
Article:
Miyazawa, Alvaro Heiji orcid.org/0000-0003-2233-9091, Cavalcanti, Ana Lucia Caneca orcid.org/0000-0002-0831-1976 and Wellings, Andrew John orcid.org/0000-0002-3338-0623 (2019) SCJ-Circus: specification and refinement of Safety-Critical Java programs. Science of Computer Programming. pp. 140-176. ISSN 0167-6423

https://doi.org/10.1016/j.scico.2019.01.002

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

Alvaro Miyazawa[a,*], Ana Cavalcanti[b], Andy Wellings[c]

Department of Computer Science, University of York, UK

[a]*Alvaro.Miyazawa@york.ac.uk*
[b]*Ana.Cavalcanti@york.ac.uk*
[c]*Andy.Wellings@york.ac.uk*

Abstract

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: it 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*, in addition to 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 multiprocessor cyclic executive pattern and produces an SCJ program 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, refinement

1. Introduction

Java is a very popular language, but 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 implementing 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 region-based memory management [3]. 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, which is comparable in complexity to the Ravenscar [4] 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 of SCJ, 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 by 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, the mission sequencer and safelet terminate when all missions have been completed.

The SCJ memory model is based on scoped memory areas. All object allocation occurs within an area and any created objects are collected at well-defined points in the programs execution. For example, each mission has a scoped memory area (called mission memory) shared with its associated handlers, which is collected at the end of the mission; each handler also has it own private scoped memory area that is collected at the end of each of its releases.

*Corresponding author.

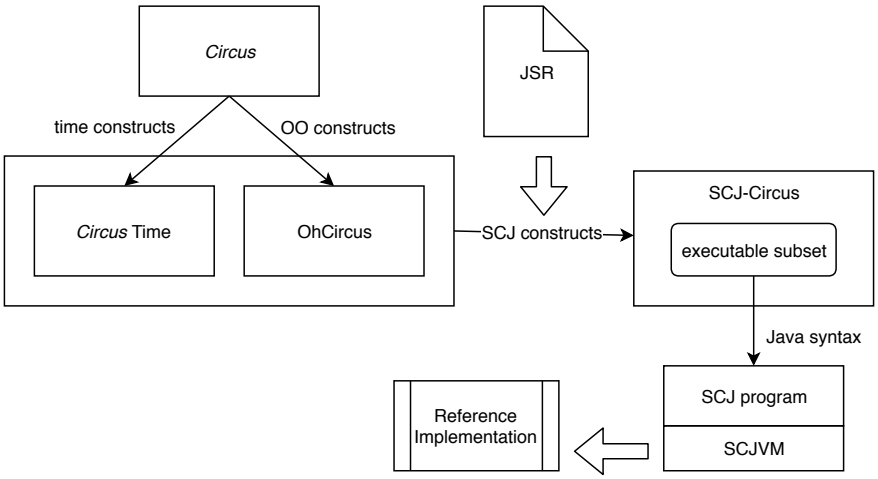


Figure 1: Languages and artefacts of *SCJ-Circus*.

The standardisation effort is being developed under the Java Community Process (JCP), and includes a Java Specification Request (JSR) for SCJ (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 the JSR. There is, however, no support available for design and static verification of SCJ programs included in the standard itself.

Cavalcanti et al. [5] proposes a formal design technique for SCJ based on the *Circus* family of languages: state-rich process algebras for refinement that combine Z [6] and CSP [7]. *Circus* has been used to verify models written in a number of different notations, such as, Simulink and Stateflow diagrams [8, 9, 10], and SysML [11, 12]. The semantics of *Circus* is based on Hoare and He's Unifying Theories of Programming [13], 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. Refinement is an important aspect of *Circus* as evidenced by its rich refinement calculus [14], and it is directly supported by the UTP. *Circus* has been extended to support a number of different programming paradigms. For example, *OhCircus* [15] supports the specification of object-oriented designs and programs, and *Circus Time* [16] supports modelling real-time programs; they are both useful in our work.

Our goal is to verify low-level models of SCJ programs by refining abstract *Circus* specifications. We build on the technique in [5] by refining the verified *Circus* architectural designs that it produces. For that, we introduce here a new member of the set of *Circus* languages: *SCJ-Circus*. Figure 1 illustrates how the different languages and artefacts used in this work relate to each other. *SCJ-Circus* combines *OhCircus* and *Circus Time*, and extends them with the SCJ constructs informed by the JSR. 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 constructs. *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 models are in direct correspondence with SCJ programs, although platform-specific aspects of an application, such as memory and thread availability, are not covered. *SCJ-Circus* models restricted to the executable subset can be directly translated into SCJ programs and executed on an SCJ Virtual Machine such as the reference implementation that is part of the Java Community Process.

Cavalcanti et al. [5] propose a refinement strategy for the verification of SCJ programs, but it stops short of reaching a model concrete enough to support automatic code generation. Here, we extend [5] first by exploring the use of *SCJ-Circus* to define a concrete target model for the refinement that is very close to an SCJ program; we specify the syntax and semantics of *SCJ-Circus*. Next, we identify patterns of *Circus* models obtained through the strategy in [5], and use them as a basis for the development by refinement of *SCJ-Circus* models. The patterns define designs with specific interaction and timing properties, namely, time-triggered missions with precedence constraints. We

present a core refinement strategy for such designs where we have 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 *SCJ-Circus* models 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 [17]. 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.

In summary, we make the following contributions in this paper: (1) an extensible and fully detailed collection of basic refinement procedures applicable to *Circus* designs that follow the SCJ paradigm; (2) a fully specified refinement strategy for the simple application pattern (single-mission and terminating); and (3) detailed extensions of that base strategy to support verification of non-terminating and multi-mission applications. With these results, we close the gap between an abstract *Circus* model and an SCJ program via a refinement-based verification.

An initial version of the strategy, which does not consider particular patterns to enable the definition of a detailed tactic as we do here, is presented in [18]. In [19], we identify the patterns that we explore here, and sketch, but not detail, the refinement strategy that supports the core pattern. Here, we not only fully specify the refinement strategy, its procedures and laws, but also present it in a way that facilitates extension to deal with other patterns. The formalisation of SCJ is first presented in [17], and is the basis for our semantics of *SCJ-Circus*.

In Section 2, we introduce Safety-Critical Java and the *Circus* family of languages. In Section 3, we present *SCJ-Circus*, its syntax and semantics. Section 4 presents our first two patterns, and Section 5 details the core refinement strategy for the first pattern. Sections 6 and 7 extend that strategy to cover our other patterns. Finally, Section 8 concludes, relating our work to those available in the literature, and discussing future work.

2. 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

SCJ aims to support the use of Java in the development of applications that require predictable performance and behaviour as well as high reliability. Validation and certification requirements demand virtual machines and libraries that are both small and highly predictable. The structure of SCJ has been designed to address issues such as memory management and concurrency, which directly impact performance, behaviour and reliability. An SCJ application is encapsulated by a safelet, responsible for interacting with the SCJ runtime environment and controlling the execution of the program. An application consists of a mission sequencer, whose goal is to coordinate the execution of the missions. For example, an application that has multiple modes of operation may represent each mode as a mission, and the sequencer is responsible for switching between missions when requested by the currently executing mission.

While a Level 0 application follows a simple cyclic executive model, in which computations are carried out periodically in a precise time line, Level 1 applications can take advantage of a multitasking programming model. A Level 1 mission may consist of multiple threads of execution, each of which is encapsulated in an SCJ asynchronous event handler. These handlers provide the main functionality of the mission, and can be invoked periodically (time-triggered) or aperiodically (event triggered). They may be mapped to individual processors in the execution environment. Hence, SCJ at Level 1 supports the execution of fully-partitioned multiprocessor applications.

Memory is carefully managed to ensure that fragmentation cannot occur and that all object allocation and deallocation is predictable in both space and execution time. A shared memory area called immortal memory contains the objects global to the application and, once created, they exist for the lifetime of the application. Objects associated with each mission are contained within a mission memory area. This area is shared between all event handlers active within that mission. Mission-allocated objects are the main mechanism that facilitates communication between the event handlers. Objects that are private to each invocation (release) of an event handler are stored in a per-release memory area. They are automatically collected at the end of each release. Finer-grain control of memory allocation and reclamation is possible by nested private memory areas. A set of object assignment rules ensure memory safety

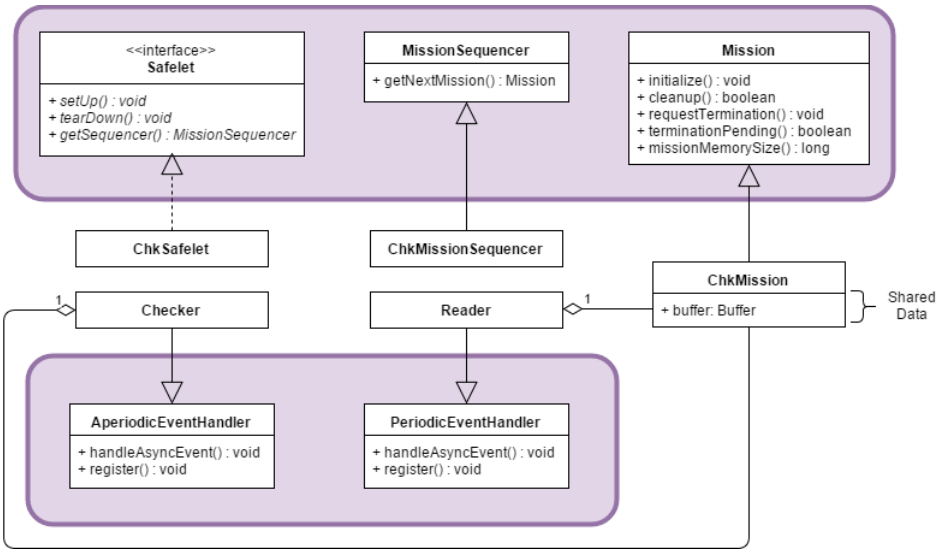


Figure 2: Our running example: communication medium

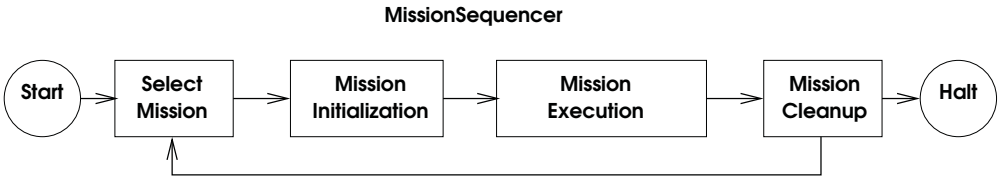


Figure 3: SCJ programming model

and that dangling references cannot result due to references between objects that have different lifetimes. For example, an object in mission memory cannot hold a reference to an object in an event handler's per-release memory area.

These main programming abstractions (safelet, mission sequencer, mission, periodic event handler, aperiodic event handler, mission memory, per-release, and private memory) are characterised by an interface or abstract class of an API that supports the development of SCJ programs via implementation and extension of these components.

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 shows a class diagram for our example. The interface `Safelet` and abstract classes `MissionSequencer`, `Mission`, `AperiodicEventHandler` and `PeriodicEventHandler` 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 `ChkMissionSequencer`. 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` the first time it is called, and `null` the second time, when the program terminates. In fact, in our example, since the mission implemented by `ChkMission` does not terminate, `getNextMission` is called just once, but the possibility of a second call returning `null` is still available.

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. Implementations of the abstract classes `AperiodicEventHandler` or `PeriodicEventHandler` must supply a method `handleAsyncEvent`, which determines the behaviour to be executed on every release of the handler. For example, in the case of a `PeriodicEventHandler` with period 2 seconds, the

```
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 4: SCJ Level 1 example: Aperiodic Event Handler

method `handleAsyncEvent` is executed every 2 seconds. In our example, the handlers are instances of `Reader` and `Checker`. Figure 4 shows the code for `Checker`, the aperiodic handler. It extends the `AperidicEventHandler` class of the SCJ API, and declares a local variable `buffer`, a constructor that receives an instance of a class `Buffer` (not shown in Figure 2) and assigns it to `buffer`, and a `handleAsyncEvent` method.

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

Here, we focus on SCJ Level 1, which is comparable to Ravenscar Ada [4], due to the level of complexity of both the programs and the execution model. Luckcuck *et al.* [20] have formalised the semantics of SCJ Level 2 in *Circus*. That formalisation can serve as the basis for an extension of our refinement strategy and that of Cavalcanti *et al.* [5] to support verification of SCJ Level 2 programs. However, the increase in complexity would likely reflect in the complexity of the strategy as well as in the level of automation achievable.

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 5. 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 *OhCircus*) 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 $\bullet$) that describes the behaviour of the process. In Figure 5, 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 [6] and guarded commands [21] for data modelling, CSP [7] for behavioural descriptions, and time constructs. Table 2.2 summarises the *Circus* operators used in this paper. For each operator, its symbol, name, and an informal description are provided.

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*.

The execution of a newly created handler is defined by the auxiliary action *Execute*. It first waits for *start* time units (**wait** *start*), and then starts two recursive actions in parallel ($A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$) synchronising on the channels *handleAsyncEventCall* and *done_handler*. The first parallel action specifies that at each step of the recursion there is an external choice ($\Box$) for communication on the channels *handleAsyncEventCall* or *done_handler*. In this

Symbol	Name	Description
wait e	wait	Waits for a exactly e time units before terminating.
wait $e_1 \,..\, e_2$	wait	Waits for any time between e_1 and e_2 time units before terminating.
$P \blacktriangleright e$	deadline	Requires that a process P terminate within e time units.
$P \blacktriangleleft e$	start-by	Requires that a process P start within e time units.
Skip	skip	Terminates immediately without any side effects.
Stop	deadlock	Refuses all interactions, but does not change the state.
$P \llbracket n_1 \mid cs \mid n_2 \rrbracket Q$	parallel composition	Run P and Q in parallel synchronising on events in cs. State changes made by P only affect state components in ns_1, and changes made by Q only affect ns_2.
$P \mid\mid\mid Q$	parallel composition	Run P and Q in parallel without synchronisation.
$\{\!\mid e \mid\!\}$	channel set	Set of all possible events associated with channel e.
$\mu X \bullet P(X)$	recursion	Behave like P with occurrences of X replaced by P itself.
$c \longrightarrow P$	prefix	Synchronise on channel c and then behave like P.
$c?x \longrightarrow P$	input	Synchronise on channel c with any possible value, store the chosen value in x, and behave like P.
$c?x : b(x) \longrightarrow P$	restricted input	Synchronise on channel c with any value such that $b(x)$ is true, store the chosen value in x, and behave like P.
$c!e \longrightarrow P$	output	Synchronise on channel c with value e and behave like P.
$v := e$	assignment	Assign value e to variable v.
$P;\ Q$	sequential composition	Behave like P, and once P terminates, behave like Q.
$P \,\Box\, Q$	external choice	Allow the environment to choose between behaving like P or Q.
$P \sqcap Q$	internal choice	Non-deterministically choose between behaving like P or Q.
$P \vartriangle c \longrightarrow Q$	interrupt	Behave like P, until synchronisation on c becomes possible, at which point, behave like Q.
var $v : T \bullet P$	local variable	Declare a local variable v of type T and behave like P within the context of the variable.
$P \setminus cs$	hiding	Run P with events in cs hidden.
$(e) \,\&\, P$	guard	If e is true, behave like P, otherwise deadlock.
$c.t \longrightarrow P$	timed prefix	Record in c the amount of time elapsed between the initial offer of c and its actual occurrence.
$\llbracket cs \rrbracket\ i : I \bullet P(i)$	iterated parallelism	Run $P(i)$ in parallel for all i in I synchronising in cs.
$\mid\mid\mid i : I \bullet P(i)$	iterated interleave	Run $P(i)$ in parallel for all i in I without synchronisation.
$;\ i : I \bullet P(i)$	iterated sequential composition	Run $P(i)$ in sequence for all i in I.

Table 1: Summary of *Circus* operators

way, the method `handleAsyncEvent` can be called through the channel *handleAsyncEventCall* or the handler can be terminated with a choice of *done_handler*. If `handleAsyncEvent` is called, then it must return, as indicated with a communication via *handleAsyncEventRet*, within *period* time units. This is specified using a *Circus Time* deadline construct $A \blacktriangleright e$ that defines that the action A must terminate within e time units.

The second parallel action in *Execute* adds a requirement that a call to `handleAsyncEvent` must be started as soon as it is available, and should be made available again only after *period* time units. This is achieved first by imposing a restriction on the communication *handleAsyncEventCall* using the start-by operator ($A \blacktriangleleft e$) that specifies that an action A must start within a certain number of time units. In the example, it must start immediately, af-

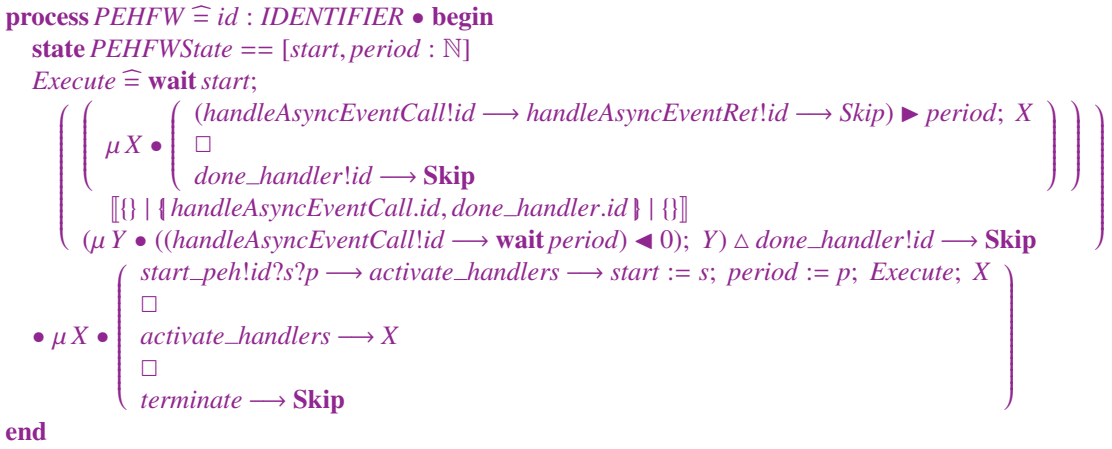


Figure 5: Framework process of the periodic event handler.

ter 0 time units. Additionally, the action **wait** *period* after the communication on *handleAsyncEventCall* ensures that *handleAsyncEventCall* is only offered after *period* time units. If the first recursion is terminated by a synchronisation on the channel *done_handler*, the second recursion must also be terminated. This is achieved by the interruption of the recursion by a synchronisation on *done_handler* using the interrupt operator (△).

Processes can be composed, via CSP operators, to define other processes. In *Circus Time*, wait and deadline operators can define time restrictions. In *OhCircus* models, we can in addition define paragraphs that declare classes used to define types. More information about these languages can be found in [14, 16, 15].

In our previous work [5], a refinement strategy is presented that supports the verification of SCJ program designs. This strategy is structured in three main phases, with the intermediate processes characterised in terms of anchors, which are special patterns of *Circus* processes that use specific combinations of notations. The strategy applies to an abstract *Circus Time* model called an `A-anchor`, which is first refined into an `O-anchor`: an object-oriented model written in *OhCircus*. The `O-anchor` is further refined into an `E-anchor`, which introduces the architectural design induced by the SCJ execution model. Finally, in the third phase, the `E-anchor` is refined into an `S-anchor` that uses *SCJ-Circus*. While the first two phases are fully explored in previous works [5], it is only briefly indicated how to proceed from the `E-anchor` to the `S-anchor`. This last phase is the focus of this paper. In particular, the patterns we identify here are special forms of `E-anchors` produced by the refinement strategy in [5].

CSP and *Circus* adopt a notion of correctness based on refinement, while other process calculi are often based on strong and weak bisimulation. While strong bisimulation is too strong for CSP and *Circus*, as it distinguishes processes based on internal behaviour, weak bisimulation does not distinguish between deadlock and divergence [7].

In the sequel, we further explain the notation as needed, and next describe our new *Circus* variant.

3. *SCJ-Circus*

As said before, *SCJ-Circus* extends *OhCircus* and *Circus Time* with abstractions specific to SCJ. In what follows, 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

Here we provide an overview of the syntax of *SCJ-Circus* and illustrate it using our running example in Figure 12. The complete specification of the syntax is in [22]; an excerpt is provided in Appendix C.

SCJ-Circus extends the syntax of *OhCircus* and *Circus Time* with paragraphs that allow the specification of safelets, mission sequencers, missions, and handlers. An *SCJ-Circus* program is a sequence of SCJCParagraphs, which can be a *Circus* paragraph (in the syntactic class CircusParagraph), or the declaration of a safelet, mission

sequencer, mission or handler. CircusParagraph includes *OhCircus* paragraphs, and, therefore, includes the definition of classes. While object-orientation is central to SCJ, our refinement strategy does not need to concern itself with this aspect because the `E-anchor`, which is the focus of our work, is already object-oriented. The structure of each of the SCJ-specific abstractions is determined by the fields and methods that must be specified for an application.

The root element of an SCJ program is a safelet, whose syntax in *SCJ-Circus* is specified as an SCJCSafelet, which 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 (SCJCSafeletProcessParagraph) and the `getSequencer` method. While the order of the paragraphs in SCJCSafelet (or any of the other SCJCParagraph elements) is not important, we fix a particular order as specified in Appendix C to simplify the presentation.

The state components model the fields of the safelet class, and the initial method, its constructor. While the specification of state components and initialisation method are optional, the `getSequencer` method is mandatory. The auxiliary actions define any extra methods implemented in a safelet class. An SCJCSafeletProcessParagraph allows the specification of an action whose body is an SCJCSafeletAction. This restricts the constructs that can be used in the definition of an action of a safelet, in particular, the type of allocation constructs as discussed later.

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 example, SCJCPeriodicHandler is defined similarly to SCJCSafelet. In this case, the **start** and **period** fields, and the **handleAsyncEvent** action are mandatory. The specification of state components and initialisation method is optional, and the order of the components is fixed just for simplicity.

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 their execution. We reflect this discipline in *SCJ-Circus* by restricting syntactically which paragraphs may include allocations, through different **new** keywords, in particular areas. A safelet may instantiate objects only in the immortal memory, and therefore may use only the keyword **newI** 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**).

In order to illustrate the syntax of *SCJ-Circus*, Figure 6 shows 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.

3.2. Semantic model

In [17], 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 described in *SCJ-Circus*. Figure 7 depicts the structure of our semantic models. Its complete *Circus Time* definition can be found in [22].

For a given program, each component of the SCJ paradigm (safelet, sequencer, and so on) is modelled by a *Circus Time* process. Such a 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 5 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 events that represent method calls. For example, the channels *safeletInitializeCall*, *safeletInitializeRet*, *getSequencerCall* and *getSequencerRet* in Figure 7 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.

$$PTB, ATB, ID, OD, PD, AD, P : \mathbb{Z}$$
$$PTB + ID \leq PD \land PD + AD \leq P$$

safelet *ChkSafelet* $\widehat{=}$ **begin**
  getSequencer $\widehat{=}$ **res return** • **return** := **new** *ChkMissionSequencer*()
end

sequencer *ChkMissionSequencer* $\widehat{=}$ **begin**
  state [*done* : $\mathbb{B}$]
  initial $\widehat{=}$ *done* := *false*
  getNextMission $\widehat{=}$ **res return** : *IDENTIFIER* • (**if** ¬ *done* ⟶ *done* := *true*; **return** := **new** *ChkMission*() [] *done* ⟶ **return** := *null* **fi**)
end

mission *ChkMission* $\widehat{=}$ **begin**
  state [*buffer* : seq $\mathbb{N}$]
  initial $\widehat{=}$ *buffer* := $\langle 0, 0 \rangle$
  initialize $\widehat{=}$ **var** *ah*, *ph* : *IDENTIFIER* • *ah* := **newHandler** *Checker*; *ph* := **newHandler** *Reader*(*ah*)
  cleanup $\widehat{=}$ **res return** : $\mathbb{B}$ • **return** := *true*
end

periodic handler *Reader* $\widehat{=}$ **begin**
  start 0 **period** *PD*
  state [*ah* : *IDENTIFIER*]
  initial $\widehat{=}$ *ah* : *IDENTIFIER* • *this.ah* := *ah*
  handleAsyncEvent $\widehat{=}$
    ((*input*?*x* ⟶ **Skip**) ◀ *ID*; *getBuffer*?*buffer* ⟶ *setBuffer*!(*tail buffer* ⌢ $\langle x \rangle$) ⟶ *release*(*ah*);
    (**wait** 0 . . *PTB*)) ▶ *PD*
end

aperiodic handler *Checker* $\widehat{=}$ **begin**

$$checker : \text{seq}\,\mathbb{N} \pfun \mathbb{B}$$
$$\forall b : \text{seq}\,\mathbb{N} \mid \#b = 2 \bullet (\textbf{if } b\,1 = b\,2 \textbf{ then } checker(b) = true \textbf{ else } checker(b) = false)$$

  handleAsyncEvent $\widehat{=}$
    (*getBuffer*?*buffer*⟶ (**if** *check*(*buffer*) = *true* ⟶ (*output*!*true* ⟶ **Skip**) ◀ *OD* [] *check*(*buffer*) = *false* ⟶ (*output*!*false* ⟶ **Skip**) ◀ *OD* **fi**); **wait** 0..*ATB*) ▶ *AD*
end

Figure 6: `S-anchor` for our running example.

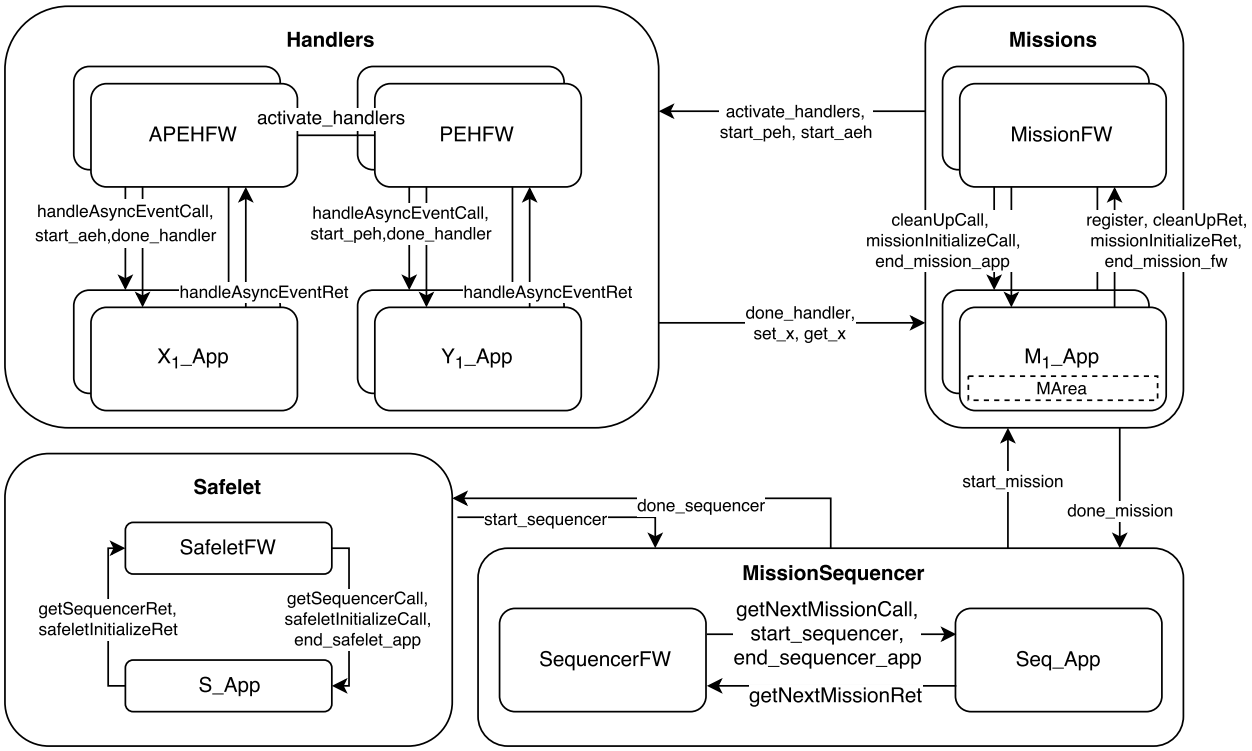


Figure 7: Structure of semantic models

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

Figure 8: Framework process for Safelet.

In the models of SCJ programs presented in [17], the application processes are combined together using 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 model. A model that adopts the structure described in this paper, where application and framework processes are composed on a per-element basis, is equivalent to the model structured as in [17].

The framework process *SafeletFW* that specifies the generic behaviour of a safelet is shown in Figure 8; 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 `initialize`, *SafeletFW* obtains a mission sequencer via the channels *getSequencerCall* and *getSequencerRet*. If the sequencer is different from *null*, *SafeletFW* starts it (using the channel *start_sequencer*); otherwise, it terminates. If the sequencer process is started, *SafeletFW* waits for the completion of its execution,

signalled via the channel *done_sequencer*. At that point, *SafeletFW* indicates to the application process that it is terminating through the channel *end_safelet_app*, waits for all other components of the application to terminate using the channel *terminate*, and finally terminates (**Skip**).

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 [14]. 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 7.

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 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*.

$$[\![\_]\!]_{SCJCProgram} : SCJCProgram \pfun CircusProgram$$

$$\forall p : WF\_SCJCProgram \bullet [\![p]\!]_{SCJCProgram} = [\![p.paragraphs]\!]_{SCJCParagraphs} \frown [\![p]\!]_{Application}$$

We use the mathematical notation of Z [6] 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 $[\![\_]\!]_{SCJCSafelet}$ for the safelet, which is used by $[\![\_]\!]_{SCJCParagraphs}$ to give semantics to an *SCJ-Circus* safelet paragraph. The complete semantics is defined in [22].

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 $[\![\_]\!]_{SCJCSafelet}$ in Figure 9. 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 ($\setminus$). Figure 10 shows, as an example, the processes that define semantics of the *ChkSafelet* paragraph of the *SCJ-Circus* model in Figure 6 as specified by the application of $[\![\_]\!]_{SCJCSafelet}$ to that paragraph.

In the definition of a semantic function, guillemets (« ») 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 *safelet_app* must be evaluated on the parameter s and the resulting syntax tree must be substituted in place of «*safelet_app*(s)».

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 $[\![\_]\!]_{SCJCSafelet}$ 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 9. 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 $N \mathrel{\widehat=} A$ 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 10 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 identifiers in 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.

$$[\![\_]\!]_{SCJCSafelet} : SCJCSafelet \pfun \mathrm{seq}\ CircusParagraph$$

$$\forall s : WF\_SCJCSafelet \bullet$$

$$[\![s]\!]_{SCJCSafelet} = \left(\begin{array}{l} \textbf{process}\ \text{«}name(s)\text{»}\_App \mathrel{\widehat=} \text{«}safelet\_app(s)\text{»} \\ \textbf{process}\ \text{«}name(s)\text{»} \mathrel{\widehat=} \\ \quad (SafeletFW(\text{«}name(s)\text{»}ID) \ [\![\ \text{«}SafeletCS(s)\text{»}\ ]\!]\ \text{«}name(s)\text{»}\_App) \setminus \text{«}SafeletCS(s)\text{»} \end{array} \right)$$

$$safelet\_app : SCJCSafelet \pfun BasicProcess$$

$$\forall s : WF\_SCJCSafelet \bullet$$

$$safelet\_app(s) =$$

$$\left(\begin{array}{l} \textbf{begin} \\ \quad \textbf{state}\ \text{«}s.state\text{»} \\ \quad \text{«}for\ each\ p : s.paragraphs\ of\ (N \mathrel{\widehat=} A)\ do\text{»} \\ \quad\quad \text{«}N\text{»}Meth \mathrel{\widehat=} \text{«}translate\_method(name(s)ID, N, A)\text{»} \\ \quad \text{«}end\text{»} \\ \quad getSequencerMeth \mathrel{\widehat=} \text{«}translate\_method(name(s)ID, getSequencer, s.getSequencer)\text{»} \\ \quad initializeApplicationMeth \mathrel{\widehat=} \text{«}translate\_method(name(s)ID, initializeApplication, s.initialize)\text{»} \\ \quad Methods \mathrel{\widehat=} \mu X \bullet \left(\begin{array}{l} getSequencerMeth;\ X \\ \Box \\ initializeApplicationMeth;\ X \\ \Box \\ \text{«}for\ each\ p : s.paragraphs\ of\ (N \mathrel{\widehat=} A)\ do\text{»} \Box \text{«}N\text{»}Meth;\ X\text{«}end\text{»} \\ \Box \\ end\_safelet\_app \longrightarrow \textbf{Skip} \end{array} \right) \\ \quad \bullet\ Methods \\ \textbf{end} \end{array} \right)$$

Figure 9: Semantic functions for safelets.

For instance, *getSequencerMeth* 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, the instantiation of the mission sequencer, **new** *ChkMissionSequencer*(), is modelled as a communication via the channel *startMissionSequencer*, which passes the identifier of the safelet to the sequencer application process. In general, any parameters required by the constructor are also passed. This communication is with the application process, because the constructor might have non-standard parameters.

All the actions of the safelet paragraph (*getSequencerMeth* and *initializeApplicationMeth*) in Figure 10 are combined in the action *Methods* that recursively ($\mu X \bullet F(X)$) offers them in external choice ($\Box$) 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 (marked by the symbol $\bullet$ at the end of the process definition) 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_App* 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 ($\setminus$).

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

$$
\begin{array}{l}
\textbf{process}\ ChkSafelet\_App \mathrel{\widehat=} \textbf{begin} \\
\quad getSequencerMeth \mathrel{\widehat=} getSequencerCall?x!ChkSafeletID \longrightarrow \\
\quad\quad \left(\begin{array}{l} \textbf{var return} : IDENTIFIER \bullet \\ \quad startMissionSequencer!ChkSafeletID!ChkMissionSequencerID \longrightarrow \\ \quad \textbf{return} := ChkMissionSequencerID; \\ getSequencerRet!x!ChkSafeletID!\textbf{return} \longrightarrow \textbf{Skip} \end{array} \right) \\
\quad initializeApplicationMeth \mathrel{\widehat=} safeletInitializeCall?x!ChkSafeletID \longrightarrow \textbf{Skip}; \\
\quad\quad safeletInitializeRet!x!ChkSafeletID \longrightarrow \textbf{Skip} \\
\quad Methods \mathrel{\widehat=} \mu X \bullet getSequencerMeth;\ X \,\Box\, initializeApplicationMeth;\ X \,\Box\, end\_safelet\_app \longrightarrow \textbf{Skip} \\
\quad \bullet\ Methods \\
\textbf{end} \\
\\
\textbf{process}\ ChkSafelet \mathrel{\widehat=} \\
\quad (SafeletFW(ChkSafeletID) \\
\quad\quad [\![\{\!|\ getSequencerCall, getSequencerRet, safeletInitializeCall, safeletInitializeRet, end\_safelet\_app\ |\!\}]\!] \\
\quad ChkSafelet\_App) \\
\setminus \{\!|\ getSequencerCall, getSequencerRet, safeletInitializeCall, safeletInitializeRet, end\_safelet\_app\ |\!\}
\end{array}
$$

Figure 10: Semantics of the Safelet in our running example.

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.

$$
MArea \mathrel{\widehat=} \textbf{var}\ buffer : \mathrm{seq}\,\mathbb{N} \bullet \left(\mu X \bullet \left(\begin{array}{l} setBuffer?o!ChkMissionID?x \longrightarrow buffer := x;\ X \\ \Box \\ getBuffer?o!ChkMissionID!buffer \longrightarrow X \\ \Box \\ end\_mission\_app \longrightarrow \textbf{Skip} \end{array} \right) \right)
$$

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 these *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 an action *IArea*, similar to *MArea*, in the safelet application process. 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. We do not consider such objects in our model, but an extension to include them is not difficult because our semantics includes a function to define a memory action, which can be used to define an immortal-memory action as well. It is worth emphasising, however, that such a program cannot have its resource allocation analysed statically.

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

$$
\begin{array}{l}
\textbf{process}\ P \mathrel{\widehat=} \textbf{begin} \\
\quad PHandler_i \mathrel{\widehat=} \mu\, X \bullet (F_i \blacktriangleright PERIODICDEADLINE \mid\mid\mid \textbf{wait}\, PERIOD);\ X \Box t \longrightarrow \textbf{Skip} \\
\quad AHandler_j \mathrel{\widehat=} \mu\, X \bullet ((c_j \longrightarrow G_j) \blacktriangleright APERIODICDEADLINE \mid\mid\mid \textbf{wait}\, PERIOD);\ X \Box t \longrightarrow \textbf{Skip} \\
\quad MArea \mathrel{\widehat=} \textbf{var}\ x : T \bullet \mu\, Y \bullet set_x?nx \longrightarrow x := nx;\ Y \Box get_x!x \longrightarrow Y \Box t \longrightarrow \textbf{Skip} \\
\quad Termination \mathrel{\widehat=} rt \longrightarrow \mu\, X \bullet rt \longrightarrow X \Box t \longrightarrow \textbf{Skip} \\
\quad Mission \mathrel{\widehat=} ((MArea \parallel (\parallel i : I \bullet PHandler_i) \parallel (\parallel j : J \bullet AHandler_j)) \llbracket\, \alpha S \mid \{\!| t, rt |\!} \mid \{\} \,\rrbracket\ Termination) \setminus \{\!| \ldots |\!\} \\
\quad MissionSequencer \mathrel{\widehat=} Mission \\
\quad Safelet \mathrel{\widehat=} MissionSequencer \\
\quad \bullet\ Safelet \\
\textbf{end} \\
\qquad\qquad \textbf{where}\ PERIODICDEADLINE \leq PERIOD \land APERIODICDEADLINE \leq PERIOD
\end{array}
$$

Figure 11: `E-anchor` pattern: single time-triggered mission with precedence constraints without optional components.

4. Patterns

In this section, we consider two main patterns of `E-anchor`. These patterns specify timing and interaction designs that are the basis for our refinement strategy. In Sections 6 and 7, the strategy is extended to cover new patterns.

Cyclic in lock step. The core pattern we consider characterises single mission applications, in which all periodic and aperiodic handlers are executed at every time step. This pattern in shown in Figure 11; it is for `E-anchors` produced by the strategy in [5], and the object of 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*.

In the pattern in Figure 11, 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.

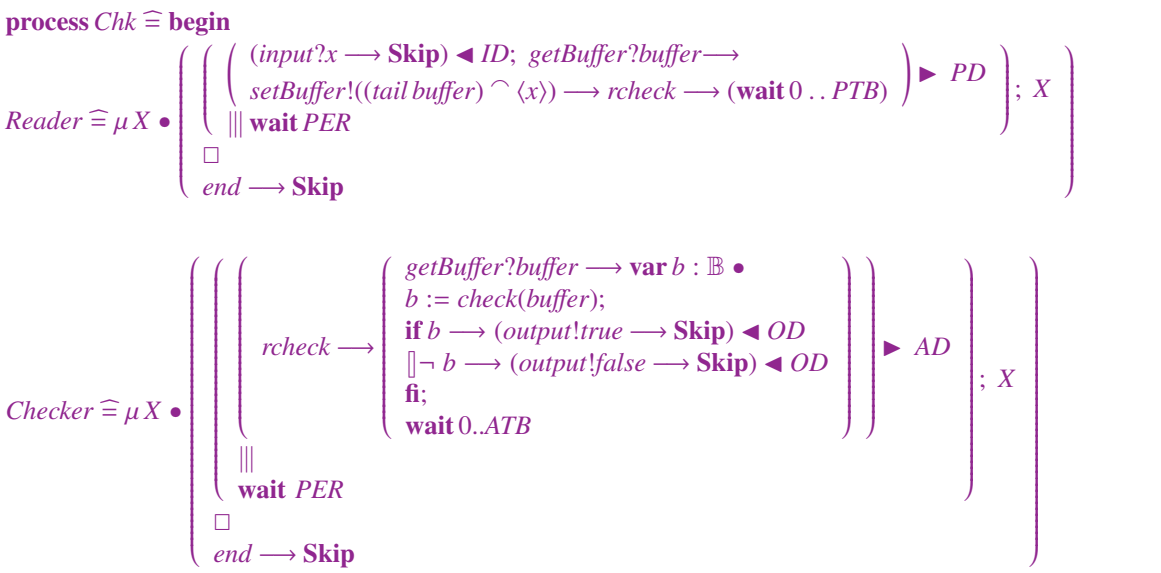
Parallelism occurs in the definition of *Mission* between the actions $PHandler_i$ and $AHandler_j$ that model the periodic and aperiodic handlers, the action *MArea* that models the mission memory, and the termination management action *Termination*. Like in the semantics of *SCJ-Circus*, the mission-memory 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 `requestTermination`). Afterwards, *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 *Circus* needs to identify the partition of variables that each parallel action can modify to avoid race conditions. (So far, these sets of variables have been empty.) In the case of the *Mission* action, however, in Figure 11, all variables in scope, denoted by αS can be modified by the parallelism of the handlers with the memory area, (action $MArea \parallel (\parallel i : I \bullet PHandler_i) \parallel (\parallel j : J \bullet AHandler_j)$), while *Termination* modifies no variables, and so is associated with the empty set $\{\}$ of variables in the use of the parallel operator $\ldots \llbracket\, \alpha S \mid \{\!| t, rt |\!} \mid \{\} \,\rrbracket \ldots$.

Periodic event handlers $PHandler_i$ are defined by recursive actions whose iterations take some fixed amount of time (**wait** *PERIOD*) whilst executing (in interleaving) an action F_i that models the behaviour of the handler release and takes at most *PERIODICDEADLINE* time units; the recursion runs until it is terminated via a synchronisation on t. Since *PERIOD*, *PERIODICDEADLINE* and *APERIODICDEADLINE* are constants used explicitly in the pattern, the constraint specified in the **where** clause of Figure 11 can be established automatically.

We illustrate the pattern in Figure 11 with the `E-anchor` in Figure 12 for our running example. The safelet, sequencer, and mission are defined by the actions *ChkSafelet*, *ChkMissionSequencer*, and *ChkMission*. The main behaviours are specified in the handlers defined by *Reader* and *Checker*. Such an `E-anchor` can be obtained using the refinement strategy described in [5] from a much more abstract *Circus* model. The action *Reader* for the periodic



where $PD \leq PER \land AD \leq PER$

Figure 12: E anchor for the communication medium example.

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

where $APERIODICDDEADLINE \leq PERIOD$

Figure 13: Pattern: cyclic not in lock-step.

handler performs a communication via a channel *input*, with a deadline *ID*, followed by an update of *buffer*, and a trigger of the aperiodic handler *Checker* via a release signal *rcheck*, all with a budget of *PTB* time units.

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

We require $PERIODICDDEADLINE \leq PERIOD \land APERIODICDEADLINE \leq PERIOD$, 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 *PD*, *PER*, and *AD*. We have $PD \leq PER \land AD \leq PER$ as required, and also need further restrictions involving *ID*, *PTB*, and *ATB* to ensure feasibility of the deadlines. These restrictions, which are omitted in Figure 12, are enforced by a global constraint on these constants.

Cyclic not in lock step. Our second pattern, in Figure 13, allows for an aperiodic handler not to be called in a cycle. It differs from that in Figure 11 just in the models of the aperiodic handlers, described in Figure 13.

An $AHandler_j$ 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 *APERIODICDEADLINE* time units, and cater for termination through the channel t. In addition, however, if a release or termination does not take place within *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 *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 than what is required to deal with arbitrary *Circus* models. For example, the technique in [23] is applicable.

After presenting our refinement strategy in the next section, we consider in Sections 6 and 7 variations of the these patterns for applications that do not terminate or have multiple missions.

5. Refinement Strategy

In this section, we present the refinement strategy for the pattern in Figure 11, and then explain how it can be adapted for that in Figure 13. Our refinement strategy is based on four phases that step-by-step introduce the complete structure of an *SCJ-Circus* program. The phases of the strategy and its target are shown in Figure 14. Each phase introduces a particular aspect of the final *SCJ-Circus* program as described below.

CF introduces the control flow of the program accounting for aspects such as requests to start a mission or a sequencer;

AP takes advantage of the control flow introduced in the previous phase to clearly isolate the application specific behaviours from the behaviour of the framework;

FW acts upon the framework behaviours isolated by the previous phase and completes them to match the full semantics of *SCJ-Circus* ; and

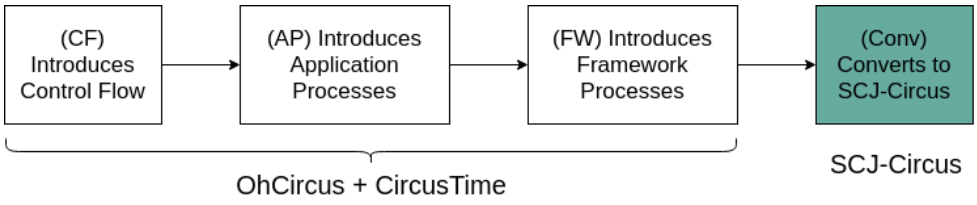
Conv uses the semantics of *SCJ-Circus* to convert the *Circus* program obtained in the previous phase into an *SCJ-Circus* program that can then be automatically translated into SCJ.

The target *SCJ-Circus* program has the form shown in Figure 14. For example, each action $PHandler_i$ in the starting model is defined in a corresponding *SCJ-Circus* **periodic handler** paragraph, whose **handleAsyncEvent** paragraph is determined by the body of $PHandler_i$. Similarly, each aperiodic, mission, mission sequencer and safelet action is defined in a corresponding *SCJ-Circus* paragraph. The first three phases (CF, AP and FW) act only on constructs of the time and object-oriented languages, *Circus Time* and *OhCircus*, whilst the last phase (Conv) produces *SCJ-Circus* specifications.

Our refinement strategy is based on a collection of specialised laws and associated refinement procedures that can be applied with a very high level of automation. While it is possible to identify a smaller set of more general laws, the proof obligations generated are likely complex and difficult to discharge automatically.

An important aspects of our refinement strategy is that most of the refinement steps introduce internal communication that correspond to elements of the structure of an SCJ program. While these communication are required to support the structure of the SCJ architecture, they are unnecessary from the point of view of program functionality, and this is what our refinement strategy ensures. 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.

Phases



Target

safelet *Safelet* $\widehat{=}$ **begin** . . . **end**
sequencer *Sequencer* $\widehat{=}$ **begin** . . . **end**
mission *Mission* $\widehat{=}$ **begin** . . . **end**
periodic handler $PHandler_i$ $\widehat{=}$ **begin** . . . **handleAsyncEvent** $\widehat{=} F_i \blacktriangleright$ *PERIODICDEADLINE* **end**
aperiodic handler $AHandler_j$ $\widehat{=}$ **begin** . . . **handleAsyncEvent** $\widehat{=} G_j \blacktriangleright$ *APERIODICDEADLINE* **end**

Figure 14: Overview of the refinement strategy

5.1. CF: Introducing the SCJ control flow

Overview. The goal of this phase is to transform the design model (Figure 11) to make the control flow of the SCJ paradigm explicit. In the patterns, this control flow is modelled implicitly via sequential and parallel compositions; after the application of this phase of our strategy, it is captured by channel synchronisations. For example, we introduce, a channel *activate_handlers* that models the synchronised start of the handlers in parallel.

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 11, a process structured as the target process shown in Figure 15. 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 ‖ to indicate a parallelism, and omit channel and name sets, if they are not relevant for the discussion.

The steps in Figure 15 define transformations that lead to the target process. They apply novel specialised laws to parallelise the safelet, sequencer and mission actions, replace synchronous communications between handlers with asynchronous communications, and separate the handlers from the mission action. In addition, parallel actions that specify the framework behaviour associated with mission execution are merged and sequentialised. Below, we detail the steps; we recall that all laws are in Appendix A. Here, we reproduce just a few new laws for illustration.

Step 1. In the first step of this phase, the Law call-intro 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]

$$F(A) \sqsubseteq (F(cs \longrightarrow ce \longrightarrow \mathbf{Skip}) \llbracket \overline{wrtV(A)} \mid \{\!| cs, ce |\!\} \mid wrtV(A) \rrbracket cs \longrightarrow A;\ ce \longrightarrow \mathbf{Skip}) \setminus \{\!| cs, ce |\!\}$$

provided

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

Law call-intro is proved by induction over the structure of the action F using distribution and step laws such as those found in [14]. The provisos guarantee that the internal channels are fresh and that the state is appropriately partitioned

Steps

1. Apply Law call-intro to the action *Safelet* with channels *cs* and *ce* replaced by *start_sequencer* and *done_sequencer*;
2. Apply procedure mission-sequencer-CF to the action *MissionSequencer*.

Target

$$
\begin{array}{l}
\textbf{process}\ CF\_P \mathrel{\widehat{=}} \textbf{begin} \\
\quad PHandler_i \mathrel{\widehat{=}} \mu X \bullet (F_i \blacktriangleright PD \,|||\, \textbf{wait}\, PERIOD);\ X \,\Box\, t \longrightarrow \textbf{Skip} \\
\quad AHandler_j \mathrel{\widehat{=}} \mu X \bullet ((c_j \longrightarrow G_j) \blacktriangleright AD \,|||\, \textbf{wait}\, PERIOD);\ X \,\Box\, t \longrightarrow \textbf{Skip} \\
\quad MArea \mathrel{\widehat{=}} \ldots \\
\quad Termination \mathrel{\widehat{=}} rt \longrightarrow \mu X \bullet (rt \longrightarrow X \,\Box\, t \longrightarrow \textbf{Skip}) \\
\quad CF\_Mission \mathrel{\widehat{=}} start\_mission \longrightarrow \\
\qquad \left(\begin{array}{l} MArea \parallel Termination \parallel \\ \left(\begin{array}{l} \parallel i : I \bullet SH_i \longrightarrow register.i \longrightarrow start\_peh.i \longrightarrow activate\_handlers \longrightarrow done\_handler.i \longrightarrow \textbf{Skip} \\ \parallel \\ \parallel j : J \bullet SH_j \longrightarrow register.j \longrightarrow start\_aeh.j \longrightarrow activate\_handlers \longrightarrow done\_handler.j \longrightarrow \textbf{Skip} \end{array} \right) \end{array} \right) ; \\
\qquad done\_mission \longrightarrow \textbf{Skip} \\
\quad Safelet \mathrel{\widehat{=}} \\
\qquad \left(\begin{array}{l} \parallel i : I \bullet SH_i \longrightarrow start\_peh.i \longrightarrow activate\_handlers \longrightarrow PHandler_i;\ done\_handler.i \longrightarrow \textbf{Skip} \\ \parallel \\ \parallel j : J \bullet SH_j \longrightarrow start\_aeh.j \longrightarrow activate\_handlers \longrightarrow \\ \quad (AHandler_j \,[\![\{\ldots\} \mid \{\!| c_{ji} |\!\} \mid \{\}]\!]\, Buffer_j) \setminus \{\!| c_{ji} |\!\};\ done\_handler.j \longrightarrow \textbf{Skip} \\ \parallel CF\_Mission \\ \parallel start\_sequencer \longrightarrow start\_mission \longrightarrow done\_mission \longrightarrow done\_sequencer \longrightarrow \textbf{Skip} \\ \parallel start\_sequencer \longrightarrow done\_sequencer \longrightarrow \textbf{Skip} \end{array} \right) \\
\quad \bullet\ Safelet \setminus \{\!| start\_sequencer, done\_sequencer, \ldots |\!\} \\
\textbf{end}
\end{array}
$$

Figure 15: Refinement strategy – CF: Introducing the SCJ control flow

to avoid racing conditions in the parallelism. We use $usedC(A)$ to refer to the set of channels used in an action A, and $usedV(A)$ and $wrtV(A)$ to refer to the variables used and modified by A.

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

$$
\begin{array}{l}
ChkSafelet \mathrel{\widehat{=}} \\
\left(\begin{array}{l} \boxed{start\_sequencer \longrightarrow done\_sequencer \longrightarrow \textbf{Skip}} \\ \quad \boxed{[\![\{\} \mid \{\!| start\_sequencer, done\_sequencer |\!\} \mid \{\ldots\}]\!]} \\ \boxed{start\_sequencer \longrightarrow} \ ChkMissionSequencer; \\ \boxed{done\_sequencer \longrightarrow \textbf{Skip}} \end{array} \right) \boxed{\setminus \{\!| start\_sequencer, done\_sequencer |\!\}}
\end{array}
$$

The boxed elements are introduced by the application of Law call-intro. Instead of calling *ChkMissionSequencer* directly, we have a call via communications on the channels *start_sequencer* and *done_sequencer*, in parallel with an action that responds to *start_sequencer* by calling *ChkMissionSequencer*.

Step 2. The second step effects a similar change in the definition of *ChkMissionSequencer*. It applies a procedure called mission-sequencer-CF, which traverses the structure of the 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. This procedure is presented in Figure 16 and described below.

Procedure mission-sequencer-CF. The definition of mission-sequencer-CF covers only the case where there is a single mission, which is itself the definition of the sequencer (like in our example). In this case, the mission sequencer action consists of a single call to a mission action as shown in Figure 11. This is identified by the condition $A = M_i$, indicating that the parameter A of the procedure is a call to an action M_i.

The alternative case, where the sequencer action is not a single call to a mission action, is not treated by our refinement strategy as indicated in Step 2. For more elaborate sequencers, we need to extend Step 2 of this procedure. Essentially, we need only to effect a distributed application of the base case (Step 1) through the structure of the mission-sequencer action. For example, 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 7, where we extend mission-sequencer-CF to cover multi-mission applications.

Step 1(a). This step essentially separates the call to *ChkMission* from its execution using parallelism and communication. For our simple case, after this step, we obtain the following definition for the sequencer, where the change effected by the refinement step is emphasised with a box.

$$
\begin{array}{l}
ChkMissionSequencer \mathrel{\widehat{=}} \\
\left(\begin{array}{l}
start\_mission.MID \longrightarrow done\_mission.MID \longrightarrow \mathbf{Skip} \\
\quad [\![\{\} \mid \{\!| start\_mission.MID, done\_mission.MID |\!\} \mid \{\ldots\}]\!] \\
start\_mission.MID \longrightarrow ChkMission; \\
done\_mission.MID \longrightarrow \mathbf{Skip}
\end{array}\right) \setminus \{\!| start\_mission.MID, done\_mission.MID |\!\}
\end{array}
$$

Steps 1(b) and 1(c). The procedure mission-sequencer-CF, however, goes further. Steps 1(b) and 1(c) restructure or, more precisely, enrich the mission action to include the control of the mission execution via *start_mission* and *done_mission*. For *ChkMission* in the example for *Step 1(a)*, the result is as follows.

$$
\begin{array}{l}
ChkMissionSequencer \mathrel{\widehat{=}} \\
\left(\begin{array}{l}
start\_mission.MID \longrightarrow done\_mission.MID \longrightarrow \mathbf{Skip} \\
\quad [\![\{\} \mid \{\!| start\_mission.MID, done\_mission.MID |\!\} \mid \{\ldots\}]\!] \\
ChkMission
\end{array}\right) \setminus \{\!| start\_mission.MID, done\_mission.MID |\!\} \\
ChkMission \mathrel{\widehat{=}} \left(\begin{array}{l}
(start\_mission.MID \longrightarrow (MArea \parallel Reader \parallel Checker) \,[\![\ldots]\!]\, Termination) \setminus \{\!| \ldots |\!\}; \\
done\_mission.MID \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}
$$

Step 1(d). Finally, we apply a separate procedure mission-CF, which is shown in Figure 18 and explained next. This procedure has the same purpose as mission-sequencer-CF, but applies to missions.

Procedure mission-sequencer-CF. In our example, after the application of this procedure, we obtain the definition for *ChkMission* in Figure 17. 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 in *ChkMission* (identified in the *ChkMission* by a box with superscript *EP*) 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*. *Reader* is now as follows.

$$
\begin{array}{l}
Reader \mathrel{\widehat{=}} StartReader.RID \longrightarrow start\_peh.RID \longrightarrow activate\_handlers \longrightarrow \\
\quad \mu X \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
(input?x \longrightarrow \mathbf{Skip}) \blacktriangleleft ID;\ getBuffer?buffer \longrightarrow \\
setBuffer!((tail\ buffer) \frown \langle x \rangle) \longrightarrow check \longrightarrow (\mathbf{wait}\, 0\,..\,PTB)
\end{array}\right) \blacktriangleright PD \\
|||\ \mathbf{wait}\ PERIOD
\end{array}\right); X \\
\Box \\
done\_handler.RID \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}
$$

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 $start\_mission.i$ 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 $start\_mission \longrightarrow \ldots$; $done\_mission \longrightarrow \mathbf{Skip}$ with name M;
   (d) Apply procedure mission-CF to M.
2. Else *"The strategy does not cover this case."*

Figure 16: Refinement strategy: procedure mission-sequencer-CF

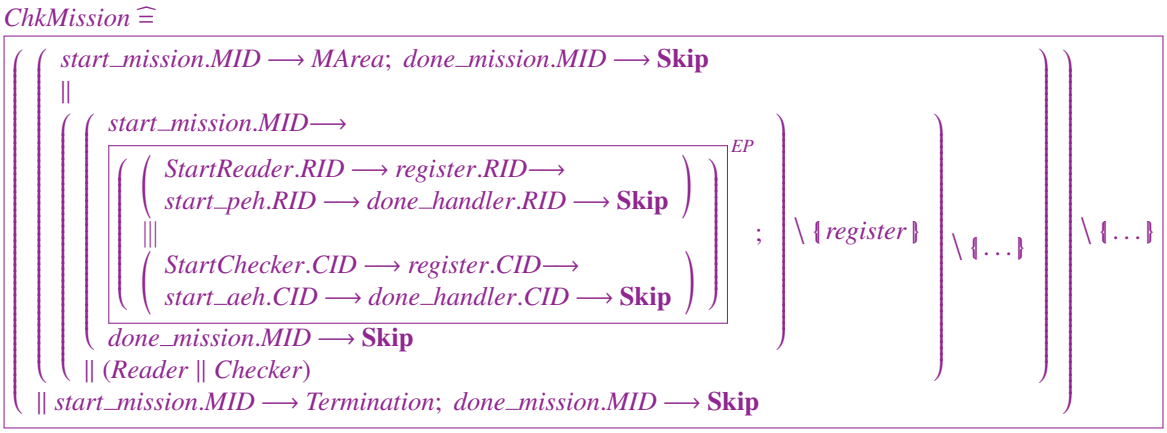


Figure 17: *ChkMission* after Step 1(d) of mission-sequencer-CF

Comparing this to the original definition of *Reader* shown in Figure 12, we observe that we have simply added control communications and replaced t with $done\_handler.RID$. The changes to *Checker* resulting from mission-CF are similar. 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.

Step 1. The first step of mission-CF (see Figure 18) 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 behaviour of the mission. This step considers each aperiodic handler $AHandler_j$; for each of them, the parallelism of handlers is reorganised (via simple associativity and commutativity laws), so that one of the parallel actions is $AHandler_j$ and the other is a component combining the rest of the handlers, including those that trigger $AHandler_j$. At this point the Law sync-async-conv is used to introduce the asynchronous communication between $AHandler_j$ and all other handlers that may release it. In what follows, we describe the details of Law sync-async-conv, which is shown in Figure 19.

In Law sync-async-conv, $\mu X \bullet ((c \longrightarrow G) \blacktriangleright D_1 \;|||\; \mathbf{wait}\, P);\; X \,\Box\, end \longrightarrow \mathbf{Skip}$ matches the specification of an aperiodic handler and $OHandlers \mathrel{\widehat{=}} F(\mu X \bullet (B(c \longrightarrow \mathbf{Skip})_1 \blacktriangleright D_2 \;|||\; \mathbf{wait}\, P);\; X \,\Box\, end \longrightarrow \mathbf{Skip})$, the specification of the other handlers. We use $F(A)_1$ 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 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 *pending* initially set to *false*. This action accepts a communication on c and, in doing so, records on *pending* that the action G whose execution is

This procedure takes a *Circus* action M_i that models as mission as its parameter.

1. For each aperiodic action $AHandler_j$ in the parallelism of handler actions use associativity and commutativity laws to obtain a parallelism between $AHandler_j$ and another parallelism with all other handlers, and apply Law sync-async-conv;
2. Apply Law prefix-introduction [14] to the action M_i to introduce channel *activate_handlers* after *start_mission*. Afterwards, apply prefix-par-dist [14] 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 $start\_mission.MID \longrightarrow H$; $done\_mission.MID \longrightarrow \mathbf{Skip}$ (except where H is *MArea* or *Termination*), apply Law copy-rule [14] 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 *startH.HID* with any parameters specific to the handler, *sh* to *start_peh.HID.S.P* and *r* to *register.HID*;
   - If H is an aperiodic event handler with id *HID*, instantiate *SH* to *startH.HID* with any parameters specific to the handler, *sh* to *start_aeh.HID* and *r* to *register.HID*.
4. Apply step laws [14] 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 $SH.HID \longrightarrow sh.HID \longrightarrow ah \longrightarrow \mu X \bullet F(X, dh.HID \longrightarrow \mathbf{Skip})$ associated with a handler H, apply Law copy-rule from right to left to introduce an action with name H.

Figure 18: Refinement strategy: procedure mission-CF

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 *end* used to terminate the handlers, and makes no changes to global variables as indicated by its associated nameset $\{\}$ in the parallelism in the right-hand side.

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 $OHandlers \llbracket \{\!| c |\!\} \rrbracket TimeReq = OHandlers$.

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.

TimeReq specifies time requirements: (a) c must take place before T time units ($(c@t : (t < T))$, and after that, it cannot happen until the next period ($\mathbf{wait}\,(P-t)$); or (b) c does not happen within the period (*wait P*). With the proviso $OHandlers \llbracket \{\!| c |\!\} \rrbracket TimeReq = OHandlers$, we require that *OHandlers* satisfies the property defined by *TimeReq*. This property also ensures that *OHandlers* cannot communicate on c infinitely often and starve the buffer.

The Law sync-async-conv also requires that the original action *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 *ChkMission* has the structure already shown in Figure 17, but with the buffer included in the parallelism between *Reader* and *Checker*.

Step 2. The second step of mission-CF (Figure 18) inserts a new event *activate_handlers* corresponding to the operation of the SCJ framework that coordinates the start of the handlers. This event is introduced after *start_mission.i*. (Law prefix-introduction introduces a hidden event.) Afterwards, we distribute the communications that record the beginning and end of the mission (that is, *start_mission.i* and *done_mission.i*, where *i* is the mission identifier) through the parallelism of handlers. This uses a standard step law of parallelism [14] and par-prefix-dist.

Law [sync-async-conv]

$$
\begin{array}{l}
\mu X \bullet ((c \longrightarrow G) \blacktriangleright D_1 \;|||\; \mathbf{wait}\, P);\; X \,\Box\, end \longrightarrow \mathbf{Skip} \;[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\; OHandlers \\
\sqsubseteq \\
((\mu X \bullet (c_i \longrightarrow (G \blacktriangleright (D_1 - T)));\; X \,\Box\, end \longrightarrow \mathbf{Skip}) \;[\![\, ns_1 \mid \{\!|\, c_i, end \,|\!\} \mid \{\} \,]\!]\; Buffer) \setminus \{\!|\, c_i \,|\!\} \\
\quad [\![ns_1 \mid cs \mid ns_2]\!] \\
OHandlers
\end{array}
$$

where

$$
\begin{array}{l}
Buffer \mathrel{\widehat{=}} \mathbf{var}\, pending : \mathbb{B} \bullet pending := false;\; \mu X \bullet \\
\quad c \longrightarrow pending := true;\; X \,\Box\, (pending)\; \& \; c_i \longrightarrow pending := false;\; X \,\Box\, end \longrightarrow \mathbf{Skip} \\
OHandlers = F(\mu X \bullet (B(c \longrightarrow \mathbf{Skip})_1 \blacktriangleright D_2 \;|||\; \mathbf{wait}\; P);\; X \,\Box\, end \longrightarrow \mathbf{Skip}) \\
TimeReq = \mu X \bullet (c@t : (t < T) \longrightarrow \mathbf{wait}\,(P - t) \,\Box\, \mathbf{wait}\; P);\; X \,\Box\, end \longrightarrow \mathbf{Skip}
\end{array}
$$

provided

- $c_i \notin usedC(LHS) \land T < D_1 < P \land D_2 < P$
- $OHandlers \;[\![\, \{\!|\, c, end \,|\!\} \,]\!]\; TimeReq = OHandlers$
- *B is not recursive and LHS is deadlock-free and feasible.*

Figure 19: Law sync-async-conv

Law par-prefix-dist is similar to the standard step law, Law prefix-par-dist in Appendix A, which extracts synchronisations from inside parallel actions. Law 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 *Mission* as shown in Figure 17 by a box with superscript *EP*, to a top level parallelism as shown in the action *CF_Mission* of the target process in Figure 15. This allows the extraction of the handler actions in the next step.

Step 3. The third step applies the Law handler-extract to each of the parallel actions modelling event handlers. It wraps the handler actions with synchronisations on new internal channels, denoted 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*, *ah*, *dh*, and *dm* match the communications *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.

Law [handler-extract]

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

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

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 *F*, remain under the control of that action. The new parallel action used for orchestration modifies no variables.This law can be proved by applying the parallelism step laws in [14]. The repeated application of Law handler-extract introduces multiple parallel actions that orchestrate the synchronisations of the handlers.

Steps 4 and 5. Step 4 applies step laws as much of possible to join the communications on *start_mission* and *done_mission* in the orchestrating actions introduced in the previous step. As a result, this step also groups these repeated actions in a single parallel action named *CF_Mission* as shown in the target process in Figure 15. Step 5 simply gives names to the handler actions. The result is the process shown in Figure 15.

5.2. AP: Introducing the application processes

Overview. This phase separates specifications of application-specific behaviours (such as, specification of the functionality of a handler release) from those of behaviours imposed by the SCJ paradigm (such as, starting a mission) that are implemented by the SCJ runtime environment (framework).

This is achieved by transforming the target of the CF phase (Figure 15) into the form shown in Figure 20. It defines a number of application actions, $Handler_i\_app$, *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.

The target in Figure 20 shows the structure of the mission actions. In the *AP* framework action *AP_Mission*, after the start of the mission (*start_mission.MID*), the `initialize` method 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 local variable *handlers*, until the call to `initialize` returns (*missionInitializeRet*!*x*!*MID*).

The steps of the AP phase are shown in Figure 20. 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

Steps

1. Apply Law p-handler-split to each periodic handler action and Law a-handler-split to each aperiodic handler action, with channels *haeC* and *haeR* replaced with *handleAsyncEventCall* and *handleAsyncEventRet*;
2. Apply procedure mission-AP to the safelet action;
3. Apply procedure sequencer-AP to the action that models the sequencer;
4. Apply Law safelet-split to the action that models the safelet.

Target

$$
\begin{array}{l}
\textbf{process}\ AP\_P \mathrel{\widehat{=}} \textbf{begin} \\
\quad Handler_i\_app \mathrel{\widehat{=}} \ldots \\
\quad AP\_Handler_i \mathrel{\widehat{=}} \ldots \\
\quad Mission\_app \mathrel{\widehat{=}} \\
\qquad \left(\begin{array}{l} start\_mission.MID \longrightarrow missionInitializeCall?x!MID \longrightarrow \\ \quad ((;\ i : I \cup J \bullet startH.i \longrightarrow \textbf{Skip});\ (;\ i : I \cup J \bullet register.i \longrightarrow \textbf{Skip}); \\ missionInitializeRet!x!MID \longrightarrow \textbf{Skip}); \\ done\_mission.MID \longrightarrow \textbf{Skip} \end{array} \right) \\
\quad AP\_Mission \mathrel{\widehat{=}} \left(\begin{array}{l} start\_mission.MID \longrightarrow missionInitializeCall?x!MID \longrightarrow \\ \left(\begin{array}{l} \textbf{var}\ handlers : \mathbb{P}\ ID \bullet \\ \left(\begin{array}{l} \mu\, X \bullet \left(\begin{array}{l} register?x \longrightarrow handlers := handlers \cup \{x\};\ X \\ \Box \\ missionInitializeRet!x!MID \longrightarrow \textbf{Skip} \end{array} \right); \\ (|||i : handlers \bullet (start\_aeh!i \longrightarrow \textbf{Skip} \Box start\_peh!i?s?p \longrightarrow \textbf{Skip}); \\ activate\_handlers \longrightarrow (|||i : handlers \bullet done\_handler.i \longrightarrow \textbf{Skip})) \end{array} \right) \end{array} \right) \\
\quad MissionSequencer\_app \mathrel{\widehat{=}} \ldots \\
\quad AP\_MissionSequencer \mathrel{\widehat{=}} \ldots \\
\quad Safelet\_app \mathrel{\widehat{=}} \ldots \\
\quad AP\_Safelet \mathrel{\widehat{=}} \ldots \\
\qquad \left(\begin{array}{l} (Safelet\_app \parallel AP\_Safelet) \setminus \{\!| \ldots |\!\} \\ \parallel \\ (MissionSequencer\_app \parallel AP\_MissionSequencer) \setminus \{\!| \ldots |\!\} \\ \parallel \\ (Mission\_app \parallel AP\_Mission) \setminus \{\!| \ldots |\!\} \\ \parallel \\ (\parallel i : I \cup J \bullet (Handler_i\_app \parallel AP\_Handler_i) \setminus \{\!| \ldots |\!\}) \end{array} \right) \setminus \{\!| \ldots |\!\} \\
\textbf{end}
\end{array}
$$

Figure 20: Refinement strategy – AP: Introducing the application processes

flow. To replace, in the 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.

Step 1. 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 *handleAsyncEventCall* and *handleAsyncEventRet* channels to model a call to the `handleAsyncEvent` method.

Law p-handler-split applies to an action $SH \longrightarrow sh.id.s.p \longrightarrow \mathbf{wait}\ s;\ \mu X \bullet ((A \blacktriangleright D \,|||\, \mathbf{wait}\ p);\ X \,\Box\, dh \longrightarrow \mathbf{Skip})$ 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. With an application of the Law p-handler-split, the behaviour $A \blacktriangleright D$ of the handler release is wrapped with new internal events *haeC* and *haeR*. 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 than 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 proved by the application of parallelism step laws.

Law [p-handler-split]

$$
\begin{array}{l}
SH \longrightarrow sh.id.s.p \longrightarrow ah \longrightarrow \mathbf{wait}\ s;\ \mu X \bullet ((A \blacktriangleright D \,|||\, \mathbf{wait}\ p);\ X \,\Box\, dh \longrightarrow \mathbf{Skip}) \\
\sqsubseteq \\
\left(\begin{array}{l}
SH \longrightarrow sh.id.s.p \longrightarrow \mu X \bullet ((haeC \longrightarrow A \blacktriangleright D;\ haeR \longrightarrow X) \,\Box\, dh \longrightarrow \mathbf{Skip}) \\
\quad [\![wrtV(A) \mid \{\!| sh, dh, haeC, haeR |\!\} \mid \{\}]\!] \\
\\
sh.id?start?period \longrightarrow ah \longrightarrow \mathbf{wait}\ start;\ \mu X \bullet \left(\begin{array}{l}
\left(\begin{array}{l} (haeC \longrightarrow haeR \longrightarrow \mathbf{Skip}) \blacktriangleleft 0 \blacktriangleright period \\ |||\ \mathbf{wait}\ period \end{array}\right);\ X \\
\Box\, dh \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}\right) \\
\setminus \{\!| haeC, haeR |\!\}
\end{array}
$$

provided $\{\!| sh, dh, haeC, haeR |\!\} \cap usedC(A) = \varnothing \wedge D \leq p.$

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

Steps 2 and 3. These steps apply specialised procedures mission-AP and sequencer-AP defined later to transform the mission and sequencer actions. 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 procedures are slightly more complex, however, in that they must restructure actions and consider the syntactic structure of the actions to apply specific laws.

Steps 4. The final step 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*.

At the end of the AP phase, the actions for the application processes are completed, but the actions $AP\_$ (see Figure 20) do not quite specify the SCJ runtime environment. These actions are the focus of the next 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 21; it consists of three steps, for each action M_i modelling a mission.

Step 1. The first step applies a Law mission-split to the safelet action. This law 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; in Figure 22, 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

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 [14] 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 21: Refinement strategy: procedure mission-AP

Law [mission-split]

$$
\begin{array}{l}
F\left(\begin{array}{l} sm \longrightarrow \left(\begin{array}{l} SH_1 \longrightarrow r.id_1 \longrightarrow sh.id_1 \longrightarrow ah \longrightarrow dh.id_1 \longrightarrow \mathbf{Skip} \\ \quad [\![\{\!| ah |\!\}]\!] \\ SH_2 \longrightarrow r.id_2 \longrightarrow sh.id_2 \longrightarrow ah \longrightarrow dh.id_2 \longrightarrow \mathbf{Skip} \end{array}\right) \setminus \{\!| r |\!\};\ dm \longrightarrow \mathbf{Skip} \end{array}\right) \setminus \{\!| SH_1, SH_2 |\!\} \\
\sqsubseteq \\
F\left(\begin{array}{l} \left(\begin{array}{l} sm \longrightarrow miC \longrightarrow \left(\begin{array}{l} SH_1 \longrightarrow r.id_1 \longrightarrow \mathbf{Skip} \\ ||| \\ SH_2 \longrightarrow r.id_2 \longrightarrow \mathbf{Skip} \end{array}\right);\ miR \longrightarrow dm \longrightarrow \mathbf{Skip} \\ \quad [\![\{\} \mid \{\!| sm, r, dm, miC, miR |\!\} \mid \{\}]\!] \\ sm \longrightarrow miC \longrightarrow \left(\begin{array}{l} r.id_1 \longrightarrow \mathbf{Skip} \\ ||| \\ r.id_2 \longrightarrow \mathbf{Skip} \end{array}\right);\ miR \longrightarrow \mathbf{Skip}; \\ \quad \left(\begin{array}{l} sh.id_1 \longrightarrow ah \longrightarrow dh.id_1 \longrightarrow \mathbf{Skip} \\ [\![\{\!| ah |\!\}]\!] \\ sh.id_2 \longrightarrow ah \longrightarrow dh.id_2 \longrightarrow \mathbf{Skip} \end{array}\right);\ dm \longrightarrow \mathbf{Skip} \end{array}\right) \setminus \{\!| miC, miR, r |\!\} \end{array}\right) \setminus \{\!| SH_1, SH_2 |\!\}
\end{array}
$$

provided $\{\!| miC, miR |\!\} \cap usedC(LHS) = \varnothing$

Figure 22: Law mission-split

register, *sh* with *start_aeh* or *start_peh*, depending on whether we have an aperiodic or a periodic handler, *ah* 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 *sh* and *dh*). The parallel handler actions synchronise just in *ah*, 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.

Like the laws for splitting handlers, Law mission-split is proved by the application of existing step laws [14].

Step 2. In the second step, 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 next steps to focus on a particular (parallel) component of the safelet action (see Figure 15).

Step 3. The Law seq-interleave, applied in the third step, 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

Law [rec-interleave]

$$
\left(\begin{array}{l}
sm \longrightarrow miC \longrightarrow SH_1 \longrightarrow SH_2 \longrightarrow r.id_1 \longrightarrow r.id_2 \longrightarrow miR \longrightarrow dm \longrightarrow \mathbf{Skip} \\
[\![\{\} \mid \{\!| sm, r, dm, miC, miR |\!\} \mid \{\}]\!] \\
sm \longrightarrow miC \longrightarrow \left(\begin{array}{l} r.id_1 \longrightarrow \mathbf{Skip} \\ ||| \\ r.id_2 \longrightarrow \mathbf{Skip} \end{array}\right); \; miR \longrightarrow \mathbf{Skip}; \\
\quad \left(\begin{array}{l} sh.id_1 \longrightarrow ah \longrightarrow dh.id_1 \longrightarrow \mathbf{Skip} \\ \quad [\![\{\!| ah |\!\}]\!] \\ sh.id_2 \longrightarrow ah \longrightarrow dh.id_2 \longrightarrow \mathbf{Skip} \end{array}\right); \; dm \longrightarrow \mathbf{Skip}
\end{array}\right) \setminus \{\!| miC, miR, r |\!\} \\
\sqsubseteq \\
\left(\begin{array}{l}
sm \longrightarrow miC \longrightarrow SH_1 \longrightarrow SH_2 \longrightarrow r.id_1 \longrightarrow r.id_2 \longrightarrow miR \longrightarrow dm \longrightarrow \mathbf{Skip} \\
\quad [\![\{\} \mid \{\!| sm, r, dm, miC, miR |\!\} \mid \{\}]\!] \\
\mathbf{var}\ handlers : \mathbb{P}\, ID \bullet handlers := \{\}; \\
sm \longrightarrow miC \longrightarrow \mu\, X \bullet \left(\begin{array}{l} r?h \longrightarrow handlers := handlers \cup \{h\}; \; X \\ \Box \\ miR \longrightarrow \mathbf{Skip} \end{array}\right); \\
\quad ([\![\{\!| ah |\!\}]\!]\, h : handlers \bullet sh.h \longrightarrow ah \longrightarrow dh.h \longrightarrow \mathbf{Skip}); \; dm \longrightarrow \mathbf{Skip}
\end{array}\right) \setminus \{\!| miC, miR, r |\!\}
$$

provided *handlers is fresh.*

Figure 23: Law rec-interleave

actions, which synchronise on instantiation and registration events, they occur in interleaving, avoiding deadlocks due to the specific choice made in the refinement. 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.

Step 4. In the final step, the Law rec-interleave, shown in Figure 23 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. 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 (which 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 rec-interleave becomes rather long. Its proof relies on unfolding the recursion the required number of times, and instantiation of the iterated parallel.

Procedure sequencer-AP. 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 7.

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 24, the procedure applies the Law sequencer-split-single to the sequencer action. This law 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 *start_sequencer* and *done_sequencer*.

1. If $A = start\_mission.i \longrightarrow done\_mission.i \longrightarrow \mathbf{Skip}$ then
   (a) Apply Law sequencer-split-single to the action A.
2. Else *"The strategy does not cover this case."*

Figure 24: Refinement strategy: procedure sequencer-AP

Steps

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

Target

$$
\begin{array}{l}
\mathbf{process}\ FW\_P \mathrel{\widehat=} \mathbf{begin} \ldots \\
\quad \bullet \left(\begin{array}{l}
(Safelet\_app \parallel SafeletFW) \setminus \{\!| \ldots |\!\} \\
\parallel \\
(MissionSequencer\_app \parallel MissionSequencerFW) \setminus \{\!| \ldots |\!\} \\
\parallel \\
(Mission\_app \parallel MissionFW) \setminus \{\!| \ldots |\!\} \\
\parallel \\
(\parallel i : I \bullet (Handler_i\_app \parallel APEHFW) \setminus \{\!| \ldots |\!\}) \\
\parallel \\
(\parallel j : J \bullet (Handler_j\_app \parallel PEHFW) \setminus \{\!| \ldots |\!\})
\end{array} \right) \setminus \{\!| \ldots |\!\} \\
\mathbf{end}
\end{array}
$$

Figure 25: Refinement strategy – FW: Introduction of framework processes

5.3. FW: Introduction of framework processes

Overview. This phase takes the potentially incomplete model of the framework identified in AP, representing the slice of the SCJ framework actually used by the application, and replaces it with the full-fledged framework model.

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 $Safelet\_app \parallel AP\_Safelet$ in the target in Figure 20, and is transformed in this phase.

These parallel actions are close to those in the processes that give semantics to an *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 *ChkSafelet* in our example in Figure 12) does not perform any initialisation. For this reason, the method *safeletInitialize* provided by the *SCJ-Circus* **safelet** paragraph and included in the framework process *SafeletFW* (see Figure 8) 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 *SCJ-Circus* program. This is possible because the missing actions model methods that are not used in the original specification, and their introduction leads to method calls that terminate immediately: for instance, a call to *safeletInitialize* whose body is just **Skip**.

The target process resulting from the application of this phase is shown in Figure 25. 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 25. The main results used are lemmas specialised to the cyclic in

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 $mission\_start.i \longrightarrow mission\_done.i \longrightarrow \mathbf{Skip}$, apply law mission-fw-cl-single.
2. Else *"The strategy does not cover this case."*

Figure 26: Refinement strategy: procedure sequencer-fw-cl

lockstep pattern. The framework actions (for example, *AP_Safelet* in the target in Figure 20) 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 safelet-fw-cl, 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, and for this reason only we do not call them laws. Obviously, the use 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.) All the lemmas are in Appendix A.

Lemma safelet-fw-cl applies to the safelet component. The application action, at this stage, is very simple. It accepts a call to `getSequencer` (via *getSequencerCall*), which returns (*getSequencerRet*) an identifier *sid*. The proviso requires *sid* not to be *null*. The framework action calls `getSequencer`, starts the sequencer (*start_sequencer*), and ends the application (*end_safelet_app*) action when the sequencer terminates (*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 is transformed to accept 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 uses 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 proviso not to be *null*, we can be sure that the sequencer is started like in the original action.

Standard step, variable block, and conditional laws can be used to prove Lemma safelet-fw-cl.

Steps 1,3,4 and 5 of the FW phase apply framework-completion lemmas like that safelet-fw-cl; all lemmas are in Appendix A. Step 2 is similar, but relies on the procedure sequencer-fw-cl in Figure 26 that identifies 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.

This is the only phase that needs to be adapted to deal with our second pattern in Figure 13. It requires a special form of the Lemma aperiodic-handler-fw-cl that considers the possibility of the handler not being executed in a cycle.

5.4. Conv: Converting to SCJ-Circus

Overview. This phase takes the final step into obtaining an *SCJ-Circus* program by refining the specification into a sequence of *SCJ-Circus* paragraphs based on the *Circus* semantics of *SCJ-Circus*. With that purpose, it introduces new process paragraphs that isolate pairs of application and framework processes corresponding to the constructs of SCJ. Afterwards, 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 14. For our example, the resulting *SCJ-Circus* specification is shown in Figure 6. 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.

Steps

1. Apply the definition of parallel processes [14] 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.

Target

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

Figure 27: Refinement strategy – Conv: Converting to *SCJ-Circus*

Starting pattern

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

Modified steps

CF Step-0 Add a new first step to phase CF that applies the law termination-intro.

Figure 28: Refinement strategy: Non-terminating pattern.

The steps for this phase are shown in Figure 27. 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.

6. Non-terminating pattern

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

In this section, we adapt our strategy to consider applications that do not terminate. The starting pattern in Figure 28 shows the *Mission* action of the non-terminating cyclic in lockstep pattern. The main difference from that in Figure 11 is the missing *Termination* action. The overall target is still that described in Figure 14.

The refinement strategy described in Section 5 cannot be applied to models that follow the starting pattern in Figure 28 because it expects the *Mission* action to have an extra parallelism with the *Termination* action. More specifically, Lemma 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.

CF Step-0. We introduce *Termination* as a first step of the CF phase using the Law termination-intro. In this way, any other extensions to the strategy can take advantage of the general pattern with termination.

Starting pattern

$$MissionSequencer \mathrel{\widehat{=}} M_1 ; M_2 ; \ldots ; M_n$$

Modified steps

CF-Step 2 Use the recursive procedure mission-sequencer-CF shown in Figure 30 instead;

AP-Step 3 Use the procedure sequencer-AP described in Figure 31 instead;

FW-Step 2 Use the procedure sequencer-fw-cl shown in Figure 32 instead.

Figure 29: Refinement strategy: multi-mission pattern

Law [termination-intro]

$$\mu X \bullet F ; X \sqsubseteq (\mu X \bullet F ; X \,\Box\, t \longrightarrow \mathbf{Skip} \;[\![\{\!| wrtV(F) |\!\} \mid \{\!| rt, t |\!\} \mid \{\}]\!]\; rt \longrightarrow \mu X \bullet rt \longrightarrow X \,\Box\, t \longrightarrow \mathbf{Skip}) \setminus \{\!| rt, t |\!\}$$

provided $\{\!| t, rt |\!\} \cap usedC(F) = \varnothing$.

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

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 have a high degree of automation with most provisos to the refinement laws being of a syntactic nature. 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.

7. 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 $M_1 ; M_2 ; \ldots ; M_n$ (see starting pattern in Figure 29). 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 29.

CF-Step 2. The procedure used in Step 2 of CF needs to be replaced with a recursive procedure that takes a sequence of missions $M_i ; M_{rest}$, applies the original steps of mission-sequencer-CF to the first mission M_i, and recursively applies itself to the remaining missions in M_{rest}. This extended procedure is shown in Figure 30. Its Step 1 is the same as in the original procedure (Figure 16); Step 2 is replaced with a step that pattern matches against sequential composition identifying the first component as M_i and recursively applies the procedure to the sequence of missions; and Step 3 is 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.

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 $start\_mission \longrightarrow \ldots ;\ done\_mission \longrightarrow \mathbf{Skip}$ with name M;
   (d) Apply procedure mission-CF to M.
2. If $A = M_i;\ B$ then
   (a) apply mission-sequencer-CF to M_i and to B separately;
   (b) apply hiding and parallelism distribution laws of [14] to obtain a parallel action.
3. Else *"The strategy does not cover this case."*

Figure 30: Refinement strategy: procedure mission-sequencer-CF

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

$$
\begin{array}{l}
MissionSequencer_1 \mathrel{\widehat{=}} \\
\left(\begin{array}{l}
start\_mission.M_1 \longrightarrow done\_mission.M_1 \longrightarrow \mathbf{Skip} \\
\quad [\![\{\} \mid \{\!| start\_mission.M_1, done\_mission.M_1 \mid \{\ldots\}]\!] \\
Mission_1
\end{array}\right) \setminus \{\!| start\_mission.M_1, done\_mission.M_1 |\!\}; \\
\left(\begin{array}{l}
start\_mission.M_2 \longrightarrow done\_mission.M_2 \longrightarrow \mathbf{Skip} \\
\quad [\![\{\} \mid \{\!| start\_mission.M_2, done\_mission.M_2 |\!\} \mid \{\ldots\}]\!] \\
Mission_2
\end{array}\right) \setminus \{\!| start\_mission.M_2, done\_mission.M_2 |\!\}
\end{array}
$$

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. Step 2(b) uses distribution laws of parallelism and hiding to merge the hidings and to transform the sequence into a single parallel action, where the mission activations are in sequence, but the mission actions are in interleaving as shown in the action $MissionSequencer_2$.

$$
\begin{array}{l}
MissionSequencer_2 \mathrel{\widehat{=}} \\
\left(\begin{array}{l}
\boxed{start\_mission.M_1 \longrightarrow done\_mission.M_1 \longrightarrow start\_mission.M_2 \longrightarrow done\_mission.M_2 \longrightarrow \mathbf{Skip}} \\
\quad [\![\{\} \mid \boxed{\{\!| start\_mission.M_1, done\_mission.M_1, start\_mission.M_2, done\_mission.M_2 |\!\}} \mid \{\ldots\}]\!] \\
\boxed{Mission_1 \;|||\; Mission_2}
\end{array}\right) \\
\setminus \{\!| \boxed{start\_mission.M_1, done\_mission.M_1, start\_mission.M_2, done\_mission.M_2} |\!\}
\end{array}
$$

This is possible because the channels $start\_mission.M_i$ and $done\_mission.M_i$ are only used by the action $Mission_i$.

AP-Step 3. The third step of the AP phase needs a slightly different definition for the procedure sequencer-AP, using a law that introduces a pattern for the action of the `getNextMission` method tailored for multiple missions in sequence. The new procedure sequencer-AP is shown in Figure 31. It has an extra step (Step 2) that applies a new specialised law sequencer-split-mult-seq, which splits a sequence of communications on *start_mission* and *done_mission* into a parallel action communicating over channels *getNextMissionCall* and *getNextMissionRet* to ob-

This procedure takes a sequencer action S as its parameter. It considers its component A that defines its behaviour after and before *start_sequencer* and *done_sequencer*.

1. If $A = start\_mission.i \longrightarrow done\_mission.i \longrightarrow \mathbf{Skip}$ then
   (a) Apply Law sequencer-split-single to the action A.
2. If $A = (;\ i : I \bullet start\_mission.i \longrightarrow done\_mission.i \longrightarrow \mathbf{Skip})$
   (a) Apply Law sequencer-split-mult-seq to the action A.
3. Else *"The strategy does not cover this case."*

Figure 31: Refinement strategy: procedure sequencer-AP

tain the identifiers of the missions being executed. The resulting action for our example is as follows.

$$
\begin{array}{l}
\ldots start\_mission.M_1 \longrightarrow done\_mission.M_1 \longrightarrow start\_mission.M_1 \longrightarrow done\_mission.M_1 \longrightarrow \ldots \\
\sqsubseteq \\
\ldots \\
\left(\begin{array}{l}
\boxed{getNextMissionCall!id!id \longrightarrow getNextMissionRet!id!id!M_1} \longrightarrow start\_mission.M_1 \longrightarrow done\_mission.M_1 \longrightarrow \\
\boxed{getNextMissionCall!id!id \longrightarrow getNextMissionRet!id!id!M_2} \longrightarrow start\_mission.M_2 \longrightarrow done\_mission.M_2 \longrightarrow \\
\boxed{getNextMissionCall!id!id \longrightarrow getNextMissionRet!id!id!null \longrightarrow end\_sequencer\_app \longrightarrow \ldots} \\
\quad \boxed{[\![\{\} \mid \{\!| getNextMissionCall, getNextMissionRet, end\_sequencer\_app, \ldots |\!\} \mid \{\}]\!]} \\
\quad \boxed{\begin{array}{l} getNextMissionCall?x!id \longrightarrow getNextMissionRet!x!id!M_1 \longrightarrow \\ getNextMissionCall?x!id \longrightarrow getNextMissionRet!x!id!M_2 \longrightarrow \\ getNextMissionCall?x!id \longrightarrow getNextMissionRet!x!id!null \longrightarrow \\ end\_sequencer\_app \longrightarrow \ldots \end{array}}
\end{array}\right) \\
\quad \backslash \{\!| getNextMissionCall, getNextMissionRet, end\_sequencer\_app |\!\}
\end{array}
$$

FW-Step 2. Finally, in the second step of the FW phase, the procedure sequencer-fw-cl is extended as shown in Figure 32 to use the Law mission-fw-cl-mult-seq to deal with sequences of missions. Like Law mission-fw-cl-single, this new law transforms the pair of application and framework actions that model the mission sequencer to make them match the semantics of *SCJ-Circus* for a sequencer. Its effect on the action produced in step *AP-Step 3* is as follows.

$$
\begin{array}{l}
\left(\begin{array}{l}
\boxed{\begin{array}{l} \mu X \bullet getNextMissionCall!id!id \longrightarrow getNextMissionRet!id!id?m \longrightarrow \\ \left(\begin{array}{l} \mathbf{if}\ m = null \longrightarrow end\_sequencer\_app \longrightarrow \mathbf{Skip} \\ [\!] m \neq null \longrightarrow start\_mission.m \longrightarrow done\_mission.m \longrightarrow X \\ \mathbf{fi} \end{array}\right) \end{array}} \\
\quad [\![\{\} \mid \{\!| getNextMissionCall, getNextMissionRet |\!\} \mid \{\}]\!] \\
\quad \boxed{\begin{array}{l} \mathbf{var}\ i : \mathbb{Z} \bullet i := 0;\ \mu X \bullet \\ \left(\begin{array}{l} getNextMissionCall?x!id \longrightarrow \left(\begin{array}{l} \mathbf{if}\ i = 0 \longrightarrow getNextMissionRet!x!id!M_1 \longrightarrow i := i + 1 \\ [\!] i = 1 \longrightarrow getNextMissionRet!x!id!M_2 \longrightarrow i := i + 1 \\ [\!] i > 1 \longrightarrow getNextMissionRet!x!id!null \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right); X \\ \Box \\ end\_sequencer\_app \longrightarrow \mathbf{Skip} \end{array}\right) \end{array}}
\end{array}\right) \\
\backslash \{\!| getNextMissionCall, getNextMissionRet, end\_sequencer\_app |\!\}
\end{array}
$$

Of note is the effect of Law mission-fw-cl-mult-seq on the result of the procedure sequencer-AP. In this case, both parallel actions are transformed into recursions, with the framework action (on the left) using the mission identifier *null* to terminate the recursion, and the application action (on the right) using a counter to decide the next mission to be returned. This pattern for the application action is commonly used and can be modified to suit particular designs.

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 $mission\_start.i \longrightarrow mission\_done.i \longrightarrow \mathbf{Skip}$, apply law mission-fw-cl-single.
2. If the action *Sequencer* of P contains an action of the form $; i : I \bullet mission\_start.i \longrightarrow mission\_done.i \longrightarrow \mathbf{Skip}$, apply law mission-fw-cl-mult-seq.
3. Else *"The strategy does not cover this case."*

Figure 32: Refinement strategy: procedure sequencer-fw-cl

8. Conclusions

In this paper, we have extended previous work [17, 5] on the semantics of SCJ and on a refinement strategy for SCJ programs. We propose a variant of *Circus* suitable for modelling using SCJ abstractions, update existing models of SCJ 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 [17] 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 [5], 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 generates 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 [5]. 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 FDR [24], but large or infinite state-spaces are usually not tractable. In such cases, it may also be possible to apply compositional techniques [25]. 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 [26]. 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 [27, 28, 29, 23]. Kalibera *et al.* [27] apply model checking and exhaustive testing to perform scheduling and race-condition analysis in SCJ programs. Haddad *et al.* [29] extend the Java Modeling Language [30] with timing properties to support worst-case execution analysis of SCJ programs, whilst Tang *et al.* [28] use annotations to analyse programs for memory safety and compliance to SCJ levels. Marriott *et al.* [23] 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 [8] 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.

The semantics of *SCJ-Circus* is closely related to the semantics of SCJ and has been extensively validated by review by an SCJ standardisation expert, by use in proving laws [31] and in case studies [32, 33]. In order to validate our approach further, the refinement strategy needs to be implemented in a tool that provides a high level of automation

to allow the verification of larger case studies. Tool support for our approach is currently under development and is based on a formalisation [34] of Unifying Theories of Programming (UTP) [13] and *Circus* in the theorem prover Isabelle [35]. As future work, we plan to complete the implementation of our strategies, and apply them to more examples, including SCJ benchmarks [36]. A more recent result [20] 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.

[1] A. Wellings, Concurrent and Real-Time Programming in Java, John Wiley & Sons, 2004.
[2] D. Locke, B. S. Andersen, M. F. B. Brosgol, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nielsen, M. Schoeberl, J. Vitek, A. Wellings, Safety-Critical Java Technology Specification, Tech. Rep. JSR-302, The Open Group, `https://jcp.org/aboutJava/communityprocess/edr/jsr302/index2.html` (2013).
[3] M. Tofte, J.-P. Talpin, Region-Based Memory Management, Information and Computation 132 (2) (1997) 109 – 176. `doi:10.1006/inco.1996.2613`.
[4] A. Burns, The Ravenscar Profile, Ada Letters XIX (1999) 49–52.
[5] A. L. C. Cavalcanti, F. Zeyda, A. Wellings, J. C. P. Woodcock, K. Wei, Safety-critical Java programs from *Circus* models, Real-Time Systems 49 (5) (2013) 614–667. `doi:10.1007/s11241-013-9182-4`.
[6] J. C. P. Woodcock, J. Davies, Using Z—Specification, Refinement, and Proof, Prentice-Hall, 1996.
[7] A. W. Roscoe, Understanding Concurrent Systems, Texts in Computer Science, Springer, 2011.
[8] A. L. C. Cavalcanti, P. Clayton, C. O'Halloran, From Control Law Diagrams to Ada via *Circus*, Formal Aspects of Computing 23 (4) (2011) 465–512. `doi:10.1007/s00165-010-0170-3`.
[9] A. Miyazawa, A. L. C. Cavalcanti, Refinement-oriented models of Stateflow charts, Science of Computer Programming 77 (10-11) (2012) 1151–1177. `doi:10.1016/j.scico.2011.07.007`.
[10] A. Miyazawa, A. L. C. Cavalcanti, Refinement-based verification of implementations of Stateflow charts, Formal Aspects of Computing 26 (2) (2013) 367–405. `doi:10.1007/s00165-013-0291-6`.
[11] A. Miyazawa, L. Lima, A. L. C. Cavalcanti, Formal models of SysML blocks, in: L. Groves, J. Sun (Eds.), Formal Methods and Software Engineering, Vol. 8144 of LNCS, Springer, 2013, pp. 249–264. `doi:10.1007/978-3-642-41202-8_17`.
[12] A. Miyazawa, A. Cavalcanti, Formal refinement in SysML, in: E. Albert, E. Sekerinski (Eds.), IFM, Vol. 8739 of LNCS, Springer, 2014, pp. 155–170. `doi:10.1007/978-3-319-10181-1_10`.
[13] C. A. R. Hoare, J. He, Unifying Theories of Programming, Prentice-Hall, 1998.
[14] M. V. M. Oliveira, Formal Derivation of State-Rich Reactive Programs Using *Circus*, Ph.D. thesis, University of York (2006).
[15] A. L. C. . Cavalcanti, A. Sampaio, J. C. P. Woodcock, Unifying Classes and Processes, Software & Systems Modeling 4 (3) (2005) 277–296. `doi:10.1007/s10270-005-0085-2`.
[16] K. Wei, J. Woodcock, A. Cavalcanti, *Circus Time* with Reactive Designs, in: B. Wolff, M.-C. Gaudel, A. Feliachi (Eds.), Unifying Theories of Programming: 4th International Symposium, UTP 2012, Paris, France, August 27-28, 2012, Revised Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 68–87. `doi:10.1007/978-3-642-35705-3_3`.
[17] F. Zeyda, L. Lalkhumsanga, A. Cavalcanti, A. Wellings, Circus Models for Safety-Critical Java Programs, The Computer Journal 57 (7) (2014) 1046–1091. `doi:10.1093/comjnl/bxt060`.
[18] A. Miyazawa, A. L. C. Cavalcanti, SCJ-Circus: a refinement-oriented formal notation for Safety-Critical Java, in: Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015., 2015, pp. 71–86. `doi:10.4204/EPTCS.209.6`.
[19] A. Miyazawa, A. Cavalcanti, Refinement Strategies for Safety-Critical Java, in: M. Cornélio, B. Roscoe (Eds.), Formal Methods: Foundations and Applications, Springer International Publishing, Cham, 2016, pp. 93–109.
[20] M. Luckcuck, A. L. C. Cavalcanti, A. Wellings, A Formal Model of the Safety-Critical Java Level 2 Paradigm, in: E. Ábrahám, M. Huisman (Eds.), Integrated Formal Methods, LNCS, Springer, 2016, pp. 226–241. `doi:10.1007/978-3-319-33693-0_15`.
[21] E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.
[22] A. Miyazawa, A. L. C. Cavalcanti, A. Wellings, *SCJ-Circus*: Report on specification and refinement of Safety-Critical Java programs, `https://www.cs.york.ac.uk/circus/hijac/files/scp-refinement-report.pdf` (2016).
[23] C. Marriott, A. L. C. Cavalcanti, SCJ: Memory-Safety Checking without Annotations, in: C. Jones, P. Pihlajasaari, J. Sun (Eds.), FM 2014: Formal Methods: 19th International Symposium, Singapore, May 12-16, 2014. Proceedings, Springer International Publishing, Cham, 2014, pp. 465–480. `doi:10.1007/978-3-319-06410-9_32`.
[24] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A. Roscoe, FDR3 — A Modern Refinement Checker for CSP, in: E. Abraham, K. Havelund (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Vol. 8413 of LNCS, 2014, pp. 187–201.
[25] P. Antonino, A. Sampaio, J. C. P. Woodcock, A Refinement Based Strategy for Local Deadlock Analysis of Networks of CSP Processes, in: C. Jones, P. Pihlajasaari, J. Sun (Eds.), FM 2014: Formal Methods: 19th International Symposium, Singapore, May 12-16, 2014. Proceedings, Springer International Publishing, Cham, 2014, pp. 62–77. `doi:10.1007/978-3-319-06410-9_5`.
[26] J. C. P. Woodcock, The Miracle of Reactive Programming, in: Unifying Theories of Programming, LNCS, Springer, 2009, pp. 202–217.
[27] T. Kalibera, P. Parizek, M. Malohlava, M. Schoeberl, Exhaustive Testing of Safety Critical Java, in: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, ACM, 2010, pp. 164–174. `doi:10.1145/1850771.1850794`.
[28] D. Tang, A. Plsek, J. Vitek, Static Checking of Safety Critical Java Annotations, in: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, ACM, 2010, pp. 148–154. `doi:10.1145/1850771.1850792`.
[29] G. Haddad, F. Hussain, G. T. Leavens, The Design of SafeJML, a Specification Language for SCJ with Support for WCET Specification, in: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, ACM, 2010, pp. 155–163. `doi:10.1145/1850771.1850793`.

[30] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll, An Overview of JML Tools and Applications, International Journal on Software Tools for Technology Transfer 7 (3) (2005) 212–232. doi:10.1007/s10009-004-0167-4.
[31] F. Zeyda, A. L. C. Cavalcanti, Laws of mission-based programming, Formal Aspects of Computing (2015) 1–50doi:10.1007/s00165-014-0317-8.
[32] A. Wellings, M. Luckcuck, A. L. C. Cavalcanti, Safety-critical java level 2: motivations, example applications and issues, in: The 11th International Workshop on Java Technologies for Real-time and Embedded Systems, ACM, 2013, pp. 48–57.
[33] N. K. Singh, A. J. Wellings, A. L. C. Cavalcanti, The cardiac pacemaker case study and its implementation in safety-critical Java and Ravenscar Ada, in: 10th International Workshop on Java Technologies for Real-time and Embedded Systems, ACM, 2012, pp. 62–71.
[34] S. Foster, F. Zeyda, J. Woodcock, Isabelle/UTP: A Mechanised Theory Engineering Framework, in: D. Naumann (Ed.), Unifying Theories of Programming, Springer International Publishing, Cham, 2015, pp. 21–41.
[35] T. Nipkow, M. Wenzel, L. C. Paulson, Isabelle/HOL: A Proof Assistant for Higher-order Logic, Springer-Verlag, Berlin, Heidelberg, 2002.
[36] F. Zeyda, A. L. C. Cavalcanti, A. Wellings, J. C. P. Woodcock, K. Wei, Refinement of the Parallel CDx, Tech. rep., University of York, Department of Computer Science, York, YO10 5GH, UK, http://www.cs.york.ac.uk/circus/publications/techreports/reports/cdx_report.pdf (July 2012).

Appendix A. Refinement laws

Law [a-handler-split*]

$$SH \longrightarrow sh \longrightarrow ah \longrightarrow \mu\, X \bullet (c \longrightarrow (A \blacktriangleright D);\ X \,\Box\, dh \longrightarrow \mathbf{Skip})$$
$$\sqsubseteq$$
$$\left(\begin{array}{l} SH \longrightarrow sh \longrightarrow \mu\, X \bullet ((haeC \longrightarrow (A \blacktriangleright D);\ haeR \longrightarrow X) \,\Box\, dh \longrightarrow \mathbf{Skip}) \\ \quad [\![wrtV(A) \mid \{\!| sh, dh, haeC, haeR |\!\} \mid \{\}]\!] \\ sh \longrightarrow ah \longrightarrow \mu\, X \bullet (c \longrightarrow haeC \longrightarrow haeR \longrightarrow X \,\Box\, dh \longrightarrow \mathbf{Skip}) \end{array}\right) \setminus \{\!| haeC, haeR |\!\}$$

provided $\{\!| sh, dh, haeC, haeR |\!\} \cap usedC(A) = \varnothing$

Lemma [aperiodic-handler-fw-cl*]

$$SH \longrightarrow start\_aeh?x!id \longrightarrow \mu\, X \bullet \left(\begin{array}{l} handleAsyncEventCall?x!id \longrightarrow (A \blacktriangleright D;\ handleAsyncEventRet!x!id \longrightarrow X) \\ \Box done\_handler.id \longrightarrow \mathbf{Skip} \end{array}\right)$$
$$[\![wrtV(A) \mid \{\!| start\_aeh, done\_handler, handleAsyncEventCall, handleAsyncEventRet |\!\} \mid \{\}]\!]$$
$$start\_aeh?x.id \longrightarrow activate\_handlers \longrightarrow \mathbf{wait}\ start;$$
$$\left(\begin{array}{l} \mu\, X \bullet \left(\begin{array}{l} (c \longrightarrow handleAsyncEventCall?x!id \longrightarrow handleAsyncEventRet!x!id \longrightarrow \mathbf{Skip});\ X \\ \Box done\_handler.id \longrightarrow \mathbf{Skip} \end{array}\right) \\ [\![\{\} \{\!| c |\!\} \mid \{\}]\!]\ Buffer \end{array}\right) \setminus \{\!| c |\!\}$$
$$\sqsubseteq$$
$$SH \longrightarrow start\_aeh?x!id \longrightarrow \mu\, X \bullet \left(\begin{array}{l} handleAsyncEventCall?x!id \longrightarrow (A \blacktriangleright D;\ handleAsyncEventRet!x!id \longrightarrow X) \\ \Box done\_handler.id \longrightarrow \mathbf{Skip} \end{array}\right)$$
$$[\![wrtV(A) \mid \{\!| start\_aeh, done\_handler, handleAsyncEventCall, handleAsyncEventRet |\!\} \mid \{\}]\!]$$
$$\mathbf{var}\ currentRelease : \mathbb{B};\ newRelease : \mathbb{B} \bullet start\_aeh!id \longrightarrow activate\_handlers \longrightarrow$$
$$\left(\begin{array}{l} \mu\, X \bullet \left(\begin{array}{l} (currentRelease = false)\ \&\ (releaseCall?o!id \longrightarrow (currentRelease := true;\ releaseRet!o!id \longrightarrow \mathbf{Skip}));\ X \\ \Box(currentRelease = true)\ \&\ (releaseCall?o!id \longrightarrow (newRelease := true;\ releaseRet!o!id \longrightarrow \mathbf{Skip}));\ X \\ \Box(currentRelease = true)\ \&\ (hasRelease \longrightarrow currentRelease := newRelease);\ X \,\Box\, done\_hander!id \longrightarrow \mathbf{Skip} \end{array}\right) \\ [\![currentRelease, newRelease \mid \{\!| hasRelease, done\_handler |\!\} \mid \{\}]\!] \\ \mu\, X \bullet hasRelease \longrightarrow handleAsyncEventCall!id!id \longrightarrow handleAsyncEventRet!id!id \longrightarrow X \,\Box\, done\_handler!id \longrightarrow \mathbf{Skip} \end{array}\right)$$
$$\setminus \{\!| hasRelease |\!\}$$

where

$$Buffer \mathrel{\widehat{=}} \mathbf{var}\ pending : \mathbb{B} \bullet pending := false;\ \mu\, X \bullet \left(\begin{array}{l} releaseCall?o!id \longrightarrow pending := true;\ releaseRet!o!id \longrightarrow X \\ \Box(pending)\ \&\ c \longrightarrow pending := false;\ X \,\Box\, done\_handler.id \longrightarrow \mathbf{Skip} \end{array}\right)$$

Law [call-intro*]

$$F(A) \sqsubseteq (F(cs \longrightarrow ce \longrightarrow \mathbf{Skip}) \llbracket \overline{wrtV(A)} \mid \{\!| cs, ce |\!\} \mid wrtV(A) \rrbracket cs \longrightarrow A;\ ce \longrightarrow \mathbf{Skip}) \setminus \{\!| cs, ce |\!\}$$

provided

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

Law [handler-extract*]

$$\begin{array}{l} sm \longrightarrow ah \longrightarrow (\mu X \bullet A;\ X \,\square\, dh \longrightarrow \mathbf{Skip});\ dm \longrightarrow \mathbf{Skip} \\ \sqsubseteq \\ \left(\begin{array}{l} (sm \longrightarrow SH \longrightarrow r \longrightarrow sh \longrightarrow ah \longrightarrow dh \longrightarrow dm \longrightarrow \mathbf{Skip}) \setminus \{\!| r |\!\} \\ \quad \llbracket \{\} \mid \{\!| sh, ah, dh, SH |\!\} \mid usedV(A) \rrbracket \\ SH \longrightarrow sh \longrightarrow ah \longrightarrow \mu X \bullet (A;\ X \,\square\, dh \longrightarrow \mathbf{Skip}) \end{array} \right) \setminus \{\!| sh, SH |\!\} \end{array}$$

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

Lemma [mission-fw-cl*]

$$
\begin{array}{l}
start\_mission!id?s \longrightarrow missionInitializeCall?x!id \longrightarrow SH_1 \longrightarrow SH_2 \longrightarrow register.id_1 \longrightarrow register.id_2 \longrightarrow \\
\quad missionInitializeRet!x!id \longrightarrow done\_mission.id \longrightarrow \mathbf{Skip} \\
[\![\{\} \mid \{\!| start\_mission, register, done\_mission, missionInitializeCall, missionInitializeRet |\!\} \mid \{\}]\!] \\
\mathbf{var}\ handlers : \mathbb{P}\, ID \bullet handlers := \{\};\ start\_mission?x \longrightarrow \\
\left(\begin{array}{l}
requestTerminationCall?x!id \longrightarrow requestTerminationRet!x!id \longrightarrow \\
\quad \mu\, X \bullet (requestTerminationCall?x!id \longrightarrow requestTerminationRet!x!id \longrightarrow X \\
\qquad \Box\ stop\_handlers \longrightarrow X \ \Box\ done\_mission.id \longrightarrow \mathbf{Skip}) \\
\quad [\![\{\} \mid \{\!| done\_mission, stop\_handlers |\!\} \mid \{handlers\}]\!] \\
\quad missionInitializeCall!id!id \longrightarrow \\
\quad \mu\, X \bullet (register?h \longrightarrow handlers := handler \cup \{h\};\ X \ \Box\ missionInitializeRet!id!id \longrightarrow \mathbf{Skip}); \\
\quad \left(\begin{array}{l} [\![\{\!| activate\_handlers, stop\_handlers |\!\}]\!]\ h : handlers \bullet (start\_aeh.h \longrightarrow \mathbf{Skip} \ \Box\ start\_peh.h \longrightarrow \mathbf{Skip}); \\ \quad activate\_handlers \longrightarrow stop\_handlers \longrightarrow done\_handler.h \longrightarrow \mathbf{Skip} \end{array}\right); \\
\quad done\_mission.x \longrightarrow \mathbf{Skip}
\end{array}\right) \\
\setminus \{\!| stop\_handlers |\!\} \\
\sqsubseteq \\
\left(\begin{array}{l}
start\_mission!id?s \longrightarrow \mu\, X \bullet \\
\left(\begin{array}{l} missionInitializeCall?x!id \longrightarrow SH_1 \longrightarrow SH_2 \longrightarrow register.id_1 \longrightarrow register.id_2 \longrightarrow missionInitializeRet!x!id \longrightarrow X \\ \Box cleanUpCall?x!id \longrightarrow cleanUpRet!x!id \longrightarrow X \ \Box\ end\_mission\_app.id \longrightarrow \mathbf{Skip} \end{array}\right) \\
[\![\{\} \mid \{\!| start\_mission, register, done\_mission, missionInitializeCall, missionInitializeRet, \\
\quad requestTerminationCall, requestTerminationRet, terminationPendingCall, terminationPendingRet |\!\} \mid \{\}]\!] \\
\mathbf{var}\ sequencer : ID;\ handlers : \mathbb{P}\, ID;\ terminating : \mathbb{B} \bullet sequencer := null;\ handlers := \{\};\ terminating := false; \\
\quad start\_mission!id?s \longrightarrow sequencer := s;\ missionInitializeCall?x!id \longrightarrow \\
\quad \mu\, X \bullet \left(\begin{array}{l} register?h \longrightarrow handlers := handlers \cup \{h\};\ X \ \Box\ missionInitializeRet!x!id \longrightarrow \mathbf{Skip} \end{array}\right); \\
\quad \left(\begin{array}{l}
\mathbf{if}\ handlers = \{\} \longrightarrow \mathbf{Skip} \\
[\!]\ handlers \neq \{\} \longrightarrow \\
\quad \left(\begin{array}{l}
(|||h : handlers \bullet [\{\}]start\_peh.h?s?p \longrightarrow \mathbf{Skip} \ \Box\ start\_aeh.h \longrightarrow \mathbf{Skip});\ activate\_handlers \longrightarrow \\
stop\_handlers \longrightarrow (|||h : handlers \bullet [\{\}]done\_handler.h \longrightarrow \mathbf{Skip});\ done\_handlers \longrightarrow \mathbf{Skip} \\
[\![\{\!| stop\_handlers, done\_handlers |\!\}]\!] \\
\qquad \mu\, X \bullet \left(\begin{array}{l}
requestTerminationCall?x!id \longrightarrow \\
\quad \left(\begin{array}{l} \left(\begin{array}{l} \mathbf{if}\ \neg\ terminating \longrightarrow (terminating := true;\ stop\_handlers \longrightarrow \mathbf{Skip}) \\ [\!]\ terminating \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right); \\ requestTerminationRet!x!id \longrightarrow \mathbf{Skip} \end{array}\right); X \\
\Box terminationPendingCall?x!id \longrightarrow terminationPendingRet!x!id!terminating \longrightarrow \mathbf{Skip};\ X \\
\Box done\_handlers \longrightarrow \mathbf{Skip}
\end{array}\right) \\
\setminus \{\!| stop\_handlers, done\_handlers |\!\}
\end{array}\right) \\
\quad \mathbf{fi}
\end{array}\right); \\
\quad cleanUpCall!id!id \longrightarrow cleanUpRet!id!id?x \longrightarrow end\_mission\_app!id \longrightarrow done\_mission!id \longrightarrow \mathbf{Skip}
\end{array}\right) \\
\setminus \{\!| stop\_handlers, cleanUpCall, end\_mission\_app |\!\}
\end{array}
$$

where *Channels SH_1 and SH_2 correspond to the constructor channels of the handlers with identifiers id_1 and id_2.*

Lemma [mission-fw-cl-single*]

$$
\begin{array}{l}
\left(\begin{array}{l}
start\_sequencer \longrightarrow \\
\quad \mu X \bullet \left(\begin{array}{l} getNextMissionCall!id!id \longrightarrow getNextMissionRet!id!id?m \longrightarrow \\ \quad \left(\begin{array}{l} \textbf{if } m = null \longrightarrow end\_sequencer\_app \longrightarrow \textbf{Skip} \\ [\!] m \neq null \longrightarrow start\_mission.m \longrightarrow done\_mission.m.true \longrightarrow X \\ \textbf{fi} \end{array}\right) \end{array}\right); \\
\quad done\_sequencer \longrightarrow \textbf{Skip} \\
[\![\{\} \mid \{\!| getNextMissionCall, getNextMissionRet, end\_sequencer\_app |\!\} \mid \{\}]\!] \\
\textbf{var } b : \mathbb{B} \bullet b := false; \\
\quad \mu X \bullet \left(\begin{array}{l} getNextMissionCall?x!id \longrightarrow \left(\begin{array}{l} \textbf{if } b \longrightarrow getNextMissionRet!x!id!null \longrightarrow \textbf{Skip} \\ [\!] \neg b \longrightarrow getNextMissionRet!x!id!MID \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array}\right); X \\ \quad \Box \\ end\_sequencer\_app \longrightarrow \textbf{Skip} \end{array}\right)
\end{array}\right) \\
\setminus \{\!| getNextMissionCall, getNextMissionRet, end\_sequencer\_app |\!\} \sqsubseteq \\
\left(\begin{array}{l}
start\_sequencer \longrightarrow \mu X \bullet getNextMissionCall!id!id \longrightarrow getNextMissionRet!id!id?m \longrightarrow \\
\quad \left(\begin{array}{l} \textbf{if } m \neq null \longrightarrow start\_mission.m \longrightarrow done\_mission.m?continue \longrightarrow \\ \quad (\textbf{if } continue \longrightarrow X [\!] \neg continue \longrightarrow \textbf{Skip fi}) \\ [\!] m = null \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array}\right); \\
\quad end\_sequencer\_app \longrightarrow done\_sequencer \longrightarrow \textbf{Skip} \\
[\![\{\} \mid \{\!| getNextMissionCall, getNextMissionRet, end\_sequencer\_app |\!\} \mid \{\}]\!] \\
\textbf{var } b : \mathbb{B} \bullet b := false; \\
\quad \mu X \bullet \left(\begin{array}{l} getNextMissionCall?x!id \longrightarrow \left(\begin{array}{l} \textbf{if } b \longrightarrow getNextMissionRet!x!id!null \longrightarrow \textbf{Skip} \\ [\!] \neg b \longrightarrow getNextMissionRet!x!id!MID \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array}\right); X \\ \quad \Box \\ end\_sequencer\_app \longrightarrow \textbf{Skip} \end{array}\right)
\end{array}\right) \\
\setminus \{\!| getNextMissionCall, getNextMissionRet, end\_sequencer\_app |\!\}
\end{array}
$$

Law [mission-split*]

$$
\begin{array}{l}
F\left(\begin{array}{l} sm \longrightarrow \left(\begin{array}{l} SH_1 \longrightarrow r.id_1 \longrightarrow sh.id_1 \longrightarrow ah \longrightarrow dh.id_1 \longrightarrow \textbf{Skip} \\ \quad [\![\{\!| ah |\!\}]\!] \\ SH_2 \longrightarrow r.id_2 \longrightarrow sh.id_2 \longrightarrow ah \longrightarrow dh.id_2 \longrightarrow \textbf{Skip} \end{array}\right) \setminus \{\!| r |\!\}; \ dm \longrightarrow \textbf{Skip} \end{array}\right) \setminus \{\!| SH_1, SH_2 |\!\} \\
\sqsubseteq \\
F\left(\begin{array}{l}
\left(\begin{array}{l} sm \longrightarrow miC \longrightarrow \left(\begin{array}{l} SH_1 \longrightarrow r.id_1 \longrightarrow \textbf{Skip} \\ ||| \\ SH_2 \longrightarrow r.id_2 \longrightarrow \textbf{Skip} \end{array}\right); \ miR \longrightarrow dm \longrightarrow \textbf{Skip} \\ \quad [\![\{\} \mid \{\!| sm, r, dm, miC, miR |\!\} \mid \{\}]\!] \\ sm \longrightarrow miC \longrightarrow \left(\begin{array}{l} r.id_1 \longrightarrow \textbf{Skip} \\ ||| \\ r.id_2 \longrightarrow \textbf{Skip} \end{array}\right); \ miR \longrightarrow \textbf{Skip}; \\ \quad \left(\begin{array}{l} sh.id_1 \longrightarrow ah \longrightarrow dh.id_1 \longrightarrow \textbf{Skip} \\ [\![\{\!| ah |\!\}]\!] \\ sh.id_2 \longrightarrow ah \longrightarrow dh.id_2 \longrightarrow \textbf{Skip} \end{array}\right); \ dm \longrightarrow \textbf{Skip} \end{array}\right) \setminus \{\!| miC, miR, r |\!\}
\end{array}\right) \setminus \{\!| SH_1, SH_2 |\!\}
\end{array}
$$

provided $\{\!| miC, miR |\!\} \cap usedC(LHS) = \varnothing$

Law [p-handler-split*]

$$
\begin{array}{l}
SH \longrightarrow sh.id.s.p \longrightarrow ah \longrightarrow \mathbf{wait}\ s;\ \mu X \bullet ((A \blacktriangleright D \ |||\ \mathbf{wait}\ p);\ X \,\Box\, dh \longrightarrow \mathbf{Skip}) \\
\sqsubseteq \\
\left(\begin{array}{l}
SH \longrightarrow sh.id.s.p \longrightarrow \mu X \bullet ((haeC \longrightarrow A \blacktriangleright D;\ haeR \longrightarrow X) \,\Box\, dh \longrightarrow \mathbf{Skip}) \\
\quad [\![wrtV(A) \mid \{\!| sh, dh, haeC, haeR |\!\} \mid \{\}]\!] \\
sh.id?start?period \longrightarrow ah \longrightarrow \mathbf{wait}\ start;\ \mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} (haeC \longrightarrow haeR \longrightarrow \mathbf{Skip}) \blacktriangleleft 0 \blacktriangleright period \\ ||| \ \mathbf{wait}\ period \end{array} \right); X \\ \Box dh \longrightarrow \mathbf{Skip} \end{array} \right)
\end{array} \right) \\
\setminus \{\!| haeC, haeR |\!\}
\end{array}
$$

provided $\{\!| sh, dh, haeC, haeR |\!\} \cap usedC(A) = \varnothing \wedge D \leq p.$

Law [par-prefix-dist*]

$$
\begin{array}{l}
(A \ [\![ns_1 \mid cs \mid ns_2]\!] \ B);\ c \longrightarrow \mathbf{Skip} \\
\sqsubseteq \\
(A;\ c \longrightarrow \mathbf{Skip} \ [\![ns_1 \mid cs \cup \{\!| c |\!\} \mid ns_2]\!] \ B;\ c \longrightarrow \mathbf{Skip})
\end{array}
$$

provided $\neg\ c \in usedC(A) \cup usedC(B).$

Lemma [periodic-handler-fw-cl*]

$$
\begin{array}{l}
SH \longrightarrow start\_peh?x!id!s!p \longrightarrow \mu X \bullet \left(\begin{array}{l} handleAsyncEventCall?x!id \longrightarrow (A \blacktriangleright D;\ handleAsyncEventRet!x!id \longrightarrow X) \\ \Box done\_handler.id \longrightarrow \mathbf{Skip} \end{array} \right) \\
[\![wrtV(A) \mid \{\!| start\_peh, done\_handler, handleAsyncEventCall, handleAsyncEventRet |\!\} \mid \{\}]\!] \\
start\_peh?x.id?start?period \longrightarrow activate\_handlers \longrightarrow \mathbf{wait}\ start; \\
\mu X \bullet \left(\begin{array}{l} ((handleAsyncEventCall?x!id \longrightarrow handleAsyncEventRet!x!id \longrightarrow \mathbf{Skip}) \blacktriangleleft 0 \blacktriangleright period \ ||| \ \mathbf{wait}\ period);\ X \\ \Box done\_handler.id \longrightarrow \mathbf{Skip} \end{array} \right) \\
\sqsubseteq \\
SH \longrightarrow start\_peh?x!id!s!p \longrightarrow \mu X \bullet \left(\begin{array}{l} handleAsyncEventCall?x!id \longrightarrow (A \blacktriangleright D;\ handleAsyncEventRet!x!id \longrightarrow X) \\ \Box done\_handler.id \longrightarrow \mathbf{Skip} \end{array} \right) \\
[\![wrtV(A) \mid \{\!| start\_peh, done\_handler, handleAsyncEventCall, handleAsyncEventRet |\!\} \mid \{\}]\!] \\
\mathbf{var}\ start, period : \mathbb{R} \bullet \\
\left(\begin{array}{l} start\_peh!id?s?p \longrightarrow activate\_handlers \longrightarrow start := s;\ period := p;\ \mathbf{wait}\ start; \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} \mu X \bullet \left(\begin{array}{l} (handleAsyncEventCall!id \longrightarrow handleAsyncEventRet!id \longrightarrow Skip) \blacktriangleright period;\ X \\ \Box \\ done\_handler!id \longrightarrow \mathbf{Skip} \end{array} \right) \\ [\![\{\} \mid \{\!| handleAsyncEventCall.id, done\_handler.id |\!\} \mid \{\}]\!] \\ (\mu Y \bullet ((handleAsyncEventCall!id \longrightarrow \mathbf{wait}\ period) \blacktriangleleft 0);\ Y) \end{array} \right) \\ \triangle\ done\_handler!id \longrightarrow \mathbf{Skip} \end{array} \right) \end{array} \right)
\end{array}
$$

Law [prefix-introduction]

$$A = (c \longrightarrow A) \setminus \{\!| c |\!\}$$

provided $c \notin usedC(A).$

Law [prefix-par-dist]

$$c \longrightarrow (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) = (c \longrightarrow A_1 \llbracket ns_1 \mid cs \cup \{\!| c |\!\} \mid ns_2 \rrbracket c \longrightarrow A_2)$$

provided $c \notin usedC(A_1) \cup usedC(A_2) \vee c \in cs$.

Law [rec-interleave*]

$$\begin{array}{l}
\left(\begin{array}{l}
sm \longrightarrow miC \longrightarrow SH_1 \longrightarrow SH_2 \longrightarrow r.id_1 \longrightarrow r.id_2 \longrightarrow miR \longrightarrow dm \longrightarrow \mathbf{Skip} \\
[\![\{\} \mid \{\!| sm, r, dm, miC, miR |\!\} \mid \{\}]\!] \\
sm \longrightarrow miC \longrightarrow \left(r.id_1 \longrightarrow \mathbf{Skip} \;|||\; r.id_2 \longrightarrow \mathbf{Skip} \right); \; miR \longrightarrow \mathbf{Skip}; \\
\quad \left(\begin{array}{l} sh.id_1 \longrightarrow ah \longrightarrow dh.id_1 \longrightarrow \mathbf{Skip} \\ \quad [\![\{\!| ah |\!\}]\!] \\ sh.id_2 \longrightarrow ah \longrightarrow dh.id_2 \longrightarrow \mathbf{Skip} \end{array} \right); \; dm \longrightarrow \mathbf{Skip}
\end{array}\right) \setminus \{\!| miC, miR, r |\!\} \\
\sqsubseteq \\
\left(\begin{array}{l}
sm \longrightarrow miC \longrightarrow SH_1 \longrightarrow SH_2 \longrightarrow r.id_1 \longrightarrow r.id_2 \longrightarrow miR \longrightarrow dm \longrightarrow \mathbf{Skip} \\
\quad [\![\{\} \mid \{\!| sm, r, dm, miC, miR |\!\} \mid \{\}]\!] \\
\mathbf{var}\; handlers : \mathbb{P}\, ID \bullet handlers := \{\}; \\
sm \longrightarrow miC \longrightarrow \mu\, X \bullet \left(\begin{array}{l} r?h \longrightarrow handlers := handlers \cup \{h\}; \; X \\ \Box miR \longrightarrow \mathbf{Skip} \end{array} \right); \\
\quad ([\![\{\!| ah |\!\}]\!] \; h : handlers \bullet sh.h \longrightarrow ah \longrightarrow dh.h \longrightarrow \mathbf{Skip}); \; dm \longrightarrow \mathbf{Skip}
\end{array}\right) \setminus \{\!| miC, miR, r |\!\}
\end{array}$$

provided *handlers is fresh.*

Lemma [safelet-fw-cl*]

$$\begin{array}{l}
\left(\begin{array}{l}
getSequencerCall?x!id \longrightarrow getSequencerRet!x!id!sid \longrightarrow end_safelet_app \longrightarrow \mathbf{Skip} \\
\quad [\![\{\} \mid \{\!| getSequencerCall, getSequencerRet, end_safelet_app |\!\} \mid \{\} \\
getSequencerCall!id!id \longrightarrow getSequencerRet!id!id?s \longrightarrow start_sequencer \longrightarrow done_sequencer \longrightarrow \\
\quad end_safelet_app \longrightarrow \mathbf{Skip}
\end{array}\right) \\
\setminus \{\!| getSequencerCall, getSequencerRet, end_safelet_app |\!\} \\
\sqsubseteq \\
\left(\begin{array}{l}
\mu\, X \bullet \left(\begin{array}{l} getSequencerCall?x!id \longrightarrow \left(\begin{array}{l} \mathbf{var\ return} : ID \bullet \mathbf{return} := sid; \\ getSequencerRet!x!id!\mathbf{return} \longrightarrow \mathbf{Skip} \end{array} \right); \; X \\ \Box \\ safeletInitializeCall?x!id \longrightarrow safeletInitializeRet!x!id \longrightarrow X \\ \Box \\ end_safelet_app \longrightarrow \mathbf{Skip} \end{array} \right) \\
[\![\{\} \mid \{\!| getSequencerCall, getSequencerRet, safeletInitializeCall, safeletInitializeRet, end_safelet_app |\!\} \mid \{\}]\!] \\
safeletInitializeCall!id!id \longrightarrow safeletInitializeRet!id!id \longrightarrow \\
\quad getSequencerCall!id!id \longrightarrow getSequencerRet!id!id?s \longrightarrow \\
\quad \left(\begin{array}{l} \mathbf{if}\; s \neq null \longrightarrow start_sequencer \longrightarrow done_sequencer \longrightarrow \mathbf{Skip} \\ [\!]\, s = null \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array} \right); \\
\quad end_safelet_app \longrightarrow \mathbf{Skip}
\end{array}\right) \\
\setminus \{\!| getSequencerCall, getSequencerRet, end_safelet_app, safeletInitializeCall, safeletInitializeRet |\!\}
\end{array}$$

provided $sid \neq null$

Law [seq-interleave*]

$$
\left(\begin{array}{l}
\left(\begin{array}{l}
sm \longrightarrow miC \longrightarrow \left(\begin{array}{l} SH_1 \longrightarrow r.id_1 \longrightarrow \mathbf{Skip} \\ ||| \\ SH_2 \longrightarrow r.id_2 \longrightarrow \mathbf{Skip} \end{array}\right); \; miR \longrightarrow dm \longrightarrow \mathbf{Skip} \\
\quad [\![\{\} \mid \{\!| sm, r, dm, miC, miR |\!\} \mid \{\}]\!] \\
FA \\
\quad [\![\{\!| SH_1, SH_2 |\!\}]\!] \\
(SH_1 \longrightarrow A_1 \parallel SH_2 \longrightarrow A_2)
\end{array}\right) \setminus \{\!| miC, miR, r |\!\}
\end{array}\right) \setminus \{\!| SH_1, SH_2 |\!\}
$$

$$
\sqsubseteq
$$

$$
\left(\begin{array}{l}
\left(\begin{array}{l}
sm \longrightarrow miC \longrightarrow SH_1 \longrightarrow SH_2 \longrightarrow r.id_1 \longrightarrow r.id_2 \longrightarrow miR \longrightarrow dm \longrightarrow \mathbf{Skip} \\
\quad [\![\{\} \mid \{\!| sm, r, dm, miC, miR |\!\} \mid \{\}]\!] \\
FA \\
\quad [\![\{\!| SH_1, SH_2 |\!\}]\!] \\
(SH_1 \longrightarrow A_1 \parallel SH_2 \longrightarrow A_2)
\end{array}\right) \setminus \{\!| miC, miR, r |\!\}
\end{array}\right) \setminus \{\!| SH_1, SH_2 |\!\}
$$

where

$$
FA \mathrel{\widehat{=}} sm \longrightarrow miC \longrightarrow \left(\begin{array}{l} r.id_1 \longrightarrow \mathbf{Skip} \\ ||| \\ r.id_2 \longrightarrow \mathbf{Skip} \end{array}\right); \; miR \longrightarrow \left(\begin{array}{l} sh.id_1 \longrightarrow ah \longrightarrow dh.id_1 \longrightarrow \mathbf{Skip} \\ \quad [\![\{\!| ah |\!\}]\!] \\ sh.id_2 \longrightarrow ah \longrightarrow dh.id_2 \longrightarrow \mathbf{Skip} \end{array}\right); \; dm \longrightarrow \mathbf{Skip}
$$

Lemma [sequencer-split-mult-seq*]

$$
\begin{array}{l}
start\_sequencer \longrightarrow (;\; i : I \bullet start\_mission.i \longrightarrow done\_mission.i \longrightarrow \mathbf{Skip}); \\
\quad done\_sequencer \longrightarrow \mathbf{Skip} \\
\sqsubseteq \\
\left(\begin{array}{l}
start\_sequencer \longrightarrow \\
\quad \mu X \bullet \left(\begin{array}{l} getNextMissionCall!id!id \longrightarrow getNextMissionRet!id!id?m \longrightarrow \\ \quad \left(\begin{array}{l} \mathbf{if}\ m = null \longrightarrow end\_sequencer\_app \longrightarrow \mathbf{Skip} \\ [\!] m \neq null \longrightarrow start\_mission.m \longrightarrow done\_mission.m \longrightarrow X \\ \mathbf{fi} \end{array}\right) \end{array}\right); \\
\quad done\_sequencer \longrightarrow \mathbf{Skip} \\
[\![\{\} \mid \{\!| getNextMissionCall, getNextMissionRet, end\_sequencer\_app |\!\} \mid \{\}]\!] \\
\mathbf{var}\ i : nat \bullet b := 1; \\
\quad \mu X \bullet \left(\begin{array}{l} getNextMissionCall?x!id \longrightarrow \\ \quad \left(\begin{array}{l} \mathbf{if}\ i > \#I \longrightarrow getNextMissionRet!x!id!null \longrightarrow \mathbf{Skip} \\ [\!] n : \mathrm{dom}\, I \bullet i = n \longrightarrow getNextMissionRet!x!id!(I\, n) \longrightarrow i := i + 1 \\ \mathbf{fi} \end{array}\right); \; X \\ \quad \Box \\ \quad end\_sequencer\_app \longrightarrow \mathbf{Skip} \end{array}\right)
\end{array}\right) \\
\setminus \{\!| getNextMissionCall, getNextMissionRet, end\_sequencer\_app |\!\}
\end{array}
$$

provided $I \in \mathrm{seq}\ ID \wedge \#I > 1 \wedge \forall i : I \bullet i \neq null$

Law [sequencer-split-single*]

$$
\begin{array}{l}
start\_sequencer \longrightarrow start\_mission.MID \longrightarrow \\
\quad done\_mission.MID \longrightarrow done\_sequencer \longrightarrow \mathbf{Skip} \\
\sqsubseteq \\
\left(\begin{array}{l}
start\_sequencer \longrightarrow \\
\quad \mu X \bullet \left(\begin{array}{l} getNextMissionCall!id!id \longrightarrow getNextMissionRet!id!id?m \longrightarrow \\ \quad \left(\begin{array}{l} \mathbf{if}\ m = null \longrightarrow end\_sequencer\_app \longrightarrow \mathbf{Skip} \\ [\!]\, m \neq null \longrightarrow start\_mission.m \longrightarrow done\_mission.m!true \longrightarrow X \\ \mathbf{fi} \end{array} \right) \end{array} \right); \\
\quad done\_sequencer \longrightarrow \mathbf{Skip} \\
[\![\{\} \mid \{\!| getNextMissionCall, getNextMissionRet, end\_sequencer\_app |\!\} \mid \{\}]\!] \\
\mathbf{var}\ b : \mathbb{B} \bullet b := false; \\
\quad \mu X \bullet \left(\begin{array}{l} getNextMissionCall?x!id \longrightarrow \left(\begin{array}{l} \mathbf{if}\ b \longrightarrow getNextMissionRet!x!id!null \longrightarrow \mathbf{Skip} \\ [\!]\, \neg\, b \longrightarrow getNextMissionRet!x!id!MID \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array} \right); X \\ \Box \\ end\_sequencer\_app \longrightarrow \mathbf{Skip} \end{array} \right)
\end{array} \right) \\
\backslash \{\!| getNextMissionCall, getNextMissionRet, end\_sequencer\_app |\!\}
\end{array}
$$

provided $MID \neq null$

Law [sync-async-conv*]

$$
\begin{array}{l}
\mu X \bullet ((c \longrightarrow G) \blacktriangleright D_1 \;|||\; \mathbf{wait}\ P);\ X \,\Box\, end \longrightarrow \mathbf{Skip}\ [\![ns_1 \mid cs \mid ns_2]\!]\ OHandlers \\
\sqsubseteq \\
((\mu X \bullet (c_i \longrightarrow (G \blacktriangleright (D_1 - T)));\ X \,\Box\, end \longrightarrow \mathbf{Skip})\ [\![ns_1 \mid \{\!| c_i, end |\!\} \mid \{\}]\!]\ Buffer) \setminus \{\!| c_i |\!\} \\
\quad [\![ns_1 \mid cs \mid ns_2]\!] \\
OHandlers
\end{array}
$$

where

$$
\begin{array}{l}
Buffer \mathrel{\widehat{=}} \mathbf{var}\ pending : \mathbb{B} \bullet pending := false;\ \mu X \bullet \\
\quad c \longrightarrow pending := true;\ X \,\Box\, (pending)\ \&\ c_i \longrightarrow pending := false;\ X \,\Box\, end \longrightarrow \mathbf{Skip} \\
OHandlers = F(\mu X \bullet (B(c \longrightarrow \mathbf{Skip})_1 \blacktriangleright D_2 \;|||\; \mathbf{wait}\ P);\ X \,\Box\, end \longrightarrow \mathbf{Skip}) \\
TimeReq = \mu X \bullet (c@t : (t < T) \longrightarrow \mathbf{wait}\,(P - t) \,\Box\, \mathbf{wait}\ P);\ X \,\Box\, end \longrightarrow \mathbf{Skip}
\end{array}
$$

provided

- *B is not recursive and LHS is deadlock-free and feasible.*
- $c_i \notin usedC(LHS) \wedge T < D_1 < P \wedge D_2 < P$
- $OHandlers\ [\![\{\!| c |\!\}]\!]\ TimeReq = OHandlers$

Law [termination-intro*]

$$
\begin{array}{l}
\mu X \bullet F;\ X \\
\sqsubseteq \\
(\mu X \bullet F;\ X \,\Box\, t \longrightarrow \mathbf{Skip}\ [\![\{\!| wrtV(F) \mid \{\!| rt, t |\!\} \mid \{\}]\!]\ rt \longrightarrow \mu X \bullet rt \longrightarrow X \,\Box\, t \longrightarrow \mathbf{Skip}) \setminus \{\!| rt, t |\!\}
\end{array}
$$

provided $\{\!| t, rt |\!\} \cap usedC(F) = \varnothing$.

Appendix B. Sample proof

Law [par-prefix-dist*]

$$(A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B);\ c \longrightarrow \mathbf{Skip} = (A;\ c \longrightarrow \mathbf{Skip} \llbracket ns_1 \mid cs \cup \{c\} \mid ns_2 \rrbracket B;\ c \longrightarrow \mathbf{Skip})$$

provided $c \notin usedC(A) \cup usedC(B)$.

Proof

$$\begin{array}{lll}
(A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B);\ c \longrightarrow \mathbf{Skip} & & \\
= (A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B);\ (c \longrightarrow \mathbf{Skip} \llbracket ns_1 \mid \{c\} \mid ns_2 \rrbracket \mathbf{Skip}) & & \textit{(Law C.78[14])} \\
= (A \llbracket ns_1 \mid cs \cup \{c\} \mid ns_2 \rrbracket B);\ (c \longrightarrow \mathbf{Skip} \llbracket ns_1 \mid \{cs \cup \{c\} c\} \mid ns_2 \rrbracket \mathbf{Skip}) & & \textit{(Law C.80[14])} \\
= (A;\ c \longrightarrow \mathbf{Skip} \llbracket ns_1 \mid cs \cup \{c\} \mid ns_2 \rrbracket B;\ c \longrightarrow \mathbf{Skip}) & & \textit{(Law C.88[14])}
\end{array}$$

□

Appendix C. Excerpts of the syntax of *SCJ-Circus*

SCJCProgram ::= {SCJCParagraph}

SCJCParagraph ::= SCJCSafelet | SCJCMissionSequencer
| SCJCMission | SCJCHandler | CircusParagraph

SCJCSafelet ::= **safelet** N $\widehat{=}$ **begin**
{SCJCSafeletProcessParagraph}
[**state** Schema-Expression)]
{SCJCSafeletProcessParagraph}
[**initialize** $\widehat{=}$ SCJCSafeletAction]
{SCJCSafeletProcessParagraph}
getSequencer $\widehat{=}$ **res return** : **sequencer** • SCJCSafeletAction
SCJCSafeletProcessParagraph*
end

SCJCPeriodicHandler ::= **periodic handler handler** N $\widehat{=}$ **begin**
{SCJCHandlerProcessParagraph}
start Expression **period** Expression
{SCJCHandlerProcessParagraph}
[**state** Schema-Expression]
{SCJCHandlerProcessParagraph}
[**initial** $\widehat{=}$ SCJCHandlerAction]
{SCJCHandlerProcessParagraph}
handleAsyncEvent $\widehat{=}$ SCJCHandlerAction
{SCJCHandlerProcessParagraph}
end

SCJCHandlerProcessParagraph ::= Paragraph
| N $\widehat{=}$ SCJCHandlerParametrisedAction
| **nameset** N == NameSetExpression

SCJCHandlerParametrisedAction ::= SCJCHandlerAction
| **val** Declaration • SCJCHandlerParametrisedAction
| **res** Declaration • SCJCHandlerParametrisedAction

SCJCHandlerAction ::= SCJCMissionAction
| N := **newPR** N
| N := **newPR** N(Expression {, Expression})
| N := **newPM** N
| N := **newPM** N(Expression {, Expression})

. . .