SCJ-Circus: Report on specification and refinement of Safety-Critical Java programs

Alvaro Miyazawa¹, Ana Cavalcanti², and Andy Wellings³

University of York
¹alvaro.miyazawa@york.ac.uk
²ana.cavalcanti@york.ac.uk
³andy.wellings@york.ac.uk

August 12, 2016
Safety-Critical Java (SCJ) is a version of Java for real-time, embedded, safety-critical applications. It supports certification via abstractions that enforce a particular program architecture, with controlled concurrency and memory models. SCJ is an Open Group standard, with a reference implementation, but little support for reasoning. Here, we present SCJ-Circus, a refinement notation for specification and verification of low-level models of SCJ programs. SCJ-Circus is part of the Circus family of state-rich process algebras. As such, SCJ-Circus includes the Circus constructs for modelling of sequential and concurrent behaviour based on Z and CSP, and the real-time and object-oriented extensions of Circus, besides the SCJ abstractions. We present the syntax of SCJ-Circus and its semantics, defined by mapping SCJ-Circus constructs to those of Circus. We also detail a refinement strategy that takes a Circus design that adheres to a multi-processor cyclic executive pattern and produces an SCJ design, described in SCJ-Circus. Finally, we show how this refinement strategy can be extended for more complex program architectures.

**Keywords:** SCJ, missions, event handlers, process algebra, semantics
Chapter 1

Introduction

Although a very popular language, Java is not suitable for programming real-time safety-critical applications [1]. Recently, however, the Open Group has defined Safety-Critical Java (SCJ) [2], a version of Java suitable for developing verifiable real-time software. It incorporates part of the Real-Time Specification for Java (RTSJ) [1], introduces new abstractions (such as, safelets and missions), and removes garbage collection in favour of scoped memory regions. All this supports predictable timing behaviours and constrained program architectures that facilitate static verification.

SCJ programs can adopt one of three profiles, called levels, which include an increasing number of abstractions. We focus on the intermediate level 1; it is comparable in complexity to the Ravenscar profile for Ada. While adequate for a wide range of applications, level 1 programs are amenable to automated formal reasoning. It enforces the programming model shown in Figure 1.1 with a simplified concurrency model.

An SCJ level 1 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. Missions are iteratively obtained from the mission sequencer, as each of them executes and terminates. Each mission is formed by a collection of periodic and aperiodic event handlers that run concurrently. A mission terminates when one of its handlers requests termination, and a safelet terminates when there are no missions left to be executed.

The SCJ memory model is based on scoped memory regions. For example, missions and handlers have associated memory regions, cleared at specific points of the program execution: end of a mission execution, and end of each release of a handler. In contrast with garbage collection, this ensures predictable execution times.

The standardisation effort includes an SCJ specification (JSR 302), a reference implementation (RI), and a Technology Compatibility Kit (TCK). The goal of the RI is to demonstrate the feasibility of implementing the proposed JSR. The TCK is a suite of test programs that check that an implementation conforms to a JSR. There is, however, no support available for design and static verification of SCJ programs.
Cavalcanti et al. \cite{3} proposes a formal design technique for SCJ based on the Circus family of languages: state-rich process algebras for refinement that combine Z \cite{4} and CSP \cite{5}. Circus has been used to verify models written in a number of different notations, such as, Simulink and Stateflow diagrams \cite{6,7}, and SysML \cite{8,9}. The semantics of Circus is based on Hoare and He’s Unifying Theories of Programming \cite{10}, which is a semantic framework that supports the formalisation of a variety of paradigms in an independent fashion, and their subsequent combination through specialised techniques. Since refinement is a central concept in the UTP, it is also an important aspect of Circus as evidenced by its rich refinement calculus \cite{11}. Circus has been extended to support a number of different programming paradigms. For example, OhCircus \cite{12} supports the specification of object-oriented designs and programs, and Circus Time \cite{13} supports modelling real-time programs; they are both useful in our work.

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 memory model adopted in Safety-Critical Java.

Abstraction can be achieved in SCJ-Circus models using the constructs of Circus for data and behavioural modelling. On the other hand, 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.

Cavalcanti et al. \cite{3} propose a refinement strategy 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 A anchor defines an abstract model and the last 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 \cite{3} 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.
Here, we extend [3] first by exploring the use of SCJ-Circus to define the S anchor; we specify the syntax and semantics of SCJ-Circus. Next, we identify patterns of E anchors that we use as a basis for the development of S anchors. They define designs with specific interaction and timing properties, namely, time-triggered missions with precedence constraints. We present a core refinement strategy for E anchors designed according to a pattern for terminating systems without optional components. We then show how that strategy can be extended and composed to support refinement of a wider variety of patterns, in particular, non-terminating and multi-mission designs with optional components. The refinement strategies derive S anchors that can be automatically translated into SCJ.

To define the semantics of SCJ-Circus, we build on a Circus semantics of SCJ programs defined in [14]. We update that semantics to reflect fundamental changes to the mode of interaction between handlers and the mission-termination protocol recently accepted by the SCJ standardisation group. We also propose a different structure for the Circus models to enable compositional refinement with respect to the SCJ-Circus constructs.

This report combines and extends the works presented in [15, 16] in that it presents laws required for the core refinement strategy, and details the steps of the extensions for the non-terminating and multi-mission designs. We also further extend the multi-mission strategy to cover patterns where the sequence of missions to be executed are not statically defined; we cater for conditional and nondeterministic choice of missions.

In Chapter 2 we introduce Safety-Critical Java and the Circus family of languages. In Chapter 3 we present SCJ-Circus, its syntax and semantics. Chapter 4 presents our patterns of design, and Chapter 5 details the core refinement strategy for the terminating pattern, first without and then with optional components. Sections 5.5 and 5.6 extend that strategy to cover non-terminating and multi-mission designs. Chapter 6 concludes by reviewing our results, relating our work to those available in the literature, and discussing future work. Finally, Appendix A summarises the novel refinement laws required by the refinement strategy, Appendix B and C present the syntax and semantics of SCJ-Circus in detail and Appendix D presents examples of specifications in SCJ-Circus.
Preliminaries

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

2.1 Safety-Critical Java

As previously mentioned, an SCJ application is formed by a safelet, a mission sequencer, a number of missions, and their 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. As also explained above, SCJ replaces the need for garbage collection by supporting the use of a memory model based on scoped memory areas. The SCJ model simplifies that of the RTSJ, supports better timing analysis, and can be more easily complemented by static-analysis techniques that guarantee absence of dangling pointers [17][18].

The components of the SCJ programming model are associated with specific memory areas as default allocation contexts. The safelet and the sequencer are associated with an immortal area; any objects created by the safelet are allocated there. Objects in the immortal memory are never cleared. There is also a mission memory area, which is cleared at the termination of each mission. In the case of a handler, we have a per-release area; it is cleared after each release. Finally, a handler can create private memory areas. A discipline of allocation ensures safety, that is, absence of dangling pointers. For example, an object in the mission memory area can refer to other objects in the mission and immortal memory areas, but not in the per-release areas, since they are cleared before the end of the mission.

Our running example is a simple application: a communication medium that flags when there is a change in the value communicated. It has a single mission containing two handlers: one periodic event handler and one aperiodic event handler. The periodic event handler reads a message at every cycle, stores it in a buffer, and releases the aperiodic event handler. Upon release, the aperiodic event handler examines the last two elements of the buffer and outputs “true” or “false”, depending on whether they are the same or not.
Figure 2.1: Our running example: communication medium

Figure 2.1 shows a class diagram that defines the structure of the SCJ program that implements the communication medium. The interface and abstract classes enclosed in rounded boxes are part of the SCJ API. The application classes extend or implement them. The safelet class implements a method `getSequencer` that returns the mission sequencer to be used in the application. In our example, the safelet is an instance of `ChkSafelet` and its `getSequencer` method returns an instance of `ChkSequencer`. A sequencer implements a method `getNextMission` used iteratively by the SCJ infrastructure to obtain a mission. In our example, it returns an instance of `ChkMission` in the first time it is called, and `null` in the second time, when then the program would terminate. In fact, in our example, since the mission implemented by `ChkMission` does not terminate, `getNextMission` is called just once.

The execution of a mission consists of the parallel execution of all its handlers. Most of the actual behaviour of the application is concentrated in the handlers. In our example, they are instances of `Reader` and `Checker`. Figure 2.2 shows the code for `Checker`, the aperiodic handler. It extends the SCJ API class `AperiodicEventHandler`, and declares a local variable `buffer`, a constructor that receives an instance of a class `Buffer` (not shown in Figure 2.1) and assigns it to `buffer`, and a `handleAsyncEvent` method that defines the behaviour of the handler releases.

The constructor of `Checker` calls the constructor of the superclass with priority 98, a new instance of an object that defines aperiodic parameters (deadline and deadline-miss handler), and storage parameters that specify the amount of memory used by the handler. The method `handleAsyncEvent` checks whether the last two 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; in practice, it is usually sent to another component of the system.
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 2.2: SCJ Level 1 example: Aperiodic Event Handler

The complete program can be found in www.cs.york.ac.uk/circus/hijac/code/checker.zip

2.2 Circus

In this section, to describe Circus and its timed variant, we use as a running example the Circus Time process PEHFW in Figure 2.3. It models the general behaviour of a periodic event handler.

The main modelling elements of a Circus specification are processes (indicated using the keyword process). In general, a Circus (Circus Time or ObCircus) 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: they have a state that is encapsulated and interact via channels.

A process definition declares state components (indicated by the keyword state), auxiliary actions, and a main action (at the end, prefixed by •) that describes the behaviour of the process. In Figure 2.3, the process PEHFW is parametrised by an identifier id of a given type IDENTIFIER for a periodic handler, and declares two state components, start and period, both of type \( \mathbb{N} \), defining the start time and period of the handler. PEHFW declares only one auxiliary action, namely, Execute. Actions are specified using a combination of Z [4] and guarded commands [19] for data modelling, CSP [5] for behavioural descriptions, and time constructs.

The main action of PEHFW is a recursion (\( \mu X \bullet \ldots \)) whose iterations start an instance of the handler via a communication through the channel start_peh. In this communication, the handler identifier id is output (!), and its start time s and period p are input. The values of s and p are then assigned to the state components start and period.
process $PEFW \equiv id : IDENTIFIER \bullet \begin{align*} \text{begin} 
\text{state } PEFW\text{State} &\equiv [\text{start, period} : \mathbb{N}] 
\text{Execute} &\equiv \text{wait start}; 
\left(\mu X \bullet \begin{align*}
&\left(\text{handleAsyncEventCall}!id \rightarrow \text{handleAsyncEventRet}!id \rightarrow \text{Skip} \triangleright \text{period}; X\right)\land
&\left(\text{start_peh}!id?s?p \rightarrow \text{activate Handlers} \rightarrow \text{start} := s; \text{period} := p; \text{Execute}; X\right)\land
&\left(\text{terminate} \rightarrow \text{Skip}\right)\land
&\left(\text{done_handler}!id \rightarrow \text{Skip}\right)\land
&\left(\text{start_peh}!id \rightarrow \text{wait period} \triangleright 0\right)\land
&\left(Y \bullet (\text{handleAsyncEventCall}!id \rightarrow \text{wait period}) \triangleright 0\right)\land
&\left(Y \bullet \text{done_handler}!id \rightarrow \text{Skip}\right)\land
&\left(\mu Y \bullet (\text{handleAsyncEventCall}!id \rightarrow \text{wait period}) \triangleright 0\right)\land
&\left(\text{start_peh}!id?s?p \rightarrow \text{activate Handlers} \rightarrow \text{start} := s; \text{period} := p; \text{Execute}; X\right)\land
&\left(\text{terminate} \rightarrow \text{Skip}\right)\land
&\left(\text{done_handler}!id \rightarrow \text{Skip}\right)
\right)
\end{align*}
\right)
\end{align*}
\text{end}
\end{align*}$

Figure 2.3: Framework process of the periodic event handler.

The execution of a newly created handler is defined by the auxiliary action \textit{Execute}. It first waits for \textit{start} time units (\textit{wait start}), and then starts two recursive actions in parallel ($A_1 [n_1 | c | n_2] A_2$) synchronising on the channels \textit{handleAsyncEventCall} and \textit{done_handler}. The first parallel action specifies that at each step of the recursion there is an external choice (□) for communication on the channels \textit{handleAsyncEventCall} or \textit{done_handler}. In this way, the method \textit{handleAsyncEvent} can be called through the channel \textit{handleAsyncEventCall} or the handler can be terminated with a choice of \textit{done_handler}. If \textit{handleAsyncEvent} is called, then it must return, as indicated with a communication via \textit{handleAsyncEventRet}, within \textit{period} time units. This is specified using a \textit{Circus Time} deadline construct $A \triangleright e$ that defines that the action $A$ must terminate within $e$ time units.

The second parallel action in \textit{Execute} adds a requirement that a call to \textit{handleAsyncEvent} must be started as soon as it is available, and should be made available again only after \textit{period} time units. This is achieved first by imposing a restriction on the communication \textit{handleAsyncEventCall} using the start-by operator ($A \triangleright e$) that specifies that an action $A$ must start within a certain number of time units. In the example, it must start immediately, after 0 time units. Additionally, the action \textit{wait period} after the communication on \textit{handleAsyncEventCall} ensures that \textit{handleAsyncEventCall} is only offered after \textit{period} time units. Since the first recursion can be terminated by a synchronisation on the channel \textit{done_handler}, the second recursion must also be terminated. This is achieved by allowing the interruption of the recursion by a synchronisation on \textit{done_handler} using the interrupt operator (Δ).

Processes can be composed, via CSP operators, to define other processes. In \textit{Circus Time}, as illustrated above, wait and deadline operators can define time restrictions. In \textit{ObCircus} models, we can in addition define paragraphs that declare classes used to define types. More information about these languages can be found in [11,13,12].
In the sequel, we further explain the notation as needed, and next describe our new *Circus* variant.
As said before, SCJ-Circus extends ObCircus and Circus Time with abstractions specific to SCJ. Below, Section 3.1 discusses the syntax of SCJ-Circus, Section 3.2 presents the Circus model of the SCJ framework (its API and programming model), and Section 3.3 describes the semantics of SCJ-Circus based on the model of Section 3.2.

3.1 Syntax

Figure 3.1 sketches the specification of the SCJ-Circus syntax. SCJ-Circus extends the syntax of ObCircus and Circus Time with paragraphs that allow the specification of safelets, mission sequencers, missions and handlers.

Figure 3.2 presents the SCJ-Circus specification of our example. It matches the structure of the code, but also specifies timing requirements. Reader reads an input every $P$ time units, with an input deadline of $ID$ time units. Each cycle of Reader takes between 0 and $PTB$ time units, and must terminate within $PD$ time units. Checker 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 periods and deadlines can be respected. These conditions require that the sum of the 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. These constraints are specified in the SCJ-Circus model as part of the declaration of these constants.

In general, as defined 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 fields and methods that must be specified for an application.
SCJProgram ::= SCJParagraph*
SCJParagraph ::= Safelet | MissionSequencer | Mission | Handler | CircusParagraph
Safelet ::= safelet N ⊑ begin
    SCJSafeletProcessParagraph*
    state Schema-Expression
    SCJSafeletProcessParagraph*
    initialize ⊑ SCJSafeletAction
    SCJSafeletProcessParagraph*
    getSequencer ⊑ res return : sequencer • SCJSafeletAction
    SCJSafeletProcessParagraph*
end
PeriodicHandler ::= periodic handler handler N ⊑ begin
    SCJHandlerProcessParagraph*
    start Expression period Expression
    SCJHandlerProcessParagraph*
    state Schema-Expression
    SCJHandlerProcessParagraph*
    initial ⊑ SCJHandlerAction
    SCJHandlerProcessParagraph*
    handleAsyncEvent ⊑ SCJHandlerAction
    SCJHandlerProcessParagraph*
end
SCJHandlerProcessParagraph ::= Paragraph
    | N ⊑ SCJHandlerParametrisedAction
    | nameset N == NameSetExpression
SCJHandlerParametrisedAction ::= SCJHandlerAction
    | val Declaration • SCJHandlerParametrisedAction
    | res Declaration • SCJHandlerParametrisedAction
SCJHandlerAction ::= SCJMissionAction
    | N := newPR N
    | N := newPR N(Expression+)
    | N := newPM N
    | N := newPM N(Expression+)
...

Figure 3.1: Syntax of SCJ-Circus (sketch)

For instance, a safelet must implement the initialize method, which allows, for instance, the allocation of global objects in the immortal memory area, and the getSequencer method. Accordingly, the SCJ-Circus paragraph corresponding to a safelet introduces a name taken from the set of valid Circus names N, and allows the specification of state components (state), an initialisation method (initial), auxiliary actions (SCJSafeletProcessParagraph) and the getSequencer method. The state components model the fields of the safelet class, and the initial method, its constructor. The auxiliary actions define any extra methods implemented in a safelet class. An SCJSafeletProcessParagraph allows the specification of an action whose body is an SCJSafeletAction. This restricts the constructs
safelet $ChkSafelet \equiv \begin{aligned} & \text{getSequencer} \equiv \text{res return} \cdot \text{return} \equiv \text{new ChkMissionSequencer()} \\ \text{end} \end{aligned}$

sequencer $ChkMissionSequencer \equiv \begin{aligned} & \text{state} \left[ \text{done} : \mathbb{B} \right] \\ & \text{initial} \equiv \text{done} \equiv \text{false} \\ & \text{getNextMission} \equiv \text{res return} : \text{IDENTIFIER} \cdot \left( \begin{array}{ll} & \text{if} \ 
eg \text{done} \rightarrow \text{done} \equiv \text{true}; \ 	ext{return} \equiv \text{new ChkMission()} \\ & \text{fi} \end{array} \right) \\ \text{end} \end{aligned}$

mission $ChkMission \equiv \begin{aligned} & \text{state} \left[ \text{buffer} : \text{seq}^N \right] \\ & \text{initial} \equiv \text{buffer} \equiv \langle 0, 0 \rangle \\ & \text{initialize} \equiv \text{var ab, ph : IDENTIFIER} \cdot \text{ab} \equiv \text{newHandler Checker}; \text{ph} \equiv \text{newHandler Reader(ab)} \\ & \text{cleanup} \equiv \text{res return} : \mathbb{B} \cdot \text{return} \equiv \text{true} \\ \text{end} \end{aligned}$

periodic handler Reader $\equiv \begin{aligned} & \text{state} \left[ \text{ab} : \text{IDENTIFIER} \right] \\ & \text{initial} \equiv \text{ab} : \text{IDENTIFIER} \cdot \text{this.ab} \equiv \text{ab} \\ & \text{handleAsyncEvent} \equiv \left( \begin{array}{ll} & (\text{input}?x \rightarrow \text{Skip}) \triangleleft \text{ID}; \text{getBuffer}?\text{buffer} \rightarrow \text{setBuffer}(\text{tail buffer} \land \langle x \rangle) \rightarrow \text{release}(ab); \\ & (\text{wait} 0..\text{PTB}) \triangleright \text{PD} \end{array} \right) \end{aligned}$

aperiodic handler Checker $\equiv \begin{aligned} & \text{checker} : \text{seq}^N \rightarrow \mathbb{B} \\ & \forall b : \text{seq}^N \mid \# b = 2 \cdot \left( \text{if} \ b_1 = b_2 \ \text{then} \ \text{checker}(b) = \text{true} \ \text{else} \ \text{checker}(b) = \text{false} \right) \\ & \text{handleAsyncEvent} \equiv \left( \begin{array}{ll} & \text{getBuffer}?\text{buffer}\rightarrow \left( \begin{array}{ll} & \text{if} \ \text{checker}(\text{buffer}) = \text{true} \rightarrow (\text{output}?\text{true} \rightarrow \text{Skip}) \triangleleft \text{OD} \\ & \text{fi} \\ & \text{[check(buffer) = false \rightarrow (output}?\text{false} \rightarrow \text{Skip}) \triangleleft \text{OD} \end{array} \right) ; \text{wait} 0..\text{ATB} \end{array} \right) \triangleright \text{AD} \\ \text{end} \end{aligned}$

Figure 3.2: SCJ Level 1 example: S-anchor

that can be used in the definition of an action of a safelet, in particular, the type of allocation constructs as discussed next.

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

These restrictions are enforced by the use of different syntactic categories for the actions and paragraphs of the different constructs. For example, the body of the getSequencer method of a safelet must be an SCJSafeletAction and the handleAsyncEvent method of a handler must be an SCJHandlerAction. The first only allows instantiation via new, 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, we refer to [20].

3.2 Semantic model

In [14], 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 to give semantics to SCJ Circus. Our Circus specifications, however, are updated to consider recent significant changes to SCJ and to cater for compositional reasoning about SCJ constructs. Figure 3.3 depicts the structure of our semantic models.

For a given program, each component of the SCJ paradigm (safelet, sequencer, and so on) is modelled by a Circus Time process. Such process is defined as the parallel composition of two processes: a general framework process that captures the behaviour of the SCJ component as an element of the SCJ programming model, and a process that captures the behaviour of the component defined in the particular application. For example, the process PEHFW in Figure 2.3 is 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 channels that represent method calls. For example, the channels safeletInitializeCall, safeletInitializeRet, getSequencerCall and getSequencerRet in Figure 3.3 are used by the safelet framework process SafeletFW to communicate with the application-specific process S_App. They are used to model calls to the methods initialize and getSequencer of the application.
In the models of SCJ programs presented in [14], 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. A model that adopts our structure, where application and framework processes are composed on a per-element basis, is equivalent to the model structured as in [14].

The framework process SafeletFW that specifies the generic behaviour of a safelet is shown in Figure 3.4; it is parametrised by the identifier of the safelet. This process, first of all, requests to the application process the initialisation of the safelet using the channels safeletInitializeCall and safeletInitializeRet. In each communication corresponding to a method call or return, the identifier of the caller and callee are included, along with the arguments, if any, in the call and with the return value, if any, in the return communication. Here, both the caller and the callee have the same identifier id used for both the framework and application processes of the safelet.

After the call to method initialize, SafeletFW obtains a mission sequencer via the channels getSequenceCall and getSequenceRet. If the sequencer is different from null, starts it (using the channel start_sequence); otherwise, it terminates. If the sequencer process is started, the safelet framework process waits for the completion of its execution, signalled via the channel done_sequence. At that point, the safelet indicates to the application process that it is terminating through the channel end_safelet_app, and finally terminates (Skip).
The complete definition of the model can be found in [20].

### 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 [11]. In order to improve readability, the semantics is presented in terms of translation rules that describe 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 as indicated in Figure 3.3.

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 \([\_]_{SCJProgram}\). It takes a well formed SCJ-Circus program \( p \) and outputs a Circus program composed of the sequence of paragraphs produced by the semantic functions \([\_]_{SCJParagraphs}\) and \([\_]_{Application}\). The first takes the sequence of SCJ-Circus paragraphs of \( p \) (that is, \( p.paragraphs \)) and outputs a sequence of Circus paragraphs. The second takes \( p \) and outputs a single paragraph that defines a parallel process that composes the processes defined by \( [p.paragraphs]_{SCJParagraphs} \) to specify the overall meaning of the SCJ-Circus program \( p \). These paragraphs use the definitions of the processes that model the SCJ framework, like PEHFW and SafeletFW.

\[
[\_]_{SCJProgram} : SCJProgram \rightarrow CircusProgram \quad \forall p : WF_{SCJProgram} \bullet [p]_{SCJProgram} = [p.paragraphs]_{SCJParagraphs} \triangleleft [p]_{Application}
\]

We use the mathematical notation of \( Z \) [4] to specify our semantic functions, and explain any non-standard use of notation in \( Z \) as needed. In what follows, we focus on the semantic function \([\_]_{SCJSafelet}\) for the safelet, which is used by \([\_]_{SCJParagraphs}\) to give semantics to an SCJ-Circus safelet paragraph. The complete semantics is defined in [20].
As already said, 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 SCJ framework. It is formalised by the function \([\_\_]SCJSafelet\) in Figure 3.5. It takes an SCJ-Circus safelet paragraph \(s\) and outputs a sequence of two Circus processes: the application process and the process that models the complete behaviour of \(s\) as the parallel composition of the framework process SafeletFW, instantiated with the identifier of \(s\), and the application process. The channels on which these processes communicate are internal to the safelet and, therefore, hidden (\(\backslash\)). Figure 3.6 shows, as an example, the processes that define semantics of the ChkSafelet paragraph of the SCJ-Circus model in Figure 3.2 as specified by the application of \([\_\_]SCJSafelet\) to that paragraph.

In the definition of a semantic function, guillemots («») are used to distinguish the Circus syntax from the meta-language used to specify the rules. For instance, «safelet_app(s)», indicates that the function \(\text{safelet_app}\) must be evaluated on the parameter \(s\) and the resulting syntax tree must be substituted in place of «safelet_app(s)>>.
process ChkSafelet_App ≡ begin
  getSequencerMeth ≡ getSequencerCall?x!ChkSafeletID→
  var return : IDENTIFIER •
    startMissionSequencer!ChkSafeletID!ChkMissionSequencerID→
    return := ChkMissionSequencerID;
    getSequencerRet!x!ChkSafeletID!return → Skip
  initializeApplicationMeth ≡ safeletInitializeCall?x!ChkSafeletID → Skip;
  safeletInitializeRet!x!ChkSafeletID → Skip
  Methods ≡ μ X • getSequencerMeth; X □ initializeApplicationMeth; X □ end_safelet_app → Skip
  • Methods
end

process ChkSafelet ≡
  (SafeletFW (ChkSafeletID)
    (\[ getSequencerCall, getSequencerRet, safeletInitializeCall, safeletInitializeRet, end_safelet_app \])
    ChkSafelet_App)
  (\[ getSequencerCall, getSequencerRet, safeletInitializeCall, safeletInitializeRet, end_safelet_app \])

Figure 3.6: Semantics of the Safelet in our running example.

The application process for a safelet paragraph $s$ is named after $s$ with the extra suffix _App; in our example, we have ChkSafelet_App. The function name gives the name of a safelet. The safelet process is named just after $s$ itself. The definition of SafeletCS relies on the function safelet_app that produces the application-specific process, and the function SafeletCS that calculates the set of internal channels.

The safelet_app function is also shown in Figure 3.5. It takes an SCJ-Circus safelet paragraph $s$ and constructs the definition of a process with the same state ($s$.state) as $s$. Each action $N \equiv A$ of $s$ is translated into a Circus action using a pair of channels to model the call and return of the method represented by the action. Similarly, the actions getSequencer and initialize are translated into the actions getSequencerMeth and initializeApplicationMeth.

For our example, the safelet paragraph does not have a state, and so, neither does its safelet process. In addition, this paragraph has no extra actions: just getSequencer and, implicitly, initialize, which is omitted because its behaviour is just Skip. The matching actions in Figure 3.6 are getSequencerMeth and initializeApplicationMeth.

The translation of actions is defined by the function translate_method, which takes as arguments the identifier of the safelet paragraph (which is used to define the communications that represent the method calls and returns), the name of the action, and its body. It defines an action for the
**Circus** process where inputs are taken via a *Call* channel, the body is executed, and then outputs are returned via a matching *Ret* channel.

For instance, `getSequencerMethod` is started with a communication `getSequencerCall?x:ChkSafeletID`. We recall that the *Call* and *Ret* channels always take as parameters the identifiers of the caller, $x$ in this example, and of the callee, `ChkSafeletID` here. This ensures that results are returned to the caller via the *Ret* channel, and correct synchronisation with the framework process for the component: `SafeletFW(ChkSafeletID)`, in this case.

The body of the method is almost unchanged, except for two points. First, results are returned via the *Ret* channel. Second, creation of objects representing components of the SCJ paradigm (sequencer, mission, and so on) is translated to a communication. In our example, we have for the new `chkMissionSequencer()` a communication via `startMissionSequencer`, which passes information to the sequencer application process. This communication is with the application process, because it models a call to the constructor, which might have non-standard parameters.

All the actions of the safelet paragraph are combined in the action *Methods* that recursively offers them in external choice as well as the possibility to terminate the recursion via a synchronisation on the channel *end_safelet_app*. The overall behaviour of the process defined by its main action is the action *Methods*.

The parallel composition of the process obtained from $s$ as defined by `safelet_app(s)` and an instance of the framework process `SafeletFW` defines the semantics of the safelet $s$. In our example, the parallel composition with `ChkSafelet_A` defines `ChkSafelet`. The parallelism requires synchronisation on the call and return channels, as well as on `end_safelet_app`. These channels are identified by `SafeletCS(s)` and made internal using the hiding operator ($\backslash$).

The functions `sequencer_app`, `mission_app`, `PEH_app` and `AEH_app` that define the application processes for sequencers, missions, periodic and aperiodic event handlers are defined similarly [22]. Of note, in the case of the application process for a mission, is the fact that it includes a component `MArea` that models the mission memory.

Allocation of objects in the immortal memory by missions and handlers, and in the mission memory by handlers, has the potential to create a problem with resources. Namely, long-running programs may exhaust the space in these areas, if the allocation is not bounded. Typically, therefore, in an SCJ program, memory is allocated in the immortal area just by the safelet and the sequencer, and in the mission area just by the mission itself. Obviously, other components may read and update existing objects allocated in these areas, but allocation of new space is restricted.

For the sake of simplicity, therefore, we take a view that the mission paragraph of an *SCJ-Circus* program defines via its state the contents of the memory area for that mission. Accordingly, each
mission application process has an action $MArea$ with local variables corresponding to those that the associated mission allocates in the mission area. This action manages these variables using set and get channels, and can be terminated using $end\_mission\_app$. For our example, its definition (inside a process $ChkMission\_App$) is as follows. 0pt 6pt

$$MArea \triangleq \text{var} \ buffer : \text{seq N} \bullet \left( \begin{array}{c} \text{setBuffer} ? o!ChkMissionID? x \rightarrow buffer := x; \ X \\ \square \\ \text{getBuffer} ? o!ChkMissionID! buffer \rightarrow X \\ \square \\ end\_mission\_app \rightarrow \text{Skip} \end{array} \right)$$

The set and get channels have as additional parameters the identifiers of the caller and callee of these operations. In this way, these channels can be used for point-to-point communication between each handler and the mission area. Access to the shared variables in the $MArea$ actions is via set and get channels. This is handled by a variant of the translate method function that takes a set with the names of the shared variables as input.

Regarding the immortal memory, we can take a similar view, and include actions $IArea$, similar to $MArea$, in the safelet and sequencer application processes. For simplicity, however, we do not include such actions. Typically, the immortal memory is used for static variables, and allocations of objects representing the safelet and the sequencer for instance. We do not consider such objects in our model, but an extension to include them is not difficult.

Next, we define patterns of Circus specifications that we consider for refinement to SCJ-Circus programs.
Patterns

The core multi-processor cyclic executive pattern that we consider is shown in Figure 4.1. It characterises applications that have a single time-triggered mission, in which all periodic and aperiodic event handlers are executed, and may terminate. For this pattern, we have our simplest refinement strategy, which transforms the synchronous releases of aperiodic event handlers in the Circus models into asynchronous releases that occur in SCj-Circus.

process $P \equiv \begin{align*} &P_{\text{Handler}} \equiv \mu X \cdot (F_i \triangleright \text{PERIODICDEADLINE} \parallel \text{wait PERIOD}); X \triangleq t \rightarrow \text{Skip} \\
&A_{\text{Handler}} \equiv \mu X \cdot ((c_j \rightarrow G_j) \triangleright \text{APERIODICDEADLINE} \parallel \text{wait PERIOD}); X \triangleq t \rightarrow \text{Skip} \\
&M \equiv \text{var } x : T \cdot \mu Y \cdot \text{set}_x \triangleright nx \rightarrow x := nx; Y \triangleq get_x \triangleright Y \triangleq t \rightarrow \text{Skip} \\
&T \equiv r \rightarrow \mu X \cdot \mu r \rightarrow X \triangleq t \rightarrow \text{Skip} \\
&M' \equiv (M \parallel (\parallel i : I \cdot P_{\text{Handler}}_i) \parallel (\parallel j : J \cdot A_{\text{Handler}}_j)) \parallel \alpha S | \parallel t, r | \{\} \parallel \text{Termination} \parallel \{\ldots\} \\
&M_{\text{Sequencer}} \equiv M \\
&\text{Safelet} \equiv M_{\text{Sequencer}} \\
\bullet \text{Safelet} \\
\end{align*}$

where $\text{PERIODICDEADLINE} \leq \text{PERIOD} \land \text{APERIODICDEADLINE} \leq \text{PERIOD}$

Figure 4.1: Pattern: single time-triggered mission with precedence constraints without optional components.

In the pattern in Figure 4.1, the application is defined by a process $P$. It has no state (since we are not treating the immortal memory in SCj-Circus). The behaviour of $P$ is defined by an action Safelet. This action is simply defined by another action MissionSequencer, which in turn is defined by an action Mission modelling the single mission of the application. The indirection introduced by defining Safelet and MissionSequencer provides a hook to generalise this pattern and its associated refinement strategy as discussed later on.

Figure 4.2 shows an E anchor for our example that follows the pattern in Figure 4.1. In this case, the safelet, sequencer, and mission are defined by the actions ChkSafelet, ChkMissionSequencer, and ChkMission.

Parallelism occurs in the definition of Mission between the actions $P_{\text{Handler}}_i$ and $A_{\text{Handler}}_j$ that model the periodic and aperiodic handlers, the action $MArea$ that models the mission memory, and the termination management action Termination. Like in the semantics of SCj-Circus, 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. Here, however, we do not parametrise the communications on the get and set channels with identifiers (for the caller and callee) and use an application-defined channel to accept a request to terminate.

Termination accepts a synchronisation on a channel \( rt \); this corresponds to a request from a handler to terminate (carried out in SCJ using the method \( \text{requestTermination} \)). Afterwards, \( \text{Termination} \) starts a recursion that at each iteration either accepts another communication on \( rt \) (that is, a further request to terminate) and recurses, or synchronises on \( t \) and terminates. The synchronisation on \( t \) terminates the handlers. This captures the SCJ termination protocol, which caters for the possibility that several handlers request termination, until termination actually occurs.

A parallelism of actions in \( \text{Circus} \) 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 \( \text{Mission} \), however, all variables \( \alpha S \) in scope can be modified by the parallelism of handlers, while \( \text{Termination} \) modifies no variables.
AHandler\(_j\) \eqdef \mu X \bullet (((c_j \rightarrow G_j) \triangleright \text{APERIODICDEADLINE} \sqcap \text{wait PERIOD}) \sqcup \text{wait PERIOD}); \ X \sqcap t \rightarrow \text{Skip}

where APERIODICDEADLINE \leq \text{PERIOD}

Figure 4.3: Pattern: cyclic not in lock-step.

Periodic event handlers \(\text{PHandler}\_i\) are defined by recursive actions whose iterations take some fixed amount of time (\text{wait PERIOD}) whilst executing (in interleaving) an action \(F_i\) that models the behaviour of the handler release and takes at most \text{APERIODICDEADLINE} time units, until it is terminated via a synchronisation on \(t\). In our example, that action is a communication via a channel \text{input}, with a deadline \text{ID} itself, followed by an update of \text{buffer}, and a trigger of the aperiodic handler \text{Checker} via a release signal \text{rcheck}, all with a time budget of \text{PTB} time units.

The execution of an aperiodic event handler \(\text{AHandler}\_j\) is triggered by a synchronous event \(c_j\); in our example, it is \text{rcheck}. This may occur at any time during the cycle, but the synchronisation on \(c_j\) and the following action \(G_j\) that models the behaviour of the handler release must terminate within \text{APERIODICDEADLINE} time units. In \text{Checker}, the action recovers the \text{buffer}, checks its contents as defined by a function \text{check} whose definition we omit, and reports via a channel \text{output} whether it is consistent or not (\text{true} or \text{false}), all with a time budget of \text{ATB} time units.

We require \text{PERIODICDEADLINE} \leq \text{PERIOD} \land \text{APERIODICDEADLINE} \leq \text{PERIOD}, so that both the periodic and the aperiodic handlers releases can terminate within the cycle of the mission. For our example, these time expressions are defined by constants \text{PD}, \text{PER}, and \text{AD}. We have \(\text{PD} \leq \text{PER} \land \text{AD} \leq \text{PER}\) as required, and also need further restrictions involving \text{ID}, \text{PTB}, and \text{ATB} to ensure feasibility of the deadlines. These restrictions, which are omitted in Figure 4.2, are enforced by a global constraint on these constants.

Our second pattern allows for an aperiodic handler not to be called in a cycle. For that, the deadlines of the aperiodic event handlers must be adapted so that they can be missed as long as they have not started. This pattern differs from that in Figure 4.1 just in the models of the aperiodic handlers, which we describe in Figure 4.3.

An \(\text{AHandler}\_j\) action in this pattern is recursive like in the previous pattern. The iterations of the recursion again cater for a release via a channel \(c_j\) associated with a handling action \(G_j\), which needs to terminate within \text{APERIODICDEADLINE} time units, and cater for termination through the channel \(t\). In addition, however, if a release or termination does not take place within \text{PERIOD} time units, the choice offered in the iteration terminates and a new iteration is started. This is what caters for the possibility that a release does not take place in a given cycle; the deadline \text{APERIODICDEADLINE} is then considered from the start of the next cycle.
The purpose of the refinement strategy that we 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 (and should have been refined to an E-anchor that follows our second pattern, and not the first). 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 (from an SCJ-Circus model) to SCJ code via trivial transformations whose soundness is easier to establish. 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 [18].

After presenting our refinement strategy in the next section, we consider in Sections 5.5 and 5.6 variations of the above patterns for applications that do not terminate or have multiple missions.
Chapter 5

Refinement Strategy

In this section, we present the refinement strategy for the pattern in Figure 4.1, and then explain how it can be adapted for that in Figure 4.3. The phases of the strategy are shown in Figure 5.1. The target is an SCJ-Circus program of the form shown in Figure 5.2. For example, each action \textit{PHandler}_i in the starting model is defined in a corresponding \textit{periodic handler} paragraph, whose \textit{handleAsyncEvent} paragraph is determined by the body of \textit{PHandler}_i. Similarly, each aperiodic, mission, mission sequencer and safelet action is defined in a corresponding paragraph.

![Diagram](image)

Figure 5.1: Phases of the refinement strategy

\begin{align*}
\text{safelet Safelet} & \equiv \text{begin} \ldots \text{end} \\
\text{sequencer Sequencer} & \equiv \text{begin} \ldots \text{end} \\
\text{mission Mission} & \equiv \text{begin} \ldots \text{end} \\
\text{periodic handler PHandler}_i & \equiv \text{begin} \ldots \text{handleAsyncEvent} \equiv F_i \triangleright \text{PERIODICDEADLINE end} \\
\text{aperiodic handler AHandler}_j & \equiv \text{begin} \ldots \text{handleAsyncEvent} \equiv G_j \triangleright \text{APERIODICDEADLINE end}
\end{align*}

Figure 5.2: Target

The four phases of our refinement strategy are as follows: CF for introduction of the SCJ control flow, AP for introduction of application processes, FW for introduction of framework processes, and Conv for conversion to \textit{SCJ-Circus}. CF transforms the design model to make 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 \textit{activate Handlers} that models the synchronised start of the handlers in parallel.

AP separates specifications of application-specific behaviours (such as, specification of the functionality of a handler release) from those of behaviours of the SCJ paradigm (such as, starting a mission)
process $CF.P \equiv \text{begin}$

$PHandler_i \equiv \mu X \bullet (F_i \triangleright PD \parallel \text{wait PERIOD}; \ X \Box t \rightarrow \text{Skip})$

$AHandler_j \equiv \mu X \bullet ((c_j \rightarrow G_j) \triangleright AD \parallel \text{wait PERIOD}; \ X \Box t \rightarrow \text{Skip})$

$MArea \equiv \ldots$

$Termination \equiv rt \rightarrow \mu X \bullet (rt \Box t \rightarrow \text{Skip})$

$CF\_Mission \equiv \text{start\_mission} \rightarrow$

$$
MArea \parallel \text{Termination} \parallel
\begin{cases}
\| i : I \bullet SH_i \rightarrow \text{register}.i \rightarrow \text{start\_peh}.i \rightarrow \text{activate\_handlers} \rightarrow \text{done\_handler}.i \rightarrow \text{Skip} \\
\| j : J \bullet SH_j \rightarrow \text{register}.j \rightarrow \text{start\_aeh}.j \rightarrow \text{activate\_handlers} \rightarrow \text{done\_handler}.j \rightarrow \text{Skip}
\end{cases}
$$

$done\_mission \rightarrow \text{Skip}$

$Safelet \equiv$

$$
\begin{cases}
\| i : I \bullet SH_i \rightarrow \text{start\_peh}.i \rightarrow \text{activate\_handlers} \rightarrow PHandler_i; \ done\_handler.i \rightarrow \text{Skip} \\
\| j : J \bullet SH_j \rightarrow \text{start\_aeh}.j \rightarrow \text{activate\_handlers} \rightarrow AHandler_j [\ldots] [\| c_{ji} \parallel \{ \} \parallel \{ \}] Buffer_j \parallel \{ c_{ji} \}; \ done\_handler.j \rightarrow \text{Skip} \\
\| CF\_Mission \\
\| start\_sequencer \rightarrow \text{start\_mission} \rightarrow \text{done\_mission} \rightarrow \text{done\_sequencer} \rightarrow \text{Skip} \\
\| start\_sequencer \rightarrow \text{done\_sequencer} \rightarrow \text{Skip}
\end{cases}
$$

$\bullet Safelet \parallel [\| \text{start\_sequencer}, \text{done\_sequencer}, \ldots \parallel]$

$end$

Figure 5.3: Refinement strategy: CF – Target

that are implemented by the SCJ runtime environment (framework). FW takes the potentially incomplete model of the framework behaviour identified in AP, representing the 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 [5.1], the first three phases act only on constructs of the time and object-oriented languages, Circus Time and ObCircus, whilst the last phase uses SCJ-Circus specifications. Next, we detail each of the phases by defining procedures for application of refinement laws that can be used to transform the models as required.

5.1 CF: Introduction of SCJ control flow

This phase isolates each of the SCJ abstractions in the starting design model into parallel actions. It derives, from a design like that in Figure [4.1], a process structured as shown in Figure [5.3]. Its main action $Safelet$ is now the parallel composition of actions corresponding to specific SCJ abstractions: handlers, missions, and sequencer. The order of execution imposed by the original specification is maintained through the use of communication channels such as start\_mission and start\_sequencer, which are hidden in the main action. For simplicity, in some cases we use just $\parallel$ to indicate a parallelism, and omit channel and name sets, if they are not relevant for the discussion.
1. Apply Law call-intro to the action Safelet with channels cs and ce replaced by start_sequencer and done_sequencer;


Figure 5.4: Refinement strategy: (CF) Introduction of SCJ control flow

Figure 5.4 defines transformations that lead to this target specification. Overall, this phase applies novel specialised laws to parallelise the safelet, sequencer and mission actions. Next, handler laws replace synchronous communications between handlers with asynchronous communications, and separate the handlers from the mission action. Finally, parallel actions that specify the framework behaviour associated with mission execution are merged and sequentialised.

In the first step of this phase, the Law call-intro below is used to separate the safelet and mission sequencer into a parallel composition. This law applies to an action of the form $F(A)$, where we use $F(A)$ to refer to any action $F$ that has as a component the action $A$. The law splits $F(A)$ into the parallel composition of two actions, one of which executes $A$. 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 synchronisations with the parallel action using these internal channels.

**Law [call-intro]**

$$F(A) \sqsubseteq (F(cs \rightarrow ce \rightarrow \text{Skip}) \parallel \text{wrtV}(A) \parallel \text{cs, ce} \parallel \text{wrtV}(A)) \parallel (cs \rightarrow A; ce \rightarrow \text{Skip}) \parallel \text{cs, ce}$$

provided

- $\text{cs, ce} \parallel \text{usedC}(F) = \emptyset$
- $\text{wrtV}(A) \cap \text{usedV}(F(\text{Skip})) = \emptyset$ and $\text{wrtV}(F(\text{Skip})) \cap \text{usedV}(A) = \emptyset$

This law is proved by induction over the structure of the action $F$ using distribution and step laws such as those found in [11]. 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 $\text{usedC}(A)$ to refer to the set of channels used in an action $A$, and $\text{usedV}(A)$ and $\text{wrtV}(A)$ to refer to the variables used and modified by $A$.  

27
After the application of this law, the Safelet action *ChkSafelet*, of our example, is as follows.

\[
\text{ChkSafelet} \equiv \\
\begin{array}{c}
\text{start\_sequencer} \rightarrow \text{done\_sequencer} \rightarrow \text{Skip} \\
\text{[[} | \text{start\_sequencer, done\_sequencer] | \{\ldots\}] \\
\text{start\_sequencer} \rightarrow \text{ChkMissionSequencer;} \\
\text{done\_sequencer} \rightarrow \text{Skip}
\end{array} \\
\text{[[} | \text{start\_sequencer, done\_sequencer]}
\]

Instead of a direct call to *ChkMissionSequencer*, we have a call via channels *start\_sequencer* and *done\_sequencer*.

A similar change is effected in the definition of *ChkMissionSequencer* in the next step. This second step of CF (shown in Figure 5.4) applies a procedure called *mission\_sequencer\_CF*, which traverses the structure of the mission sequencer action and extracts its missions (to be executed in sequence) to form an action that composes in parallel the isolated mission sequencer and all its missions.

The definition of *mission\_sequencer\_CF* presented in Figure 5.5 covers only the case where there is a single mission, which is itself the definition of the sequencer (like in our example). To cover more elaborate sequencers, we need only to effect a distributed application of this base case through the structure of that mission-sequencer action. For instance, if we have a sequential composition of mission actions, the isolated mission-sequencer action obtained in this step has a sequential composition of calls via channels to the parallel mission actions. This is considered in Section 5.6, where we extend *mission\_sequencer\_CF* to cover multi-mission applications. For our simple case, after step 1(a) of the procedure *mission\_sequencer\_CF*, we obtain the following definition for the sequencer.

\[
\text{ChkMissionSequencer} \equiv \\
\begin{array}{c}
\text{start\_mission.MID} \rightarrow \text{done\_mission.MID} \rightarrow \text{Skip} \\
\text{[[} | \text{start\_mission.MID, done\_mission.MID] | \{\ldots\}] \\
\text{start\_mission.MID} \rightarrow \text{ChkMission:} \\
\text{done\_mission.MID} \rightarrow \text{Skip}
\end{array} \\
\text{[[} | \text{start\_mission.MID, done\_mission.MID]}
\]

The procedure *mission\_sequencer\_CF*, however, goes further. The steps 1(b) and 1(c) restructure or, more precisely, enrich the mission action, *Chkmision* in the example above, to include the control of the mission execution via *start\_mission* and *done\_mission*. The result is as follows.

\[
\text{ChkMissionSequencer} \equiv \\
\begin{array}{c}
\text{start\_mission.MID} \rightarrow \text{done\_mission.MID} \rightarrow \text{Skip} \\
\text{[[} | \text{start\_mission.MID, done\_mission.MID] | \{\ldots\}] \\
\text{Chkmision} \\
\text{Chkmision} \equiv \\
\text{(start\_mission.MID} \rightarrow \text{(MArea} || \text{Reader} || \text{Checker) [\ldots] Termination)} \rightarrow \text{done\_mission.MID} \rightarrow \text{Skip}
\end{array}
\]
This procedure takes a *Circus* action $A$ as its parameter.

1. If $A = M_i$ then
   
a) Apply Law call-intro to the action $A$ with channels $cs$ and $ce$ replaced by $\text{start\_mission.i}$ and $\text{done\_mission.i}$;
   
b) Apply Law copy-rule to the call action $M$ in $A$;
   
c) Apply Law copy-rule from right to left to the action 
   
   \[
   \text{start\_mission} \rightarrow \ldots; \text{done\_mission} \rightarrow \text{Skip}
   \]
   
   with name $M$;
   
d) Apply procedure mission-CF to $M$.

2. Else “The strategy does not cover this case.”

Figure 5.5: Refinement strategy: procedure mission-sequencer-CF

\[
\text{ChkMission} \equiv
\begin{aligned}
\text{start\_mission.MID} & \rightarrow M\text{Area}; \text{done\_mission.MID} \rightarrow \text{Skip} \\
\text{start\_mission.MID} & \rightarrow \left( \text{StartReader.RID} \rightarrow \text{register.RID} \rightarrow \text{start\_peh.RID} \rightarrow \text{done\_handler.RID} \rightarrow \text{Skip} \right); \ \text{\textbf{\textbackslash register \textbf{|} \textbf{\textbackslash \ldots}}} \ \text{\textbf{|} \textbf{\textbackslash \ldots}} \\
\text{start\_mission.MID} & \rightarrow \left( \text{StartChecker.CID} \rightarrow \text{register.CID} \rightarrow \text{start\_aeh.CID} \rightarrow \text{done\_handler.CID} \rightarrow \text{Skip} \right); \ \text{\textbf{\textbackslash register \textbf{|} \textbf{\textbackslash \ldots}}} \ \text{\textbf{|} \textbf{\textbackslash \ldots}} \\
\text{start\_mission.MID} & \rightarrow \text{Termination}; \text{done\_mission.MID} \rightarrow \text{Skip}
\end{aligned}
\]

Figure 5.6: ChkMission after Step 1(c) of mission-sequencer-CF

Finally, in step 1(d), we apply a separate procedure mission-CF, which is shown in Figure 5.7 and explained below. This procedure has the same purpose as mission-sequencer-CF, but applies to missions. In our example, after its application, we obtain the definition for ChkMission in Figure 5.6.

We note that the original actions MArea and Termination as well as the handler actions Reader and Checker are all in parallel now, controlled by the start\_mission and done\_mission signals. The extra parallel action defines the creation, via StartReader.RID and StartChecker.CID in the example, of the handlers, their registration, via a channel register, their activation, via start\_peh or start\_aeh, and their deactivation, via done\_handler. The new definitions for Reader and Checker include the control channels and termination via done\_handler. For instance, the definition of Reader is now as follows.

29
This procedure takes a *Circus* action $M_t$ that models as mission as its parameter.

1. For each aperiodic action $AHandler$, in the parallelism of handler actions use associativity and commutativity laws to obtain a parallelism between $AHandler$, and another parallelism with all other handlers, and apply Law `sync-async-conv`;

2. Apply Law `prefix-introduction` to the action $M_t$ to introduce channel `activate_handlers` after `start_mission`. Afterwards, apply `prefix-par-dist` and `par-prefix-dist` exhaustively to distribute the communications on `start_mission.i`, `activate_handlers` and `done_mission.i` over all parallel actions;

3. For each parallel action `start_mission.MID → H`, `done_mission.MID → Skip` (except where $H$ is `MArea` or `Termination`), apply Law `copy-rule` to $H$ and Law `handler-extract` to the whole action instantiated as follows:
   - If $H$ is a periodic event handler with id $HID$, period $P$ and start time $S$, instantiate $SH$ to `start_H.HID` with any parameters specific to the handler, $sb$ to `start_很多玩家.HID.S.P` and $r$ to `register.HID`;
   - If $H$ is an aperiodic event handler with id $HID$, instantiate $SH$ to `start_H.HID` with any parameters specific to the handler, $sb$ to `start_很多玩家.HID` and $r$ to `register.HID`.

4. Apply step laws exhaustively to merge the `start_mission` and `done_mission` communications in the left-hand side actions of the parallelisms introduced in the previous step;

5. For each action $SH.HID → sb.HID → ab → \mu X \bullet F(X, db.HID → Skip)$ associated with a handler $H$, apply Law `copy-rule` from right to left to introduce an action with name $H$.

Figure 5.7: Refinement strategy: procedure `mission-CF`

The changes to *Checker* resulting from `mission-CF` are similar.

```plaintext
Reader ≜ StartReader.RID → start__players.RID → activate_handlers→
          (input?!x → Skip) ◁ ID; getBuffer?buffer→
          setBuffer!((tail buffer) ∪ ⟨x⟩) → check → (wait 0 . PTB) ▷ PD ; X
          μ X •
          || wait PERIOD
          □
          done_handler.RID → Skip
```

Comparing this to the original definition of `Reader` shown in Figure 4.2, we observe that we have simply added control communications and replaced $t$ with `done_handler.RID`. In summary, the procedure `mission-CF` takes a *Circus* action that models a mission, and refines it into a parallelism of actions modelling the mission and its handlers.

The first step of `mission-CF` (see Figure 5.7) transforms the synchronous communications among event handlers to asynchronous communications to match the semantics of handlers in `SCJ-Circus`. The refinement guarantees that the asynchrony does not have any visible effect on the overall be-
haviour of the mission. This step considers each aperiodic event handler $A\text{Handler}_j$; for each of them, the parallelism of event handlers is reorganised (via simple associativity and commutativity laws), so that one of the parallel actions is $A\text{Handler}_j$, and the other is a component combining the rest of the handlers, including those that trigger $A\text{Handler}_j$. At this point the Law \textit{sync-async-conv} below is used to introduce the asynchronous communication between $A\text{Handler}_j$ and all other handlers that may release it.

In the Law \textit{sync-async-conv}, $\mu X \cdot ((c \rightarrow G) \triangleright D_1 \parallel \text{wait } P): X \cap \text{end} \rightarrow \text{Skip}$ matches the specification of an aperiodic handler and $O\text{Handlers} \equiv F(\mu X \cdot (B(c \rightarrow \text{Skip}) \triangleright D_2 \parallel \text{wait } P): X \cap \text{end} \rightarrow \text{Skip})$, the specification of the other handlers. We use $F(\cdot)_1$ to specify that there is exactly one occurrence of the action $A$ as a component of the action $F$. So, the other handlers are specified by an action $F$ that includes a recursion, whose non-recursive body $B$ has a single release of the aperiodic handler via a channel $c$. This recursion specifies a handler itself.

The Law \textit{sync-async-conv} makes the synchronous communication on $c$ asynchronous using a buffer of one position. The buffer is represented by a variable block that declares a boolean variable $\text{pending}$ initially set to $false$. This action accepts a communication on $c$ and, in doing so, records on $\text{pending}$ that the action $G$ whose execution is triggered by $c$ can then take place. The actual triggering of $G$ is now carried out via a fresh channel $c_i$. We use LHS to refer to the action on the left-hand side of the law, and require that $c_i$ is not used in that action. The buffer can be terminated via the same channel $\text{end}$ used to terminate the handlers, and makes no changes to global variables as indicated by its associated nameset $\{\}$ in the parallelism.

Because the communication on $c_i$ is hidden, it is urgent when it becomes available. Also, the aperiodic handler is always immediately available to communicate on $c_i$ because (a) it is available at the beginning; (b) once there is a synchronisation on $c_i$, it finishes and recurses before the end of the period $(D_1 - T < P)$; and (c) there is no further request to communicate on $c_i$ until the next period due to the proviso $O\text{Handlers} \parallel \{ \parallel c \parallel \} \text{TimeReq} = O\text{Handlers}.$

The value $T$ is a deadline that is available to the other handlers to call the aperiodic handler. This deadline needs to be given as input in the application of this law. After the refinement, the deadline on the release $G$ of the aperiodic handler is adjusted to $D_1 - T$, where $D_1$ is its overall deadline as originally specified.

$\text{TimeReq}$ specifies time requirements: (a) $c$ must take place before $T$ time units ($(c \cap t : (t < T))$, and after that, it cannot happen until the next period ($\text{wait } (P - t)$); or (b) $c$ does not happen within the period ($\text{wait } P$). With the proviso $O\text{Handlers} \parallel \{ \parallel c \parallel \} \text{TimeReq} = O\text{Handlers}$, we require that $O\text{Handlers}$ satisfies the property defined by $\text{TimeReq}$. This property also ensures that $O\text{Handlers}$ cannot communicate on $c$ infinitely often and starve the buffer.
Law [sync-async-conv]

\[
\mu X \cdot ((c \rightarrow G) \triangleright D_1 \parallel \text{wait } P); \ X \square \text{end} \rightarrow \text{Skip} \ [n_1 | cs | n_2] \ O\text{Handlers} \\
\subseteq \\
((\mu X \cdot (c_i \rightarrow (G \triangleright (D_1 - T)))); \ X \square \text{end} \rightarrow \text{Skip} \ [n_1 | | c_i \square | c_i \omega | \text{end} | |] \ Buffer) \ \ \ [n_1 | cs | n_2] \\
O\text{Handlers}
\]

where

Buffer \equiv \text{var pending : } \mathbb{B} \cdot \text{pending := false;} \ \mu X \cdot \\
c \rightarrow \text{pending := true;} \ X \square (\text{pending} \ \& \ c_i \rightarrow \text{pending := false;} \ X \square \text{end} \rightarrow \text{Skip} \\
O\text{Handlers} = F(\mu X \cdot (B(c \rightarrow \text{Skip})_1 \triangleright D_2 \parallel \text{wait } P); \ X \square \text{end} \rightarrow \text{Skip}) \\
TimeReq = \mu X \cdot (c@t : (t \prec T) \rightarrow \text{wait } (P - t) \square \text{wait } P); \ X \square \text{end} \rightarrow \text{Skip}

provided

- \(B \) is not recursive.
- \(c_i \notin \text{usedC}(\text{LHS})\)
- \(T < D_1 < P \land D_2 < P\)
- \(O\text{Handlers} [\parallel c \parallel] \ TimeReq = O\text{Handlers}\)
- \(\text{LHS} \) is deadlock-free and feasible.

This law also requires that the original action \(\text{LHS}\) is deadlock-free and feasible because behaviours that can lead to a deadlock, and, therefore, infeasibility of deadlines, under synchronous communication, such as accumulation of calls to a handler, are not blocked when we use asynchronous communication and, therefore, can potentially introduce new behaviours. Essentially, the last proviso of this law guarantees that the original actions can never perform calls that, were it not for the introduction of asynchronous communication, would deadlock the process or violate its timing restrictions. Proof of this law is by induction of the structure of \(F\) and \(B\).

For our example, at the end of the Step 1, the definition of \textit{ChkMission} has the structure already shown in Figure[5.6] but with the buffer included in the parallelism between \textit{Reader} and \textit{Checker}.

Step 2 \textbf{mission-CF} (see Figure[5.7]) inserts a new event \textit{activate_handlers} corresponding to the operation of the SCJ framework that coordinates the start of the handlers. This event is introduced after \textit{start_mission.i}. The simple Law \textbf{prefix-introduction} introduces a hidden event. Afterwards, we distribute the communications that record the beginning and end of the mission (that is, \textit{start_mission.i}}
and \texttt{done\_mission.}i, where \(i\) is the identifier of the mission) through the parallelism of handlers. This uses a standard step law of parallelism \([11]\) and \texttt{par\_prefix\_dist} below.

This law is similar to the standard step law of parallelism, which extracts synchronisations from inside parallel actions. Law \texttt{par\_prefix\_dist}, however, distributes a synchronisation back into the parallel actions. Essentially, Step 2 expands the parallelism between handlers, which is internal to the action \texttt{Mission} as illustrated in Figure \ref{fig:parallel_handler} to a top level parallelism as shown in Figure \ref{fig:top_level_parallelism}. This allows the extraction of the handler actions in the next step.

\textbf{Law [par\_prefix\_dist]}

\begin{align*}
(A \parallel ns_1 \mid cs \mid ns_2 \parallel B); \ c \rightarrow \text{Skip} \\
\subseteq \{ (A; \ c \rightarrow \text{Skip} \mid ns_1 \mid cs \parallel \; \mid ns_2 \parallel B; \ c \rightarrow \text{Skip}) \}
\end{align*}

provided \(\rightarrow c \in \text{used}\!(A) \cup \text{used}\!(B).\)

Step 3 applies the Law \texttt{handler\_extract} below to each of the parallel actions modelling event handlers. It wraps the handler actions with synchronisations on new internal channels, represented by \(SH, \ r, \) and \(sh\) in the law. These new interactions correspond to the initialisation of a mission, including creation (\(SH\)), registration (\(r\)), and starting (\(sh\)) of handlers. The channels \(sm, \ ab, \ db, \) and \(dm\) match the communications \texttt{start\_mission, activate\_handlers, done\_handler, and done\_mission} already in the mission action. All these synchronisations are orchestrated by a new parallel action that models the mission-execution cycle as a sequence of communications.

\textbf{Law [handler\_extract]}

\begin{align*}
sm \rightarrow ab \rightarrow (\mu X \bullet A; \; X \square db \rightarrow \text{Skip}); \; dm \rightarrow \text{Skip} \\
\subseteq \{ (sm \rightarrow SH \rightarrow r \rightarrow sh \rightarrow ab \rightarrow db \rightarrow dm \rightarrow \text{Skip}) \} \setminus \{ r \} \\
\setminus \{ \parallel sh, \; Sh \} \\
SH \rightarrow sh \rightarrow ab \rightarrow \mu X \bullet (A; \; X \square db \rightarrow \text{Skip})
\end{align*}

provided \(\parallel sh, Sh, db, ab \parallel \cap \text{used}\!(A) = \emptyset.\)

All the new events are hidden either locally (\(r\)) or in the parallelism (\(sh\) and \(SH\)). The variables modified by the handler release, specified by \(A\), remain under the control of that action. The new parallel action used for orchestration modifies no variables. This law can be easily proved by applying the parallelism step laws in \([11]\).
The repeated application of handler-extract introduces multiple parallel actions that orchestrate the synchronisations of the handlers. Step 4 applies step laws as much of possible in order to join the communications on start_mission and done_mission in these orchestrating action. As a result, this step also groups these repeated actions in a single parallel action named CF_Mission as shown in Figure 5.3 The last Step 5 simply gives names to the handler actions.

5.2 (AP) Introduction of application processes

The starting point of this phase is the target of the CF phase shown in Figure 5.3 its own target is in Figure 5.8 It defines a number of application actions, Handler_app, Mission_app, and so on. The original process CF_P is refined to AP_P, whose main action is the parallel composition of actions that model SCJ components. Each such action is itself a parallel composition of an application action and a new action that captures the SCJ framework behaviour relevant for the application. These do not define the complete behaviour of the framework processes from Section 3.2 but their application-specific behaviours are modelled by calls to application actions via Call and Ret channels.

Figure 5.8 shows the structure of the mission actions. In the AP framework action AP_Mission, after the start of the mission (start_mission.MID), initialize is called (missionInitializeCall!x!MID). This triggers, in the application action Mission_app, the creation (startH.i) and registration (register.i) of each of the handlers i. AP_Mission itself enters a loop where the registration of handlers is accepted and recorded in a variable handlers, until the call to initialize returns (missionInitializeRet!x!MID).

The steps of the AP phase are shown in Figure 5.9 Overall, we take the process obtained in phase CF and identify the application-specific behaviours and the behaviours expected of the framework, which are isolated and composed in parallel. 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 the CF phase to model the SCJ control flow. To replace in this control action the application-specific behaviours with calls via appropriate channels, we use specialised Laws p-handler-split, a-handler-split, mission-split, sequencer-split and safelet-split.

The first step of this phase applies the Laws p-handler-split and a-handler-split to each handler, depending on whether it is a periodic or an aperiodic handler, to separate the application-specific and the generic framework behaviours. These behaviours are composed in parallel using the channels handleAsyncEventCall and handleAsyncEventRet to model a call to the handleAsyncEvent method. We present below first Law p-handler-split.

This law applies to an action SH → sb.id.s.p → wait s; μX · ((A □ D)|| wait p); X ⊗ db → Skip modelling a periodic handler. In our example, the handlers start immediately after the start of the mission, so the wait action wait s expected after the start of the mission is just wait 0 and is omitted.
process AP_P ≡ begin
  Handler_app ≡ ... 
  AP_Handler ≡ ... 
  Mission_app ≡
    (start_mission.MID → missionInitializeCall?x!MID →
      (\( i : I \cup J \bullet \text{startH}.i \rightarrow \text{Skip} \); \( i : I \cup J \bullet \text{register}.i \rightarrow \text{Skip} \);
      missionInitializeRet!x!MID → Skip);
    done_mission.MID → Skip
  start_mission.MID → missionInitializeCall?x!MID →
    \( \mu X \bullet \square (\text{register}?x \rightarrow \text{handlers} := \text{handlers} \cup \{x\}; X)
    \text{missionInitializeRet!x!MID → Skip}
    (||i : \text{handlers} \bullet (\text{start}_\text{aeh}!i \rightarrow \text{Skip} \boxdot \text{start}_\text{peh}!i?i?p \rightarrow \text{Skip})
    \text{activate_handlers} → (||i : \text{handlers} \bullet \text{done_handler}.i \rightarrow \text{Skip}))
  var handlers : \mathbb{P}ID •
  AP_Mission ≡
    (\text{Safelet_app }\parallel\text{AP_Safelet}) \setminus \{ \ldots \}
    || (\text{MissionSequence_app }\parallel\text{AP_MissionSequence}) \setminus \{ \ldots \}
    • || (\text{Mission_app }\parallel\text{AP_Mission}) \setminus \{ \ldots \}
    || (|| i : I \cup J \bullet (\text{Handler_app }\parallel\text{AP_Handler}) \setminus \{ \ldots \})
  done
end

Figure 5.8: Refinement strategy: target of phase AP

1. Apply Law \text{p-handler-split} to each periodic handler action and Law \text{a-handler-split} to each aperiodic handler action, with channels \text{hs}C and \text{hs}R replaced with \text{handleAsyncEventCall} and \text{handleAsyncEventRet};
2. Apply procedure \text{mission-AP} to the safelet action;
3. Apply procedure \text{sequencer-AP} to the action that models the sequencer;
4. Apply Law \text{safelet-split} to the action that models the safelet.

Figure 5.9: Refinement strategy: (AP) Introduction of application processes

With an application of Law \text{p-handler-split}, the behaviour \( A \uplus D \) of the handler release is wrapped with new internal events \text{hs}C and \text{hs}R. The timing restrictions are moved to the framework action, with the exception only of the deadline \( D \) for termination of \( A \). Moreover, since the proviso ensures
that this deadline is less than or equal to the period \( p \) of the handler, then we can via refinement introduce in the framework action a (spurious) requirement for termination before the end of the period.

This law is easily proved by the application of parallelism step laws.

**Law [p-handler-split]**

\[
SH \rightarrow sh.\ id.\ s.\ p \rightarrow ab \rightarrow \text{wait } s; \mu X \bullet ((A \triangleright D) || (X \triangleright \text{db} \rightarrow \text{Skip}))
\]

\[
\subseteq \begin{cases} 
SH \rightarrow sh.\ id.\ s.\ p \rightarrow \mu X \bullet ((\text{haeC} \rightarrow A \triangleright D; \text{haeR} \rightarrow X) \triangleright \text{db} \rightarrow \text{Skip}) \\
\text{val}V(A) \parallel \text{sh}, \text{dh}, \text{haeC}, \text{haeR} \parallel \{(), ()\} \\
\text{sh}.\ id.?\text{start}?\text{period} \rightarrow ab \rightarrow \text{wait start}; \mu X \bullet \begin{cases} 
\left((\text{haeC} \rightarrow \text{haeR} \rightarrow \text{Skip}) \bullet 0 \bullet \text{period}\right) \parallel X \\
\text{db} \rightarrow \text{Skip}
\end{cases}
\end{cases}
\]

\[
\text{provided} \parallel \text{sh}, \text{dh}, \text{haeC}, \text{haeR} \parallel \cap \text{usedC}(A) = \emptyset \land D \leq p.
\]

The Law **a-handler-split** is similar to **p-handler-split** above. The model of an aperiodic handler, however, does not have a leading \text{wait} statement, because aperiodic handlers do not have an offset for starting. Accordingly, also the channel \text{sh} used to start a handler is not used to communicate a start time and a period.

**Law [a-handler-split]**

\[
SH \rightarrow sh \rightarrow ab \rightarrow \mu X \bullet (c \rightarrow (A \triangleright D); X \triangleright \text{db} \rightarrow \text{Skip})
\]

\[
\subseteq \begin{cases} 
SH \rightarrow sh \rightarrow \mu X \bullet ((\text{haeC} \rightarrow (A \triangleright D); \text{haeR} \rightarrow X) \triangleright \text{db} \rightarrow \text{Skip}) \\
\text{val}V(A) \parallel \text{sh}, \text{dh}, \text{haeC}, \text{haeR} \parallel \{(), ()\} \\
\text{sh} \rightarrow ab \rightarrow \mu X \bullet (c \rightarrow \text{haeC} \rightarrow \text{haeR} \rightarrow X \triangleright \text{db} \rightarrow \text{Skip})
\end{cases}
\]

\[
\text{provided} \\
\bullet \parallel \text{sh}, \text{dh}, \text{haeC}, \text{haeR} \parallel \cap \text{usedC}(A) = \emptyset
\]

Steps 2 and 3 apply specialised procedures **mission-AP** and **sequencer-AP** defined later to transform the actions that model the missions and the sequencer. These procedures have an effect on these actions similar to that of the Laws **p-handler-split** and **a-handler-split** on the handler actions. The
For each action $M_i$ that models a mission:

1. Apply to the safelet action, a version of Law mission-split, singling out the component $M_i$ as required. The version needed is that for the number of handlers in $M_i$. The new channels $miC$ and $miR$ must be instantiated as missionInitializeCall and missionInitializeRet.

2. Apply exhaustively to the safelet distributivity laws for hiding [11] to distribute the hiding of handler constructor channels inwards towards the result of the previous step;

3. Apply Law seq-interleave to the action hiding the $SH_i$ channels;

4. Apply Law rec-interleave to the action hiding the missionInitializeCall, missionInitializeRet, and register channels.

Figure 5.10: Refinement strategy: procedure mission-AP

procedures are slightly more complex, however, in that they must restructure actions and consider the syntactic structure of the actions to apply specific laws.

Step 4 applies the Law safelet-split to the safelet action to separate the behaviours that obtain the identifier of the sequencer and initialise the safelet. These are triggered by synchronisations on the channels getSequencerCall, getSequencerRet, safeletInitializeCall and safeletInitializeRet. We omit Law safelet-split because it is very similar and simpler than Law mission-split presented below. It can be found in [20].

At the end of the AP phase, the actions for the application processes are completed, but the actions $AP_-$ (see Figure 5.8) do not quite specify the SCJ runtime environment. These actions are the focus of the next FW phase, but before describing it, we detail the procedures mission-AP and sequencer-AP used in this phase.

**Procedure mission-AP** This is shown in Figure 5.10; it consists of three steps, for each action $M_i$ modelling a mission.

Step 1 applies a Law mission-split to the safelet action. This law, as described below, considers a component of the safelet action that models a mission. In this step, we take that component to be $M_i$.

We need a version of Law mission-split for the number of handlers used in $M_i$; below, we show a version for two handlers. In the strategy, $id_1$ and $id_2$ match the handlers identifiers. Moreover, as perhaps expected, we match $sm$ with start_mission.id, where id is the mission identifier, $SH_1$ and $SH_2$ with the handler constructor channels, $r$ with register, $sb$ with start_ab or start_peh, depending on whether we have an aperiodic or a periodic handler, $ab$ with activate_handlers, $dh$ with done_handler, and $dm$ with done_mission.
In all cases, the Law **mission-split** refines the mission action into a parallelism of two actions, one dealing with application-specific behaviours, such as handler instantiation (using channels $SH_i$), and the other dealing with framework-specific behaviours, such as handler activation and deactivation (using channels $sb$ and $db$). The parallel handler actions synchronise just in $ab$, so their behaviour on the other events is interleaved. In the refined action, the interleaving is made explicit. A new synchronisation point is enforced, via the new channel $miR$, right after the $r$ events, but the context $F$ ensures that the events up until then ($SH_1$, $SH_2$, $r.id_1$, and $r.id_2$) are hidden.

**Law [mission-split]**

\[
F \begin{pmatrix}
sm \rightarrow \left( SH_1 \rightarrow r.id_1 \rightarrow sh.id_1 \rightarrow ab \rightarrow db.id_1 \rightarrow Skip \right) \\
\left( SH_2 \rightarrow r.id_2 \rightarrow sh.id_2 \rightarrow ab \rightarrow db.id_2 \rightarrow Skip \right)
\end{pmatrix} \setminus \left\{ \| r \| : dm \rightarrow Skip \right\} \setminus \parallel SH_1, SH_2 \parallel
\]

\[
F \begin{pmatrix}
sm \rightarrow miC \left( SH_1 \rightarrow r.id_1 \rightarrow Skip \right) | \parallel SH_2 \rightarrow r.id_2 \rightarrow Skip \parallel \\
\langle \{1\} | \parallel sm, r, dm, miC, miR \parallel | \{1\} \rangle
\end{pmatrix} \langle \parallel miC, miR, r \parallel \rangle \setminus \parallel SH_1, SH_2 \parallel
\]

\[
F \begin{pmatrix}
sm \rightarrow miC \left( r.id_1 \rightarrow Skip \right) | \parallel r.id_2 \rightarrow Skip \parallel \\
\langle \{1\} | \parallel sm, r, dm, miC, miR \parallel | \{1\} \rangle
\end{pmatrix} \langle \parallel miC, miR, r \parallel \rangle \setminus \parallel SH_1, SH_2 \parallel
\]

\[
\langle \{1\} | \parallel sm, r, dm, miC, miR \parallel | \{1\} \rangle
\]

\[
\parallel SH_1, SH_2 \parallel
\]

\[
\parallel miC, miR, r \parallel
\]

\[
\parallel SH_1, SH_2 \parallel
\]

\[
\parallel miC, miR, r \parallel \cap usedC(LHS) = \emptyset
\]

Similarly to the laws for splitting handlers, this law is easily proved by the application of existing step laws \([\Pi]\).

In Step 2, the hiding of the channels corresponding to handler constructors is localised (using standard distributivity laws) around the parallelism introduced in the previous step and the handler actions. This allows the laws in the next steps to be applied to a particular (parallel) component of the safelet action (see Figure \[5.3\]).

The Law **seq-interleave** below, applied in Step 3, serialises the interleaving of handler instantiations and registrations in the application action. This is possible because, although we fix an order for the instantiations and registrations, these events are hidden, and the framework and handler actions allow any order to be chosen. In those actions, which synchronise on instantiation and registration events, they occur in interleaving, avoiding deadlocks due to the specific choice made in the refine-
ment. It is because we need to identify this flexible context that Law seq-interleave becomes rather specific, although it is simple: proof relies on step laws.

**Law [seq-interleave]**

\[
\left( \begin{array}{c}
sm \rightarrow miC \rightarrow (SH_1 \rightarrow r.id_1 \rightarrow \text{Skip}) ;
\| \| \| sm, r, dm, miC, miR \| \| \| 1 \| \\
SH_2 \rightarrow r.id_2 \rightarrow \text{Skip} \right) \right) \quad \| \| miC, miR, r \| \\
FA \quad \| \| SH_1, SH_2 \| \\
(\|SH_1 \| SH_2 \|) \quad \| \| SH_1, SH_2 \| \\
(\|SH_1 \| SH_2 \|) \quad \| \| SH_1, SH_2 \|
\right)
\]

where

\[
FA \equiv sm \rightarrow miC \rightarrow (r.id_1 \rightarrow \text{Skip}) ;
\| \| r.id_2 \rightarrow \text{Skip} \right) \quad \| \| sh.id_1 \rightarrow ah \rightarrow db.id_1 \rightarrow \text{Skip} \right) ;
\| \| ah \| \\
miR \rightarrow (sh.id_2 \rightarrow ah \rightarrow db.id_2 \rightarrow \text{Skip}) \\
\| \| sh.id_2 \rightarrow ah \rightarrow db.id_2 \rightarrow \text{Skip} \right) ;
\| \| dm \rightarrow \text{Skip}
\]

In the final step, the Law rec-interleave transforms the parallelisms in the framework action into a recursion and an iterated parallelism. The objective is to make the framework action (named FA in Law seq-interleave) generic, in the sense that it can deal with any number of handlers. This action FA is exactly the right parallel action in the action to which Law rec-interleave applies. The first interleaving (of registrations) is transformed into a recursion: a loop that accepts the registration of any number of handlers and records their identifier in a fresh local variable handlers.

The second parallelism of actions (that orchestrate the handlers instantiation, activation, and termination) is transformed into an iterated parallelism. Iteration is over handler identifiers h from the set handlers. Just like in the original action, it defines a parallelism of actions that orchestrate a given handler.

The context of the framework action, defined by the application and handler actions, ensure that the extra generality of the new framework action is not exploited. For example, although the new framework action allows the registration of any number of handlers, the context in which it occurs ensures that exactly two handlers are registered.
Again, it is because we need to identify this constrained context that the Law \textbf{rec-interleave} becomes rather long. Its proof relies on unfolding the recursion the required number of times, and instantiation of the iterated parallel.

\textbf{Law [rec-interleave]}

\[
\begin{align*}
\text{(sm $\rightarrow$ miC $\rightarrow$ SH$_1$ $\rightarrow$ SH$_2$ $\rightarrow$ r.id$_1$ $\rightarrow$ r.id$_2$ $\rightarrow$ miR $\rightarrow$ dm $\rightarrow$ Skip)} & \quad \text{\{ } \text{sm, r, dm, miC, miR \} } \quad \text{\{ \} } \\
\text{(sm $\rightarrow$ miC $\rightarrow$ r.id$_1$ $\rightarrow$ Skip)} & \quad \text{\{ } \text{\{ r.id$_1$ $\rightarrow$ Skip \} } \quad \text{\{ miC, miR, r \}} \\
\text{(sm $\rightarrow$ r.id$_2$ $\rightarrow$ Skip)} & \quad \text{\{ } \text{\{ r.id$_2$ $\rightarrow$ Skip \} } \quad \text{\{ miC, miR, r \}} \\
\text{(sb.id$_1$ $\rightarrow$ ab $\rightarrow$ db.id$_1$ $\rightarrow$ Skip)} & \quad \text{\{ } \text{\{ ab \} } \quad \text{\{ dm $\rightarrow$ Skip \}} \\
\text{(sb.id$_2$ $\rightarrow$ ab $\rightarrow$ db.id$_2$ $\rightarrow$ Skip)} & \quad \text{\{ } \text{\{ ab \} } \quad \text{\{ dm $\rightarrow$ Skip \}}
\end{align*}
\]

\text{provided handlers is fresh.}

\textbf{Procedure sequencer-AP} \quad \text{It splits the action that models the sequencer into framework and application-specific actions composed in parallel. This procedure defines a conditional strategy that identifies the structure of the sequencer and applies the appropriate steps, or fails the application of the strategy. This procedure is structured in this way in order to support further extension as described in Section 5.6.}

In this section, we cover only the simple case where the mission sequencer executes a single mission and terminates (if and when the mission terminates). In this case, shown in Figure 5.11 the procedure applies the Law \textbf{sequencer-split-single} to the sequencer action. This law (omitted here and available in [23]) refines the sequencer action into an application-specific action accepting synchronisations on the channel getNextMissionCall and getNextMissionRet, and a framework action that uses these channels to iteratively obtain the next mission and execute it.
This procedure takes a sequencer action $S$ as its parameter. It considers its component $A$ that defines its behaviour after and before $\text{start\_sequencer}$ and $\text{done\_sequencer}$.

1. If $A = \text{start\_mission}\cdot i \rightarrow \text{done\_mission}\cdot i \rightarrow \text{Skip}$ then
   a) Apply Law $\text{sequencer\_split\_single}$ to the action $A$.
2. Else “The strategy does not cover this case.”

Figure 5.11: Refinement strategy: procedure $\text{sequencer\_AP}$

5.3 (FW) Introduction of framework processes

This phase applies to the parallel composition of pairs of application and framework actions that describe each of the components of the SCJ design. For example, the safelet component, at the start of this phase, is described by the parallelism $\text{Safelet\_app} \parallel \text{AP\_Safelet}$ in Figure 5.8 and is transformed in this phase.

These parallel actions are close to those in the processes that give semantics to an $\text{SCJ\_Circus}$ program, but still lack features of the framework that are not used in the original E-Anchor. For example, the safelet action in our pattern (action $\text{ChkSafelet}$ in our example in Figure 4.2) does not perform any initialisation. For this reason, the method $\text{safelet\_Initialize}$ provided by the $\text{SCJ\_Circus safelet}$ paragraph and included in the framework process $\text{SafeletFW}$ (see Figure 3.4) is not included in the safelet framework action at the start of this phase.

It is this phase’s goal to introduce all such missing actions so that we can match the resulting actions to those of the application and framework processes defined in the semantics of an $\text{SCJ\_Circus}$ program. This is possible because the missing actions model methods are not used in the original specification, and their introduction leads to method calls that terminate immediately: for instance, a call to $\text{safelet\_Initialize}$ whose body is just $\text{Skip}$.

The process resulting from the application of this phase is shown in Figure 5.12. Its main action is the parallel composition of pairs of application and framework actions, where the application actions are those introduced in the AP phase, enriched to call the extra framework methods, and the framework actions that model the SCJ API.

The steps of this phase are shown in Figure 5.13. The main results used are lemmas specialised to the cyclic in lockstep pattern. The framework actions (for example, $\text{AP\_Safelet}$ in Figure 5.8) obtained in the AP phase are the same for all applications that follow our pattern, since this pattern restricts the flow of execution of missions and handlers and most of the framework-specific behaviours are introduced in the CP and AP phases. For this reason, very specialised results, like Lemma $\text{safelet}$—
process $FW_P \equiv \begin{align*}
   &\begin{cases}
     (\text{Safelet\_app} \parallel \text{SafeletFW}) \setminus \ldots \setminus \\
     (\text{MissionSequencer\_app} \parallel \text{MissionSequencerFW}) \setminus \ldots \setminus \\
     (\text{Mission\_app} \parallel \text{MissionFW}) \setminus \ldots \setminus \\
     (\text{\mid} i : I \bullet (\text{Handler\_app} \parallel \text{APEFW}) \setminus \ldots \setminus \\
     (\text{\mid} j : J \bullet (\text{Handler\_app} \parallel \text{PEFW}) \setminus \ldots \setminus \\
   \end{cases}
\end{align*}
end

Figure 5.12: Refinement strategy: target of phase $FW$

1. Apply Lemma $\text{safelet-fw-cl}$ to the parallel action that models the safelet in $AP_P$;
2. Apply procedure $\text{sequencer-fw-cl}$ to the parallel action that models the mission sequencer in $AP_P$;
3. Apply Lemma $\text{mission-fw-cl}$ to the parallel action that models a mission in $AP_P$;
4. Apply Lemma $\text{periodic-handler-fw-cl}$ to each parallel action that models a periodic handler in $AP_P$;
5. Apply Lemma $\text{aperiodic-handler-fw-cl}$ to each parallel action that models a aperiodic handler in $AP_P$;

Figure 5.13: Refinement strategy: (FW) Introduction of framework processes

$fw-cl$ described below, can be used to introduce the full-blown framework actions relying solely on syntactic conditions over the application actions.

The specialised lemmas are already described in terms of the channels actually used in the framework semantics. Obviously, the use of of a more general result is possible, but we keep the specific lemmas for clarity. (This also has a potential impact on the efficiency of the implementation of the strategy using a theorem prover.)

Lemma $\text{safelet-fw-cl}$ applies to the safelet component. The application action, at this stage, is very simple. It accepts a call to $\text{getSequencer}$ (via $\text{getSequencerCall}$), which returns ($\text{getSequencerRet}$) an identifier $sid$. The proviso requires $sid$ not to be $null$. The framework action calls $\text{getSequencer}$, starts the sequencer ($\text{start_sequencer}$), and ends the application ($\text{end_safelet_app}$) action when the sequencer terminates ($\text{done_sequencer}$).
With the use of Lemma safelet-fw-cl, we can justify the refinement of the application action to become much more elaborate. It now accepts a call to getSequencer or safeletInitialize, or a signal to terminate via end_safelet_app. In addition, when getSequencer is called, the new application action has a variable block that introduces a new local variable return to record the sequencer identifier sid. This matches the meaning of an SCJ method that returns a value. When safeletInitialize is called, the new application action does nothing.

The new framework action calls safeletInitialize, and then enforces the original sequence of events: just getSequencer is called, and then a termination takes place. A new conditional in the framework action checks the value of the sequencer identifier s returned by the application. Because this is sid, which is guaranteed by the provise not to be null, we can be sure that the sequencer is started like in the original action.

**Lemma [safelet-fw-cl]**

\[
\begin{align*}
&\text{\{getSequencerCall?x:id $\rightarrow$ getSequencerRet?x:id!sid $\rightarrow$ end_safelet_app $\rightarrow$ Skip} \\
&\text{\{if } \parallel \text{getSequencerCall, getSequencerRet, end_safelet_app } \parallel \text{ if } \} \\
&\text{getSequencerCall?x:id $\rightarrow$ getSequencerRet?x:id!sid?s $\rightarrow$ start_sequencer $\rightarrow$ done_sequencer $\rightarrow$ end_safelet_app $\rightarrow$ Skip} \\
&\text{\{getSequencerCall, getSequencerRet, end_safelet_app } \parallel \text{ if } \}
\end{align*}
\]

\[
\begin{align*}
\mu X \bullet &\text{safeletInitializeCall?x:id $\rightarrow$ safeletInitializeRet?x:id $\rightarrow$ X} \\
&\text{if } s = \text{null } \rightarrow \text{start_sequencer $\rightarrow$ done_sequencer $\rightarrow$ Skip} \\
&\text{end_safelet_app $\rightarrow$ Skip} \\
&\text{getSequencerCall, getSequencerRet, end_safelet_app, safeletInitializeCall, safeletInitializeRet } \parallel \text{ if } \}
\end{align*}
\]

provided sid $\neq$ null

Standard step, variable block, and conditional laws can be used to prove this lemma.

Steps 1,3,4 and 5 of the FW phase apply framework-completion lemmas like that above; all lemmas are in [?]. Step 2 is similar, but relies on the procedure sequencer-fw-cl in Figure 5.14 that identifies
This procedure takes as its parameter a *Circus* process $P$ of the form $AP\_P\_FW$.

1. If the action *Sequencer* of $P$ has a single occurrence of the action $\text{mission\_start}.i \rightarrow \text{mission\_done}.i \rightarrow \text{Skip}$, apply law *mission-fw-cl-single*.

2. Else “The strategy does not cover this case.”

![Figure 5.14: Refinement strategy: procedure sequencer-fw-cl](image)

handler $S\_Handler \equiv \ldots$

mission $S\_Mission \equiv \ldots$

sequencer $S\_MissionSequencer \equiv \ldots$

safelet $S\_Safelet \equiv \ldots$

![Figure 5.15: Refinement strategy: target of phase Conv](image)

the pattern of the sequencer framework action, and applies the appropriate laws and lemma. In the case of a cyclic in lockstep single-mission application, only one case is necessary. It identifies that the sequencer action only activates a single mission and applies a Lemma (*mission-fw-cl-single*) similar to the others to complete the sequencer framework action.

### 5.4 (Conv) Conversion to *SCJ-Circus*

This phase introduces new process paragraphs that isolate pairs of application and framework processes corresponding to the constructs of SCJ. Next, the semantics of *SCJ-Circus* is used to convert the newly introduced processes into the corresponding *SCJ-Circus* paragraphs. In our example, processes *SafeletFW* and *ChkSafelet_App* are paired and extracted to a process *ChkSafelet*, which is converted into a paragraph labelled *safelet*.

The target of this phase is the target of the strategy: a specification formed by *SCJ-Circus* paragraphs as shown in Figure 5.2. For our example, the resulting *SCJ-Circus* specification is shown in Figure 3.2. 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 5.16. The first step uses each pair of application and framework process to define a new process using the reverse of the copy-rule, and the second step uses the semantics of *SCJ-Circus* to transform these newly defined processes into *SCJ-Circus* paragraphs.
1. Apply the definition of parallel processes from right to left exhaustively to replace the basic process \( FW_P \), whose main action is parallel, with a parallelism of application processes and framework processes.

2. For each newly introduced component process, apply the definition of the appropriate SCJ abstraction from right to left.

Figure 5.16: Refinement strategy: (Conv) Conversion to SCJ-Circus

\[
\text{Mission} \triangleq (MArea \parallel (\parallel i : I \bullet P\text{Handler}_i) \parallel (\parallel j : Jn \bullet A\text{Handler}_j))
\]

Figure 5.17: Non-terminating cyclic in lockstep pattern

5.5 Non-terminating pattern

In the strategy in the previous section, handlers in the original specification may require termination. In the Step 2 of the AP phase, the termination treatment in the mission, originally in an action named \textit{Termination} in our pattern, is integrated into the mission framework action. In the Step 3 of the FW phase, the Lemma \textit{mission-fw-cl} renames the channel used in the original specification to \textit{requestTerminationCall}.

In this section, we adapt our strategy to consider applications that do not terminate. Figure 5.17 shows the \textit{Mission} action of the non-terminating cyclic in lockstep pattern. The main difference from that in Figure 4.1 is the missing \textit{Termination} action. The target of our refinement strategy is the same: that described in Figure 5.2.

The refinement strategy described in Section 5 cannot be applied to models that follow the pattern in Figure 5.17 because it expects the \textit{Mission} action to have an extra parallelism with the \textit{Termination} action. More specifically, Lemma \textit{mission-fw-cl}, used in Step 3 of FW does not apply. Instead of modifying this mission-specific result to introduce the mechanism of termination, we propose to extend the refinement strategy in Section 5 slightly. We introduce \textit{Termination} as a first step of the CF phase using the Law \textit{termination-intro} below. In this way, any other extensions to the strategy can take advantage of the general pattern with termination.

\textbf{Law} \textit{[termination-intro]}

\[
\begin{align*}
\mu X \bullet F; X & \subseteq (\mu X \bullet F; X \diamond t \rightarrow \text{Skip} \parallel \parallel \text{wrt} V(F) \parallel \parallel rt, t \parallel \parallel \mid \mid rt \rightarrow \mu X \bullet rt \rightarrow X \diamond t \rightarrow \text{Skip}) \parallel \parallel rt, t \parallel \\
\text{provided} \parallel t, rt \parallel \cap \text{used}C(F) = \emptyset.
\end{align*}
\]
MissionSequencer ≡ M₁; M₂; . . . ; Mₙ

Figure 5.18: Multi-mission pattern

This law takes a Circus action A of the form \( \mu X \cdot F; X \) and two channels \( t \) and \( rt \), and refines A into a parallelism between \( \mu X \cdot F; X \parallel t \rightarrow \text{Skip} \), 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 \( t \).

The extra parallel action introduced by this law is Termination. The refinement is valid because \( F \) does not use \( rt \), and so Termination waits forever for a synchronisation of this channel, and never actually terminates the mission.

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.

5.6 Multi-mission pattern

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 \( M₁; M₂; . . . ; Mₙ \) (see Figure 5.18). In this case, we need to modify the existing refinement strategy at very specific points to cater for a sequence of missions. We describe the changes next; they are summarised in Figure 5.19:

| CF-Step 2 | Use the recursive procedure mission-sequencer-CF shown in Figure 5.20 instead; |
| AP-Step 3 | Use the procedure sequencer-AP described in Figure 5.21 instead; |
| FW-Step 2 | Use the procedure sequencer-fw-cl shown in Figure 5.22 instead. |

Figure 5.19: Refinement strategy: multi-mission pattern

The procedure used in Step 2 of CF needs to be replaced with a recursive procedure that takes a sequence of missions \( Mᵢ; M_{rest} \), applies the original steps of mission-sequencer-CF for the first mission \( Mᵢ \), and recursively applies itself to the remaining missions in \( M_{rest} \). This extended procedure is shown in Figure 5.20. Its Step 1 is the same as in the original procedure (Figure 5.5); Step 2 is replaced with the recursive application of the procedure to the sequence of missions, and Step 3 is
This procedure takes a *Circus* action $A$ as its parameter.

1. If $A = M_i$ then
   a) Apply Law *call-intro* to the action $A$ with channels $cs$ and $ce$ replaced by $start\_mission.i$ and $done\_mission.i$;
   b) Apply Law *copy-rule* to the call action $M$ in $A$;
   c) Apply Law *copy-rule* from right to left to the action $start\_mission \rightarrow \ldots$; $done\_mission \rightarrow \text{Skip}$ with name $M$;
   d) Apply procedure *mission-CF* to $M$.

2. If $A = M_i \land B$ then
   a) apply *mission-sequencer-CF* to $M_i$ and to $B$ separately;
   b) apply hiding and parallelism distribution laws of [[1]] to obtain a parallel action.

3. Else “The strategy does not cover this case.”

---

Figure 5.20: Refinement strategy: procedure *mission-sequencer-CF*

---

used to catch cases not covered by the strategy. Further extensions need to elaborate on Step 3, like we do here in regards to the original procedure.

The result of the recursive applications (Step 2(2)) to a two-mission application is as follows.

\[
\text{MissionSequencer} \equiv
\begin{align*}
\left ( start\_mission.M_1 \rightarrow done\_mission.M_1 \rightarrow \text{Skip} \\
\text{Mission}_1 \\
\left ( start\_mission.M_2 \rightarrow done\_mission.M_2 \rightarrow \text{Skip} \\
\text{Mission}_2 \right ) \right ) & \parallel \\
\left ( \left \{ \{ \} \parallel start\_mission.M_1, done\_mission.M_1 \mid \{ \ldots \} \right \} \parallel \\
\left \{ \{ \} \parallel start\_mission.M_2, done\_mission.M_2 \mid \{ \ldots \} \right \} \right ) & \parallel
\end{align*}
\]

Like in the core strategy, each mission is replaced by a parallel action that calls a specific mission using $start\_mission$ and $done\_mission$, with the parallel actions composed in sequence. The next Step 2(b) uses distribution laws of parallelism and hiding to merge the hidings and to transform the sequence of parallel actions into a single parallel action, where the mission activations are in
sequence, but the mission actions are in interleaving as shown below.

\[
\text{MissionSequencer} = \begin{cases} 
\text{start}_\text{mission}.M_1 \rightarrow \text{done}_\text{mission}.M_1 \rightarrow \text{start}_\text{mission}.M_2 \rightarrow \text{done}_\text{mission}.M_2 \rightarrow \text{Skip} \\
\text{Mission}_1 \mid \text{Mission}_2 \\
\text{start}_\text{mission}.M_1, \text{start}_\text{mission}.M_2, \text{done}_\text{mission}.M_2 \\
\end{cases}
\]

This is possible because the channels \(\text{start}_\text{mission}.M_i\) and \(\text{done}_\text{mission}.M_i\) are only used by the action \(\text{Mission}_i\).

Steps 3 of the AP phase also needs a slightly different definition for the procedure \text{sequencer-AP}, using a law that introduces a particular pattern for the implementation of the \text{getNextMission} method tailored for multiple missions in sequence. The new procedure \text{sequencer-AP} is shown in Figure 5.21. It has an extra Step 2 that applies a new specialised law \text{sequencer-split-mult-seq}, which splits a sequence of communications on \text{start}_\text{mission} and \text{done}_\text{mission} into a parallel action communicating over channels \text{getNextMissionCall} and \text{getNextMissionRet} to obtain the identifiers of the missions being executed. The resulting action for our example above is sketched below.

\[
\ldots \text{start}_\text{mission}.M_1 \rightarrow \text{done}_\text{mission}.M_1 \rightarrow \text{start}_\text{mission}.M_1 \rightarrow \text{done}_\text{mission}.M_1 \rightarrow \ldots
\]

\[
\begin{cases} 
\text{getNextMissionCall}\!x!id \rightarrow \text{getNextMissionRet}\!x!id!id!M_1 \rightarrow \text{start}_\text{mission}.M_1 \rightarrow \text{done}_\text{mission}.M_1 \rightarrow \\
\text{getNextMissionCall}\!x!id \rightarrow \text{getNextMissionRet}\!x!id!id!M_2 \rightarrow \text{start}_\text{mission}.M_2 \rightarrow \text{done}_\text{mission}.M_2 \rightarrow \\
\text{getNextMissionCall}\!x!id \rightarrow \text{getNextMissionRet}\!x!id!null \rightarrow \\
\text{end}_\text{sequencer}_\text{app} \rightarrow \ldots \\
\text{getNextMissionCall}\!x!id \rightarrow \text{getNextMissionRet}\!x!id!id!M_1 \rightarrow \\
\text{getNextMissionCall}\!x!id \rightarrow \text{getNextMissionRet}\!x!id!id!M_2 \rightarrow \\
\text{getNextMissionCall}\!x!id \rightarrow \text{getNextMissionRet}\!x!id!null \rightarrow \\
\text{end}_\text{sequencer}_\text{app} \rightarrow \ldots \\
\text{getNextMissionCall}, \text{getNextMissionRet}, \text{end}_\text{sequencer}_\text{app} \rightarrow \ldots \\
\end{cases}
\]

Finally, in Step 2 of the FW phase, the procedure \text{sequencer-fw-cl} is also extended as shown in Figure 5.22 to use the Law \text{mission-fw-cl-mult-seq} to deal with sequences of missions. Like Law \text{mission-fw-cl-mult-single}, this new law transforms the pair of application and framework actions that model the mission sequencer to make them match the semantics of \text{SCJ-Circus} for a sequencer. The effect
This procedure takes a sequencer action $S$ as its parameter. It considers its component $A$ that defines its behaviour after and before $\text{start\_sequencer}$ and $\text{done\_sequencer}$.

1. If $A = \text{start\_mission}.i \rightarrow \text{done\_mission}.i \rightarrow \text{Skip}$ then
   a) Apply Law $\text{sequencer\_split\_single}$ to the action $A$.
2. If $A = (; i : I \cdot \text{start\_mission}.i \rightarrow \text{done\_mission}.i \rightarrow \text{Skip})$
   a) Apply Law $\text{sequencer\_split\_mult\_seq}$ to the action $A$.
3. Else “The strategy does not cover this case.”

Figure 5.21: Refinement strategy: procedure $\text{sequencer\_AP}$

Of this law on the action above is shown below.

$$
\mu X \cdot \text{getNextMissionCall}\!\!\!:i!id \rightarrow \text{getNextMissionRet}\!\!\!:i!id?m \rightarrow
\begin{cases}
\text{if} \ m = \text{null} \rightarrow \text{end\_sequencer\_app} \rightarrow \text{Skip} \\
\text{if} \ m \neq \text{null} \rightarrow \text{start\_mission}.m \rightarrow \text{done\_mission}.m \rightarrow X
\end{cases}
\begin{array}{c}
\{1, \{\text{getNextMissionCall}, \text{getNextMissionRet}\} \mid \epsilon\} \\
\text{var} \ i : \mathbb{Z} \cdot i := 0; \ \mu X \cdot \text{getNextMissionCall}\!\!\!:x!id \rightarrow
\begin{cases}
\text{if} \ i = 0 \rightarrow \text{getNextMissionRet}\!\!\!:x!id!M_1 \rightarrow i := i + 1 \\
\text{if} \ i = 1 \rightarrow \text{getNextMissionRet}\!\!\!:x!id!M_2 \rightarrow i := i + 1 \\
\text{if} \ i > 1 \rightarrow \text{getNextMissionRet}\!\!\!:x!id!\text{null} \rightarrow \text{Skip}
\end{cases} \\
\text{end\_sequencer\_app} \rightarrow \text{Skip}
\end{array}
$$

Of particular note is the effect of Law $\text{mission\_fw\_cl\_mult\_seq}$ on the result of the procedure $\text{sequencer\_AP}$ shown above. In this case, both parallel actions are transformed into recursions, with the framework action (left parallel action) using the mission identifier $\text{null}$ to terminate the recursion, and the application action (right parallel action) using a counter to decide the next mission to be returned. We note that this pattern for the application action is commonly used and can be modified to suit particular design decisions.
This procedure takes as its parameter a *Circus* process $P$ of the form $AP_P_{FW}$.

1. If the action *Sequencer* of $P$ has a single occurrence of the action $\text{mission}_\text{start}.i \rightarrow \text{mission}_\text{done}.i \rightarrow \text{Skip}$, apply law $\text{mission-fw-cl-single}$.

2. If the action *Sequencer* of $P$ contains an action of the form $i : I \bullet \text{mission}_\text{start}.i \rightarrow \text{mission}_\text{done}.i \rightarrow \text{Skip}$, apply law $\text{mission-fw-cl-mult-seq}$.

3. Else “The strategy does not cover this case.”

Figure 5.22: Refinement strategy: procedure $\text{sequencer-fw-cl}$
Conclusions

In this paper, we have extended previous work \[14, 3\] on both the semantics of SCJ and refinement strategies for SCJ programs. We propose a variant of Circus suitable for modelling using SCJ abstractions, update existing models of Circus designs to reflect changes to the SCJ specification and support compositional verification, and formalise the semantics of SCJ-Circus in terms of these updated models. We also detail a refinement strategy for SCJ programs. We describe each step necessary, and present the new specialised laws required.

Significant differences between our model of SCJ and that in \[14\] include: (1) the shift from the use of events to trigger the execution of aperiodic event handlers in a previous version of the SCJ specification, to the direct use of the asynchronous method release of an aperiodic event handler, and (2) modelling of handlers using two framework processes PEHFW (for a periodic event handler) and APEHFW (for an aperiodic event handler) so that the distinction between periodic and aperiodic handlers are made at the framework, instead of the application level.

Our strategy differs from that in \[3\], which takes an abstract model and refines it to a concrete program model: the E-anchor, which uses patterns based on SCJ, but not its API. Our strategy refines this SCJ-based model to a program that makes full use of the SCJ library to implement control aspects specific to SCJ. In this sense, our refinement strategy is similar to compilation, but the target SCJ-Circus programs include library calls not present in the starting model. Moreover, the strategy requires input, namely, deadlines for releases of aperiodic handlers to specialise the time design further. Finally, application of the strategy generate proof obligations and requires theorem proving.

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 \[3\]. The only law whose provisos are not syntactic is Law sync-async-conv. Its proof obligation involves the verification of deadlock-freedom, feasibility, and equality (refinement in both directions). In some cases, it is possible to verify deadlock-freedom and refinement using model-checkers such as FDR3\[21\], but large or infinite state-spaces are usually not tractable. In such cases, it may also be possible to apply compositional techniques \[22\]. In the case of infinite state-spaces, theorem proving is likely to be necessary. Verification of feasibility involves establishing the absence of miracles in the
semantics of the process given in the UTP [23]. Whilst this can be achieved through theorem proving, automated techniques based on the syntax of processes, rather than on their semantic models, are yet to be developed.

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 [24] [17] [25] [18]. Kalibera et al. [24] apply model checking and exhaustive testing to perform scheduling and race-condition analysis in SCJ programs. Haddad et al. [25] extend the Java Modeling Language [26] with timing properties to support worst-case execution analysis of SCJ programs, whilst Tang et al. [17] use annotations to analyse programs for memory safety and compliance to SCJ levels. Marriott et al. [18] perform automatic verification of memory-safety without requiring the user to annotate the program.

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 [6] 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. In addition, the extensions to the refinement strategy for the core pattern that we have presented illustrate how we can tackle more elaborate patterns by reusing our results. Our core pattern and strategy focus on a single mission, and form a strong basis to consider various forms of multi-mission (multi-mode) applications.

As future work, we plan to implement our strategies in a theorem prover such as Isabelle/HOL, and apply them to more examples, including SCJ benchmarks [27]. A more recent result [28] on the semantics of SCJ level 2 also paves the way to generalise SCJ-Circus and our strategy to cater for such programs.

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

Refinement Laws

This section summarises the novel refinement laws used in our refinement strategy.

Law [call-intro]

\[ F(A) \subseteq (F(cs \rightarrow ce \rightarrow \text{Skip}) \parallel \text{wrt}(\text{V}(A)) \parallel \{cs, ce\} \parallel \text{wrt}(\text{V}(A)) \parallel cs \rightarrow A; ce \rightarrow \text{Skip}) \setminus \{cs, ce\} \]

provided

- \(\{cs, ce\} \cap \text{used}(F) = \emptyset\)
- \(\text{wrt}(\text{V}(A)) \cap \text{used}(F(\text{Skip})) = \emptyset\) and \(\text{wrt}(F(\text{Skip})) \cap \text{used}(A) = \emptyset\)

Law [par-prefix-dist]

\[ (A \parallel ns_1 | cs | ns_2 \parallel B) ; c \rightarrow \text{Skip} \subseteq (A ; c \rightarrow \text{Skip} \parallel ns_1 | cs \parallel \{c\} | ns_2 \parallel B ; c \rightarrow \text{Skip}) \]

provided \(\neg c \in \text{used}(A) \cup \text{used}(B)\).

Law [handler-extract]

\[ sm \rightarrow ab \rightarrow (\mu X \cdot A ; X \rightarrow db \rightarrow \text{Skip}); dm \rightarrow \text{Skip} \subseteq (sm \rightarrow SH \rightarrow r \rightarrow sh \rightarrow ab \rightarrow db \rightarrow dm \rightarrow \text{Skip}) \setminus \{r\} \setminus \{sh, SH\} \]

provided \(\{sh, SH, db, ab\} \cap \text{used}(A) = \emptyset\).
Law [sync-async-conv]

\[ \mu X \bullet ((c \rightarrow G) \triangleright D_1 \parallel \text{wait } P) ; \ X \nabla \text{end} \rightarrow \text{Skip} [ n_1 | cs | n_2 ] \ O\text{Handlers} \]

where

\[ \text{Buffer} \triangleq \text{var pending} : \mathbb{B} \bullet \text{pending} := \text{false} ; \ \mu X \bullet \ c \rightarrow \text{pending} := \text{true} ; \ X \nabla (\text{pending}) \triangleq \ c_i \rightarrow \text{pending} := \text{false} ; \ X \nabla \text{end} \rightarrow \text{Skip} \]

\[ \text{O\text{Handlers}} = F(\mu X \bullet (B(c \rightarrow \text{Skip})_1 \triangleright D_2 \parallel \text{wait } P) ; \ X \nabla \text{end} \rightarrow \text{Skip}) \]

\[ \text{TimeReq} = \mu X \bullet ((c@t : (t < T) \rightarrow \text{wait } (P - t) \nabla \text{wait } P) ; \ X \nabla \text{end} \rightarrow \text{Skip}) \]

provided

- \( B \) is not recursive.
- \( c_i \notin \text{usedC}(\text{LHS}) \)
- \( T < D_1 < P \land D_2 < P \)
- \( \text{O\text{Handlers}} [ c ] \) \( \text{TimeReq} = \text{O\text{Handlers}} \)
- \( \text{LHS} \) is deadlock-free and feasible.

Law [p-handler-split]

\[ \text{SH} \rightarrow sb.id.s.p \rightarrow ab \rightarrow \text{wait } s ; \ \mu X \bullet ((A \triangleright D \parallel \text{wait } p) ; \ X \nabla db \rightarrow \text{Skip}) \]

\[ \triangleq \left( \begin{array}{l}
\text{SH} \rightarrow sb.id.s.p \rightarrow \mu X \bullet ((\text{haeC} \rightarrow A \triangleright D ; \text{haeR} \rightarrow X) \nabla db \rightarrow \text{Skip}) \\
\text{wrt} V(A) \mid \{ sb, db, \text{haeC}, \text{haeR} \} \}
\end{array} \right) \]

\[ sb.id?\text{start}?\text{period} \rightarrow ab \rightarrow \text{wait start} ; \ \mu X \bullet \left( \begin{array}{l}
\text{period} \rightarrow 0 \rightarrow \text{period} \\
\text{wait} \text{period} \rightarrow db \rightarrow \text{Skip}
\end{array} \right) ; \ X \]

\[ \text{provision} [ sb, db, \text{haeC}, \text{haeR} ] \cap \text{usedC}(A) = \emptyset \land D \leq p. \]
Law [a-handler-split]

\[ SH \rightarrow sb \rightarrow ab \rightarrow \mu X \bullet (c \rightarrow (A \triangleright D); X \Box db \rightarrow \text{Skip}) \]
\[ \subseteq \]
\[ SH \rightarrow sb \rightarrow \mu X \bullet (\text{baeC} \rightarrow (A \triangleright D); \text{baeR} \rightarrow X \Box db \rightarrow \text{Skip}) \]
\[ \text{wrt V} (A) \mid \{ sb, db, \text{baeC, baeR} \mid \{ c \} \} \]
\[ sb \rightarrow ab \rightarrow \mu X \bullet (c \rightarrow \text{baeC} \rightarrow \text{baeR} \rightarrow X \Box db \rightarrow \text{Skip}) \]

provided \( \{ sb, db, \text{baeC, baeR} \} \cap \text{usedC} (A) = \varnothing \)

Law [safelet-split]

\[ \text{start\_sequencer} \rightarrow \text{done\_sequencer} \rightarrow \text{Skip} \]
\[ \subseteq \]
\[ \text{getSequencerCall!id:id} \rightarrow \text{getSequencerRet!id:id?x} \rightarrow \]
\[ \text{start\_sequencer} \rightarrow \text{done\_sequencer} \rightarrow \text{Skip} \]
\[ \{ \{ c \} \mid \{ \text{getSequencerCall, setSequencerCall} \mid \{ \} \} \}
\[ \text{getSequencerCall?x!id} \rightarrow \text{getSequencerRet!x!id!sid} \rightarrow \text{Skip} \]

provided \( \text{sid} \neq \text{null} \)

Law [mission-split]

\[ F \; ( sm \rightarrow ( SH_1 \rightarrow r.\text{id}_1 \rightarrow sh.\text{id}_1 \rightarrow \text{ab} \rightarrow \text{db.}\text{id}_1 \rightarrow \text{Skip} ) \]
\[ \subseteq \]
\[ \{ \{ c \} \mid \{ \text{ab} \} \} \]
\[ ( SH_2 \rightarrow r.\text{id}_2 \rightarrow sh.\text{id}_2 \rightarrow \text{ab} \rightarrow \text{db.}\text{id}_2 \rightarrow \text{Skip} ) \]
\[ \text{r.}\text{id}_1 \rightarrow \text{Skip} \]
\[ \text{SH}_1 \rightarrow \text{r.}\text{id}_1 \rightarrow \text{Skip} ; \text{miR} \rightarrow \text{dm} \rightarrow \text{Skip} \]
\[ \{ \{ c \} \mid \{ \text{ab} \} \} \]
\[ \text{r.}\text{id}_2 \rightarrow \text{Skip} \]
\[ \text{miR} \rightarrow \text{Skip} ; \text{dm} \rightarrow \text{Skip} \]
\[ \text{r.}\text{id}_1 \rightarrow \text{Skip} \]
\[ \text{SH}_2 \rightarrow \text{r.}\text{id}_2 \rightarrow \text{Skip} \]
\[ \{ \{ \text{miC, miR} \} \mid \{ \} \} \]
\[ \text{provided} \; \{ \text{miC, miR} \} \cap \text{usedC} (LHS) = \varnothing \]
Law [seq-interleave]

\[
\begin{align*}
& \left( \text{sm} \rightarrow \text{miC} \rightarrow \left( \begin{array}{c}
\text{SH}_1 \rightarrow r.id_1 \rightarrow \text{Skip} \\
\text{SH}_1 \rightarrow r.id_2 \rightarrow \text{Skip}
\end{array} \right) \right) : \text{miR} \rightarrow dm \rightarrow \text{Skip} \\
& \text{FA} \\
& \left( \begin{array}{c}
\| \| \\
\| \| \\
\| \|
\end{array} \right) \left( \begin{array}{c}
\| \| \\
\| \| \\
\| \|
\end{array} \right) \\
& \left( \begin{array}{c}
\text{SH}_1 \rightarrow A_1 \parallel \text{SH}_2 \rightarrow A_2
\end{array} \right)
\end{align*}
\]

where

\[
\text{FA} \equiv \text{sm} \rightarrow \text{miC} \rightarrow \left( \begin{array}{c}
r.id_1 \rightarrow \text{Skip} \\
r.id_2 \rightarrow \text{Skip}
\end{array} \right) : \text{miR} \rightarrow \left( \begin{array}{c}
\text{sh.id}_1 \rightarrow ab \rightarrow \text{db.id}_1 \rightarrow \text{Skip} \\
\text{sh.id}_2 \rightarrow ab \rightarrow \text{db.id}_2 \rightarrow \text{Skip}
\end{array} \right) ; \text{dm} \rightarrow \text{Skip}
\]

Law [rec-interleave]

\[
\begin{align*}
& \left( \text{sm} \rightarrow \text{miC} \rightarrow \text{SH}_1 \rightarrow \text{SH}_2 \rightarrow r.id_1 \rightarrow r.id_2 \rightarrow \text{miR} \rightarrow dm \rightarrow \text{Skip} \right) \\
& \left( \begin{array}{c}
r.id_1 \rightarrow \text{Skip} \\
r.id_2 \rightarrow \text{Skip}
\end{array} \right) : \text{miR} \rightarrow \text{Skip} \\
& \text{FA} \\
& \left( \begin{array}{c}
\| \| \\
\| \| \\
\| \|
\end{array} \right) \left( \begin{array}{c}
\| \| \\
\| \| \\
\| \|
\end{array} \right) \\
& \left( \begin{array}{c}
\text{sh.id}_1 \rightarrow ab \rightarrow \text{db.id}_1 \rightarrow \text{Skip} \\
\text{sh.id}_2 \rightarrow ab \rightarrow \text{db.id}_2 \rightarrow \text{Skip}
\end{array} \right) : \text{dm} \rightarrow \text{Skip}
\end{align*}
\]

\[
\begin{align*}
& \left( \text{sm} \rightarrow \text{miC} \rightarrow \text{SH}_1 \rightarrow \text{SH}_2 \rightarrow r.id_1 \rightarrow r.id_2 \rightarrow \text{miR} \rightarrow dm \rightarrow \text{Skip} \right) \\
& \left( \begin{array}{c}
r.id_1 \rightarrow \text{Skip} \\
r.id_2 \rightarrow \text{Skip}
\end{array} \right) : \text{miR} \rightarrow \text{Skip} \\
& \text{FA} \\
& \left( \begin{array}{c}
\| \| \\
\| \| \\
\| \|
\end{array} \right) \left( \begin{array}{c}
\| \| \\
\| \| \\
\| \|
\end{array} \right) \\
& \left( \begin{array}{c}
\text{sh.id}_1 \rightarrow ab \rightarrow \text{db.id}_1 \rightarrow \text{Skip} \\
\text{sh.id}_2 \rightarrow ab \rightarrow \text{db.id}_2 \rightarrow \text{Skip}
\end{array} \right) : \text{dm} \rightarrow \text{Skip}
\end{align*}
\]

provided handlers is fresh.
Law [sequencer-split-single]

\[
\begin{align*}
\text{start} & \text{sequencer} \rightarrow \text{start} \text{mission}. \text{MID} \rightarrow \\
\text{done} & \text{mission}. \text{MID} \rightarrow \text{done} \text{sequencer} \rightarrow \text{Skip}
\end{align*}
\]

\[
\begin{align*}
\text{start} & \text{sequencer} \rightarrow \\
\mu X & \cdot \\
\hspace{1cm} & \begin{cases}
\text{getNextMissionCall}!\text{id}\text{!id} \rightarrow \text{getNextMissionRet}!\text{id}\text{!id}?m \rightarrow \\
\text{if } m = \text{null} \rightarrow \text{end} \text{sequencer} \text{app} \rightarrow \text{Skip} \\
\text{if } m \neq \text{null} \rightarrow \text{start} \text{mission} \text{.} \text{m} \rightarrow \text{done} \text{mission} \text{.} \text{m} \rightarrow X
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{done} & \text{sequencer} \rightarrow \text{Skip}
\end{align*}
\]

\[
\begin{align*}
\ll & \{1 \mid \{ \text{getNextMissionCall}, \text{getNextMissionRet}, \text{end} \text{sequencer} \text{app} \} \mid 1\} \\
\text{var} & b : B \cdot b := \text{false};
\end{align*}
\]

\[
\begin{align*}
\mu X & \cdot \\
\hspace{1cm} & \begin{cases}
\text{getNextMissionCall}!?\text{id}\text{!id} \rightarrow \\
\text{if } b \rightarrow \text{getNextMissionRet}!\text{x}\text{id}\text{!id}\text{!null} \rightarrow \text{Skip} \\
\text{if } \neg b \rightarrow \text{getNextMissionRet}!\text{x}\text{id}\text{!MID} \rightarrow \text{Skip}
\end{cases}
\]
\]

\[
\begin{align*}
\\ll & \{ \text{getNextMissionCall}, \text{getNextMissionRet}, \text{end} \text{sequencer} \text{app} \} \\
\text{provided} & \text{MID } \neq \text{null}
\end{align*}
\]
Lemma [safelet-fw-cl]

\[
\begin{align*}
&\text{getSequencerCall?x!id } \rightarrow \text{getSequencerRet!x!id!sid } \rightarrow \text{end_safelet_app } \rightarrow \text{Skip} \\
&\text{\{\{ getSequencerCall, getSequencerRet, end_safelet_app \} | \{ \}} \\
&\text{getSequencerCall!id!id } \rightarrow \text{getSequencerRet!id!id?s } \rightarrow \text{start_sequencer } \rightarrow \text{done_sequencer } \\
&\text{end_safelet_app } \rightarrow \text{Skip} \\
&\text{\{ getSequencerCall, getSequencerRet, end_safelet_app \} }
\end{align*}
\]

\[
\begin{align*}
&\mu X \cdot \\
&\text{getSequencerCall?x!id } \rightarrow \begin{align*}
&\text{var return : ID } \bullet \text{return := sid;} \\
&\text{getSequencerRet!x!id!return } \rightarrow \text{Skip}
\end{align*} \\
&\text{end_safelet_app } \rightarrow X \\
&\text{\{ getSequencerCall, getSequencerRet, safeletInitializeCall, safeletInitializeRet, end_safelet_app \} | \{ \}} \\
&\text{safeletInitializeCall!id!id } \rightarrow \text{safeletInitializeRet!x!id } \rightarrow X \\
&\text{getSequencerCall!id!id } \rightarrow \text{getSequencerRet!id!id?s } \rightarrow \begin{align*}
&\text{if s \neq null } \rightarrow \text{start_sequencer } \rightarrow \text{done_sequencer } \rightarrow \text{Skip} \\
&\text{if } s = \text{null } \rightarrow \text{Skip}
\end{align*} \\
&\text{end_safelet_app } \rightarrow \text{Skip} \\
&\text{\{ getSequencerCall, getSequencerRet, end_safelet_app, safeletInitializeCall, safeletInitializeRet \} }
\end{align*}
\]

provided \( sid \neq \text{null} \)
Lemma [mission-fw-cl]

\[
\begin{align*}
\text{start\_mission!id'?s} &\rightarrow \text{missionInitializeCall!x?id} \rightarrow SH_1 \rightarrow SH_2 \rightarrow \text{register.id}_1 \rightarrow \text{register.id}_2 \rightarrow \\
&\text{missionInitializeRet!x?id} \rightarrow \text{done\_mission.id} \rightarrow \text{Skip} \\
\end{align*}
\]

\[\begin{array}{l}
\{1\} \oplus \text{start\_mission, register, done\_mission, missionInitializeCall, missionInitializeRet} \parallel \{1\} \\
\text{var handlers : P ID • handlers := \{}; \text{start\_mission?x} \rightarrow \\
\begin{cases}
\text{requestTerminationCall!x?id} \rightarrow \text{requestTerminationRet!x?id} \rightarrow \\
\mu X \bullet \text{(requestTerminationCall!x?id} \rightarrow \text{requestTerminationRet!x?id} \rightarrow X
\end{cases}
\end{array}\]

\[\begin{array}{l}
\{1\} \oplus \text{done\_mission, stop\_handlers} \parallel \{\text{handlers}\} \\
\text{missionInitializeCall!id?id} \rightarrow \\
\begin{cases}
\mu X \bullet \text{(register?b \rightarrow handlers := handler \cup \{b\}; X \not\in \text{missionInitializeRet!id?id} \rightarrow \text{Skip})} \\
\{\text{activate\_handlers, stop\_handlers} \parallel \{\text{handlers}\} \bullet \text{activate\_handlers} \rightarrow \text{stop\_handlers} \rightarrow \text{done\_handler.b} \rightarrow \text{Skip} \}
\end{cases}
\end{array}\]

\[\text{done\_mission.x} \rightarrow \text{Skip} \]

\[\begin{array}{l}
\{\text{stop\_handlers}\} \\
\end{array}\]

\[\begin{array}{l}
\text{start\_mission!id'?s} \rightarrow \mu X \bullet \\
\text{missionInitializeCall!x?id} \rightarrow SH_1 \rightarrow SH_2 \rightarrow \text{register.id}_1 \rightarrow \text{register.id}_2 \rightarrow \text{missionInitializeRet!x?id} \rightarrow X
\end{array}\]

\[\begin{array}{l}
\{1\} \oplus \text{cleanUpCall!x?id} \rightarrow \text{cleanUpRet!x?id} \rightarrow X \parallel \text{end\_mission_app.id} \rightarrow \text{Skip} \\
\{1\} \oplus \text{start\_mission, register, done\_mission, missionInitializeCall, missionInitializeRet, requestTerminationCall, requestTerminationRet, terminationPendingCall, terminationPendingRet} \parallel \{1\} \\
\text{var sequencer : ID; handlers : P ID: terminating : \{B \bullet \text{sequence} := \text{null}; handlers := \{}; \text{terminating} := \text{false}; \\
\text{start\_mission!id'?s} \rightarrow \text{sequencer := s; missionInitializeCall!x?id} \rightarrow \\
\mu X \bullet \{\text{register?b \rightarrow handlers := handlers \cup \{b\}; X \not\in \text{missionInitializeRet!x?id} \rightarrow \text{Skip}\} \\
\begin{cases}
\text{if handlers = \{\} \rightarrow \text{Skip} \\
\text{\{\text{\# handlers \neq \{\}}} \rightarrow \\
\begin{cases}
\{\text{\# handlers} \bullet \text{\{\#\} start\_p.eh.b?s？p \rightarrow \text{\# start\_p.eh.b} \rightarrow \text{Skip}; activate\_handlers} \rightarrow \\
\text{stop\_handlers} \rightarrow \text{\{\# handlers} \bullet \text{\{\#\} done\_handler.b} \rightarrow \text{Skip}; done\_handlers} \rightarrow \text{Skip} \\
\end{cases}
\end{cases}
\end{cases}
\end{array}\]

\[\begin{array}{l}
\mu X \bullet \\
\text{requestTerminationCall!x?id} \rightarrow \\
\begin{cases}
\text{if \not\text{terminating} \rightarrow \{\text{\# terminating} := \text{true}; \text{stop\_handlers} \rightarrow \text{\# Skip} \\
\text{\# terminating} \rightarrow \text{\# Skip} \\
\end{cases}
\end{array}\]

\[\begin{array}{l}
\text{\# requestTerminationRet!x?id} \rightarrow \text{\# Skip} \\
\text{\# terminationPendingCall!x?id} \rightarrow \text{terminationPendingRet!x?id?termination} \rightarrow \text{\# Skip}; \text{X} \\
\text{\# done\_handlers} \rightarrow \text{\# Skip} \\
\end{array}\]

\[\begin{array}{l}
\{\text{\# stop\_handlers, done\_handlers}\} \\
\text{\# cleanUpCall!id?id?x} \rightarrow \text{cleanUpRet!id?id?x} \rightarrow \text{end\_mission_app:id} \rightarrow \text{done\_mission!id} \rightarrow \text{\# Skip} \\
\{\text{\# stop\_handlers, cleanUpCall, end\_mission_app}\}
\end{array}\]

where Channels $SH_1$ and $SH_2$ correspond to the constructor channels of the two handlers whose identifiers are $id_1$ and $id_2$.  

59
Lemma [periodic-handler-fw-cl]

\[
SH \rightarrow \text{start}_\text{peh}/x!id!@p \rightarrow \mu X \cdot \begin{cases}
\text{handleAsyncEventCall}/x!id! \rightarrow (A \triangleright D; \text{handleAsyncEventRet}/x!id! \rightarrow X) \\
\Box \text{done_handler}.id \rightarrow \text{Skip}
\end{cases}
\]

\[
\text{wrt} \ A \ | \ {[\text{start}_\text{peh}, \text{done_handler}, \text{handleAsyncEventCall}, \text{handleAsyncEventRet}]} | ({}]
\]

\[
\text{start}_\text{peh}!x.id!@start?\text{period} \rightarrow \text{activate_handlers} \rightarrow \text{wait} \text{start};
\]

\[
\mu X \cdot \begin{cases}
((\text{handleAsyncEventCall}/x!id! \rightarrow \text{handleAsyncEventRet}/x!id! \rightarrow \text{Skip}) \triangleright 0 \triangleright \text{period} || \text{wait period}); \ X
\end{cases}
\]

\[
E
\]

\[
SH \rightarrow \text{start}_\text{peh}/x!id!@p \rightarrow \mu X \cdot \begin{cases}
\text{handleAsyncEventCall}/x!id! \rightarrow (A \triangleright D; \text{handleAsyncEventRet}/x!id! \rightarrow X) \\
\Box \text{done_handler}.id \rightarrow \text{Skip}
\end{cases}
\]

\[
\text{wrt} \ A \ | \ {[\text{start}_\text{peh}, \text{done_handler}, \text{handleAsyncEventCall}, \text{handleAsyncEventRet}]} | ({}]
\]

\[
\text{var} \text{start, period} : \mathbb{R} \cdot
\]

\[
\text{start}_\text{peh}/id!@p \rightarrow \text{activate_handlers} \rightarrow \text{start} := s; \ \text{period} := p; \ \text{wait} \text{start};
\]

\[
\mu X \cdot \begin{cases}
(\text{handleAsyncEventCall}/id! \rightarrow \text{handleAsyncEventRet}/id! \rightarrow \text{Skip}) \triangleright \text{period}; \ X
\end{cases}
\]

\[
\Box \text{done_handler}.id \rightarrow \text{Skip}
\]

\[
\text{wrt} \ A \ | \ {[\text{handleAsyncEventCall}.id, \text{done_handler}.id]} | ({}]
\]

\[
(\mu Y \cdot ((\text{handleAsyncEventCall}/id! \rightarrow \text{wait period}) \triangleright 0); \ Y) \triangle \text{done_handler}.id \rightarrow \text{Skip}
\]
Lemma [aperiodic-handler-fw-cl]

\begin{align*}
SH & \rightarrow \text{start\_aeh}\?x!id \rightarrow \mu X \bullet \left( \begin{array}{l}
\text{handleAsyncEventCall}\?x!id \rightarrow (A \triangleright D; \text{handleAsyncEventRet}\!x!id \rightarrow X) \\
\Box \text{done}\_\text{handler}\.id \rightarrow \text{Skip}
\end{array} \right) \\
\text{wrt} V(A) & \mid \Box \text{start\_aeh, done}\_\text{handler, handleAsyncEventCall, handleAsyncEventRet} \mid \{ | \}
\text{start}\_\text{aeh}\?x!.id \rightarrow \text{activate}\_\text{handlers} \rightarrow \text{wait};
\end{align*}

\begin{align*}
\mu X \bullet \left( (c \rightarrow \text{handleAsyncEventCall}\?x!id \rightarrow \text{handleAsyncEventRet}\!x!id \rightarrow \text{Skip}) \; X \right) \quad \Box \Box \Box
\end{align*}

\begin{align*}
SH & \rightarrow \text{start}\_\text{aeh}\?x!id \rightarrow \mu X \bullet \left( \begin{array}{l}
\text{handleAsyncEventCall}\?x!id \rightarrow (A \triangleright D; \text{handleAsyncEventRet}\!x!id \rightarrow X) \\
\Box \text{done}\_\text{handler}\.id \rightarrow \text{Skip}
\end{array} \right) \\
\text{wrt} V(A) & \mid \Box \text{start\_aeh, done}\_\text{handler, handleAsyncEventCall, handleAsyncEventRet} \mid \{ | \}
\text{var} \text{currentRelease} : B; \text{newRelease} : B \bullet \\
\text{start}\_\text{aeh}\!id \rightarrow \text{activate}\_\text{handlers} \rightarrow
\mu X \bullet \left( \begin{array}{l}
\text{currentRelease} = \text{false} & \& (\text{releaseCall}\?o!id \rightarrow (\text{currentRelease} := \text{true}; \text{releaseRet}\!o!id \rightarrow \text{Skip})); X \\
\Box
\text{currentRelease} = \text{true} & \& (\text{releaseCall}\?o!id \rightarrow (\text{newRelease} := \text{true}; \text{releaseRet}\!o!id \rightarrow \text{Skip})); X \\
\Box
\text{currentRelease} = \text{true} & \& (\text{hasRelease} \rightarrow \text{currentRelease} := \text{newRelease}); X \\
\Box
\text{done}\_\text{handler}\.id \rightarrow \text{Skip}
\end{array} \right) \quad \Box \Box \Box
\end{align*}

\begin{align*}
\text{var} \text{currentRelease, newRelease} : B \bullet \Box \Box \Box
\mu X \bullet \left( \begin{array}{l}
\text{hasRelease} \rightarrow \text{handleAsyncEventCall}\!id!.id \rightarrow \text{handleAsyncEventRet}\!id!.id \rightarrow X \\
\Box
\text{done}\_\text{handler}\.id \rightarrow \text{Skip}
\end{array} \right) \quad \Box \Box \Box
\end{align*}

where

\begin{align*}
\text{Buffer} & \triangleq \text{var} \text{pending} : B \bullet \text{pending} := \text{false}; \mu X \bullet
\text{releaseCall}\?o!id \rightarrow \text{pending} := \text{true}; \text{releaseRet}\!o!id \rightarrow X \\
\Box
(\text{pending}) & \& c \rightarrow \text{pending} := \text{false}; X \\
\Box
\text{done}\_\text{handler}\.id \rightarrow \text{Skip}
\end{align*}
Lemma [mission-fw-cl-single]

\[
\begin{array}{c}
\text{start\_sequencer} \\
\quad \begin{cases}
\text{getNextMissionCall}!\text{id}\text{id} & \text{getNextMissionRet}!\text{id}\text{id}?m \\
\text{if } m = \text{null} & \text{end\_sequencer\_app} \rightarrow \text{Skip} \\
\quad \text{if } m \neq \text{null} & \text{start\_mission.m} \rightarrow \text{done\_mission.m.true} \rightarrow X
\end{cases}
\end{array}
\]

\[\mu X \cdot\]

\[
\text{done\_sequencer} \rightarrow \text{Skip}
\]

\[
\llbracket 1 \mid \llbracket\text{getNextMissionCall, getNextMissionRet, end\_sequencer\_app} \rrbracket \rrbracket 1\rrbracket
\]

var \( b : \mathbb{B} \quad b := \text{false} ;
\]

\[
\begin{array}{c}
\mu X \cdot
\quad \begin{cases}
\text{getNextMissionCall}!\text{x}\text{id} & \text{if } b \rightarrow \text{getNextMissionRet}!\text{x}\text{id}\text{null} \rightarrow \text{Skip} \\
\quad \text{if } \neg b \rightarrow \text{getNextMissionRet}!\text{x}\text{id}\text{MID} \rightarrow \text{Skip } X
\end{cases}
\end{array}
\]

\[
\text{end\_sequencer\_app} \rightarrow \text{Skip}
\]

\[
\llbracket\text{getNextMissionCall, getNextMissionRet, end\_sequencer\_app} \rrbracket \subseteq
\]

\[
\begin{array}{c}
\text{start\_sequencer} \\
\quad \begin{cases}
\text{getNextMissionCall}!\text{id}\text{id} & \mu X \cdot \text{getNextMissionRet}!\text{id}\text{id}?m \\
\text{if } m \neq \text{null} & \text{start\_mission.m} \rightarrow \text{done\_mission.m?continue} \\
\quad \text{if } \text{continue} \rightarrow X \quad \text{continue} \rightarrow \text{Skip } fi
\end{cases}
\end{array}
\]

\[\text{end\_sequencer\_app} \rightarrow \text{done\_sequencer} \rightarrow \text{Skip}
\]

\[
\llbracket 1 \mid \llbracket\text{getNextMissionCall, getNextMissionRet, end\_sequencer\_app} \rrbracket \rrbracket 1\rrbracket
\]

var \( b : \mathbb{B} \quad b := \text{false} ;
\]

\[
\mu X \cdot
\quad \begin{cases}
\text{getNextMissionCall}!\text{x}\text{id} & \text{if } b \rightarrow \text{getNextMissionRet}!\text{x}\text{id}\text{null} \rightarrow \text{Skip} \\
\quad \text{if } \neg b \rightarrow \text{getNextMissionRet}!\text{x}\text{id}\text{MID} \rightarrow \text{Skip } X
\end{cases}
\]

\[
\text{end\_sequencer\_app} \rightarrow \text{Skip}
\]

\[
\llbracket\text{getNextMissionCall, getNextMissionRet, end\_sequencer\_app} \rrbracket
\]

provided
Lemma [sequencer-split-mult-seq]

\[
\text{startsequencer} \rightarrow (; \ i \ : \ I \ \bullet \ \text{startmission.i} \ \rightarrow \ \text{donemission.i} \ \rightarrow \ \text{Skip});
\]

\[
\text{donesequencer} \rightarrow \ \text{Skip}
\]

\[\subseteq
\left(\begin{array}{c}
\text{startsequencer} \\
\mu X \bullet \\
\text{donesequencer} \rightarrow \ \text{Skip}
\end{array}\right)
\]

\[\left[\left[\{1\} \ | \ | \ \text{getNextMissionCall, getNextMissionRet, endsequencer_app} \ | \ \{1\}\right]\right]
\]

\[
\text{var} \ i : \ \text{nat} \ \bullet \ \ b := 1;
\]

\[
\mu X \bullet \\
\text{getNextmissioncall}!\text{id}\text{id} \rightarrow \\
\ \text{getNextmissionret}!\text{id}\text{id}?\text{m} \rightarrow \\
\ \text{if} \ m = \text{null} \ \rightarrow \ \text{endsequencer_app} \rightarrow \ \text{Skip}
\]

\[
\left[\left[\begin{array}{c}
\text{if} \ m \neq \text{null} \\
\ \rightarrow \ \text{startmission.m} \ \rightarrow \ \text{donemission.m} \ \rightarrow \ X
\end{array}\right]\right]\]

\[\text{donesequencer} \rightarrow \ \text{Skip}
\]

\[\left[\begin{array}{c}
\text{\{1\} | | \text{getNextMissionCall, getNextMissionRet, endsequencer_app} \ | \ \{1\}\}
\end{array}\right]
\]

\[
\text{provided} \ I \in \text{seq} \ \text{ID} \ \wedge \ \# I > 1 \ \wedge \ \forall \ i : I \ \bullet \ i \neq \text{null}
\]
Appendix B

SCJ Syntax

B.1 Program

Just like in Circus [11], a program is a sequence of zero or more paragraphs.

\[
\text{Program} \quad ::= \quad \text{Paragraph}^* 
\]

Well formedness  A program should include only one safelet and one mission sequencer paragraph.

B.2 Paragraphs

The syntactic category of paragraphs includes all those of Circus, the class definition paragraphs of ObCircus [12], and SCJ [2] paragraphs [29] to define a safelet, a mission sequencer, missions, and handlers.

We envisage that a process in an SCJ-Circus program encapsulates an object of a class type including a method that needs to interact with the external environment. These methods cannot be defined in a class due to its reactive behaviour. We observe that classes define sets of passive (data) objects. Processes, on the other hand, are not values. This restricts the kind of program designs that we can describe in SCJ-Circus. Reactive objects have to be defined by a process, and this limits the use of inheritance in the definition of these objects. Inheritance of processes is an interesting open question, whose main challenge is imposed by substitutability.
The syntactic category `Paragraph` is that of Z paragraphs as defined in [30], and we do not say more here. The definitions of channels and channel sets are just like in `Circus`; we reproduce the definition of the relevant syntactic categories below. Further information about them can be found in [11]. We observe that N is the syntactic category of valid identifiers, which has no further definition here.

## B.3 Channels

Channels are used to specify instantaneous and atomic interactions with the environment.

```
ChannelDeclaration  ::=  SimpleChannelDeclaration
  |  SimpleChannelDeclaration;  ChannelDeclaration
```

A channel declaration can be a simple declaration, or a simple declaration followed by a channel declaration.

```
SimpleChannelDeclaration  ::=  N+
  |  N+ : Expression
  |  | N+ | N+:Expression
  |  | Schema-Expression
```

A simple channel declaration may just give a list of one or more channel names. In this case, the channels are typeless, and used just for synchronisation. Channels used to communicate values are given a type defined by an expression.

A simple channel declaration can also be preceded by a list of names in brackets. In this case, the channel is generic: the names in brackets are formal parameters that can be used as types in the expression. This is akin to the use of generic definitions in Z.
Schema-Expression is the syntactic category of Z schema expressions as defined in [32]. A channel declaration given by a schema expression introduces all components of the schema as channels. The predicate of the schema expression should be true, and so typically omitted.

A channel set expression defines a channel set.

\[
\text{ChannelSetExpression} \; ::= \; \{ \} \\
\quad | \; \{ N^+ \} \\
\quad | \; N \\
\quad | \; \text{ChannelSetExpression} \cup \text{ChannelSetExpression} \\
\quad | \; \text{ChannelSetExpression} \cap \text{ChannelSetExpression} \\
\quad | \; \text{ChannelSetExpression} \setminus \text{ChannelSetExpression}
\]

It can be the empty channel set \{ \}, a channel set given by the enumeration of its channels, a reference to a definition of another set, or the union, intersection, or difference of two channel sets.

### B.4 Processes

Processes are defined just like in Circus Time.

\[
\text{ProcessDeclaration} \; ::= \; \text{process} N \bowtie \text{ProcessDefinition} \\
\quad | \; \text{process} N[N^+] \bowtie \text{ProcessDefinition}
\]

\[
\text{ProcessDefinition} \; ::= \; \text{Declaration} \cdot \text{ProcessDefinition} \\
\quad | \; \text{Declaration} \odot \text{ProcessDefinition} \\
\quad | \; \text{Process}
\]

Declaration contains the Z declarations as defined in [32].

\[
\text{Process} \quad ::= \quad \text{BasicProcess} \\
\quad | \; \text{CSPProcess} \\
\quad | \; \text{ProcessInstantiation} \\
\quad | \; \text{IteratedProcess} \\
\quad | \; \text{TimedCSPProcess}
\]
B.4.1 Basic processes

BasicProcess ::=
   begin
   ProcessParagraph
   state Schema-Expression
   ProcessParagraph
   • Action
   end

B.4.2 CSP processes

CSPProcess ::=
   Process ⊗ Process
   | Process ⊔ Process
   | Process || ChannelSetExpression || Process
   | Process || ChannelSetExpression | ChannelSetExpression || Process
   | Process || Process
   | Process ⊖ Process
   | Process \ ChannelSetExpression
   | Process[N + := N + ]

B.4.3 Process instantiations

ProcessInstantiation ::=
   N
   | N(Expression + )
   | (Declaration • ProcessDefinition)(Expression + )
   | N[Expression + ]
   | (Declaration ⊙ ProcessDefinition)[Expression + ]
   | N[Expression + ]
B.4.4 Iterated processes

\[
\text{IteratedProcess} ::= ;\;\text{Declaration} \cdot \text{Process} \\
\quad \mid \Box \text{Declaration} \cdot \text{Process} \\
\quad \mid \Diamond \text{Declaration} \cdot \text{Process} \\
\quad \mid \llbracket \text{ChannelSetExpression} \rrbracket \text{Declaration} \cdot \text{Process} \\
\quad \mid \llll\text{Declaration} \cdot \text{Process}
\]

B.4.5 Timed CSP processes

\[
\text{TimedCSPProcess} ::= \text{Process} \;\text{Expression} \\
\quad \mid \text{Process} \;\Delta\text{Expression} \;\text{Process} \\
\quad \mid \text{Expression} \;\leftarrow \text{Process} \\
\quad \mid \text{Process} \;\rightarrow \text{Expression}
\]

B.4.6 Process paragraphs

\[
\text{ProcessParagraph} ::= \text{Paragraph} \\
\quad \mid \text{N} \;\hat{=} \;\text{ParametrisedAction} \\
\quad \mid \text{nameset N} \;\equiv \;\text{NameSetExpression}
\]

\[
\text{ParametrisedAction} ::= \text{Action} \\
\quad \mid \text{Declaration} \;\cdot \;\text{ParametrisedAction} \\
\quad \mid \text{valDeclaration} \;\cdot \;\text{ParametrisedAction} \\
\quad \mid \text{resDeclaration} \;\cdot \;\text{ParametrisedAction} \\
\quad \mid \text{vresDeclaration} \;\cdot \;\text{ParametrisedAction}
\]
NameSetExpression ::= {} 
| \{N^+\} 
| N 
| NameSetExpression \cup NameSetExpression 
| NameSetExpression \cap NameSetExpression 
| NameSetExpression \setminus NameSetExpression

B.5 Actions

Action ::= Command 
| CSPAction 
| ActionInstantiation 
| IteratedAction 
| TimedCSPAction

B.5.1 Commands

Command ::= PrimitiveCommand 
| if GuardedActions fi 
| do GuardedActions od 
| var Declaration • Action

PrimitiveCommand ::= Schema-Expression 
| N^+ : [ Predicate, Predicate ] 
| [ Predicate ] 
| [ Predicate ] 
| N^+ := Expression^+ 
| N := new N 
| N := new N(Expression^+) 
| MethodCall-ObjectTarget
Predicate contains the Z predicates as defined in [30].

\[\text{MethodCall-ObjectTarget ::= Expression.N} \]
\[\text{Expression.N} \]
\[\text{Expression.N(Expression^+)}\]

\[\text{GuardedActions ::= GuardedAction} \]
\[\text{GuardedAction & GuardedActions} \]

\[\text{GuardedAction ::= Predicate \rightarrow Action}\]

**B.5.2 CSP Actions**

\[\text{CSPAction ::=} \]
\[\text{Skip} \]
\[\text{Stop} \]
\[\text{Chaos} \]
\[\text{Communication \rightarrow Action} \]
\[\text{Predicate \& Action} \]
\[\text{Action \& Action} \]
\[\text{Action \& Action} \]
\[\text{NameSetExpression | ChannelSetExpression | NameSetExpression \& Action} \]
\[\text{NameSetExpression | ChannelSetExpression | ChannelSetExpression | NameSetExpression \& Action} \]
\[\text{NameSetExpression | NameSetExpression \& Action} \]
\[\text{Action: Action} \]
\[\text{Action \& Action} \]
\[\text{Action \& ChannelSetExpression} \]
\[\text{μN • Action} \]

\[\text{Communication ::=} \]
\[\text{N CommunicationParameter^*} \]
\[\text{N|Expression^*|CommunicationParameter^*}\]
CommunicationParameter ::= ?N
  | ?N : Predicate
  | !Expression
  | .Expression

B.5.3 Action instantiations

ActionInstantiation ::= N
  | N(Expression+)
  | (ParametrisedAction)(Expression+)

B.5.4 Iterated actions

IteratedAction ::= ; Declaration • Action
  | □ Declaration • Action
  | □ Declaration • Action
  | ChannelSetExpression • Declaration • NameSetExpression • Action
  | Declaration • NameSetExpression • Process
B.5.5 Timed CSP actions

\[\text{TimedCSPActions} ::= \text{Communication}\oplus N \rightarrow \text{Action} \]
\[| \text{Communication} \rightarrow \text{Action} \]
\[| \text{Communication}\oplus N \rightarrow \text{Action} \]
\[| \text{wait} \text{Expression} \]
\[| \text{wait} \text{Expression} \Rightarrow \text{Expression} \]
\[| \text{wait} N : \text{Expression} \Rightarrow \text{Expression} \cdot \text{Action} \]
\[| \text{Action} \triangleright \text{Action} \]
\[| \text{Action} \triangle \text{Expression} \cdot \text{Action} \]
\[| \text{Expression} \triangleleft \text{Action} \]
\[| \text{Action} \triangleright \text{Expression} \]

B.6 Classes

Opt Opt

\[\text{ClassDeclaration} ::= \text{class} N \equiv \text{ClassDefinition} \]
\[| \text{class} N \equiv \text{extends} N \text{ ClassDefinition} \]
ClassDefinition ::= begin
    ClassParagraph*
    state Schema-Expression
    ClassParagraph*
    initial ParametrisedCommand
    ClassParagraph*
end

| begin
    ClassParagraph*
    state Predicate
    ClassParagraph*
    initial ParametrisedCommand
    ClassParagraph*
end

B.7 Class paragraphs

ClassParagraph ::= Paragraph
    | Qualifier N ≡ ParametrisedMethodBody

Qualifier ::= public
    | protected
    | private
    | logical

73
B.8 Methods

\[
\text{ParametrisedMethodBody} ::= \text{MethodBody} \\
| \text{val Declaration} \odot \text{ParametrisedMethodBody} \\
| \text{res Declaration} \odot \text{ParametrisedMethodBody} \\
| \text{vres Declaration} \odot \text{ParametrisedMethodBody}
\]

\[
\text{MethodBody} ::= \text{PrimitiveCommand} \\
| \text{SuperCall} \\
| \text{MethodBodyInstantiation} \\
| \text{MethodBody} \cdot \text{MethodBody} \\
| \text{if GuardedMethodBodies fi} \\
| \text{do GuardedMethodBodies od} \\
| \text{var Declaration} \odot \text{MethodBody} \\
| \mu N \odot \text{MethodBody}
\]

\[
\text{MethodBodyInstantiation} ::= N \\
| N(\text{Expression}^+) \\
| (\text{ParametrisedMethodBody})(\text{Expression}^+)
\]

\[
\text{SuperCall} ::= \text{super.N} \\
| \text{super.N}(\text{Expression}^+)
\]

\[
\text{GuardedMethodBodies} ::= \text{GuardedMethodBody} \\
| \text{GuardedMethodBody} \| \text{GuardedMethodBodies}
\]

\[
\text{GuardedMethodBody} ::= \text{Predicate} \rightarrow \text{MethodBody}
\]
B.9 Expressions

\[
\text{Expression} ::= \text{ZExpression} \\
\quad | \text{self} \\
\quad | \text{null} \\
\quad | \text{Expression.N} \\
\quad | \text{Expression instanceof N} \\
\quad | (N)\text{Expression}
\]

\text{ZExpression} is the syntactic category of Z expressions as defined in [30].

B.10 SCJ paragraphs

\[
\text{SCJParagraph} ::= \text{SCJSafelet} \\
\quad | \text{SCJSequencer} \\
\quad | \text{SCJMission} \\
\quad | \text{SCJHandler} \\
\quad | \text{SCJClass}
\]

B.11 Safelet

A safelet must provide an implementation of the getSequencer method. Additionally, it must provide an implementation of the immortalMemorySize method, but this is abstracted in SCJCircus and must be supplied separately for code generation.
SCJSafelet  ::=  safelet N ⊑ begin
    SCJSafeletProcessParagraph
    state Schema-Expression
    SCJSafeletProcessParagraph
    initialize ⊑ SCJSafeletAction
    SCJSafeletProcessParagraph
    getSequencer ⊑ res return : sequencer • SCJSafeletAction
    SCJSafeletProcessParagraph
end

SCJSafeletProcessParagraph  ::=  Paragraph
| N ⊑ SCJSafeletParametrisedAction
| nameset N ⊑ NameSetExpression

SCJSafeletParametrisedAction  ::=  SCJSafeletAction
| val Declaration • SCJSafeletParametrisedAction
| res Declaration • SCJSafeletParametrisedAction

We restrict the parameter passing mechanisms to those available in Java.

A safelet action is any action as described in Section 5 extended with the instantiation assignments shown below. This production represents the extension of the Action production with the safelet specific assignments. For instance, the sequential composition of Action, should allow for the sequential composition of two safelet assignments (x := newI N₁; y := newI N₂).

SCJSafeletAction  ::=  Action
| N := newI N
| N := newI N(Expression⁺)
B.12 Sequencer

The mission sequencer must provide a constructor (initial? – should the prescribed parameters be implicit and assumed not to be modified by initial?), and a getNextMission method. The standard additionally requires a signalTermination method that is not used in icecap.

\[
\text{SCJSequencer} ::= \text{sequencer N} \triangleq \text{begin} \\
\quad \text{SCJSequencerProcessParagraph}^* \\
\quad \text{state Schema-Expression} \\
\quad \text{SCJSequencerProcessParagraph}^* \\
\quad \text{initial} \triangleq \text{SCJSequencerAction} \\
\quad \text{SCJSequencerProcessParagraph}^* \\
\quad \text{getNextMission} \triangleq \text{res return : mission} \bullet \text{SCJSequencerAction} \\
\quad \text{SCJSequencerProcessParagraph}^* \\
\quad \text{end}
\]

\[
\text{SCJSequencerProcessParagraph} ::= \text{Paragraph} \\
\quad | \text{N} \triangleq \text{SCJSequencerParametrisedAction} \\
\quad | \text{nameset N} == \text{NameSetExpression}
\]

\[
\text{SCJSequencerParametrisedAction} ::= \text{SCJSequencerAction} \\
\quad | \text{val Declaration} \bullet \text{SCJSequencerParametrisedAction} \\
\quad | \text{res Declaration} \bullet \text{SCJSequencerParametrisedAction}
\]

We restrict the parameter passing mechanisms to those available in Java.

According to the standard the “getNextMission method may allocate the returned mission within the newly instantiated MissionMemory allocation area”. Should we therefore add newM in SCJSequencerAction? Or are we to assume that newMission instantiates the mission in mission memory? Notice that the sequencer could, in the constructor, instantiate a mission in the immortal memory and then pass it in the getNextMission method.
SCJSequencerAction ::= Action
| N := newIN
| N := newIN(Expression+)
| N := newMN
| N := newMN(Expression+)

B.13 Mission

The Mission class needs a static method getMission that returns the current mission. The process MissionFW should provide this operation: after the mission is started, it should always offer the action $\mu X \cdot \text{getMissionCall} \rightarrow \text{getMissionRet}!\text{mission} \rightarrow X$, which must be terminated when the mission terminates. Since (in level 1) there is at most one mission started at any time, only the active mission would answer. The method call, however, should be atomic, it should not be possible to terminate a mission before answering to a getMission call.

The abstract method missionMemorySize is abstracted in SCJCircus. For code generation the method must be completed with external information provided by the user (e.g., memory configuration).

SCJMission ::= mission N \triangleq begin
SCJMissionProcessParagraph*
state SchemaExpression
SCJMissionProcessParagraph*
initial \triangleq SCJSequencerAction
SCJMissionProcessParagraph*
initialize \triangleq SCJMissionInitializeAction
SCJMissionProcessParagraph*
cleanup \triangleq res return : $\Delta = \Delta \text{SCJMissionAction}$
SCJMissionProcessParagraph*
end
SCJMissionProcessParagraph ::= Paragraph
| N ⇒ SCJMissionParametrisedAction
| nameset N ⇒ NameSetExpression

SCJMissionParametrisedAction ::= SCJMissionAction
| val Declaration • SCJMissionAction
| res Declaration • SCJMissionAction

SCJMissionAction ::= SCJSequencerAction

The operator newHandler is used only to identify clearly what are the handlers of a mission.

SCJMissionInitializeAction ::= SCJSequencerAction
| newHandler N
| newHandler N(Expression+)

B.14 Handlers

SCJHandler ::= SCJPeriodicHandler
| SCJAperiodicHandler
SCJPeriodicHandler ::= periodic handler handler N \equiv \text{begin}
  SCJHandlerProcessParagraph^*
  start Expression period Expression
  SCJHandlerProcessParagraph^*
  state Schema-Expression
  SCJHandlerProcessParagraph^*
  initial \equiv SCJHandlerAction
  SCJHandlerProcessParagraph^*
  handleAsyncEvent \equiv SCJHandlerAction
  SCJHandlerProcessParagraph^*
\text{end}

SCJAperiodicHandler ::= aperiodic handler handler N \equiv \text{begin}
  SCJHandlerProcessParagraph^*
  state Schema-Expression
  SCJHandlerProcessParagraph^*
  initial \equiv SCJHandlerAction
  SCJHandlerProcessParagraph^*
  handleAsyncEvent \equiv SCJHandlerAction
  SCJHandlerProcessParagraph^*
\text{end}

SCJAperiodicLongHandler ::= aperiodic handler handler N \equiv \text{begin}
  SCJHandlerProcessParagraph^*
  state Schema-Expression
  SCJHandlerProcessParagraph^*
  initial \equiv SCJHandlerAction
  SCJHandlerProcessParagraph^*
  handleAsyncEvent \equiv \text{val } n : \mathbb{Z} \bullet SCJHandlerAction
  SCJHandlerProcessParagraph^*
\text{end}
SCJHandlerProcessParagraph ::= Paragraph
| N \equiv SCJHandlerParametrisedAction
| nameset N \equiv NameSetExpression

SCJHandlerParametrisedAction ::= SCJHandlerAction
| val Declaration \cdot SCJHandlerParametrisedAction
| res Declaration \cdot SCJHandlerParametrisedAction

We restrict the parameter passing mechanisms to those available in Java.

SCJHandlerAction ::= SCJMissionAction
| N \equiv \text{newPR} N
| N \equiv \text{newPR} N(\text{Expression}^+)
| N \equiv \text{newPM} N
| N \equiv \text{newPM} N(\text{Expression}^+)

B.15 SCJClass

An SCJClass is a class that interacts with SCJ active elements (e.g., Mission) by calling their operations, and, therefore, cannot be modelled as an \textit{ObCircus} class.

SCJClass ::= scjclass N \equiv \begin{align*}
\text{SCJClassParagraph}^* \\
\text{state} \ Schema-Expression \\
\text{SCJClassParagraph}^* \\
\text{initial} \equiv \text{SCJClassAction} \\
\text{SCJClassParagraph}^* \\
\end{align*}
\text{end}
SCJClassParagraph ::= Paragraph
| N \equiv SCJClassParametrisedAction
| nameset N == NameSetExpression

SCJClassParametrisedAction ::= SCJClassAction
| val Declaration \& SCJClassParametrisedAction
| res Declaration \& SCJClassParametrisedAction

We restrict the parameter passing mechanisms to those available in Java.

An SCJClass action is any action as described in Section 5.

SCJClassAction ::= Action
Appendix C

SCJCircus – semantics

In this chapter, the semantic functions are specified in a style based on the Z notation as partial functions over abstract syntax trees. Meta-notation elements such as if-then-else constructs are identified by the use of guillemots “if”...“then”...“else”...“fi”.

$$\llbracket \text{Program} \rrbracket : \text{Program} \rightarrow \text{Program}$$

$$\forall p : \text{WF}_\text{Program} \bullet \llbracket p \rrbracket_{\text{SCJProgram}} = \llbracket p.\text{paragraphs} \rrbracket_{\text{SCJParagraphs}}$$

$$\llbracket \text{Paragraphs} \rrbracket : \text{seq CircusParagraph} \rightarrow \text{seq CircusParagraph}$$

$$\forall ps : \text{seq WF}_\text{Paragraph} \bullet$$

$$\llbracket ps \rrbracket_{\text{SCJParagraphs}} = \begin{cases} \text{if } \# ps = 0 \\
\text{then } \langle \rangle \\
\text{else } \langle \text{head } ps \rangle_{\text{SCJParagraph}} \triangleright \llbracket \text{tail } ps \rrbracket_{\text{SCJParagraphs}} \end{cases}$$

$$\llbracket \text{Paragraph} \rrbracket : \text{CircusParagraph} \rightarrow \text{seq CircusParagraph}$$

$$\forall p : \text{WF}_\text{Paragraph} \bullet$$

$$\llbracket p \rrbracket_{\text{SCJParagraph}} = \begin{cases} \text{if } p \in \text{ran SCJSafelet then } \llbracket (\text{SCJSafelet } \sim) p \rrbracket_{\text{SCJSafelet}} \\
\text{else if } p \in \text{ran SCJSequencer then } \llbracket (SCJSequencer)^{\sim} p \rrbracket_{\text{SCJSequencer}} \\
\text{else if } p \in \text{ran SCJMission then } \llbracket (SCJMission)^{\sim} p \rrbracket_{\text{SCJMission}} \\
\text{else if } p \in \text{ran SCJPeriodicHandler then } \llbracket (SCJPeriodicHandler)^{\sim} p \rrbracket_{\text{APEH}} \\
\text{else if } p \in \text{ran SCJAperiodicHandler then } \llbracket (SCJAperiodicHandler)^{\sim} p \rrbracket_{\text{APEH}} \\
\text{else } \langle p \rangle \end{cases}$$
C.1 Safelet

\[ \text{\textsc{Safelet}} : \text{SCJSafelet} \rightarrow \text{seq CircusParagraph} \]

\[ \forall s: \text{WF\_SCJSafelet} \bullet [s]_{\text{SCJSafelet}} = \]

\[ \langle \text{ProcessDefinition} \]
\[ \text{«name(s)>>,} \]

\[ \text{HidingProcess} \]

\[ \text{ParallelProcess} \]
\[ \text{NamedAction(SafeletFW),} \]
\[ \text{safeflet\_chan(s),} \]
\[ \text{NamedAction(«name(s)>>\_App)} \]
\[ \),} \]
\[ \text{safeflet\_chan(s)} \]
\[ \rangle \]

\[ \text{ProcessDefinition} \]
\[ \text{«name(s)>>\_App,} \]
\[ \text{«safeflet\_app(s)>>} \]
\[ \rangle \]

\]

\[ \text{\textsc{Safelet}} : \text{SCJSafelet} \rightarrow \text{seq CircusParagraph} \]

\[ \forall s: \text{WF\_SCJSafelet} \bullet [s]_{\text{SCJSafelet}} = \]

\[ \left( \begin{array}{c}
\text{process «name(s)>>\_App } \cong \text{ «safeflet\_app(s)>>} \\
\text{process «name(s)>> } \cong \\
\text{(SafeletFW(«name(s)>>\_ID) } \parallel \text{ «SafeletCS(s)>> } \parallel \text{ «name(s)>>\_App)}
\end{array} \right) \]
SafeletCS : SCJSafelet → ChannelSet

∀ s : WF_SCJSafelet •
SafeletCS(s) = \{ getSequencerCall.x."name(s)ID", getSequencerRet.x."name(s)ID",
safeletInitializeCall.x."name(s)ID", safeletInitializeRet.x."name(s)ID",
«for each p : s.paragraphs of (N \equiv A) do»
  «N*Call.x."name(s)ID", «N*Ret.x."name(s)ID",
  «for»
startSafelet.x."name(s)ID" | x : ID \}
safelet_app : SCJSafelet \rightarrow \text{BasicProcess}

\forall s : WF_{\text{SCJSafelet}} \bullet
\begin{align*}
\text{safelet_app}(s) =
\begin{cases}
\text{begin} \\
\text{state} \ «s.state» \\
\text{«for each } p : s.paragraphs \text{ of } (\text{N} \equiv A) \text{ do} » \\
\text{«N»Meth} \equiv \text{«translate\_method(name(s)ID, N, A)»} \\
\text{«end»} \\
\text{getSequencerMeth} \equiv \text{«translate\_method(name(s)ID, getSequencer, s.getSequencer)»} \\
\text{initializeApplicationMeth} \equiv \text{«translate\_method(name(s)ID, initializeApplication, s.initialize)»} \\
\text{Methods} \equiv \mu X \bullet
\quad \text{getSequencerMeth}; X \\
\quad \Box \text{initializeApplicationMeth}; X \\
\text{«for each } p : s.paragraphs \text{ of } (\text{N} \equiv A) \text{ do} » \\
\quad \Box \text{«N»Meth}; X \\
\text{«end»} \\
\Box \\
\text{end}_\text{safelet_app} \rightarrow \text{Skip} \\
\bullet \text{Methods} \\
\end{cases}
\end{align*}
\text{end}

The result of applying these translation rules to the ACCS safelet is as follows.

\text{processACCSSafelet\_App} \equiv \begin{cases}
\text{begin} \\
\text{getSequencerMeth} \equiv \text{getSequencerCall}?x!\text{ACCSSafeletID} \rightarrow (\text{var return} : \text{ID} \bullet \\
\text{return} := \text{ACCSMissionSequencerID}; \\
\text{getSequencerRet}!x!\text{ACCSSafeletID}!\text{return} \rightarrow \text{Skip}) \\
\text{initializeApplicationMeth} \equiv \text{safeletInitializeCall}?x!\text{ACCSSafeletID} \rightarrow \\
\text{Skip}; \text{safeletInitializeRet}!x!\text{ACCSSafeletID} \rightarrow \text{Skip}
\end{cases}
Methods \( \equiv \mu X \cdot \)

\begin{align*}
& \quad \text{getSequencerMeth; } X \\
& \quad \square \\
& \quad \text{initializeApplicationMeth; } X \\
& \quad \square \\
& \quad \text{end_safelet_app } \rightarrow \text{Skip}
\end{align*}

\begin{itemize}
\item Methods
\end{itemize}

\textbf{end}

\textbf{process SafeletFW } \equiv \text{id : ID } \cdot \text{begin}

\begin{align*}
\text{Execute } \equiv & \text{getSequencerCall!id!id } \rightarrow \text{getSequencerRet!id!id?} \\
& \left( \begin{array}{l}
\text{if } s \neq \text{null } \rightarrow \text{start_sequencer } \rightarrow \text{done_sequencer } \rightarrow \text{Skip} \\
\text{fi}
\end{array} \right)
\end{align*}

\begin{itemize}
\item safeletInitializeCall!id!id \rightarrow \text{safeletInitializeRet!id!id } \rightarrow \text{Execute}; \\
\text{end_safelet_app } \rightarrow \text{terminate } \rightarrow \text{Skip}
\end{itemize}

\textbf{process ACCSSafelet } \equiv \\
(SafeletFW(ACCSSafeletID) \\
\begin{array}{l}
\text{\{getSequencerCall, getSequencerRet, safeletInitializeCall, safeletInitializeRet, end_safelet_app \}} \text{\}} \\
\text{ACCSSafelet_App}
\end{array}\)

\textbf{translate_method } : N \times N \times \text{ParametrisedAction} \rightarrow \text{Action}

\begin{align*}
\forall m, n : N; \ a : \text{WF_ParametrisedAction } \bullet \\
\text{translate_method}(m, n, a) = \\
\begin{cases}
\text{«n»Call!x!«m»ID «[p : \text{decl.vparams} \bullet \text{p.name}»} \\
\text{«if } \# \text{decl.rparams} > 0 \text{ then } \var \text{decl.rparams}(1) \bullet \text{«fi»} \\
\text{«n»!(p : \text{decl.vparams} \land \text{decl.rparams})}; \\
\text{«n»Ret!x!«m»ID «if } \# \text{decl.rparams} > 0 \text{ then } !\text{decl.rparams}(1).\text{name} \text{fi» } \rightarrow \text{Skip}
\end{cases}
\end{align*}
C.2 Sequencer

The rules for the sequencer are

```
\[ scJSequencer : SCJSequencer \rightarrow \text{seq CircusParagraph} \]

\forall s : WF_{SCJSequencer} \bullet
\begin{align*}
\llbracket s \rrbracket SCJSequencer = \\
\text{process } \langle \text{name(s)} \rangle_{\text{App}} \equiv \langle \text{sequencer\_app(s)} \rangle \\
\text{process } \langle \text{name(s)} \rangle \equiv \\
(\text{SequencerFW} [ \langle \text{sequencer\_chan(s)} \rangle \cup \langle \text{done\_sequencer} \rangle ] \langle \text{name(s)} \rangle_{\text{App}}) \\
\langle \text{sequencer\_chan(s)} \rangle
\end{align*}
```

```
sequencer\_app : SCJSequencer \rightarrow \text{BasicProcess}

\forall s : WF_{SCJSequencer} \bullet
\begin{align*}
\text{sequencer\_app(s)} = \\
\text{begin} \\
\text{state } \langle s\text{.state} \rangle \\
\langle s\text{.paragraphs} \rangle \\
\langle \text{for each } p : s\text{.paragraphs of } (N \equiv \text{SCJSafeletParametrisedAction}) \text{ do} \rangle \\
\langle N\text{Meth} \equiv \langle \text{translate\_method(name(s)ID,N,p.body)} \rangle \rangle \\
\langle \text{end} \rangle \\
\text{Init} = \langle s\text{.initial} \rangle \\
\text{getNextMissionMeth} \equiv \\
\langle \text{translate\_method(name(s)ID,getNextMission,s\text{.getNextMission.action})} \rangle \\
\begin{align*}
\text{getNextMissionMeth; } X \\
\langle \text{for each } p : s\text{.paragraphs of } (N \equiv A) \text{ do} \rangle \\
\langle \text{X\_Meth; } X \rangle \\
\langle \text{end} \rangle \\
\langle \text{end\_sequencer\_app} \rightarrow \text{Skip} \rangle
\end{align*}
\end{align*}
```

```
SequencerCS : SCJSequencer → ChannelSet

∀ s : WF_SCJSequencer •
SequencerCS(s) = { getNextCall x «name(s)ID», getNextCallRet x «name(s)ID», «for each p : s.paragraphs of (N ≡ A) do»
«N»Call x «name(s)ID», «N»Ret x «name(s)ID», «for»,
end_sequencer_app | x : ID }

sequencerACCSMissionSequencer ≡ begin
state mission_done : B
initial ≡ mission_done := false
getNextMission ≡ res return : ID •
if ¬ mission_done → mission_done := true; return := newACCSMission()
[mission_done → return := null
fi
end

Applying the the mission sequencer of the ACCS example, we obtain the following processes.

process ACCSMissionSequencer_App ≡ begin
state[mission_done : B]
Init ≡ mission_done := false

getNextMissionMeth ≡ getNextMissionCall?x!ACCSMissionSequencerID→
var return : ID •
if ¬ mission_done → mission_done := true; return := newACCSMission()
[mission_done → return := null
fi
getNextMissionRet!x!ACCSMissionSequencerID!return → Skip

Methods ≡ μ X • getNextMissionMeth: X □ end_sequencer_app → Skip

89
• Init; Methods
end

process SequencerFW ≜ id : ID • begin
  Execute ≜ getNextMissionCall!id!id → getNextMissionRet!id!id?m→
  (if m ≠ null → start_mission.m → done_mission.m?continue→
   (if continue → Execute
    (\ continue → Skip)
   fi
   if m = null → Skip
   fi)
  fi

• start_sequencer → Execute; end_sequencer_app → done_sequencer → terminate → Skip
end

process ACCSMissionSequencer ≜
  (SequencerFW (ACCSMissionSequencerID)
   [\ getNextMissionCall, getNextMissionRet, end_sequencer_app []]
   ACCSMissionSequencer_app) \ [\ getNextMissionCall, getNextMissionRet, end_sequencer_app []]

C.3 Mission

\[[\_]: SCJM \rightarrow seq CircusParagraph
\forall s : WF_SCJM •

\begin{align*}
\exists s : SCJM = &\begin{cases}
\text{process } \text{name}(s) \text{App} \cong \text{mission_app}(s) \\
\text{process } \text{name}(s) \cong \\
(MissionFW \\
[\text{mission_chan}(s) \cup \| \text{terminate } \|]
\text{name}(s) \text{App} \setminus \text{Mission_chan}(s))
\end{cases}
\end{align*}
mission_app : SCJMission ↦→ BasicProcess

∀ s : WF_S CJMission •

begin
  MArea ≜ «mission_memory(s)»
  «for each p : s.paragraphs of (N ≜ SCJSafeLetParameterisedAction) do»
    «N»Meth ≜ «translate_method_shared(name(s)ID, N, p.body, shared_variables(s))»
  «end»
  Init = start«name(s)»«cparams(s)» → «s.initial»
  initializeMeth ≜ initializeCall.id →
    «translate(s.initialize)»; initializeRet.id → Skip
  cleanUpMeth ≜ cleanUpCall.id →
    (var return : B • «s.cleanup.action»;
     cleanUpRet.id.return → Skip)

  Methods ≜ μX •
    initializeMeth: X
    □
    cleanUpMeth: X
    «for each p : s.paragraphs of (N ≜ A) do»
      □
      «N»Meth: X
    «end»
    □
    end_mission_app.«name(s)>>ID → Skip

  mission_app(s) =
    (initializeMeth; X
     □
cleanUpMeth; X
    «for each p : s.paragraphs of (N ≜ A) do»
      □
      «N»Meth; X
    «end»
     □
    end_mission_app.«name(s)>>ID → Skip)

  Step ≜
    ([1] | «mission_memory_CS(s)>> ∪ «end_mission_app >> | [1]])
    MArea
    \ «mission_memory_CS(s)>>
  • μX • Step; X □ terminate → Skip end
\[ \text{mission\_memory} : \text{SCJMission} \rightarrow \text{seq CircusParagraph} \]

\[
\forall m : \text{SCJMission} \bullet
\begin{align*}
\text{var} & \langle m.\text{state} \rangle \bullet \mu X \bullet \\
& \langle \text{for each } v : m.\text{state do} \rangle \\
& \quad \text{get}_v \langle \langle v.\text{name}\rangle \text{?}!\langle \text{name}(m)\rangle \text{?}!\langle \text{ID}\rangle \text{?}! v.\text{name} \rangle \rightarrow X \\
& \quad \square \\
& \quad \text{set}_v \langle \langle v.\text{name}\rangle \text{?}!\langle \text{name}(m)\rangle \text{?}!\langle \text{ID}\rangle \text{?}! v.\text{name} \rangle \rightarrow \langle v.\text{name} \rangle := x; X \\
& \quad \square \\
& \langle \text{end} \rangle \\
& \langle \text{end\_mission\_app.} \langle \text{name}(s)\rangle \text{ID} \rightarrow \text{Skip} \rangle \\
\end{align*}
\]

\[ \text{MissionCS} : \text{SCJMission} \rightarrow \text{ChannelSet} \]

\[
\forall s : \text{WF\_SCJMission} \bullet
\begin{align*}
\text{MissionCS}(s) &= \{ \text{getNextMissionCall} x.\langle \text{name}(s)\rangle \text{ID}, \text{getNextMissionRet} x.\langle \text{name}(s)\rangle \text{ID}, \\
& \quad \text{requestTerminationCall} x.\langle \text{name}(s)\rangle \text{ID}, \text{requestTerminationRet} x.\langle \text{name}(s)\rangle \text{ID}, \\
& \quad \text{terminationPendingCall} x.\langle \text{name}(s)\rangle \text{ID}, \text{terminationPendingRet} x.\langle \text{name}(s)\rangle \text{ID}, \\
& \quad \text{getSequencerCall} x.\langle \text{name}(s)\rangle \text{ID}, \text{getSequencerRet} x.\langle \text{name}(s)\rangle \text{ID}, \\
& \quad \langle \text{foreachp} : s.\text{paragraphs} \text{sof } (N \equiv A) \text{do} \rangle \\
& \quad \langle \text{N\text{Call}} x.\langle \text{name}(s)\rangle \text{ID}, \langle \text{N\text{Ret}} x.\langle \text{name}(s)\rangle \text{ID}, \rangle \\
& \quad \langle \text{for} \rangle \\
& \quad \langle \text{foreachh} : m.\text{handlers} \text{do} \rangle \\
& \quad \langle \text{start\text{h}} b.\text{name} \rangle \langle \text{name}(s)\rangle \text{ID}, \langle \text{N\text{Call}} x.\langle \text{name}(s)\rangle \text{ID}, \langle \text{N\text{Ret}} x.\langle \text{name}(s)\rangle \text{ID}, \rangle \\
& \quad \langle \text{for} \rangle \\
& \quad \langle \text{start\text{sequencer}} x.\langle \text{name}(s)\rangle \text{ID}, \text{start\text{mission}}.\langle \text{name}(s)\rangle \text{ID}.x, \\
& \quad \langle \text{done\text{mission}}.\langle \text{name}(s)\rangle \text{ID}.x, \text{register}, \text{start\text{peh}}, \text{start\text{aeh}}, \\
& \quad \langle \text{start\text{handlers}}, \text{stop\text{handlers}}, \text{done\text{handlers}}, \text{done\text{handler}}, \text{activate\text{handlers}} \mid x : \text{ID} \rangle \rangle \\
\end{align*}
\]
mission\text{ACCSMission} \equiv \text{begin}
\text{state}
[\text{shaft, speedo, throttle, cruise, engine, brake, gear, lever : ID}]
\text{initial} \equiv \text{Skip}
\begin{align*}
\text{shaft} & : = \text{new WheelShaft}(); \\
\text{speedo} & : = \text{new SpeedMonitor(shaft, 500)}(); \\
\text{throttle} & : = \text{new ThrottleController(speedo)}(); \\
\text{cruise} & : = \text{new Controller(throttle, speedo)}(); \\
\text{engine} & : = \text{new Engine(cruise)}(); \\
\text{brake} & : = \text{new Brake(cruise)}(); \\
\text{gear} & : = \text{new Gear(cruise)}(); \\
\text{lever} & : = \text{new Lever(cruise)}(); \\
\text{ shaft.register}(); \\
\text{ speedo.register}(); \\
\text{ throttle.register}(); \\
\text{ engine.register}(); \\
\text{ brake.register}(); \\
\text{ gear.register}(); \\
\text{ lever.register}();
\end{align*}
\text{initialize} \equiv
\begin{align*}
\text{lever} & : = \text{new Lever(cruise)}(); \\
\text{ shaft.register}(); \\
\text{ speedo.register}(); \\
\text{ throttle.register}(); \\
\text{ engine.register}(); \\
\text{ brake.register}(); \\
\text{ gear.register}(); \\
\text{ lever.register}();
\end{align*}
\text{cleanup} \equiv \text{res return : B • return} : = \text{true}
\text{end}

The result of applying the translation rules to the ACCS mission example results in the following processes.

\text{process\text{ACCSMission}\_App} \equiv \text{begin}
\[ MArea \equiv \text{var } shaft, \text{ speedo, throttle, cruise, engine, brake, gear, lever : ID } \cdot \mu X \cdot \]
\[
\begin{align*}
\text{get}_\text{shaft}?o!\text{ACCSMissionID}$&\Rightarrow \text{shaft} \rightarrow X \square \text{set}_\text{shaft}?o!\text{ACCSMissionID}$\Rightarrow \text{shaft} := x; X \\
\text{get}_\text{speedo}?o!\text{ACCSMissionID}$&\Rightarrow \text{speedo} \rightarrow X \square \text{set}_\text{speedo}?o!\text{ACCSMissionID}$\Rightarrow \text{speedo} := x; X \\
\text{get}_\text{throttle}?o!\text{ACCSMissionID}$&\Rightarrow \text{throttle} \rightarrow X \square \text{set}_\text{throttle}?o!\text{ACCSMissionID}$\Rightarrow \text{throttle} := x; X \\
\text{get}_\text{cruise}?o!\text{ACCSMissionID}$&\Rightarrow \text{cruise} \rightarrow X \square \text{set}_\text{cruise}?o!\text{ACCSMissionID}$\Rightarrow \text{cruise} := x; X \\
\text{get}_\text{engine}?o!\text{ACCSMissionID}$&\Rightarrow \text{engine} \rightarrow X \square \text{set}_\text{engine}?o!\text{ACCSMissionID}$\Rightarrow \text{engine} := x; X \\
\text{get}_\text{brake}?o!\text{ACCSMissionID}$&\Rightarrow \text{brake} \rightarrow X \square \text{set}_\text{brake}?o!\text{ACCSMissionID}$\Rightarrow \text{brake} := x; X \\
\text{get}_\text{gear}?o!\text{ACCSMissionID}$&\Rightarrow \text{gear} \rightarrow X \square \text{set}_\text{gear}?o!\text{ACCSMissionID}$\Rightarrow \text{gear} := x; X \\
\text{get}_\text{lever}?o!\text{ACCSMissionID}$&\Rightarrow \text{lever} \rightarrow X \square \text{set}_\text{lever}?o!\text{ACCSMissionID}$\Rightarrow \text{lever} := x; X \\
\text{end}_\text{mission}_\text{app}.\text{ACCSMissionID}$&\Rightarrow \text{Skip}
\end{align*}
\]

\[ Init \equiv \text{start}_\text{Mission}!\text{ACCSMissionID} \rightarrow \text{Skip} \]

\[ cleanUpMeth \equiv \text{cleanUp}_\text{Call}?x!\text{ACCSMissionID} \rightarrow \]
\[
\begin{align*}
\text{(var } \text{return : B } \cdot \text{return} := \text{true}; \text{ cleanUpRet}\!x!\text{ACCSMissionID}!\text{return} \rightarrow \text{Skip})
\end{align*}
\]
initializeMeth \equiv \text{missionInitializeCall}'x!\text{ACCSMissionID} \rightarrow
\begin{align*}
&\text{startWheelShaft!ACCSMissionID!WheelShaftID} \rightarrow \\
&\quad \text{set_shaft!ACCSMissionID!ACCSMissionID!WheelShaftID} ; \\
&\text{startSpeedMonitor!ACCSMissionID!SpeedMonitorID!shaft!500} \rightarrow \\
&\quad \text{set_speedo!ACCSMissionID!ACCSMissionID!SpeedMonitorID} ; \\
&\text{startThrottleController!ACCSMissionID!ThrottleControllerID!speedo} \rightarrow \\
&\quad \text{set_throttle!ACCSMissionID!ACCSMissionID!ThrottleControllerID} ; \\
&\text{startController!ACCSMissionID!ControllerID!throttle!speedo} \rightarrow \\
&\quad \text{set_cruise!ACCSMissionID!ACCSMissionID!ControllerID} ; \\
&\text{startEngine!ACCSMissionID!EngineID!cruise} \rightarrow \\
&\quad \text{set_engine!ACCSMissionID!ACCSMissionID!EngineID} ; \\
&\text{startBrake!ACCSMissionID!BrakeID!cruise} \rightarrow \\
&\quad \text{set_brake!ACCSMissionID!ACCSMissionID!BrakeID} ; \\
&\text{startGear!ACCSMissionID!GearID!cruise} \rightarrow \\
&\quad \text{set_gear!ACCSMissionID!ACCSMissionID!GearID} ; \\
&\text{startLever!ACCSMissionID!LeverID!cruise} \rightarrow \\
&\quad \text{set_lever!ACCSMissionID!ACCSMissionID!LeverID} ; \\
&\text{register!shaft} \rightarrow \text{Skip} ; \\
&\text{register!speedo} \rightarrow \text{Skip} ; \\
&\text{register!throttle} \rightarrow \text{Skip} ; \\
&\text{register!engine} \rightarrow \text{Skip} ; \\
&\text{register!brake} \rightarrow \text{Skip} ; \\
&\text{register!gear} \rightarrow \text{Skip} ; \\
&\text{register!lever} \rightarrow \text{Skip} ; \\
&\text{missionInitializeRet!'x!\text{ACCSMissionID}} \rightarrow \text{Skip} .
\end{align*}

Methods \equiv \mu X \left( \begin{array}{c}
\text{initializeMeth; } X \\
\text{cleanUpMeth; } X \\
\text{end\_mission\_app!ACCSMissionID} \rightarrow \text{Skip}
\end{array} \right)
Step ≡ (Init; Methods \\
\begin{align*}
\mu X \cdot \text{Step}; \ X \cap \text{terminate} \rightarrow \text{Skip}
\end{align*}
end

process MissionFW ≡ id : ID • begin
  state[sequencer : ID; handlers : \{ ID \}; terminating : B]

Init ≡ sequencer := null; handlers := \{\}; terminating := false

requestTerminationMeth ≡ requestTerminationCall?x!id →
  \begin{align*}
  &\text{if } \neg \text{terminating } \rightarrow (\text{terminating} := \text{true}; \text{stop handlers} \rightarrow \text{Skip}) \\
  &\text{terminationPendingMeth} \rightarrow \text{Skip}
  \end{align*}
fi

requestTerminationRet!x!id → Skip

terminationPendingMeth ≡ terminationPendingCall?x!id →
  terminationPendingRet!x!id!terminating → Skip

Methods ≡ µ X • \begin{align*}
\mu X \cdot \text{requestTerminationMeth}; \ X \\
\square \\
\text{terminationPendingMeth}; \ X \\
\square \\
\text{done handlers} \rightarrow \text{Skip}
\end{align*}

Initialize ≡ missionInitializeCall?x!id →
  µ X • \begin{align*}
  \text{register}?h \rightarrow \text{handlers} := \text{handlers} \cup \{h\}; \ X \\
  \square \\
  \text{missionInitializeRet!x!id} \rightarrow \text{Skip}
\end{align*}

StartHandlers ≡ (|||h : handlers • [1]\text{start_peh}b?s?p → Skip \square \text{start_aeh}b → Skip)
StopHandlers \equiv (\|b : \text{handlers} \bullet [\bullet \text{done_handler}, b \rightarrow \text{Skip}])

\[
\begin{align*}
\text{Execute} & \equiv \\
& \begin{cases}
\text{if } \text{handlers} = \{\} \rightarrow \text{Skip} \\
\quad \begin{cases}
\text{StartHandlers; activate_handlers} \rightarrow \\
\quad \text{stop_handlers} \rightarrow \text{StopHandlers} \\
\quad \| \text{stop_handlers, done_handlers} \|
\end{cases} \\
\quad \text{Methods}
\end{cases} \\
& \text{fi}
\end{align*}
\]

\[
\begin{align*}
\text{process ACCSMission} & \equiv \\
(M) \text{MissionFW} \\
\quad \| \| \text{cleanUpCall, cleanUpRet, missionInitializeCall, missionInitializeRet, terminate} \| \\
\quad \text{ACCSMission_App} \setminus \| \| \text{cleanUpCall, cleanUpRet, missionInitializeCall, missionInitializeRet} \|
\end{align*}
\]

\text{C.4 Periodic Event Handler}

\[
\begin{align*}
\text{\text{[\_]}_{\text{PEH}} : \text{SCJPeriodicHandler} \rightarrow \text{seq CircusParagraph}} \\
\forall s : \text{WF_SCJPeriodicHandler} \bullet \\
\quad [s]_{\text{PEH}} = \\
\quad \begin{cases}
\text{process «name(s)»}_{\text{App}} \equiv \langle \text{PEH_app(s)} \rangle \\
\text{process «name(s)»} \equiv \\
\quad (\text{PEHFW} \\
\quad \| \| \text{APEH-chan(s)} \| \| \text{terminate} \|) \\
\quad \langle \text{name(s)}\rangle_{\text{App}} \setminus \langle \text{PEH-chan(s)} \rangle
\end{cases}
\end{align*}
\]

97
**PEH_app**: SCJPeriodicHandler → BasicProcess

∀ s : WF_SCJPeriodicHandler •

\[
\begin{array}{l}
\text{begin} \\
\quad \text{state} \leftarrow s\.\text{state} \text{«s.state»} \\
\quad \text{«for each } p : s\.\text{paragraphs of } (N \equiv \text{SCJSafeletParametrisedAction}) \text{ do»} \\
\quad \quad \text{«N}\text{.Meth} \equiv \text{translate_method_shared(name(s)ID, N, p.body,} \\
\quad \quad \quad \quad \text{shared_variables(s))»} \\
\quad \text{«end»} \\
\quad \text{Init} = \text{translate_constructor(s.initial)} \\
\quad \text{handleAsyncEventMeth} \equiv \text{handleAsyncEventCall.id} \rightarrow \\
\quad \quad \text{«s.handleAsyncEvent»; handleAsyncEventRet.id} \rightarrow \text{Skip} \\
\quad \text{Methods} \equiv \mu X \bullet \\
\quad \quad \text{handleAsyncEventMeth; X} \\
\quad \quad \text{«for each } p : s\.\text{paragraphs of } (N \equiv A) \text{ do»} \\
\quad \quad \quad \Box «N\text{.Meth; X} \\
\quad \quad \text{«end»} \\
\quad \bullet \mu X \bullet \text{Init; Methods; X} \Box \text{terminate} \rightarrow \text{Skip} \\
\end{array}
\]

**PEHCS**: SCJPeriodicHandler → ChannelSet

∀ s : WF_SCJPeriodicHandler •

\[
\begin{array}{l}
\text{PEHCS}(s) = \| \text{handleAsyncEventCall.x, handleAsyncEventRet.x,} \\
\quad \text{«for each } p : s\.\text{paragraphs of } (N \equiv A) \text{ do»} \\
\quad \quad \text{«N}\text{.Call.x «name(s)ID», «N}\text{.Ret.x «name(s)ID»,} \\
\quad \quad \text{«for»} \\
\quad \quad \quad \text{done_handler.«name(s)ID», start «name(s)», start_peh.«name(s)ID»} \mid x : ID \|}
\end{array}
\]
periodic handler handlerSpeedMonitor ≡ begin
  start 0 period period
  state
    [calibration : Z,
     iterationsInOneHour : Z,
     cmInKilometer : Z,
     wheel_shaft : ID,
     numberRotations : Z,
     lastNumberRotations : Z,
     currentSpeed : Z]
  initial ≡ val shaft : ID; period : Z •
    (wheel_shaft := shaft;
     calibration := wheel_shaft.getCalibration();
     iterationsInOneHour := (3600 * 1000) div period
     getCurrentSpeed ≡ res return : Z • return := currentSpeed
  handleAsyncEvent ≡
    (numberRotations := wheel_shaft.getCount();
     var difference : Z • difference := numberRotations – lastNumberRotations;
     currentSpeed := (difference * calibration * iterationsInOneHour) div cmInKilometer;
     lastNumberRotations := numberRotations)
end

process SpeedMonitor_App ≡ begin

  state SpeedMonitorState
    calibration, iterationsInOneHour, cmInKilometer : Z
    wheel_shaft : ID
    numberRotations, lastNumberRotations, currentSpeed : Z
\[ \text{Init} \equiv \text{startSpeedMonitor} \cdot \text{start} \cdot \text{period} \rightarrow \text{start}_{\text{peb}} \cdot \text{SpeedMonitorID} \cdot \text{0!period} \rightarrow \]

\[
\left( \begin{array}{l}
\text{wheel} \_\text{shaft} := \text{shaft}; \\
\text{calibration} := \text{wheel} \_\text{shaft} \cdot \text{getCalibration}(); \\
\text{iterationsInOneHour} := (3600 \times 1000) \div \text{period}
\end{array} \right)
\]

\[ \text{handleAsyncEvent} \equiv \text{handleAsyncEventCall}!\text{SpeedMonitorID} \rightarrow \]

\[
\left( \begin{array}{l}
\text{numberRotations} := x; \\
\text{var difference} := \mathbb{Z} \cdot \text{difference} := \text{numberRotations} - \text{lastNumberRotations}; \\
\text{currentSpeed} := (\text{difference} \times \text{calibration} \times \text{iterationsInOneHour}) \div \text{cmInKilometer}; \\
\text{lastNumberRotations} := \text{numberRotations}
\end{array} \right)
\]

\[ \text{handleAsyncEventRet!SpeedMonitorID} \rightarrow \text{Skip} \]

\[ \text{getCurrentSpeedMeth} \equiv \text{getCurrentSpeedCall}?x!\text{SpeedMonitorID} \rightarrow \]

\[
(\text{var return} := \mathbb{Z} \cdot \text{return} := \text{currentSpeed}; \text{getCurrentSpeedCall}?x!\text{SpeedMonitorID}!\text{return} \rightarrow \text{Skip})
\]

\[ \text{Methods} \equiv \mu X \cdot \left( \begin{array}{l}
\text{handleAsyncEventMeth}; X
\end{array} \right)
\]

\[ \Box \]

\[ \text{getCurrentSpeedMeth}; X
\]

\[ \Box \]

\[ \text{done_handler!SpeedMonitorID} \rightarrow \text{Skip} \]

\[ \mu X \cdot \left( \begin{array}{l}
\text{Init}; \text{Methods}; X
\end{array} \right)
\]

\[ \Box \]

\[ \text{terminate} \rightarrow \text{Skip} \]

end
process \( \text{PEHFW} \triangleq id : \text{ID} \bullet \begin{align*} &\begin{align*} \text{state } \text{PEHFWState} &= [\text{start}, \text{period} : \mathbb{N}] \\ \text{Execute} &\triangleq \text{wait} \text{start}; \\ &\begin{pmatrix} \mu X \bullet \begin{pmatrix} \begin{pmatrix} \mu Y \bullet (\text{handleAsyncEventCall}!id \rightarrow \text{handleAsyncEventRet}!id \rightarrow \text{Skip}) \triangleright \text{period}; X \end{pmatrix} \\ \text{done_handler}!id \rightarrow \text{Skip} \end{pmatrix} \\ \{1\} | \{\text{handleAsyncEventCall}!id, \text{done_handler}!id\} | \{1\} \end{pmatrix} \end{align*} \end{align*} \end{align*}

end

process \( \text{SpeedMonitor} \triangleq \begin{align*} &\begin{align*} (\text{PEHFW}(\text{SpeedMonitorID}) \\ \{\text{handleAsyncEventCall, handleAsyncEventRet, done_handler, start_peh, terminate}\} \\ \text{SpeedMonitor}_\text{App}) \setminus \{\text{handleAsyncEventCall, handleAsyncEventRet}\} \end{align*} \end{align*} \end{align*}

C.5 Aperiodic Event Handler

\[
\forall s : \text{WF.SCJAperiodicHandler} \bullet
\begin{align*}
&\begin{pmatrix} \mu s.APEH = \begin{pmatrix} \mu s.APEH.FW \begin{pmatrix} \mu \text{APEH.chan}(s) \cup \{\text{terminate}\} \\ \text{name(s)}\_\text{App} \setminus \text{APEH.chan}(s) \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} \text{process } \text{name(s)}\_\text{App} \triangleq \text{APEH.app(s)} \\
\text{process } \text{name(s)} \equiv \text{APEHFW} \\
\end{pmatrix} \end{pmatrix}
\end{align*}
\]
APEH_app : SCJApaperidicHandler → BasicProcess

∀ s : WF_SCJApaperidicHandler •

begin
  state «s.state»
  «for each p : s.paragraphs of (N ≡ SCJSafeletParametrisedAction) do»
    «N»Meth ≡ «translate_method_shared(name(s)ID, N, p.body,
    shared_variables(s))»
  «end»
  Init = «translate_constructor(s.initial)»
  handleAsyncEventMeth ≡ handleAsyncEventCall.id→
  «s.handleAsyncEvent»; handleAsyncEventRet.id → Skip
  Methods ≡ μ X •
    handleAsyncEventMeth; X
  «for each p : s.paragraphs of (N ≡ A) do»
    □ «N»Meth; X
  «end»
  • μ X • Init; Methods; X □ terminate → Skip
end

APEHCS : SCJApaperidicHandler → ChannelSet

∀ s : WF_SCJApaperidicHandler •

APEHCS(s) = [] handleAsyncEventCall.x.<name(s)ID>, handleAsyncEventRet.x.<name(s)ID>,
  «for each p : s.paragraphs of (N ≡ A) do»
    «N»Call.x.<name(s)ID>, «N»Ret.x.<name(s)ID>,
  «for»
    done_handler.<name(s)ID>, start.<name(s)>, start_aeh.<name(s)ID> | x : ID []

aperiodic handler handler WheelShaft ≡ begin
  state count : Z
  initial ≡ count := 0
  handleAsyncEvent ≡ count := count + 1
  getCallibration ≡ res return : Z • return := 100
  getCount ≡ res return : Z • return := count
end
Applying the translation rules to the aperiodic event handler WheelShaft, we obtain the following processes.

\[
\text{process \em WheelShaft\_App} \triangleq \text{begin} \\
\text{state} S = [\text{count} : \mathbb{Z}] \\
\text{Init} \triangleq \text{startWheelShaft}!\text{WheelShaftID} \rightarrow \text{start\_aeh}!\text{WheelShaftID} \rightarrow \text{count} := 0
\]

\[
\text{handleAsyncEventMeth} \triangleq \text{handleAsyncEventCall}!\text{WheelShaftID} \rightarrow \\
\text{count} := \text{count} + 1; \text{handleAsyncEventRet}!\text{WheelShaftID} \rightarrow \text{Skip}
\]

\[
\text{getCalibrationMeth} \triangleq \text{getCalibrationCall}!\text{WheelShaftID} \rightarrow \\
(\text{var return} : \mathbb{Z} \cdot \text{return} := 100; \text{getCalibrationRet}!\text{WheelShaftID}!\text{return} \rightarrow \text{Skip})
\]

\[
\text{getCountMeth} \triangleq \text{getCountCall}!\text{WheelShaftID} \rightarrow \\
(\text{var return} : \mathbb{Z} \cdot \text{return} := \text{count}; \text{getCountRet}!\text{WheelShaftID}!\text{return} \rightarrow \text{Skip})
\]

\[
\text{Methods} \triangleq \mu X \cdot \\
\quad \begin{cases} 
\text{handleAsyncEventMeth}: X \\
\text{getCalibrationMeth}: X \\
\text{getCountMeth}: X \\
\text{done\_handler}!\text{WheelShaftID} \rightarrow \text{Skip}
\end{cases}
\]

\[
\mu X \cdot \text{Init}; \text{Methods}; X \triangleright \text{terminate} \rightarrow \text{Skip}
\]

end
process APEHFW ≡ id : ID • begin
state S == [currentRelease : B; newRelease : B]
releaseMeth ≡ µ X • 
\[
\left(\begin{array}{c}
\text{currentRelease} = \text{false} \land \text{releaseCall} \circ \text{id} \rightarrow \text{currentRelease} := \text{true}; \text{releaseRet} \circ \text{id} \rightarrow \text{Skip}); \ X \\
\text{currentRelease} = \text{true} \land \text{releaseCall} \circ \text{id} \rightarrow \text{newRelease} := \text{true}; \text{releaseRet} \circ \text{id} \rightarrow \text{Skip}); \ X \\
\text{currentRelease} = \text{true} \land \text{hasRelease} \rightarrow \text{currentRelease} := \text{newRelease}); \ X \\
\text{done\_handler} \circ \text{id} \rightarrow \text{Skip}
\end{array}\right)
\]

\text{treatReleases} ≡ µ X • \[
\left(\begin{array}{c}
\text{hasRelease} \rightarrow \text{handleAsyncEventCall} \circ \text{id} \circ \text{id} \rightarrow \text{handleAsyncEventRet} \circ \text{id} \circ \text{id} \rightarrow \ X \\
\text{done\_handler} \circ \text{id} \rightarrow \text{Skip}
\end{array}\right)
\]

Methods =
\(\bar{\text{releaseMeth}}\) \[
\left(\begin{array}{c}
\text{currentRelease, newRelease} \lor \text{hasRelease, done\_handler} \lor \text{1}
\end{array}\right)
\]
\(\text{treatReleases} \lor \text{hasRelease} \lor \text{1}\)
\[
\left(\begin{array}{c}
\text{start\_aeh} \circ \text{id} \rightarrow \text{activate\_handlers} \rightarrow \text{Methods}; \ Y \\
\text{activate\_handlers} \rightarrow \ Y \\
\text{terminate} \rightarrow \text{Skip}
\end{array}\right)
\]

process WheelShaft ≡ (APEHFW(WheelShaftID))
\(\bar{\text{start\_aeh, done\_handler, handleAsyncEventCall, handleAsyncEventRet, terminate} \lor \text{1}}\)
WheelShaft)
process MyApp ≡

\[
\begin{align*}
&\text{ACCSSafelet} \\
&\quad [[\text{startSafelet}, \text{getSequencerCall}, \text{getSequencerRet}, \text{start_sequencer}, \text{done_sequencer}, \text{terminate}]] \\
&\quad \text{ACCSMissionSequencer} \\
&\quad [[\text{getNextMissionCall}, \text{getNextMissionRet}, \text{start_mission}, \text{done_mission}, \text{terminate}]] \\
&\quad \text{ACCSMission} \\
&\quad [[\text{register}, \text{start_peh}, \text{start_aeh}, \text{done_handler}, \text{terminate}]] \\
&\quad \text{WheelShaft} \\
&\quad [[\text{activate_handlers}, \text{terminate}, \text{release}.x.y, \text{release}.y.x | x = \text{WSID} \land y \in \{\text{SMID, TCID, CID, EID, BID, GID, LID}\}]] \\
&\quad \text{SpeedMonitor} \\
&\quad [[\text{activate_handlers}, \text{terminate}, \text{release}.x.y, \text{release}.y.x | x = \text{SPID} \land y \in \{\text{TCID, CID, EID, BID, GID, LID}\}]] \\
&\quad \text{ThrottleController} \\
&\quad [[\text{activate_handlers}, \text{terminate}, \text{release}.x.y, \text{release}.y.x | x = \text{TCID} \land y \in \{\text{CID, EID, BID, GID, LID}\}]] \\
&\quad \text{Controller} \\
&\quad [[\text{activate_handlers}, \text{terminate}, \text{release}.x.y, \text{release}.y.x | x = \text{CID} \land y \in \{\text{EID, BID, GID, LID}\}]] \\
&\quad \text{Engine} \\
&\quad [[\text{activate_handlers}, \text{terminate}, \text{release}.x.y, \text{release}.y.x | x = \text{EID} \land y \in \{\text{BID, GID, LID}\}]] \\
&\quad \text{Brake} \\
&\quad [[\text{activate_handlers}, \text{terminate}, \text{release}.x.y, \text{release}.y.x | x = \text{BID} \land y \in \{\text{GID, LID}\}]] \\
&\quad \text{Gear} \\
&\quad [[\text{activate_handlers}, \text{terminate}, \text{release}.x.y, \text{release}.y.x | x = \text{G} \land y \in \{\text{LID}\}]] \\
&\quad \text{Lever}
\end{align*}
\]
\[
\text{translate}_\text{call} : \text{target} \times \mathbb{N} \times (\text{seqExpression}) \mapsto \text{Action}
\]

\[
\forall id : \text{ID}; \ n : \mathbb{N}; \ es : \text{seqExpression} \mid (\text{ID.N}(es) \in \text{WF}_\text{Call}) \bullet
\text{translate}_\text{call}(id, n, es) =
\begin{cases}
\begin{array}{l}
\text{«if operation}(n, id).\text{returnType} = \text{Void then}» \\
\text{«n}!\text{Call!}\langle id \rangle\!\langle \_ \rangle : es \mapsto \text{«n}!\text{Ret!}\langle id \rangle \mapsto \text{Skip}» \\
\text{«else»} \\
\text{«n}!\text{Call!}\langle id \rangle\!\langle \_ \rangle : es \mapsto \text{«n}!\text{Ret!}\langle id \rangle?x \mapsto \text{return} := x»
\end{array}
\end{cases}
\]

\[
\text{Action} : \text{Action} \mapsto \text{Action}
\]

\[
\forall a : \text{Action} \bullet
\begin{cases}
\begin{array}{l}
\text{«if a instanceof Call then»} \\
\text{«if a.object.id} \in (\text{SafeletID} \cup \text{MissionID} \cup \text{SequencerID} \cup \text{HandlerID}) \text{then}» \\
\text{«translate}_\text{call}(a.object.id, a.name, a.parameters)» \\
\text{«else a»} \\
\text{«fi»}
\end{array}
\end{cases}
\]

\[
\begin{cases}
\begin{array}{l}
\text{«else a»} \\
\text{«fi»}
\end{array}
\end{cases}
\]

106
Examples

D.1 ACCS in SCJ Circus

safelet $\text{ACCSafelet} \triangleq \text{begin}$
  $\text{initial} \triangleq \text{Skip}$
  $\text{getSequencer} \triangleq \text{res return : ID \bullet return := newACCSMissionSequencer()}$
  $\text{initialize} \triangleq \text{Skip}$
end

$\text{sequencer} \text{ACCSMissionSequencer} \triangleq \text{begin}$
  $\text{mission\_done} : \mathbb{B}$
  $\text{initial} \triangleq \text{mission\_done} := \text{false}$
  $\text{getNextMission} \triangleq \text{res return : ID \bullet}$
    $\begin{cases} 
    \text{if} \neg \text{mission\_done} \rightarrow \text{mission\_done} := \text{true}; \text{return := newACCSMission()} \\
    \text{fi} 
    \end{cases}$
end
mission \( \text{ACCSMission} \) \( \triangleq \) begin
  state
  \[ \text{[shaft, speedo, throttle, cruise, engine, brake, gear, lever : ID]} \]
  initial \( \triangleq \) Skip
    \( \begin{align*}
      \text{shaft} & \text{ := new WheelShaft();} \\
      \text{speedo} & \text{ := new SpeedMonitor(shaft, 500);} \\
      \text{throttle} & \text{ := new ThrottleController(speedo);} \\
      \text{cruise} & \text{ := new Controller(throttle, speedo);} \\
      \text{engine} & \text{ := new Engine(cruise);} \\
      \text{brake} & \text{ := new Brake(cruise);} \\
      \text{gear} & \text{ := new Gear(cruise);} \\
    \end{align*} \)
  initialize \( \triangleq \) lever := new Lever(cruise);
    shaft.register();
    speedo.register();
    throttle.register();
    engine.register();
    brake.register();
    gear.register();
    lever.register();
  cleanup \( \triangleq \) res return : B • return := true
end

aperiodic handler handler \( \text{WheelShaft} \) \( \triangleq \) begin
  state \( \text{count : Z} \)
  initial \( \triangleq \) count := 0
  handleAsyncEvent \( \triangleq \) count := count + 1
  getCallibration \( \triangleq \) res return : Z • return := 100
  getCount \( \triangleq \) res return : Z • return := count
end
periodic handler handlerSpeedMonitor ≡ begin
  start 0 period period
  state
    [calibration : Z,
     iterationsInOneHour : Z,
     cmInKilometer : Z,
     wheel_shaft : ID,
     numberRotations : Z,
     lastNumberRotations : Z,
     currentSpeed : Z]
  initial ≡ val shaft : ID; period : Z •
    (wheel_shaft := shaft;
     calibration := wheel_shaft.getCalibration();
     iterationsInOneHour := (3600 * 1000) div period)
  handleAsyncEvent ≡ res return : Z • return := currentSpeed
    (numberRotations := wheel_shaft.getCount();
     var difference : Z • difference := numberRotations − lastNumberRotations;
     currentSpeed := (difference * calibration * iterationsInOneHour) div cmInKilometer;
     lastNumberRotations := numberRotations)
end
periodic handler handler ThrottleController ≜ begin
    start 0 period THROTTLE_PERIOD
    state
        THROTTLE_PERIOD : ℤ := 100
        VOLTAGE_INCREMENT : float := 0.01
        speedo : ID
        schedule,brottle : bool := false
        accelerating : bool := false
        maintainSpeed : bool := false
        cruiseSpeed : ℤ := 0
        voltage : float := 0.0
    initial ≜ val speedMonitor : ID • speedo = speedMonitor
    setCruiseSpeed ≜ val kph : ℤ •
        cruiseSpeed := kph; maintainSpeed := true; accelerating := false
    accelerate ≜ accelerating := true
    writeVoltage ≜ ( var Portio_port • io_port = newPort(PORTA); io_port.setVoltage(voltage * 10) );
    increaseVoltage ≜
        ( if voltage ≤ 8 → voltage := voltage + VOLTAGE_INCREMENT; voltage := min(voltage, 8) )
        ( if (voltage ≤ 8) → Skip )
    writeVoltage()
    decreaseVoltage ≜
        ( if voltage ≥ 0 → voltage := voltage - VOLTAGE_INCREMENT; voltage := max(voltage, 0) )
        ( if (voltage ≥ 0) → Skip )
    writeVoltage()
    resetVoltage ≜ voltage := 0; writeVoltage()
    schedulePeriodic ≜ schedule_throttle := true
    deschedulePeriodic ≜ schedule_throttle := false
handleAsyncEvent ≡

\[
\begin{align*}
\text{if } \text{schedule_throttle} & \rightarrow (\text{if } \text{accelerating} \rightarrow \text{increaseVoltage}() \\
& \text{fi}) \\
\text{fi} \\
\text{if } \text{maintainSpeed} & \rightarrow \\
\text{var} \ currentSpeed : \mathbb{Z} \cdot currentSpeed := \text{speedo}.\text{getCurrentSpeed}(); \\
\text{if}(\text{cruiseSpeed} - currentSpeed > 2) & \rightarrow \text{increaseVoltage()} \\
\text{fi} \\
\text{if}(\text{cruiseSpeed} - currentSpeed < -2) & \rightarrow \text{resetVoltage()} \\
\text{fi} \\
\text{if}(\text{cruiseSpeed} - currentSpeed \leq 2) \land (\text{cruiseSpeed} - currentSpeed \geq -2) & \rightarrow \\
\text{var} \ volts : \text{float} \cdot volts := 2 \times (\text{cruiseSpeed} - currentSpeed + 2); \\
\text{if}(volts > \text{voltage}) & \rightarrow \text{increaseVoltage()} \\
\text{fi} \\
\text{fi} \\
\text{fi} \\
\text{fi} \\
\text{end}
\end{align*}
\]

PORTA == 0x020
PORTB == 0x021

class Port ≡ begin
state PortState == [VOLTAGE : \mathbb{Z}; Address : \mathbb{Z}]
initial ≡ val address : \mathbb{Z} \cdot Address := address
public getVoltage ≡ res return : \mathbb{Z} \cdot return := VOLTAGE
public setVoltage ≡ val voltage : \mathbb{Z} \cdot VOLTAGE := voltage
end

class Controller ≡ begin
**state** ControllerState

throttle : ID
speedo : ID
engineActive : Bool
topGear : Bool
braking : Bool
accelerating : Bool
cruising : Bool
throttleStarted : Bool

engineActive = false
topGear = false
braking = false
accelerating = false
cruising = false
throttleStarted = false

**initial** ≜ val throttle : ID; speedo : ID •
  this.throttle := throttle; this.speedo := speedo

**public** engineOn ≜ engineActive := true; braking := false; topGear := false; cruising := false

**public** engineOff ≜ engineActive := false; braking := false; topGear := false;
  (if cruising → cruising := false; throttle.deschedulePeriodic() )
  fi

fi
public brakePressed \( \triangleq \) 
\[
\begin{align*}
& \text{if engineActive } \rightarrow \\
& \quad \text{if cruising } \rightarrow \text{cruising} := \text{false}; \text{throttle.deschedulePeriodic()} \\
& \quad \text{else } \rightarrow \text{Skip} \\
& \quad \text{fi} \\
& \text{braking} := \text{true} \\
& \text{else } \rightarrow \text{Skip} \\
& \text{fi}
\end{align*}
\]

public brakeReleased \( \triangleq \) 
\[
\begin{align*}
& \text{if engineActive } \rightarrow \text{braking} := \text{false} \\
& \text{else } \rightarrow \text{Skip} \\
& \text{fi}
\end{align*}
\]

public topGearEngaged \( \triangleq \) 
\[
\begin{align*}
& \text{if engineActive } \rightarrow \text{topGear} := \text{true} \\
& \text{else } \rightarrow \text{Skip} \\
& \text{fi}
\end{align*}
\]

public topGearDisengaged \( \triangleq \) 
\[
\begin{align*}
& \text{if engineActive } \rightarrow \text{topGear} := \text{false}; \\
& \quad \text{if cruising } \rightarrow \text{cruising} := \text{false}; \\
& \quad \text{throttle.deschedulePeriodic()} \\
& \quad \text{else } \rightarrow \text{Skip} \\
& \quad \text{fi} \\
& \text{else } \rightarrow \text{Skip} \\
& \text{fi}
\end{align*}
\]

113
public activate \triangleq \begin{cases} 
\text{if}(\text{engineActive} \land \text{topGear} \land \neg \text{braking}) \rightarrow \\
\text{cruising} := \text{true}; \\
\text{var cruise\_speed : } \mathbb{R} \\
\text{cruise\_speed} := \text{speedo.get\_Current\_Speed}(); \\
\text{throttle.set\_Cruise\_Speed}(*\text{cruise\_speed}); \\
\text{throttle.schedule\_Periodic}(); \\
\text{throttle\_Started} := \text{true} \\
\end{cases} \\
\text{fi} \\
\text{\textbf{if}}(\text{engineActive} \land \text{topGear} \land \neg \text{braking}) \rightarrow \text{Skip} \end{cases}

public deactivate \triangleq \begin{cases} 
\text{if}(\text{engineActive} \land \text{topGear} \land \neg \text{braking} \land \text{cruising}) \rightarrow \\
\text{cruising} := \text{false}; \\
\text{throttle.deschedule\_Periodic}() \\
\end{cases} \\
\text{\textbf{fi}} \\
\text{\textbf{if}}(\text{engineActive} \land \text{topGear} \land \neg \text{braking} \land \text{cruising}) \rightarrow \text{Skip} \end{cases}

public startAccelerating \triangleq \begin{cases} 
\text{if}(\text{engineActive} \land \text{topGear} \land \neg \text{braking}) \rightarrow \\
\text{accelerating} := \text{true}; \\
\text{\textbf{if}}(\text{throttle\_Started}) \rightarrow \text{throttle.schedule\_Periodic}() \\
\end{cases} \\
\text{\textbf{fi}} \\
\text{\textbf{if}}(\text{engineActive} \land \text{topGear} \land \neg \text{braking}) \rightarrow \text{Skip} \end{cases}
public stopAccelerating ≝
( if(engineActive ∧ topGear ∧ ¬ braking ∧ accelerating) →
  accelerating := false; cruising := true;
  (var aux : Z • aux := speedo.getCurrentSpeed(); throttle.setCruiseSpeed(aux))
  ¬ (engineActive ∧ topGear ∧ ¬ braking ∧ accelerating) → Skip
) fi

public resume ≝
( if(topGear ∧ ¬ braking ∧ throttleStarted) →
  cruising := true; throttle.schedulePeriodic()
  ¬ (topGear ∧ ¬ braking ∧ throttleStarted) → Skip
) fi

end

ENGINE_OFF == 1
ENGINE_ON == 2
BRAKE_ON == 3
BRAKE_OFF == 4
TOP_GEAR_ENGAGED == 5
TOP_GEAR_DISENGAGED == 6
ACTIVATE == 7
DEACTIVATE == 8
START_ACCELERATING == 9
STOP_ACCELERATING == 10
RESUME == 11
aperiodic handler handler\(\text{Engine} \triangleq \text{begin}\)

\text{state EngineState} == [cruise : ID; event : N]
\text{initial} \triangleq \text{val} cruise : ID \cdot this.cruise := cruise

\text{handleAsyncEvent} \triangleq \begin{cases} 
\text{if} \ \text{event} = \text{ENGINE\_ON} \rightarrow \text{cruise}.\text{engineOn}() \\
\text{if} \ \text{event} = \text{ENGINE\_OFF} \rightarrow \text{cruise}.\text{engineOff}() \\
\text{fi}
\end{cases}
\text{event} = \text{ENGINE\_ON} \lor \text{event} = \text{ENGINE\_OFF} \rightarrow \text{Skip}

\text{setEvent} \triangleq \text{val} event : N \cdot this.event := event
\text{end}

aperiodic handler handler\(\text{Brake} \triangleq \text{begin}\)

\text{state BrakeState} == [cruise : ID; event : N]
\text{initial} \triangleq \text{val} cruise : ID \cdot this.cruise := cruise

\text{handleAsyncEvent} \triangleq \begin{cases} 
\text{if} \ \text{event} = \text{BRAKE\_ON} \rightarrow \text{cruise}.\text{brakePressed}() \\
\text{if} \ \text{event} = \text{BRAKE\_OFF} \rightarrow \text{cruise}.\text{brakeReleased}() \\
\text{fi}
\end{cases}
\text{event} = \text{BRAKE\_ON} \lor \text{event} = \text{BRAKE\_OFF} \rightarrow \text{Skip}

\text{setEvent} \triangleq \text{val} event : N \cdot this.event := event
\text{end}

aperiodic handler handler\(\text{Gear} \triangleq \text{begin}\)

\text{state GearState} == [cruise : ID; event : N]
\text{initial} \triangleq \text{val} cruise : ID \cdot this.cruise := cruise

\text{handleAsyncEvent} \triangleq \begin{cases} 
\text{if} \ \text{event} = \text{TOP\_GEAR\_ENGAGED} \rightarrow \text{cruise}.\text{topGearEngaged}() \\
\text{if} \ \text{event} = \text{TOP\_GEAR\_DISENGAGED} \rightarrow \text{cruise}.\text{topGearDisengaged}() \\
\text{fi}
\end{cases}
\text{event} = \text{TOP\_GEAR\_ENGAGED} \lor \text{event} = \text{TOP\_GEAR\_DISENGAGED} \rightarrow \text{Skip}

\text{setEvent} \triangleq \text{val} event : N \cdot this.event := event
\text{end}
aperiodic handler handlerLever ≜ begin
  state LeverState == [cruise : ID; event : ℕ]
  initial ≜ val cruise : ID • this.cruise := cruise
  handleAsyncEvent ≜
    \ if event = ACTIVATE -> cruise.activate()
    \ event = DEACTIVATE -> cruise.deactivate()
    \ event = START_ACCELERATING -> cruise.startAccelerating()
    \ event = STOP_ACCELERATING -> cruise.stopAccelerating()
    \ event = RESUME -> cruise.resume()
    \ event ∉ {ACTIVATE, DEACTIVATE, START_ACCELERATING, STOP_ACCELERATING, RESUME} -> Skip 
    fi
  setEvent ≜ val event : ℕ • this.event := event
end

D.2 Simple Example SCJ Level 1

channel startMyPeriodicEvh : ID.ID.ID
channel handleAsyncEventCall, handleAsyncEventRet : ID.ID

periodic handler MyPeriodicEvh ≜ begin
  state S
    aevh : ID
    count : ℤ
  start 0 period 1000
  initial ≜ val handler : ID • count := 0; aevh := handler
```
handleAsyncEvent \( \triangleq \) count := count + 1; 
\( \begin{cases} 
\text{if count} = 5 \rightarrow \text{aevh.release(); count} := 0 \\
\[\text{otherwise}\] 
\text{if count} \neq 5 \rightarrow \text{Skip} \\
\end{cases} \)

end

channel startMyAperiodicEvh : ID.ID.ID

aperiodic handler MyAperiodicEvh \( \triangleq \) begin

<table>
<thead>
<tr>
<th>state S</th>
</tr>
</thead>
<tbody>
<tr>
<td>missSeq : ID</td>
</tr>
<tr>
<td>count : Z</td>
</tr>
</tbody>
</table>

initial \( \triangleq \) val sequencer : ID \( \bullet \) missSeq := sequencer; count := 0

handleAsyncEvent \( \triangleq \) count := count + 1; 
\( \begin{cases} 
\text{if count} = 2 \rightarrow \text{missSeq.requestSequenceTermination()} \\
\[\text{otherwise}\] 
\text{if count} \neq 2 \rightarrow \text{Skip} \\
\end{cases} \)

end

channel startMyMission : ID.ID.ID

missionMyMission \( \triangleq \) begin
```

118
state \( S \)

\[ \text{missSeq} : ID \]

\[
\text{initial} \triangleq \text{val \ sequencer} : ID \bullet \text{missSeq} := \text{sequencer}
\]

\[
\text{initialize} \triangleq \text{var \ aevh, pevh} \bullet
\begin{align*}
\text{aevh} & \text{: newMyAperiodicEvh(\text{missSeq}); aevh\text{.register();}} \\
\text{pevh} & \text{: newMyPeriodicEvh(\text{aevh}); pevh\text{.register();}}
\end{align*}
\]

\text{end}

\[
\text{sequencerMySequencer} \triangleq \text{begin}
\begin{align*}
\text{state} \; S & \; = \; [\text{\mission} : \text{ID}] \\
\text{initial} & \text{: mission} \; := \; \text{newMyMission(this)}
\end{align*}
\]

\[
\text{getNextMission} \triangleq \text{res return} : ID \bullet \text{var aux} : \mathbb{B} \bullet \text{aux} := \text{mission\text{.terminationPending();}}
\begin{align*}
\text{if aux} \rightarrow \text{return} & \; := \; \text{null} \\
\rightarrow \text{aux} & \rightarrow \text{return} \; := \; \text{mission}
\end{align*}
\]

\text{end}

\[
\text{safeletMyApp} \triangleq \text{begin}
\begin{align*}
\text{initialize} & \; := \; \text{Skip} \\
\text{getSequencer} & \; := \; \text{res return} : ID \bullet \text{return} \; := \; \text{newMySequencer()}
\end{align*}
\]

\text{end}
D.2.1 E-Anchor

\[\text{process } \text{System} \equiv \begin{align*}
\text{state } S &= [\text{count}_1 : N; \text{count}_2 : N] \\
\text{PeriodicHandler} &\equiv \text{count}_1 := 0; \\
&\mu X \bullet \\
&\begin{cases}
\text{wait} 0..TB; \text{count}_1 := \text{count}_1 + 1; \\
\text{if } \text{count}_1 = 5 \rightarrow \text{release} \rightarrow \text{count}_1 := 0 \\
\text{if } \text{count} \neq 5 \rightarrow \text{Skip} \\
\text{stop} \text{_handlers} \rightarrow \text{Skip}
\end{cases}; \text{sync} \rightarrow X
\end{align*}
\]

\[\text{AperiodicHandler} \equiv \text{count}_2 := 0; \\
\mu X \bullet \\
\begin{cases}
\text{release} \rightarrow \text{count}_2 := \text{count}_2 + 1 \\
\text{if } \text{count}_2 = 2 \rightarrow \text{terminate} \rightarrow \text{Skip} \\
\text{if } \text{count} \neq 2 \rightarrow \text{Skip} \\
\text{stop} \text{_handlers} \rightarrow \text{Skip}
\end{cases}; \text{sync} \rightarrow X
\]

\[\text{Cycles} \equiv \mu X \bullet \text{wait} 1000; \text{sync} \rightarrow X
\]

\[\text{Handlers} = \begin{cases}
\text{PeriodicHandler} \\
\text{AperiodicHandler} \\
\text{Cycles}
\end{cases}\]

\[\text{Termination} \equiv \text{terminate} \rightarrow \text{Stopping}
\]

\[\text{Stopping} \equiv \mu X \bullet (\text{terminate} \rightarrow X \Box \text{stop} \text{_handlers} \rightarrow X)
\]

\[\text{Mission} = \begin{cases}
\text{Termination} \\
\text{Handler}
\end{cases}\]

D.2.2 Extra examples

Cyclic in lockstep

\[M \text{ interarrival time for aperiodic event } A D \text{ deadline for aperiodic task } AT B \text{ budget for aperiodic task}\]
Constraints: $PTB + INP_{DL} \leq PD, PTB \leq PD \leq PERIOD$

**channel input : N**
**channel print : String**

**process System1 \equiv begin**

\[
\begin{align*}
    three_0 & \equiv P(seq N) \\
    \forall s : seq N \cdot s \in three_0 \iff (\#s \geq 3 \land (0,0,0) \text{ suffix } s)
\end{align*}
\]

**MArea \equiv \text{var buffer : seq } N \cdot \mu X \cdot**

\[
\begin{align*}
    \text{setBuffer?x} & \rightarrow buffer := x; \ X \\
    \square & \\
    \text{getBuffer!buffer} & \rightarrow X \\
    \square & \\
    \text{stop} & \rightarrow \text{Skip}
\end{align*}
\]

**PeriodicHandler \equiv**

\[
\begin{align*}
    \mu X \cdot \\
    \begin{cases} 
        \begin{cases} 
            \begin{cases} 
                \begin{cases} 
                    (\text{input?x} \rightarrow (\text{wait } 0..PTB)) \leftrightarrow INP_{DL}; \\
                    \text{setBuffer!(buffer \concat (x))} \rightarrow \text{Skip}; \ release \rightarrow \text{Skip}
                \end{cases} \\
                \text{setBuffer?x} \rightarrow buffer := x; \ X \\
            \end{cases} \\
            \text{wait PERIOD} \\
            \square \\
            \text{stop} \rightarrow \text{Skip}
        \end{cases} \\
    \end{cases}
\end{align*}
\]

The pattern in the paper suggests that using `wait m` to guarantee the aperiodic event handler’s interarrival time. However, in this case, we want the aperiodic event handler to be in lock step, since there is no restriction on how long the handler waits for the release, one execution could take longer
than the period.

\[
\text{AperiodicHandler} \equiv \\
\mu X \bullet \left( \\
\begin{array}{l}
\text{release} \rightarrow \text{getBuffer?buffer} \rightarrow \text{wait 0..ATB;}
\text{if buffer} \in \text{three}_0 \rightarrow \text{print!true} \rightarrow \text{Skip} \\
\text{or buffer} \notin \text{three}_0 \rightarrow \text{print!false} \rightarrow \text{Skip}
\end{array}
\right) \rightarrow \text{PERIOD; X}
\]

\[
\text{PeriodicHandler} \\
\text{Handlers} = \left( [1] \parallel \text{stop, release} \parallel [1] \right) \parallel \text{release} \parallel \text{AperiodicHandler}
\]

\[
\text{Termination} \equiv \text{terminate} \rightarrow \text{Stopping}
\]

\[
\text{Stopping} \equiv \mu X \bullet (\text{terminate} \rightarrow X \parallel \text{stop} \rightarrow X)
\]

\[
\text{Mission} = \left( \begin{array}{l}
\text{Termination} \\
\text{Handlers} \\
\text{setBuffer, getBuffer} \parallel [1]
\end{array} \right) \parallel \text{setBuffer, getBuffer} \parallel \text{MArea}
\]

end

\text{safetletSystem1} \equiv \begin{aligned}
\text{initial} & \equiv \text{Skip} \\
\text{getSequencer} & \equiv \text{return : ID} \bullet \text{return} := \text{newMissionSequencer}() \\
\text{initialize} & \equiv \text{Skip} \\
\text{end}
\end{aligned}
process System1_App ≜ begin
  getSequencerMeth ≜ getSequencerCall!x!SID → (var return : ID •
    startMissionSequencer!SID!MSID → return := MSID;
    getSequencerRet!x!SID!return → Skip)
  initializeApplicationMeth ≜ safeletInitializeCall!x!SID → Skip;
  safeletInitializeRet!x!SID → Skip
  Methods ≜ µ X • (getSequencerMeth; X □ initializeApplicationMeth; X
                   □ end_safelet_app → Skip
     • µ X • startSafelet?!x!SID → Methods; X)
end

sequencerMissionSequencer ≜ begin
  state[mision_done : B]
  initial ≜ mission_done := false
  getNextMission ≜ res return : ID •
    (if ¬ mission_done → mission_done := true; return := newMission()
     fi)
end

process MissionSequencer_app ≜ begin
  state[mision_done : B]
  Init ≜ mission_done := false
  getNextMissionMeth ≜ getNextMissionCall!x!SID →
    (var return : ID •
     if ¬ mission_done → mission_done := true;
     startMission!MSID!MID → return := MID
     fi
     getNextMissionRet!x!MSID!return → Skip
    );
  Methods ≜ µ X • getNextMission; X □ done_sequence → Skip
  • µ X • start_sequence?!x!MSID → Init; Methods; X
end
missionMission ≡ begin
  state[aeh, peh : ID; buffer : seq N]
  initial ≡ Skip
    initialize ≡ (buffer := (); aeh := newAEH(); peh := newPEH(aeh, 1000);
      aeh.register(); peh.register())
  cleanup ≡ res return := true
end

process Mission_app ≡ begin state[aeh, peh : ID; buffer : seq N]
  MArea ≡ var aeh, peh : ID; buffer : seq N • μ X •
    (get_aeh?x!MID!aeh → X □ set_aeh?x!MID!!aeh := x; X
      □
    get_peh?x!MID!peh → X □ set_peh?x!MID!!peh := x; X
      □
    get_buffer?x!MID!buffer → X □ set_buffer?x!MID!!buffer := x; X
      □
  )
  end_mission_app.MID → Skip
  Init ≡ startMission?x!MID → Skip
  cleanUpMeth ≡ cleanUpCall?x!MID → (var return := true; cleanUpRet!x!MIF!return → Skip)
  initializeMeth ≡ missionInitializeCall?x!MID →
    (startAEH!MID!AID → set_aeh!MID!AID!aeh; startPEH!MID!PID → set_peh!MID!PID!peh;
      register!aeh → register!peh → Skip;
      missionInitializeRet!x!MID → Skip
    )
  Methods ≡ μ X •
    (initializeMeth; X
      □ cleanUpMeth; X
      □ end_mission_app!MID → Skip
    )
  (Init; Methods
    [ {aeh, peh} | { set_buffer.MID, get_buffer.MID, set_aeh.MID, get_aeh.MID, set_peh.MID, get_peh.MID } ] ; X
    MArea
  )
end
periodic handler $PEH \equiv \begin{array}{l}
\text{state} [aeh : ID] \\
\text{initial} \equiv \text{val } b : ID \land aeh := b \\
\text{handleAsyncEvent} \equiv \begin{cases}
(inp\text{put?}x \rightarrow \text{Skip}) \rhd \text{INP}_DL; \\
buffer := buffer \setminus \langle x \rangle; \ aeh.\text{release}(); \ \text{wait } 0..\text{PTB}
\end{cases}
\end{array}$
end

process $PEH_{\text{app}} \equiv \begin{array}{l}
\text{state} [aeh : ID] \\
\text{Init} \equiv \text{val } b : ID \land aeh := b \\
\text{handleAsyncEventMeth} \equiv \text{handleAsyncEventCall?}!\text{PID}\rightarrow \\
\text{handleAsyncEventRet!}!\text{PID} \rightarrow \text{Skip}
\end{array}$

Methods $\equiv \begin{array}{l}
\mu X \begin{cases}
\text{handleAsyncEventMeth}; X
\end{cases}
\end{array}$

$\bullet \begin{array}{l}
\mu X \bullet \text{startPEH?}!\text{PID}?b \rightarrow \text{start.peh}!\text{PID}!0!\text{PERIOD} \rightarrow \text{Init}(b); \ 	ext{Methods}; X
\end{array}$

aperiodic handler $AEH \equiv \begin{array}{l}
\text{initial} \equiv \text{Skip} \\
\text{handleAsyncEvent} \equiv \begin{cases}
\text{if buffer } \in \text{three}_0 \rightarrow \text{print!true} \rightarrow \text{Skip} \rhd \text{OUT}_DL \\
\text{fi}
\end{cases}
\end{array}$
process \textit{AEH\_app} ≜ \texttt{begin} \\
\textit{Init} ≜ \texttt{Skip} \\
\textit{handleAsyncEventMeth} ≜ \textit{handleAsyncEventCall}!\texttt{AID} → \\
\texttt{getBuffer!AID!MID?buffer} → (\texttt{buffer} ∈ \texttt{three}_0 → (\texttt{print!true} → \texttt{Skip}) \Downarrow \texttt{OUT\_DL}) \\
\texttt{fi} \\
\texttt{handleAsyncEventRet!x!AID} → \texttt{Skip} \\
\textit{Methods} ≜ \mu X \bullet \left( \textit{handleAsyncEventMeth}: X \right) \\
\bullet \mu X \bullet \textit{startAEH}?x!\texttt{AID} → \texttt{start\_aeh!AID} → \texttt{Init}; \textit{Methods}; X \\
\texttt{end} \\

\textbf{Cyclic not in lockstep} \\

Constraints: \texttt{PTB + INP\_DL} \leq \textit{PERIOD}, \texttt{ATB} \leq \textit{PERIOD} \\

\texttt{channel input} : \mathbb{N} \\
\texttt{channel print} : \texttt{String} \\

\texttt{process System2} ≜ \texttt{begin} \\
\texttt{three}_0 : \mathbb{P}(\texttt{seq N}) \\
\forall s : \texttt{seq N} \bullet s ∈ \texttt{three}_0 \iff (\#s \geq 3 \land (0, 0, 0\text{ suffix }s))
$\text{MArea} \triangleq \text{var buffer : seq } \mathbb{N} \bullet \mu X \bullet \begin{cases} \text{setBuffer}?x \rightarrow \text{buffer} := x; X \\ \text{stop} \rightarrow \text{Skip} \end{cases}$

$\text{PeriodicHandler} \triangleq \mu X \cdot \begin{cases} (\text{input}?x \rightarrow (\text{wait } 0..\text{PTB})) \triangleleft \text{INP.DL}; \\
\text{getBuffer}?\text{buffer} \rightarrow \text{setBuffer}!(\text{buffer} \odot \langle x \rangle) \rightarrow \text{Skip}; \\
\text{getBuffer}?\text{buffer} \rightarrow \begin{cases} \text{if buffer} \in \text{three} \rightarrow \text{release} \rightarrow \text{Skip} \\
\text{fi}
\end{cases} \triangleleft \text{PD} \\
\text{wait PERIOD} \\
\text{stop} \rightarrow \text{Skip} \end{cases}$

$\text{AperiodicHandler} \triangleq \mu X \cdot \begin{cases} (\text{release} \rightarrow \text{print}!\text{true} \rightarrow \text{Skip}) \triangleright \text{PERIOD}; X \\
\text{stop} \rightarrow \text{Skip} \end{cases}$

$\text{Handlers} = \begin{cases} \text{PeriodicHandler} \\
\text{AperiodicHandler} \end{cases}$

$\text{Termination} \triangleq \text{terminate} \rightarrow \text{Stopping}$

$\text{Stopping} \triangleq \mu X \bullet (\text{terminate} \rightarrow X \square \text{stop} \rightarrow X)$

$\text{Mission} = \begin{cases} \text{Termination} \\
\text{Handlers} \\
\text{MArea} \end{cases}$

 Mission end

Cyclic potentially missing releases

Constraints: $\text{PTB} + \text{INP.DL} \leq \text{PERIOD}$
channel input : N
channel print : String

process System3 ≜ begin

\[
\text{three}_0 : \mathbb{P} (\text{seq N})
\]
\[
\forall s : \text{seq N} \bullet s \in \text{three}_0 \iff (\# s \geq 3 \land (0, 0, 0) \text{ suffix } s)
\]

MArea ≜ var buffer : seq N • \mu X • (setBuffer ? x → buffer := x; X)

PeriodicHandler ≜

\[
\mu X \bullet (\text{input} ? x → (\text{wait } 0..\text{PTB}) \uparrow \text{INP DL}; \text{setBuffer} ! (\text{buffer } \bowtie \langle x \rangle) → \text{Skip}; \text{release} → \text{Skip} ); X
\]
Buffer = \textbf{var} \ b : \mathbb{B} \cdot \ b := \text{false}; \ \mu \ X \cdot \left( \begin{array}{l} \text{release} \rightarrow \ b := \text{true}; \ X \\ \text{stop} \rightarrow \text{Skip} \end{array} \right)

AperiodicHandler \equiv

\mu \ X \cdot \left( \begin{array}{l} \text{trigger} \rightarrow \text{getBuffer?buffer}\rightarrow \\
\text{wait} \ 0..\text{ATB}; \ \left( \begin{array}{l} \text{if} \ \text{buffer} \in \text{three} \rightarrow \text{println} \rightarrow \text{Skip} \\
\text{buffer} \notin \text{three} \rightarrow \text{println} \rightarrow \text{Skip} \end{array} \right) \end{array} \right) ; \ X

\text{Handlers} = \left( \begin{array}{l} \text{PeriodicHandler} \\
\text{AperiodicHandler} \end{array} \right) \setminus \{ \text{release, trigger} \}

\text{Buffer}

\text{Termination} \equiv \text{terminate} \rightarrow \text{Stopping}

\text{Stopping} \equiv \mu \ X \cdot (\text{terminate} \rightarrow X \Box \text{stop} \rightarrow X)

\text{Mission} = \left( \begin{array}{l} \text{Termination} \\
\text{Handlers} \end{array} \right) \setminus \{ \text{terminate, stop} \}

\{ \text{setBuffer, getBuffer} \}

\text{MArea}

\bullet \text{Mission}

\text{end}

\text{Not in lockstep} - \text{not necessarily finishing before another inputs is available}

\text{Constraints}: PTB + \text{INP_DL} \leq \text{PERIOD}
channel input : N
channel print : String

process System4 ≜ begin

three0 : P(seq N)
∀ s : seq N • s ∈ three0 ↔ (♯ s ≥ 3 ∧ (0, 0, 0) suffix s)

MArea ≜ var buffer : seq N • μ X •
(setBuffer? x → buffer := x; X)
□

getBuffer! buffer → X
□

(stop → Skip)

PeriodicHandler ≜
μ X •

(setBuffer? x → (wait 0..PTB)) ▽ INP_DL;
getBuffer? buffer → setBuffer!(buffer ∧ ⟨x⟩) → Skip;

getBuffer? buffer →
(if buffer ∈ three0 → release → Skip)
fi

buffer ∉ three0 → Skip

wait PERIOD
□

(stop → Skip)
AperiodicHandler \( \equiv \mu X \left( \begin{array}{l}
(trigger \rightarrow \text{print!true} \rightarrow \text{Skip}); X \\
\quad \Box \\
stop \rightarrow \text{Skip}
\end{array} \right) \)

Buffer \( \equiv \text{var } b : B \bullet b := \text{false}; \mu X \left( \begin{array}{l}
(b \& trigger \rightarrow b := \text{false}); X \\
\quad \Box \\
release \rightarrow b := \text{true}; X \\
\quad \Box \\
stop \rightarrow \text{Skip}
\end{array} \right) \)

Handlers \( \equiv \left( \begin{array}{l}
\begin{array}{l}
\text{PeriodicHandler} \\
\text{Buffer}
\end{array}
\end{array} \right) \setminus \langle \text{release, trigger} \rangle

Termination \( \equiv \text{terminate} \rightarrow \text{Stopping} \)

Stopping \( \equiv \mu X \bullet (\text{terminate} \rightarrow X \Box stop \rightarrow X) \)

Mission \( \equiv \left( \begin{array}{l}
\begin{array}{l}
\text{Termination} \\
\text{Handlers} \\
\text{MArea}
\end{array}
\end{array} \right) \setminus \langle \text{terminate, stop} \rangle \setminus \langle \text{setBuffer, getBuffer} \rangle

\bullet \text{Mission}
\end{array} \right) \)
Bibliography


132


