Law copy-rule to the action *Mission* in *MissionSequencer*;
4. For each aperiodic action $\mathit{AHandler}_j$ in the parallelism of handler actions use associativity and commutativity laws to obtain a parallelism between $\mathit{AHandler}_j$ and another parallelism with all other handlers, and apply Law sync-async-conv;
5. Apply Laws prefix-par-dist [14] and par-prefix-dist to the action *Mission* to distribute the communications on *start_mission* and *done_mission* over all parallel actions;
6. Apply Law handler-extract to each parallel action except *MArea* and *Termination*;
7. Apply step laws of [14] to merge the left hand side actions of the parallelisms introduced in the previous step.

Fig. 9. Refinement strategy: (CF) Introduction of SCJ control flow

For the treatment of event handlers, we first introduce asynchronous communication between the event handlers using the Law sync-async-conv (see step 4 of Figure 9). This law, which we omit here, replaces the synchronous communication between two actions by an asynchronous communication based on a buffer.

$$
\begin{array}{l}
\mathit{Handler}_i\_\mathit{app} \mathrel{\widehat=} \ldots \\
\mathit{Mission\_app} \mathrel{\widehat=} \ldots \\
\mathit{MissionSequencer\_app} \mathrel{\widehat=} \ldots \\
\mathit{Safelet\_app} \mathrel{\widehat=} \ldots \\
\mathit{AP\_P} \mathrel{\widehat=} \mathit{AP\_P\_FW} \parallel \left(\begin{array}{l} \mathit{Safelet\_app} \interleave \mathit{MissionSequencer\_app} \interleave \mathit{Mission\_app} \parallel \\ (\interleave i : I \cup J \bullet \mathit{Handler}_i\_\mathit{app}) \end{array} \right)
\end{array}
$$

Fig. 10. Refinement strategy: target of phase **AP**

Next, parallelism distribution laws are used in step 5 to expand the parallelism between handlers (previously internal to the action *Mission*) to a top level parallelism as shown in Figure 8. In the next step, a simple law, handler-extract omitted here, is used wrap the handler actions with synchronisations on new internal channels *SH*, *register*, *start_peh*, *start_aeh*, *activate_handlers* and *done_handler* to represent the interactions corresponding to the initialisation of a mission, including creation (*SH*), registration (*register*), starting (*start_peh* and *start_aeh*) and activation (*activate_handlers*) of handlers, and to the termination of a mission, including the cleaning of handlers (*done_handler*). All these synchronisations are orchestrated by a new parallel action that models the mission execution cycle. Its repeated occurrence for each handler is eliminated in favour of a single parallel action named *CF_Mission* in Figure 8.

4.2 (AP) Introduction of application processes

The starting point of this phase is the target of the previous phase in Figure 8, and its target is shown in Figure 10: 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 the original process, where application-specific behaviours have been replaced by calls to actions of the application processes via *Call* and *Ret* channels that model method calls.

The steps of this phase are shown in Figure 11. Overall, we use the process obtained in phase CF to identify the behaviours that are application specific and construct application processes. Next, each action modelling an SCJ abstraction is split into two parallel actions: one containing application-specific behaviours, and the other containing the interactions introduced during CF to model the SCJ control flow. In this control action, the application-specific behaviour is replaced by calls via appropriate channels. This is achieved by specialised laws handler-split, mission-split, sequencer-split and safelet-split, for each of the different SCJ constructs. Finally the initial basic process is split into a parallelism of processes. Since, following the application of the specialised split laws, the main action of the basic process is a parallelism, this is a simple application of the definition of process parallelism in *Circus*.

Due to space restrictions, we present just the Law handler-split. The others are similar and simpler. For handlers, the new parallel actions communicate through channels that model a call to the *handleAsyncEvent* method. Accordingly, this

1. Apply Law handler-split to each handler action with channels *haeC* and *haeR* replaced by *handleAsyncEventCall* and *handleAsyncEventRet*;
2. Apply Law mission-split to the action that models the mission;
3. Apply Law seq-interleave [1] to turn the interleaving on the left hand side of the parallel action introduced in step 2 into a sequential composition;
4. Apply Law rec-interleave [1] to turn the interleaving on the right hand side of the parallel action in step 2 into a recursion;
5. Apply Law sequencer-split [1] to the action that models the sequencer;
6. Apply Law safelet-split [1] to the action that models the safelet;
7. Apply the definition of parallel processes [14] from right to left to replace the basic process, whose main action is parallel, with a parallelism of application processes and the remains of the original process.

Fig. 11. Refinement strategy: (AP) Introduction of application processes

law takes an action modelling a handler, and splits it by distributing application aspects such as constructor channels SH and release behaviour F to one side, and framework behaviours such as start and end channels (sh and dh) to the other side. This law is easily proved by the application of parallelism step laws [14].

Law [handler-split]

$$
\begin{array}{l}
SH.n \longrightarrow sh.n \longrightarrow \mu X \bullet (F;\ X);\ dh.n \longrightarrow \mathbf{Skip} \\
\sqsubseteq \\
\left(\begin{array}{l}
SH.n \longrightarrow sh.n \longrightarrow \mu X \bullet (haeC \longrightarrow F;\ haeR \longrightarrow X);\ dh.n \longrightarrow \mathbf{Skip} \\
[\![wrtV(F) \mid \{\!| sh.n, dh.n, haeC, haeR |\!\} \mid \{\}]\!] \\
sh.n \longrightarrow \mu X \bullet (haeC \longrightarrow haeR \longrightarrow X);\ dh.n \longrightarrow \mathbf{Skip}
\end{array}\right) \\
\setminus \{\!| haeC, haeR |\!\}
\end{array}
$$

provided $\{\!| sh, SH, dh |\!\} \cap usedC(F) = \varnothing$.

In step 3, the strategy applies a law to transform the interleaving of the instantiation and registration of all handlers (on the application side) into a sequence. This is possible because all the parallel actions that synchronise on the interleaved events do so in interleaving (avoiding deadlock) and these events are internal. Step 4 transforms the interleaving on the framework side of the mission into a recursive action that at each step allows the registration of a handler, and once all handlers have been registered, executes them in interleaving.

At the end of this phase, the application processes are completed, but the remaining process AP_P_FW does not quite specify the SCJ runtime environment. This process is the focus of the next phase.

4.3 (FW) Introduction of framework processes

This phase applies to the part of the model that remains after the application processes are extracted. It consists of a process AP_P_FW containing portions

1. Apply Law safelet-fw-cl to the action *Safelet* of CF_P_FW;
2. Apply Law sequencer-fw-cl to the action *Sequencer* of CF_P_FW;
3. Apply Law mission-fw-cl to the action *Mission* of CF_P_FW;
4. Apply Law periodic-handler-fw-cl to the actions $PHandler_i$ of CF_P_FW;
5. Apply Law aperiodic-handler-fw-cl to the action $AHandler_j$ of CF_P_FW;
6. Apply the definition of parallel processes [14] from right to left to replace the process CF_P_FW, with a parallelism of framework processes.

Fig. 12. Refinement strategy: (FW) Introduction of framework processes

$$FW\_P \mathrel{\widehat=} \left(\begin{array}{l} \left(\begin{array}{l} SafeletFW \parallel SequencerFW \parallel MissionFW(mission) \parallel \\ (\interleave i : I \cup J \bullet HandlerFW(handler_i)) \end{array}\right) \\ \parallel \\ \left(\begin{array}{l} Safelet\_app \interleave MissionSequencer\_app \interleave Mission\_app \parallel \\ (\interleave i : I \cup J \bullet Handler_i\_app) \end{array}\right) \end{array}\right)$$

Fig. 13. Refinement strategy: target of phase **FW**

of the framework that are explicitly used in the design. The result is a new process that defines the complete framework behaviours. For instance, our running example never asks a mission for the sequencer that oversees its execution. This is, however, a service provided by the framework. We can introduce the richer description of the framework because the application process is guaranteed not to request the additional behaviour. The steps of this phase are shown in Figure 12.

The process resulting from the application of this phase is shown in Figure 13. It is the parallel composition of the application processes introduced in the previous phase and the framework processes that model the SCJ API.

The main laws used in this phase are specialised to the cyclic in lockstep pattern as indicated by the suffix -cl. The single non-application process AP_P_FW obtained in the previous phase is the same for all applications that follow our target pattern. This is because the pattern is very restrictive with respect to the execution of missions and handlers, and most of the framework specific behaviours are introduced by the laws in the previous steps. For this reason, the specialised laws can be used to introduce the full blown framework processes relying solely on syntactic conditions over the application processes. This is done to each abstraction in steps 1–5. These framework processes are specified in [10].

At the final step 6, the process whose main action is the parallel composition of the actions completed by the previous steps is split into a parallelism of framework processes. The result is a parallelism of processes as shown in Figure 13.

4.4 (Conv) Conversion to ***SCJ-Circus***

This phase rearranges the parallel processes shown in Figure 13 by pairing framework and application processes according to the abstraction they model, and introducing new process paragraphs that isolate these pairs. For instance,

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

Fig. 14. Refinement strategy: target of phase **Conv**

1. Systematically apply Law par-par-dist to rearrange the parallelism in Figure 13, until it is structured as a parallelism of pairs of processes;
2. For each pair of processes, apply the copy rule from right to left to introduce the corresponding action paragraph and replace the process by a call;
3. For each newly introduced action, apply the definition of the appropriate SCJ abstraction from right to left.

Fig. 15. Refinement strategy: (Conv) Conversion to *SCJ-Circus*

SafeletFW is paired with *Safelet_app*, and extracted into a process *Safelet*. Next, the semantics of *SCJ-Circus* is used to convert the newly introduced processes into the corresponding *SCJ-Circus* paragraphs. For example, the process *Safelet* is converted into a paragraph identified by the keyword **safelet**.

The target of this phase is a specification formed by *SCJ-Circus* paragraphs as shown in Figure 14. Each action that models an SCJ abstraction in the original design is modelled by an *SCJ-Circus* paragraph. These paragraphs overtly specify only the application specific behaviours, leaving the framework aspects implicit.

The steps for this phase are shown in Figure 15. The first step extracts pairs of application and framework processes two by two using the Law par-par-dist below. This is carried out exhaustively until there are no more pairs to extract.

Law [par-par-dist]

$$(A \llbracket s_A \mid a_1 \mid s_B \rrbracket B) \llbracket s_A \cup s_B \mid a_2 \cup b \mid s_C \cup s_D \rrbracket (C \llbracket s_C \mid c \mid s_D \rrbracket D)$$
$$=$$
$$(A \llbracket s_A \mid a_2 \mid s_C \rrbracket C) \llbracket s_A \cup s_C \mid a_1 \cup c \mid s_B \cup s_D \rrbracket (B \llbracket s_B \mid b \mid s_D \rrbracket D)$$

provided

- $usedC(A) \cap usedC(B) \subseteq a_1 \land a_1 \cap usedC(C, D) = \varnothing$
- $usedC(C) \cap usedC(D) \subseteq c \land c \cap usedC(A, B) = \varnothing$
- $usedC(A) \cap usedC(C) \subseteq a_2 \land a_2 \cap usedC(B, D) = \varnothing$
- $usedC(B) \cap usedC(D) \subseteq b \land b \cap usedC(A, C) = \varnothing$

This law relies on the strict partition of the communication network between the four parallel processes. It uses the fact that the channels used by the processes C and D, which are matched to application processes in our strategy, to communicate with each other are not used by A and B, and, conversely, that the channels used by A and B (matched to the framework processes in our strategy) to communicate with each other are not used by the application processes.

Next, each pair of application and framework processes is used to define a new process using the reverse of the copy-rule, and the semantics of *SCJ-Circus* is used to transform these newly defined processes into *SCJ-Circus* paragraphs.

5 Non-terminating pattern

Figure 16 shows the *Mission* action of the non-terminating cyclic in lockstep pattern. The main difference from that in Figure 3 is the missing Termination action. The target of our refinement strategy is the same: an *SCJ-Circus* program in the form described in Figure 7.

$$Mission \mathrel{\widehat=} (MArea \parallel (\parallel i : I \bullet PHandler_i) \parallel (\parallel j : Jn \bullet AHandler_j)$$

Fig. 16. Non-terminating cyclic in lockstep pattern

The refinement strategy described in Section 4 cannot be applied to models that follow the pattern in Figure 16 because it expects the *Mission* action to have an extra parallelism: see step 6 of CF and step 3 of FW. Instead of modifying the mission specific laws to introduce the mechanisms of termination, it is possible to extend the refinement strategy in the Section 4 by introducing this parallel action as a first step using the Law termination-intro, omitted here, before applying it.

This law takes a *Circus* action A of the form $\mu X \bullet F;\ X$ and two channels t and rt, and refines A into a parallelism between $\mu X \bullet F;\ X \Box t \longrightarrow \mathbf{Skip}$, and an action that waits for a termination request on a channel rt and then behaves as a recursive action that either accepts an event on the channel rt and recurses, or accepts an event on the channel t and terminates. The parallelism synchronises on both t and rt, which are made internal via the hiding operator. This law relies on the fact that A does not terminate, and does not use rt or t.

It may seem inefficient to complicate the model, but we note that the refinement steps of the whole refinement strategy are mostly automatic. Moreover, the phase FW is already about completing the framework model to reflect the SCJ paradigm. The termination protocol is part of the framework model already.

6 Multiple terminating missions

For an application with multiple missions in sequence, the pattern only differs in the specification of the action *MissionSequencer*, which is defined as the sequential composition of a number of missions $M_1;\ M_2;\ \ldots;\ M_n$. In this case, it is possible to modify the existing refinement strategy at very specific points to cater for a sequence of missions.

Step 2 of CF needs to be replaced with an iteration that, for each mission M_i, applies the Law call-intro to *MissionSequencer* with A instantiated as M_i and the channels cs and ce replaced by $start\_mission.M_i\_ID$ and $done\_mission.M_i\_ID$. With that, the mission-sequencer action is refined to a sequence of pairs of synchronisations on $start\_mission$ and $done\_mission$, in parallel with actions that call the mission actions. We have one parallel action for each mission, with the call wrapped by the $start\_mission$ and $done\_mission$ events. This is similar to the result obtained for the first pattern: the only difference is that, in this case, we have several parallel calls to missions.

Next, for each mission, the modified strategy applies the remaining steps described originally, including those of the following phases. We have to take into account, however, that the steps 5 and 6 of AP and 1 and 2 of FW only need to be applied in the first iteration. These steps are related to the application and framework processes for the safelet and the mission sequencer, and need to be carried out just once. Moreover, step 5 of AP needs a slightly different refinement law, which introduces a particular pattern for the implementation of the `getNextMission` tailored for multiple missions in sequence.

7 Conclusions

In this paper, we detail a refinement strategy for SCJ specifications. We describe each step necessary, and present some of the specialised laws required. This strategy differs from the refinement strategy for SCJ in [5] in that the latter takes an abstract model and refines it into a concrete program using specification patterns based on SCJ, but not its API. The strategy we present here refines a concrete SCJ-based model into a program that makes full use of the standard SCJ library to implement control aspects that are specific to SCJ. In that sense, our refinement strategy is similar to compilation, except that the target *SCJ-Circus* programs include library calls that are not present in the starting model. Moreover, some of the applications of refinement laws in the strategy generate proof obligations. Theorem proving is required when applying the strategy.

Despite that, since the *Circus* models used as a starting point for our strategy already embed an SCJ design, the level of automation achievable in applying the strategy is much higher than in [5]. For the particular pattern that we target here, most of the laws used have only syntactic provisos.

Our refinement strategy, possibly in combination with the one in [5], complements other verification techniques for SCJ. The work in [8] proposes an annotation-free technique for the verification of memory safety of SCJ programs based on a translation to a notation similar to *SCJ-Circus*. Also, [6] extends the widely used Java Modelling Language [2] with timing annotations to support worst case execution time analysis of SCJ programs.

It is worth mentioning that the pattern on which we focus here is fairly common in safety critical systems. For instance, the refinement strategy for control law diagrams proposed in [4] targets Ada implementations that follow a similar pattern, and it may be possible to adapt both refinement strategies to support the verification of SCJ implementations of control law diagrams.

As future work, we plan to refactor our strategy by extracting a refinement strategy that targets missions. With this structure, our strategy can be more easily generalised. We plan to specify strategies that target common patterns of mission combination, as well as missions following different patterns. Finally, we plan to implement our strategies in a theorem prover such as Isabelle/HOL, and apply them to existing examples such as a collision detection system [19].

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

References

1. A. Miyazawa and A. Cavalcanti. Report on refinement strategies for Safety-Critical Java, 2015. `http://www-users.cs.york.ac.uk/~alvarohm/report2015b.pdf`.
2. L. Burdy et al. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, June 2005.
3. A. Cavalcanti, A. Sampaio, and J. Woodcock. Unifying classes and processes. *Software & Systems Modeling*, 4(3):277–296, 2005.
4. A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via *Circus*. *Formal Aspects of Computing*, 23(4):465 – 512, 2011.
5. A. L. C. Cavalcanti et al. Safety-Critical Java programs from *Circus* models. *Real-Time Systems*, 49(5):614–667, 2013.
6. G. Haddad et al. The design of SafeJML, a specification language for SCJ with support for WCET specification. In JTRES '10, pages 155–163. ACM, 2010.
7. D. Locke et al. Safety-Critical Java technology specification. Technical report.
8. C. Marriott and A. L. C. Cavalcanti. SCJ: Memory-safety checking without annotations. In *FM*, volume 8442 of *LNCS*, pages 465–480. Springer, 2014.
9. A. Miyazawa and A. Cavalcanti. Formal refinement in SysML. In *iFM 2014*, volume 8739 of *LNCS*, pages 155–170. Springer, 2014.
10. A. Miyazawa and A. Cavalcanti. Refinement of *Circus* models into *SCJ-Circus*, 2015. `http://www-users.cs.york.ac.uk/~alvarohm/report2015a.pdf`.
11. A. Miyazawa and A. L. C. Cavalcanti. Refinement-oriented models of Stateflow charts. *Science of Computer Programming*, 77(10-11):1151–1177, 2012.
12. A. Miyazawa and A. L. C. Cavalcanti. *SCJ-Circus*: a refinement-oriented formal notation for Safety-Critical Java. In *Refinement Workshop*, 2015.
13. A. Miyazawa, L. Lima, and A. Cavalcanti. Formal models of SysML blocks. In *ICFEM 2013*, volume 8144 of *LNCS*, pages 249–264. Springer, 2013.
14. M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs Using Circus*. PhD thesis, University of York, 2006.
15. A. W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2011.
16. K. Wei, J. C. P. Woodcock, and A. L. C. Cavalcanti. *Circus Time* with Reactive Designs. In *UTP 2012*, volume 7681 of *LNCS*, pages 68–87. Springer, 2012.
17. A. Wellings. *Concurrent and Real-Time Programming in Java.* John Wiley & Sons, 2004.
18. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.
19. F. Zeyda, A. Cavalcanti, A. Wellings, J. Woodcock, and K. Wei. Refinement of the Parallel CDx. Technical report, University of York, 2012.