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 its own private scoped memory area that is collected at the end of each of its releases.
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...
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
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 AperiodicEventHandler 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 •) 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 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 (µX • ...) 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 (A1 [n1 | cs | n2] A2) 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 (□) for communication on the channels handleAsyncEventCall or done_handler. In this
way, the method handleAsyncEvent can be called through the channel handleAsyncEventCall or the handler can be terminated with a choice of doneHandler. 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 ▶ 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 ◐ e) that specifies that an action A must start within a certain number of time units. In the example, it must start immediately, af-

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait e</td>
<td>wait</td>
<td>Waits for a exactly e time units before terminating.</td>
</tr>
<tr>
<td>wait e1..e2</td>
<td>wait</td>
<td>Waits for any time between e1 and e2 time units before terminating.</td>
</tr>
<tr>
<td>P ▶ e</td>
<td>deadline</td>
<td>Requires that a process P terminate within e time units.</td>
</tr>
<tr>
<td>P ◐ e</td>
<td>start-by</td>
<td>Requires that a process P start within e time units.</td>
</tr>
<tr>
<td>Skip</td>
<td>skip</td>
<td>Terminates immediately without any side effects.</td>
</tr>
<tr>
<td>Stop</td>
<td>deadlock</td>
<td>Refuses all interactions, but does not change the state.</td>
</tr>
<tr>
<td>P [ n1</td>
<td>cs</td>
<td>n2 ] Q</td>
</tr>
<tr>
<td>[]</td>
<td>Q</td>
<td>parallel composition</td>
</tr>
<tr>
<td>μX • P(X)</td>
<td>recursion</td>
<td>Behave like P with occurrences of X replaced by P itself.</td>
</tr>
<tr>
<td>c → P</td>
<td>prefix</td>
<td>Synchronise on channel c and then behave like P.</td>
</tr>
<tr>
<td>c?x → P</td>
<td>input</td>
<td>Synchronise on channel c with any possible value, store the chosen value in x, and behave like P.</td>
</tr>
<tr>
<td>c?x : b(x) → P</td>
<td>restricted input</td>
<td>Synchronise on channel c with any value such that b(x) is true, store the chosen value in x, and behave like P.</td>
</tr>
<tr>
<td>c!e → P</td>
<td>output</td>
<td>Synchronise on channel c with value e and behave like P.</td>
</tr>
<tr>
<td>v := e</td>
<td>assignment</td>
<td>Assign value e to variable v.</td>
</tr>
<tr>
<td>P; Q</td>
<td>sequential composition</td>
<td>Behave like P, and once P terminates, behave like Q.</td>
</tr>
<tr>
<td>P □ Q</td>
<td>external choice</td>
<td>Allow the environment to choose between behaving like P or Q.</td>
</tr>
<tr>
<td>P ▪ Q</td>
<td>internal choice</td>
<td>Non-deterministically choose between behaving like P or Q.</td>
</tr>
<tr>
<td>P △ c → Q</td>
<td>interrupt</td>
<td>Behave like P, until synchronisation on c becomes possible, at which point, behave like Q.</td>
</tr>
<tr>
<td>var v : T • P</td>
<td>local variable</td>
<td>Declare a local variable v of type T and behave like P within the context of the variable.</td>
</tr>
<tr>
<td>P \ cs</td>
<td>hiding</td>
<td>Run P with events in cs hidden.</td>
</tr>
<tr>
<td>(e) ⊆ P</td>
<td>guard</td>
<td>If e is true, behave like P, otherwise deadlock.</td>
</tr>
<tr>
<td>c@t → P</td>
<td>timed prefix</td>
<td>Record in c the amount of time elapsed between the initial offer of c and its actual occurrence.</td>
</tr>
<tr>
<td>[cs] i : I • P(i)</td>
<td>iterated parallelism</td>
<td>Run P(i) in parallel for all i in I synchronising in cs.</td>
</tr>
<tr>
<td>[[[i : I • P(i)]]</td>
<td>iterated interleave</td>
<td>Run P(i) in parallel for all i in I without synchronisation.</td>
</tr>
<tr>
<td>; i : I • P(i)</td>
<td>iterated sequential composition</td>
<td>Run P(i) in sequence for all i in I.</td>
</tr>
</tbody>
</table>

Table 1: Summary of Circus operators
process PEHFW \equiv id : IDENTIFIER \bullet begin
  state PEHFWState \equiv [start, period : N]
  Execute \equiv \text{wait start};
  \begin{align*}
  (\mu X \bullet &
  \begin{cases}
  \begin{array}{l}
  \text{start} \rightarrow \text{activate handlers} \rightarrow \begin{cases}
  \text{start} := s; \text{period} := p; \text{Execute}, X
  \end{cases}
  \end{array}
  \end{cases}
  \end{align*}
end

Figure 5: Framework process of the periodic event handler.

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

Processes can be composed, via CSP operators, to define other processes. In \text{Circus Time}, \text{wait} and \text{deadline} operators can define time restrictions. In \text{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 \text{Circus} processes that use specific combinations of notations. The strategy applies to an abstract \text{Circus Time} model called an \text{A-anchor}, which is first refined into an \text{O-anchor}: an object-oriented model written in \text{OhCircus}. The \text{O-anchor} is further refined into an \text{E-anchor}, which introduces the architectural design induced by the SCJ execution model. Finally, in the third phase, the \text{E-anchor} is refined into an \text{S-anchor} that uses \text{SCJ-Circus}. While the first two phases are fully explored in previous works [5], it is only briefly indicated how to proceed from the \text{E-anchor} to the \text{S-anchor}. This last phase is the focus of this paper. In particular, the patterns we identify here are special forms of \text{E-anchors} produced by the refinement strategy in [5].

CSP and \text{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 \text{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 \text{Circus} variant.

3. SCJ-Circus

As said before, \text{SCJ-Circus} extends \text{OhCircus} and \text{Circus Time} with abstractions specific to SCJ. In what follows, Section 3.1 discusses the syntax of \text{SCJ-Circus}, Section 3.2 presents the \text{Circus} model of the SCJ framework (its API and programming model), and Section 3.3 describes the semantics of \text{SCJ-Circus} based on the model of Section 3.2.

3.1. Syntax

Here we provide an overview of the syntax of \text{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.

\text{SCJ-Circus} extends the syntax of \text{OhCircus} and \text{Circus Time} with paragraphs that allow the specification of safelets, mission sequencers, missions, and handlers. An \text{SCJ-Circus} program is a sequence of \text{SCJCParagraphs}, which can be a \text{Circus} paragraph (in the syntactic class \text{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 SCJParagraph 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.
\[
\begin{align*}
\text{PTB, ATB, ID, OD, PD, AD, P} & : \mathbb{Z} \\
\text{PTB} + \text{ID} & \leq \text{PD} \land \text{PD} + \text{AD} \leq P
\end{align*}
\]

safelet \( \text{ChkSafelet} \stackrel{\equiv}{=} \begin{aligned}
\text{getSequencer} & \equiv \text{return } \bullet \text{return} \leftarrow \text{new ChkMissionSequencer}() \\
\end{aligned} \]
end

sequencer \( \text{ChkMissionSequencer} \stackrel{\equiv}{=} \begin{aligned}
\text{state} [\text{done} : \mathbb{B}] \\
\text{initial} & \equiv \text{done} := \text{false} \\
\text{getNextMission} & \equiv \text{return} \bullet \left(\begin{array}{l}
\text{if } \text{done} \rightarrow \text{done} := \text{true }; \text{return} \leftarrow \text{new ChkMission}() \\
\text{fi}
\end{array}\right)
\end{aligned} \]
end

mission \( \text{ChkMission} \stackrel{\equiv}{=} \begin{aligned}
\text{state} [\text{buffer} : \text{seq N}] \\
\text{initial} & \equiv \text{buffer} := (0, 0) \\
\text{initialize} & \equiv \text{var ah, ph : IDENTIFIER ; ah := newHandler Checker ; ph := newHandler Reader(ah)} \\
\text{cleanup} & \equiv \text{return} \bullet \left(\begin{array}{l}
\text{null}
\end{array}\right)
\end{aligned} \]
end

periodic handler \( \text{Reader} \stackrel{\equiv}{=} \begin{aligned}
\text{start} 0 \text{period PD} \\
\text{state} [\text{ah} : \text{IDENTIFIER}] \\
\text{initial} & \equiv \text{ah} : \text{IDENTIFIER ; ah.ah} := \text{ah} \\
\text{handleAsyncEvent} & \equiv \left(\begin{array}{l}
\text{getBuffer?buffer} \rightarrow \text{setBuffer!}(\text{tail buffer} \ominus \langle x \rangle) \rightarrow \text{release(ah)}; \\
\text{wait} 0 \ldots \text{PTB} \rightarrow \text{PD}
\end{array}\right)
\end{aligned} \]
end

aperiodic handler \( \text{Checker} \stackrel{\equiv}{=} \begin{aligned}
\text{checker} : \text{seq N} \rightarrow \mathbb{B} \\
\forall b : \text{seq N} | \# b = 2 \bullet \left(\begin{array}{l}
\text{if } b 1 = b 2 \text{ then checker}(b) = \text{true else checker}(b) = \text{false}
\end{array}\right)
\end{aligned} \]
end

\[
\text{getBuffer?buffer} \rightarrow \left(\begin{array}{l}
\text{if check(buffer) = true } \rightarrow \text{(output!true } \rightarrow \text{Skip) } \ominus \text{OD} \\
\text{fi}
\end{array}\right) ; \text{wait} 0 \ldots \text{ATB} \rightarrow \text{AD}
\]
end

Figure 6: S-anchor for our running example.
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 \( \llbracket \cdot \rrbracket_{\text{SCJProgram}} \). 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 \( \llbracket \cdot \rrbracket_{\text{SCJParagraphs}} \) and \( \llbracket \cdot \rrbracket_{\text{Application}} \). The former takes the sequence of SCJ-Circus paragraphs of \( p \) (that is, \( p{\text{paragraphs}} \)) 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 \( \llbracket \cdot \rrbracket_{\text{SCJParagraphs}} \) to specify the overall meaning of the SCJ-Circus program \( p \). These functions use the definitions of the processes that model the SCJ framework, like PEHFW and SafeletFW.

\[
\llbracket \cdot \rrbracket_{\text{SCJCProgram}} : \text{SCJCProgram} \rightarrow \text{CircusProgram}
\]

\[
\forall p : \text{WF\_SCJCProgram} \bullet \llbracket p \rrbracket_{\text{SCJCProgram}} = \llbracket p{\text{paragraphs}} \rrbracket_{\text{SCJParagraphs}} \downarrow \llbracket p \rrbracket_{\text{Application}}
\]

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 \( \llbracket \cdot \rrbracket_{\text{SCJSafelet}} \) for the safelet, which is used by \( \llbracket \cdot \rrbracket_{\text{SCJParagraphs}} \) 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 \( \llbracket \cdot \rrbracket_{\text{SCJCSafelet}} \) 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 (\( \llbracket \cdot \rrbracket \)). 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 \( \llbracket \cdot \rrbracket_{\text{SCJCSafelet}} \) 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 \( \text{App} \); 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 \( \llbracket \cdot \rrbracket_{\text{SCJSafelet}} \) 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 \((s,\text{state})\) as \( s \). Each action \( N \in \text{A} \) 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.
For instance, \texttt{getSequencerMeth} is started with a communication \texttt{getSequencerCall?(s)!ChkSafeletID}. We recall that the \texttt{Call} and \texttt{Ret} channels always take as parameters the identifiers of the caller, \(x\) in this example, and of the callee, \texttt{ChkSafeletID} here. This ensures that results are returned to the caller via the \texttt{Ret} channel, and correct synchronisation with the framework process for the component: \texttt{SafeletFW(ChkSafeletID)}, in this case.

The body of the method is almost unchanged, except for two points. First, results are returned via the \texttt{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, \texttt{new ChkMissionSequencer()}, is modelled as a communication via the channel \texttt{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 (\texttt{getSequencerMeth} and \texttt{initializeApplicationMeth}) in Figure 10 are combined in the action \texttt{Methods} that recursively (\(\mu X \bullet F(X)\)) offers them in external choice (\(\sqcup\)) as well as the possibility to terminate the recursion via a synchronisation on the channel \texttt{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 \texttt{Methods}.

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

The functions \texttt{sequencer_app}, \texttt{mission_app}, \texttt{PEH_app} and \texttt{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
process ChkSafelet_App ≡ begin
    getSequencerMeth ≡ getSequencerCall?x!ChkSafeletID →
    var return : IDENTIFIER •
        startMissionSequencer!ChkSafeletID!ChkMissionSequencerID →
    return := ChkMissionSequencerID;
    getSequencerRet!x!ChkSafeletID!return → Skip
initializeApplicationMeth ≡ safeletInitializeCall?x!ChkSafeletID → Skip;
    safeletInitializeRet!x!ChkSafeletID → Skip
Methods ≡ µ X • getSequencerMeth; X □ initializeApplicationMeth; X □ end_safelet_app → Skip
end

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

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

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

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

MArea ≡ var buffer : seq ℕ • µ X •
        setBuffer?o!ChkMissionID?x → buffer := x; X
        □
        getBuffer?o!ChkMissionID!buffer → X
        □
        end_mission_app → Skip

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.
process \( P \equiv \) begin
\[ \]
\[ P_{\text{Handler}} \equiv \mu X \cdot (F; \parallel \text{PERIODICDEADLINE} \parallel \text{wait PERIOD}) \; X \sqcap t \rightarrow \text{Skip} \]
\[ A_{\text{Handler}} \equiv \mu X \cdot ((C_j \rightarrow G_j) \parallel \text{APERIODICDEADLINE} \parallel \text{wait PERIOD}) \; X \sqcap t \rightarrow \text{Skip} \]
\[ \text{MArea} \equiv \forall x : T \cdot \mu Y \cdot \text{set}_i \text{rm} \; x := nx ; \; Y \sqcap \text{get}_i \text{rm} \; Y \sqcap t \rightarrow \text{Skip} \]
\[ \text{Termination} \equiv rt \rightarrow \mu X \cdot rt \rightarrow X \sqcap t \rightarrow \text{Skip} \]
\[ \text{Mission} \equiv ((\text{MArea} \parallel ((i : J \parallel P_{\text{Handler}}) \parallel (j : J \parallel A_{\text{Handler}})) \parallel \{aS \mid \{t, rt\} \mid \}) \text{Termination}) \setminus \ldots \]
\[ \text{MissionSequencer} \equiv \text{Mission} \]
\[ \text{Safelet} \equiv \text{MissionSequencer} \]
\[ \bullet \; \text{Safelet} \]
\[ \end \]
\[ \text{where} \; \text{PERIODICDEADLINE} \leq \text{PERIOD} \wedge \text{APERIODICDEADLINE} \leq \text{PERIOD} \]

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 \( P_{\text{Handler}} \) and \( A_{\text{Handler}} \) that model the periodic and aperiodic handlers, the action \( \text{MArea} \) that models the mission memory, and the termination management action Termination. Like in the semantics of SCJ-Circus, the mission-memory action \( \text{MArea} \) 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 \( aS \), can be modified by the parallelism of the handlers with the memory area, (action \( \text{MArea} \parallel (i : I \parallel P_{\text{Handler}}) \parallel (j : J \parallel A_{\text{Handler}}) \)), while Termination modifies no variables, and so is associated with the empty set \( \{\} \) of variables in the use of the parallel operator \( \ldots [aS \mid \{t, rt\} \mid \{\} \ldots] \).

Periodic event handlers \( P_{\text{Handler}} \) are defined by recursive actions whose iterations take some fixed amount of time (\( \text{wait PERIOD} \)) whilst executing (in interleaving) an action \( F \) 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
Our second pattern, in Figure 13, allows for an aperiodic handler not to be called in a cycle. Cyclic not in lock-step.

omitted in Figure 12, are enforced by a global constraint on these constants.

ID further restrictions involving expressions are defined by constants.

odic and the aperiodic handlers releases can terminate within the cycle of the mission. For our example, these time

It di

ers from that in Figure 11 just in the models of the aperiodic handlers, described in Figure 13.

process Chk ≜ begin

Reader ≜ μX • ( (input?x → Skip) ▶ ID; getBuffer?buffer →

setBuffer!(tail buffer) → wait PER) ▶ PD ) ; X

end → Skip

Checker ≜ μX • ( (rcheck → (if b → (output!true → Skip) ▶ OD

wait PER)

end → Skip

MArea ≜ var buffer : seqN • μX • setBuffer?x → buffer := x; X □ getBuffer!buffer → X □ end → Skip

Termination ≜ requestTermination → μX • requestTermination → X □ end → Skip

ChkMission ≜ ((MArea || Reader || Checker) \ (b = true) \ (end, requestTermination) \ \{ \} \ Termination) \ \{ \} ..

ChkMissionSequencer ≜ ChkMission

ChkSafelet ≜ ChkMissionSequencer

end

where PD ≤ PER ∧ AD ≤ PER

Figure 12: E anchor for the communication medium example.

AHandler_i ≜

μX • (((c_i → G_j) ▶ APERIODICDEADLINE □ wait PERIOD) || wait PERIOD); X □ t → Skip

where APERIODICDEADLINE ≤ 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_i is triggered by a synchronous event c_i; in our example, it is rcheck. This may occur at any time during the cycle, but the synchronisation on c_i and the following action G_j 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 PERIODICDEADLINE ≤ PERIOD ∧ APERIODICDEADLINE ≤ 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 ≤ PER ∧ AD ≤ 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;

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, in the starting model is defined in a corresponding SCJ-Circus periodic handler paragraph, whose handleAsyncEvent paragraph is determined by the body of PHandler. 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.
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]

$$F(A) \equiv (F(cs \rightarrow ce \rightarrow \text{Skip}) \mid \text{wrtV}(A) \mid \{cs, ce\} \mid \text{wrtV}(A)) \mid cs \rightarrow A; ce \rightarrow \text{Skip}) \setminus \{cs, ce\}$$

provided

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

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 `startsequencer` and `donesequencer`;


**Target**

```
process CF_P ≜ begin
  PHandler_i ≜ μX • (F_i ⊗ PD || wait PERIOD); X □ t → Skip
  AHandler_j ≜ μX • ((c_j → G_j) ⊗ AD || wait PERIOD); X □ t → Skip
  MArea ≜ ...
  Termination ≜ rt → μX • (rt → X □ t → Skip)

CF_Mission ≜ startMission →
  MArea || Termination ||
  (i:I • SH_i → register.i → start_peh.i → activate_handlers → done_handler.i → Skip)
  (j:J • SH_j → register.j → start_aeh.j → activate_handlers → done_handler.j → Skip)
  MArea → Skip
  ChkMissionSequencer;

Safelet ≜
  (i:I • SH_i → start_peh.i → activate_handlers → PHandler_i; done_handler.i → Skip)
  (j:J • SH_j → start_aeh.j → activate_handlers →
    (AHandler_j [(...)] | {c_j} | {j Buffer_i}] \ {c_j}; done_handler.j → Skip)
  CF_Mission
  MSequence ≜[...]
  start_sequencer → startMission → doneMission → donesequencer → Skip
  start_sequencer → doneSequencer → Skip
  Safelet \{startSequence doneSequence,...\}
end
```

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.

```
ChkSafelet ≜
  start_sequencer → donesequencer → Skip
  [{} | {start_sequencer, donesequencer} | {...}] \{start_sequencer, donesequencer\}
  start_sequencer → ChkMissionSequencer;
  doneSequencer → Skip
```

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 `donesequencer`, 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 \( \text{ChkMission} \) 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.

\[
\text{ChkMissionSequencer} \equiv \begin{cases} 
\text{start}_{-}\text{mission}.MID \rightarrow \text{done}_{-}\text{mission}.MID \rightarrow \text{Skip} \\
\{[] | \{\text{start}_{-}\text{mission}.MID, \text{done}_{-}\text{mission}.MID\} | \{\ldots\}\} \\
\text{start}_{-}\text{mission}.MID \rightarrow \text{ChkMission}; \\
\text{done}_{-}\text{mission}.MID \rightarrow \text{Skip} 
\end{cases} \]

\{\text{start}_{-}\text{mission}.MID, \text{done}_{-}\text{mission}.MID\}

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 \( \text{start}_{-}\text{mission} \) and \( \text{done}_{-}\text{mission} \). For \( \text{ChkMission} \) in the example for Step 1(a), the result is as follows.

\[
\text{ChkMissionSequencer} \equiv \begin{cases} 
\text{start}_{-}\text{mission}.MID \rightarrow \text{done}_{-}\text{mission}.MID \rightarrow \text{Skip} \\
\{[] | \{\text{start}_{-}\text{mission}.MID, \text{done}_{-}\text{mission}.MID\} | \{\ldots\}\} \\
\text{ChkMission} \\
\end{cases} \]

\{\text{start}_{-}\text{mission}.MID, \text{done}_{-}\text{mission}.MID\}

\[\text{ChkMission} \equiv \begin{cases} 
(\text{start}_{-}\text{mission}.MID \rightarrow (\text{MArea} \parallel \text{Reader} \parallel \text{Checker} \parallel \ldots \parallel \text{Termination}) \{\ldots\}) \\
\text{done}_{-}\text{mission}.MID \rightarrow \text{Skip} 
\end{cases} \]

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 \( \text{ChkMission} \) in Figure 17. The original actions \( \text{MArea} \) and \( \text{Termination} \) as well as the handler actions \( \text{Reader} \) and \( \text{Checker} \) are all in parallel now, controlled by the \( \text{start}_{-}\text{mission} \) and \( \text{done}_{-}\text{mission} \) signals. The extra parallel action in \( \text{ChkMission} \) (identified in the \( \text{ChkMission} \) by a box with superscript \( \text{EP} \)) defines the creation, via \( \text{StartReader.RID} \) and \( \text{StartChecker.CID} \) in the example, of the handlers, their registration, via a channel \( \text{register} \), their activation, via \( \text{start}_{-}\text{peh} \) or \( \text{start}_{-}\text{peh} \), and their deactivation, via \( \text{done}_{-}\text{handler} \). The new definitions for \( \text{Reader} \) and \( \text{Checker} \) include the control channels and termination via \( \text{done}_{-}\text{handler} \). \( \text{Reader} \) is now as follows.

\[
\text{Reader} \equiv \text{StartReader.RID} \rightarrow \text{start}_{-}\text{peh}.\text{RID} \rightarrow \text{activate}_{-}\text{handlers} \rightarrow \\
\mu X \bullet \left( \begin{array}{l}
(\text{input}_x \rightarrow \text{Skip}) \triangleright \text{ID}; \text{getBuffer}?!\text{buffer} \\
(\text{setBuffer}!((\text{tail buffer}) \land (\langle x \rangle) \rightarrow \text{check} \rightarrow (\text{wait} 0 \ldots \text{PTB}) ) \triangleright \text{PD} ) ; X
\end{array} \right)
\]

\[
\text{done}_{-}\text{handler}.\text{RID} \rightarrow \text{Skip}
\]
This procedure takes a Circus action A as its parameter.

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

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

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

\[
\text{ChkMission} = \begin{cases} 
\text{start\_mission.MID} \rightarrow \text{MArea}; \text{done\_mission.MID} \rightarrow \text{Skip} \\
\text{start\_mission.MID} \rightarrow \\
\text{StartReader.RID} \rightarrow \text{register.RID} \rightarrow \\
\text{start\_peh.RID} \rightarrow \text{done\_handler.RID} \rightarrow \text{Skip} \\
\text{StartChecker.CID} \rightarrow \text{register.CID} \rightarrow \\
\text{start\_ach.CID} \rightarrow \text{done\_handler.CID} \rightarrow \text{Skip} \\
\text{done\_mission.MID} \rightarrow \text{Skip} \\
\text{(Reader} \parallel \text{Checker}) \\
\text{start\_mission.MID} \rightarrow \text{Termination}; \text{done\_mission.MID} \rightarrow \text{Skip} 
\end{cases}
\]

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

Comparing this to the original definition of Reader shown in Figure[2] we observe that we have simply added control communications and replaced \( i \) with \( \text{done\_handler.RID} \). 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 \( AH_{\text{Handler}} \); for each of them, the parallelism of handlers is reorganised (via simple associativity and commutativity laws), so that one of the parallel actions is \( AH_{\text{Handler}} \) and the other is a component combining the rest of the handlers, including those that trigger \( AH_{\text{Handler}} \). At this point the Law sync-async-conv is used to introduce the asynchronous communication between \( AH_{\text{Handler}} \) 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, \( \mu X \bullet ((c \rightarrow G) \parallel D_1) \parallel \text{wait} \, P) \). \( X \square \text{end} \rightarrow \text{Skip} \) matches the specification of an aperiodic handler and \( OH_{\text{Handlers}} \equiv F(\mu X \bullet (B(c \rightarrow \text{Skip})_1 \parallel D_2) \parallel \text{wait} \, P) \). \( X \square \text{end} \rightarrow \text{Skip} \), 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.$MID \rightarrow H_i$; done_mission.$MID \rightarrow \text{Skip}$ (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 peça.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 peça.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 $SH.HID \rightarrow sh.HID \rightarrow ah \rightarrow \mu X . F(X, dh.HID \rightarrow \text{Skip})$ 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 ($D_i - T < P$); and (c) there is no further request to communicate on $c_i$ until the next period due to the proviso $OHandlers [\{ c \}] TimeReq = OHandlers$.

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 $D_i - T$, where $D_i$ is its overall deadline as originally specified.

$TimeReq$ specifies time requirements: (a) $c$ must take place before $T$ time units ($(c\forall t : t < T)$), and after that, it cannot happen until the next period (wait $(P - t)$); or (b) $c$ does not happen within the period (wait $P$). With the proviso $OHandlers [\{ c \}] TimeReq = OHandlers$, 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]

\[ \mu X \bullet ((c \rightarrow G) ▶ D_1 || wait P); X \square end \rightarrow Skip [ns_1 | cs | ns_2] \]

\( O\text{Handlers} \)

\[ (\mu X \bullet (c_i \rightarrow (G ▶ (D_1 - T)))); X \square end \rightarrow Skip) [ns_1 | \{c_i, end\} | \{\}\] Buffer \) \( \setminus \{c_i\} \)

\( O\text{Handlers} \)

where

\[ Buffer \equiv \text{var } pending : B \bullet pending := false; \mu X \bullet \]

\( c \rightarrow pending := true; X \square (pending) \otimes c_i \rightarrow pending := false; X \square end \rightarrow Skip \)

\( O\text{Handlers} = F(\mu X \bullet (B(c \rightarrow Skip) \bullet D_2 || wait P)); X \square end \rightarrow Skip \)

\( TimeReq = \mu X \bullet (c \otimes t : (t < T) \rightarrow wait (P - t) \square wait P); X \square end \rightarrow Skip \)

provided

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

Figure 19: Law sync-async-conv
Law \( \text{par-prefix-dist} \) is similar to the standard step law, Law \( \text{prefix-par-dist} \) in \( \text{Appendix A} \) which extracts synchronisations from inside parallel actions. Law \( \text{par-prefix-dist} \), however, distributes a synchronisation back into the parallel actions. Essentially, Step 2 expands the parallelism between handlers, which is internal to the action \( \text{Mission} \) as shown in Figure 17 by a box with superscript \( \text{EP} \), to a top level parallelism as shown in the action \( \text{CF\_Mission} \) 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 \( \text{handler-extract} \) 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 \( \text{start\_mission}, \text{activate\_handlers}, \text{done\_handler}, \) and \( \text{done\_mission} \) 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]**

\[
\begin{align*}
\text{sm} &\rightarrow (\mu X \bullet F; X); \text{dm} \rightarrow \text{Skip} \\
&\subseteq \left\{ \begin{array}{l}
(sm \rightarrow (\mu X \bullet F; X)), dm \rightarrow \text{Skip} \setminus \{r\} \\
(SH.n \rightarrow r.n \rightarrow sH.n \rightarrow ah \rightarrow dh.n \rightarrow dm \rightarrow \text{Skip}) \setminus \{r\} \\
\{\text{used}(F)\}, sh.n \rightarrow sh.n \rightarrow ah \rightarrow \mu X \bullet (F; X); dh.n \rightarrow \text{Skip} \\
\{sh, SH, dh\}
\end{array} \right.
\]

provided \( \{sh, SH, dh\} \cap \text{used}(F) = \emptyset \).

All the new events are hidden either locally (\( r \)) or in the parallelism (\( sh \) and \( SH \)). The variables modified by the handler release, specified by \( F \), 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 \( \textbf{[14]} \). The repeated application of Law \( \text{handler-extract} \) 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 \( \text{start\_mission} \) and \( \text{done\_mission} \) 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 \( \text{CF\_Mission} \) 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, \( \text{Handler\_app}, \text{Mission\_app} \), and so on. The original process \( \text{CF\_P} \) is refined to \( \text{AP\_P} \), 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 \( \text{Call} \) and \( \text{Ret} \) channels.

The target in Figure 20 shows the structure of the mission actions. In the AP framework action \( \text{AP\_Mission} \), after the start of the mission (\( \text{start\_mission}\_\text{MID} \)), the \text{initialize} method is called (\( \text{mission\_Initialize\_Call}\_x\_\text{MID} \)). This triggers, in the application action \( \text{Mission\_app} \), the creation (\( \text{startH}\_i \)) and registration (\( \text{register}\_i \)) of each of the handlers \( i \). \( \text{AP\_Mission} \) itself enters a loop where the registration of handlers is accepted and recorded in a local variable \( \text{handlers} \), until the call to \text{initialize} returns (\( \text{mission\_Initialize\_Ret}\_x\_\text{MID} \)).

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

\[
\text{process} AP_P \equiv \begin{align*}
\text{Handler}_{\text{app}} & \equiv \ldots \\
\text{AP}\_\text{Handler}_i & \equiv \ldots \\
\text{Mission}\_\text{app} & \equiv \\
\text{AP}\_\text{Mission} & \equiv \\
\text{MissionSequencer}\_\text{app} & \equiv \ldots \\
\text{AP}\_\text{MissionSequencer} & \equiv \ldots \\
\text{Safelet}\_\text{app} & \equiv \ldots \\
\text{AP}\_\text{Safelet} & \equiv \ldots \\
& \equiv (\text{Safelet}\_\text{app} \parallel \text{AP}\_\text{Safelet}) \setminus \{\ldots\} \\
& \parallel (\text{MissionSequencer}\_\text{app} \parallel \text{AP}\_\text{MissionSequencer}) \setminus \{\ldots\} \\
& \parallel (\text{Mission}\_\text{app} \parallel \text{AP}\_\text{Mission}) \setminus \{\ldots\} \\
& \parallel (\{\{i \in I \cup J \parallel (\text{Handler}_{\text{app}} \parallel \text{AP}\_\text{Handler}_i) \setminus \{\ldots\})\} \\
\end{align*}
\]

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 \texttt{p-handler-split}, \texttt{a-handler-split}, \texttt{mission-split}, \texttt{sequencer-split} and \texttt{safelet-split}.

\textbf{Step 1.} The first step of this phase applies the Laws \texttt{p-handler-split} and \texttt{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 \texttt{handleAsyncEventCall} and \texttt{handleAsyncEventRet} channels to model a call to the \texttt{handleAsyncEvent} method.

Law \texttt{p-handler-split} applies to an action $SH \rightarrow sh.id.s.p \rightarrow ah \rightarrow wait s; \mu X\bullet ((A \triangleright D \parallel wait p); X \square dh \rightarrow \text{Skip})$ modelling a periodic handler. In our example, the handlers start immediately after the start of the mission, so the wait action \texttt{wait} is expected after the start of the mission is just \texttt{wait} 0 and is omitted. With an application of the Law \texttt{p-handler-split}, the behaviour $A \triangleright D$ of the handler release is wrapped with new internal events \texttt{haeC} and \texttt{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.

\textbf{Law [p-handler-split]} \hspace{1cm}

$SH \rightarrow sh.id.s.p \rightarrow ah \rightarrow wait s; \mu X\bullet ((A \triangleright D \parallel wait p); X \square dh \rightarrow \text{Skip})$ \hspace{1cm} \hspace{1cm} $\sqsubseteq$

$SH \rightarrow sh.id.s.p \rightarrow \mu X\bullet ((\texttt{haeC} \rightarrow A \triangleright D; \texttt{haeR} \rightarrow X) \square dh \rightarrow \text{Skip})$

[\texttt{wrtV(A) | {sh, dh, haeC, haeR} | [\delta]}]

$sh.id?start?period \rightarrow ah \rightarrow wait start; \mu X\bullet \left( \left( (\texttt{haeC} \rightarrow \texttt{haeR} \rightarrow \text{Skip} \right) \triangleright 0 \triangleright period \right); X \right)$

[\texttt{dh \rightarrow Skip}]

\texttt{provided} $\{sh, dh, haeC, haeR\} \cap usedC(A) = \emptyset \land D \leq p$.

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

\textbf{Steps 2 and 3.} These steps apply specialised procedures \texttt{mission-AP} and \texttt{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 \texttt{p-handler-split} and \texttt{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.

\textbf{Steps 4.} The final step applies the Law \texttt{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 \texttt{getSequencerCall}, \texttt{getSequencerRet}, \texttt{safeletInitializeCall} and \texttt{safeletInitializeRet}.

At the end of the AP phase, the actions for the application processes are completed, but the actions \texttt{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 \texttt{mission-AP} and \texttt{sequencer-AP} used in this phase.

\textbf{Procedure \texttt{mission-AP}.} This is shown in Figure[21] it consists of three steps, for each action $M_i$ modelling a mission.

\textbf{Step 1.} The first step applies a Law \texttt{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 \texttt{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 \texttt{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 $\text{mission-split}$, 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 $\text{missionInitializeCall}$ and $\text{missionInitializeRet}$.

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 $\text{seq-interleave}$ to the action hiding the $SH_i$ channels;

4. Apply Law $\text{rec-interleave}$ to the action hiding the $\text{missionInitializeCall}$, $\text{missionInitializeRet}$, and $\text{register}$ channels.

**Figure 21:** Refinement strategy: procedure $\text{mission-AP}$

\[\text{Law [mission-split]}\]

\[
F \left\{ \begin{array}{c}
\text{sm} \rightarrow \left( \begin{array}{c}
SH_1 \rightarrow r.id_1 \rightarrow sh.id_1 \rightarrow ah \rightarrow dh.id_1 \rightarrow \text{Skip} \\
\{ ah \}
\end{array} \right) \setminus \{ r \}; \text{dm} \rightarrow \text{Skip} \\
\{ SH_1, SH_2 \}
\end{array} \right. \\
\leq \\
\left\{ \begin{array}{c}
\text{sm} \rightarrow \text{miC} \rightarrow \left( \begin{array}{c}
SH_1 \rightarrow r.id_1 \rightarrow \text{Skip} \\
\mid \mid \\
SH_2 \rightarrow r.id_2 \rightarrow \text{Skip}
\end{array} \right); \text{miR} \rightarrow \text{dm} \rightarrow \text{Skip} \\
\{ \{ sm, r, dm, miC, miR \} \cup \{ \} \}
\end{array} \right. \\
\left\{ \begin{array}{c}
\text{sm} \rightarrow \text{miC} \rightarrow \left( \begin{array}{c}
r.id_1 \rightarrow \text{Skip} \\
\mid \mid \\
r.id_2 \rightarrow \text{Skip}
\end{array} \right); \text{miR} \rightarrow \text{Skip}; \\
\{ miC, miR, r \} \setminus \{ SH_1, SH_2 \}
\end{array} \right. \\
\left\{ \begin{array}{c}
sh.id_1 \rightarrow ah \rightarrow dh.id_1 \rightarrow \text{Skip} \\
\{ ah \}
\end{array} \right. \\
\left\{ \begin{array}{c}
sh.id_2 \rightarrow ah \rightarrow dh.id_2 \rightarrow \text{Skip} \\
\{ ah \}
\end{array} \right. \\
\text{dm} \rightarrow \text{Skip}
\end{array} \right. \\
\right. \\
\text{provided } \{ miC, miR \} \cap \text{usedC(LHS)} = \emptyset
\]

**Figure 22:** Law mission-split

register, $sh$ with $\text{start}_\text{adh}$ or $\text{start}_\text{peh}$, depending on whether we have an aperiodic or a periodic handler, $ah$ with $\text{activate}_\text{handlers}$, $dh$ with $\text{done}_\text{handler}$, and $dm$ with $\text{done}_\text{mission}$.

In all cases, the Law $\text{mission-split}$ 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 $\text{mission-split}$ 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 $\text{seq-interleave}$, 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
nates (if and when the mission terminates). In this case, shown in Figure 24, the procedure applies the Law

\textbf{Law [rec-interleave]}

\[
\begin{align*}
\text{sm} & \rightarrow \text{miC} \rightarrow \text{SH}_1 \rightarrow \text{SH}_2 \rightarrow r.id_1 \rightarrow r.id_2 \rightarrow \text{miR} \rightarrow \text{dm} \rightarrow \text{Skip} \\
\{ | \{ \text{sm}, r, \text{dm}, \text{miC}, \text{miR} \} | | \}
\end{align*}
\]

\[
\begin{align*}
\text{sm} & \rightarrow \text{miC} \rightarrow \text{||} \rightarrow \text{Skip} \\
\text{sm} & \rightarrow \text{miC} \rightarrow \text{r.id}_1 \rightarrow \text{Skip} \\
\text{sm} & \rightarrow \text{miC} \rightarrow \text{r.id}_2 \rightarrow \text{Skip} \\
\text{sm} & \rightarrow \text{miC} \rightarrow \text{sh.id}_1 \rightarrow \text{ah} \rightarrow \text{dh.id}_1 \rightarrow \text{Skip} \\
\text{sm} & \rightarrow \text{miC} \rightarrow \text{sh.id}_2 \rightarrow \text{ah} \rightarrow \text{dh.id}_2 \rightarrow \text{Skip} \\
\text{sm} & \rightarrow \text{miC} \rightarrow \mu X \cdot \left( \begin{array}{c} 
\text{r?h} \rightarrow \text{handlers} := \text{handlers} \cup \{ \text{h} \}; X \\
\text{miR} \rightarrow \text{Skip} \\
\text{|| ah} \rightarrow \text{dh.h} \rightarrow \text{Skip}; \text{dm} \rightarrow \text{Skip} 
\end{array} \right) \\
\{ | \{ \text{sm}, r, \text{dm}, \text{miC}, \text{miR} \} | | \}
\end{align*}
\]

\text{provided} \text{ handlers} \text{ 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.

\textit{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 \textit{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 \textit{h} from the set \textit{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.

\textbf{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 \( \text{start} \_\text{sequencer} \) and \( \text{done} \_\text{sequencer} \).

1. If \( A = \text{start} \_\text{mission} \_i \rightarrow \text{done} \_\text{mission} \_i \rightarrow \text{Skip} \) then
   (a) Apply Law \( \text{sequencer-split-single} \) to the action \( A \).
2. Else “The strategy does not cover this case.”

Figure 24: Refinement strategy: procedure \( \text{sequencer-AP} \)

**Steps**

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

**Target**

\[
\begin{align*}
\text{process } FW \_P \equiv & \text{ begin } \\
& (\text{Safelet}_\text{app} \parallel \text{SafeletFW}) \setminus \{\ldots\} \\
& (\text{MissionSequencer}_\text{app} \parallel \text{MissionSequencerFW}) \setminus \{\ldots\} \\
& (\text{Mission}_\text{app} \parallel \text{MissionFW}) \setminus \{\ldots\} \\
& (\| i : I \bullet (\text{Handler}_i \_\text{app} \parallel \text{APEHFW}) \setminus \{\ldots\}) \\
& (\| j : J \bullet (\text{Handler}_j \_\text{app} \parallel \text{PEHFW}) \setminus \{\ldots\}) \\
\text{end}
\end{align*}
\]

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 \( \text{Safelet}_\text{app} \parallel AP \_\text{Safelet} \) 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 \( \text{ChkSafelet} \) in our example in Figure 12) does not perform any initialisation. For this reason, the method \( \text{safeletInitialize} \) provided by the SCJ-Circus safelet paragraph and included in the framework process \( \text{SafeletFW} \) (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 \( \text{safeletInitialize} \) whose body is just \( \text{Skip} \).

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 $mission\_start.i \rightarrow mission\_done.i \rightarrow \text{Skip}$, 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

\[
\begin{align*}
\text{handler } S\_\text{Handler}_i \equiv & \ldots \\
\text{mission } S\_\text{Mission} \equiv & \ldots \\
\text{sequencer } S\_\text{MissionSequencer} \equiv & \ldots \\
\text{safelet } S\_\text{Safelet} \equiv & \ldots
\end{align*}
\]

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

Starting pattern

\[Mission \equiv (\text{MArea} \parallel (\parallel i : I \cdot \text{PHandler}_i) \parallel (\parallel j : Jn \cdot \text{AHandler}_j))\]

Modified steps

CF Step-0 Add a new first step to phase CF that applies the law \(\text{termination-intro}\).

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 \(\text{Termination}\) in our pattern, is integrated into the mission framework action. In Step 3 of the FW phase, the Lemma \(\text{mission-fw-cl}\) renames the channel used in the original specification to \(\text{requestTerminationCall}\).

In this section, we adapt our strategy to consider applications that do not terminate. The starting pattern in Figure 28 shows the \(\text{Mission}\) action of the non-terminating cyclic in lockstep pattern. The main difference from that in Figure 11 is the missing \(\text{Termination}\) 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 \(\text{Mission}\) action to have an extra parallelism with the \(\text{Termination}\) action. More specifically, Lemma \(\text{mission-fw-cl}\), 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 \(\text{Termination}\) as a first step of the CF phase using the Law \(\text{termination-intro}\). In this way, any other extensions to the strategy can take advantage of the general pattern with termination.
Starting pattern

\[ \text{MissionSequencer} \equiv M_1; M_2; \ldots; M_n \]

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 \([\text{termination-intro}]\)

\[
\begin{align*}
\mu X \cdot F; X & \sqsubseteq \\
(\mu X \cdot F; X \parallel t \rightarrow \text{Skip}) & \sqsubseteq (\mu X \cdot rt \rightarrow X \parallel t \rightarrow \text{Skip}) \setminus \{rt, t\}\end{align*}
\]

provided \( \{t, rt\} \cap \text{usedC}(F) = \emptyset \).

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

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

It may seem inefficient to complicate the model, but we note that the refinement steps of the whole refinement strategy 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 \( \text{start}_i \text{mission} \) and \( \text{done}_i \text{mission} \);
   (b) Apply Law copy-rule to the call action \( M \) in \( A \);
   (c) Apply Law copy-rule from right to left to the action \( \text{start}_i \text{mission} \rightarrow \ldots \); \( \text{done}_i \text{mission} \rightarrow \textbf{Skip} \) 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.”

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

\[
\text{MissionSequencer}_1 \equiv \begin{cases} 
\text{start}_1 \text{mission}.M_1 \rightarrow \text{done}_1 \text{mission}.M_1 \rightarrow \textbf{Skip} \\
[[]] \{\text{start}_1 \text{mission}.M_1, \text{done}_1 \text{mission}.M_1 | [\ldots]\}\} \ \{\text{start}_1 \text{mission}.M_1, \text{done}_1 \text{mission}.M_1\} ; \\
\text{Mission}_1 \\
\text{start}_2 \text{mission}.M_2 \rightarrow \text{done}_2 \text{mission}.M_2 \rightarrow \textbf{Skip} \\
[[]] \{\text{start}_2 \text{mission}.M_2, \text{done}_2 \text{mission}.M_2 | [\ldots]\}\} \ \{\text{start}_2 \text{mission}.M_2, \text{done}_2 \text{mission}.M_2\} \\
\text{Mission}_2
\end{cases}
\]

Like in the core strategy, each mission is replaced by a parallel action that calls a specific mission using \( \text{start}_i \text{mission} \) and \( \text{done}_i \text{mission} \), 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 \( \text{MissionSequencer}_2 \).

\[
\text{MissionSequencer}_2 \equiv \begin{cases} 
\text{start}_1 \text{mission}.M_1 \rightarrow \text{done}_1 \text{mission}.M_1 \rightarrow \text{start}_1 \text{mission}.M_1 \rightarrow \text{done}_1 \text{mission}.M_2 \rightarrow \textbf{Skip} \\
[[]] \{\text{start}_1 \text{mission}.M_1, \text{done}_1 \text{mission}.M_1, \text{start}_1 \text{mission}.M_2, \text{done}_1 \text{mission}.M_2 | [\ldots]\}\} \\
\text{Mission}_1 || \text{Mission}_2 \\
\{\text{start}_1 \text{mission}.M_1, \text{done}_1 \text{mission}.M_1, \text{start}_1 \text{mission}.M_2, \text{done}_1 \text{mission}.M_2\}
\end{cases}
\]

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

\textit{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 \( \text{getNextMission} \) 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 \( \text{start}_i \text{mission} \) and \( \text{done}_i \text{mission} \) into a parallel action communicating over channels \( \text{getNextMissionCall} \) and \( \text{getNextMissionRet} \) 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 $A = start\_mission.i \rightarrow done\_mission.i \rightarrow \textbf{Skip}$ then
   (a) Apply Law sequencer-split-single to the action $A$.
2. If $A = (; \ i : I \cdot start\_mission.i \rightarrow done\_mission.i \rightarrow \textbf{Skip})$
   (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

... $start\_mission.M_1 \rightarrow done\_mission.M_1 \rightarrow start\_mission.M_1 \rightarrow done\_mission.M_1 \rightarrow ...$

... $\begin{align*}
\text{getNextMissionCall}(id!id &\rightarrow \text{getNextMissionRet}(id!id!M_1 \rightarrow start\_mission.M_1 \rightarrow done\_mission.M_1 \rightarrow) \\
\text{getNextMissionCall}(id!id &\rightarrow \text{getNextMissionRet}(id!id!M_1 \rightarrow start\_mission.M_1 \rightarrow done\_mission.M_2 \rightarrow) \\
\text{getNextMissionCall}(id!id &\rightarrow \text{getNextMissionRet}(id!id!null \rightarrow end\_sequencer\_app \rightarrow ... \begin{align*}
\mu X \bullet \text{getNextMissionCall}(id!id &\rightarrow \text{getNextMissionRet}(id!id!m \rightarrow) \\
(\text{if } m = \text{null } \rightarrow \text{end\_sequencer\_app } \rightarrow \text{Skip}) \\
(\text{if } m \neq \text{null } \rightarrow \text{start\_mission.m } \rightarrow \text{done\_mission.m } \rightarrow X) \\
\end{align*}
\text{var i : Z } i := 0; \mu X \bullet \\
\text{getNextMissionCall}(x!id \rightarrow ... \begin{align*}
\text{if } i = 0 &\rightarrow \text{getNextMissionRet}(x!id!M_1 \rightarrow i := i + 1) \\
\text{if } i = 1 &\rightarrow \text{getNextMissionRet}(x!id!M_2 \rightarrow i := i + 1) \\
\text{if } i > 1 &\rightarrow \text{getNextMissionRet}(x!id!null \rightarrow \text{Skip}) \\
\end{align*} \\
\end{align*}
\text{end\_sequencer\_app } \rightarrow \text{Skip}
\end{align*}$

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.

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 $mission\_start.i \rightarrow mission\_done.i \rightarrow \text{Skip}$, apply law mission-fw-cl-single.
2. If the action Sequencer of $P$ contains an action of the form $i : 1 \bullet mission\_start.i \rightarrow mission\_done.i \rightarrow \text{Skip}$, 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 $\text{release}$ 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 [12] [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 of Unifying Theories of Programming (UTP) and Circus in the theorem prover Isabelle. As future work, we plan to complete the implementation of our strategies, and apply them to more examples, including SCJ benchmarks. A more recent result on the semantics of SCJ Level 2 also paves the way to generalise SCJ-Circus and our strategy to cater for such programs.

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

Appendix A. Refinement laws

Law [a-handler-split*]

\[ SH \rightarrow sh \rightarrow ah \rightarrow \mu X \cdot (c \rightarrow (A \leftarrow D); X \square dh \rightarrow \text{Skip}) \]

\[ \square \]

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

Lemma [aperiodic-handler-fw-cl*]

\[ SH \rightarrow \text{start}_aeh?x!id \rightarrow \mu X \cdot \{ \text{handleAsyncEventCall}?x!id \rightarrow (A \leftarrow D; \text{handleAsyncEventRet}!x!id \rightarrow X) \} \]

\[ \square \]

\[ \{ \text{start}_aeh, \text{done}_handler, \text{handleAsyncEventCall, handleAsyncEventRet} \} \cap \{ | | \} \]

\[ \text{start}_aeh?x!id \rightarrow \text{activate}_\text{handlers} \rightarrow \text{wait} \text{ start} ; \]

\[ \mu X \cdot \{ (c \rightarrow \text{handleAsyncEventCall}?x!id \rightarrow \text{handleAsyncEventRet}!x!id \rightarrow \text{Skip}); X \square \text{done}_\text{handler} \rightarrow \text{Skip} \} \]

\[ \{ | | \} \}

\[ \text{Buffer} \]

\[ \square \]

\[ SH \rightarrow \text{start}_aeh?x!id \rightarrow \mu X \cdot \{ \text{handleAsyncEventCall}?x!id \rightarrow (A \leftarrow D; \text{handleAsyncEventRet}!x!id \rightarrow X) \} \]

\[ \square \]

\[ \{ \text{start}_aeh, \text{done}_\text{handler, handleAsyncEventCall, handleAsyncEventRet} \} \cap \{ | | \} \]

\[ \text{var} \text{currentRelease} : B; \text{newRelease} : B \bullet \text{start}_aeh?id \rightarrow \text{activate}_\text{handlers} \rightarrow \]

\[ (\text{currentRelease} = \text{false}) \& (\text{releaseCall}?0!id \rightarrow (\text{currentRelease} = \text{true}; \text{releaseRet}?0!id \rightarrow \text{Skip})); X \]

\[ \mu X \cdot \{ (\text{currentRelease} = \text{true}) \& (\text{releaseCall}?0!id \rightarrow (\text{newRelease} = \text{true}; \text{releaseRet}?0!id \rightarrow \text{Skip})); X \]

\[ (\text{currentRelease} = \text{true}) \& (\text{hasRelease} \rightarrow \text{currentRelease} = \text{newRelease}); X \square \text{done}_\text{handler} \rightarrow \text{Skip} \]

\[ \{ | | \} \}

\[ \mu X \cdot \text{hasRelease} \rightarrow \text{handleAsyncEventCall}!x!id \rightarrow \text{handleAsyncEventRet}!x!id \rightarrow X \square \text{done}_\text{handler} \rightarrow \text{Skip} \]

\[ \{ \text{HasRelease} \} \]

\[ \text{where} \]

\[ \text{Buffer} \equiv \text{var} \text{pending} : B \bullet \text{pending} := \text{false}; \mu X \cdot \{ \text{releaseCall}?0!id \rightarrow \text{pending} := \text{true}; \text{releaseRet}?0!id \rightarrow X \}

\[ (\text{pending}) \& c \rightarrow \text{pending} := \text{false}; X \square \text{done}_\text{handler} \rightarrow \text{Skip} \]
Law [call-intro*]

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

provided

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

Law [handler-extract*]

\[ sm \to ah \to (\mu X \cdot A; X \to dh \to \text{Skip}); \quad dm \to \text{Skip} \]
\[ \subseteq \left( \begin{array}{c}
(sm \to SH \to r \to sh \to ah \to dh \to dm \to \text{Skip}) \setminus \{ r \} \\
\{ [1] \mid \{ sh, ah, dh, SH \} \mid usedV(A) \} \\
SH \to sh \to ah \to \mu X \cdot (A; X \to dh \to \text{Skip})
\end{array} \right) \setminus \{ sh, SH \}
\]

provided \{ sh, SH, dh, ah \} \cap usedC(A) = \emptyset.
Lemma \([\text{mission-fw-cl}^*]\)

\[
\begin{align*}
\text{start\_mission!id}\!?: & \to \text{missionInitializeCall!x!id} \to S_1 \to S_2 \to \text{register!id}_1 \to \text{register!id}_2 \to \\
\text{missionInitializeRet!x!id} & \to \text{done!mission!id} \to \text{Skip} \\
\{1\} \mid \text{start\_mission, register, done\_mission, missionInitializeCall, missionInitializeRet} \mid \{1\}
\end{align*}
\]

\[
\begin{align*}
\text{var} \text{handlers} : \mathbb{P}\text{ID} \to \text{ID}; \text{start\_mission}!x{}\!\to{}\! \\
\text{start\_mission!id} & \to \mu X \cdot \\
\text{requestTerminationCall!x!id} & \to \text{requestTerminationRet!x!id} \\
\{1\} \mid \text{done\_mission, stop\_handlers} \mid \{\text{handlers}\}
\end{align*}
\]

\[
\begin{align*}
\text{missionInitializeCall!x!id} & \to S_1 \to S_2 \to \text{register!id}_1 \to \text{register!id}_2 \to \text{missionInitializeRet!x!id} \to X \\
\{1\} \mid \text{start\_mission, register, done\_mission, missionInitializeCall, missionInitializeRet, requestTerminationCall, requestTerminationRet, terminationPendingCall, terminationPendingRet} \mid \{1\}
\end{align*}
\]

\[
\begin{align*}
\text{var} \text{sequencer} : \text{ID}; \text{terminating} : \mathbb{B}; \text{sequencer} := \text{null}; \text{handlers} := \{\}; \text{terminating} := \text{false}; \\
\text{start\_mission!id} & \to \text{sequencer} := s; \text{missionInitializeCall!x!id} \\
\mu X \cdot \text{register!h} \to \mu X \cdot \text{handlers} := \text{handlers} \cup \{h\}; X \to \text{missionInitializeRet!x!id} \to \text{Skip} \\
\text{if} \text{handlers} = \{\} & \to \text{Skip} \\
\text{if} \text{handlers} \neq \{\} & \to \\
\{|||\text{h!}: \text{handlers} \neq \{\}\mid \text{start\_ah!h} \to \text{Skip} \mid \text{activate\_handlers} \mid \text{stop\_handlers} \mid \text{done\_handlers} \mid \{\text{done\_mission!id} \to \text{Stop}\} \mid \{\text{stop\_handlers, done\_handlers}\} \mid \{\text{requestTerminationCall!x!id} \to \text{Stop}\} \mid \{\text{requestTerminationRet!x!id} \to \text{Stop}\} \mid \{\text{terminationPendingCall!x!id} \to \text{Stop}\} \mid \{\text{terminationPendingRet!x!id} \to \text{Stop}\} \mid \{\text{stop\_handlers, done\_handlers}\} \mid \{\text{cleanUpCall!id} \to \text{Stop}\} \mid \{\text{end\_mission!app!id} \to \text{Stop}\} \mid \{\text{done\_mission!id} \to \text{Stop}\} \mid \{\text{stop\_handlers, cleanUpCall, end\_mission!app}\} \mid \{\text{cleanUpRet!id}!x \to \text{Stop}\} \mid \{\text{end\_mission!app!id} \to \text{Stop}\} \mid \{\text{done\_mission!id} \to \text{Stop}\} \mid \{\text{stop\_handlers, cleanUpCall, end\_mission!app}\}
\end{align*}
\]

\[\text{where} \quad \text{Channels } S_1 \text{ and } S_2 \text{ correspond to the constructor channels of the handlers with identifiers } \text{id}_1 \text{ and } \text{id}_2.\]
**Lemma** \([\text{mission-fw-cl-single}^*]\)

\[
\begin{align*}
\text{start}_{\text{sequencer}} & \rightarrow \begin{cases}
\mu X \cdot \text{getNextMissionCall!id} \rightarrow \text{getNextMissionRet!id'?m} \rightarrow \\
\text{if } m \neq \text{null} \rightarrow \text{end}_{\text{sequencer}_\text{app}} \rightarrow \text{Skip} \\
\text{fi}
\end{cases};
\end{align*}
\]

done_{\text{sequencer}} \rightarrow \text{Skip}

\[
\begin{align*}
\{!() | \{\text{getNextMissionCall, getNextMissionRet, end}_{\text{sequencer}_\text{app}}\} | ()\}
\end{align*}
\]

\[
\begin{align*}
\text{var } b : B \cdot b := \text{false};
\mu X \cdot \begin{cases}
\text{getNextMissionCall?x}!id \rightarrow \begin{cases}
\text{if } b \rightarrow \text{getNextMissionRet!x}!id'null \rightarrow \text{Skip} \\
\text{fi}
\end{cases};X
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{end}_{\text{sequencer}_\text{app}} \rightarrow \text{Skip}
\end{align*}
\]

\[
\begin{align*}
\{!() | \{\text{getNextMissionCall, getNextMissionRet, end}_{\text{sequencer}_\text{app}}\} | ()\}
\end{align*}
\]

\[
\begin{align*}
\text{start}_{\text{sequencer}} & \rightarrow \mu X \cdot \text{getNextMissionCall!id} \rightarrow \text{getNextMissionRet!id'?m} \rightarrow \\
\text{if } m \neq \text{null} \rightarrow \begin{cases}
\text{start}_m \rightarrow \text{done}_m \rightarrow \text{continue}_X \rightarrow \text{continue} \rightarrow \text{Skips} \text{fi}
\end{cases};
\end{align*}
\]

done_{\text{sequencer}_\text{app}} \rightarrow \text{done}_{\text{sequencer}} \rightarrow \text{Skip}

\[
\begin{align*}
\{!() | \{\text{getNextMissionCall, getNextMissionRet, end}_{\text{sequencer}_\text{app}}\} | ()\}
\end{align*}
\]

\[
\begin{align*}
\text{var } b : B \cdot b := \text{false};
\mu X \cdot \begin{cases}
\text{getNextMissionCall?x}!id \rightarrow \begin{cases}
\text{if } b \rightarrow \text{getNextMissionRet!x}!id'null \rightarrow \text{Skip} \\
\text{fi}
\end{cases};X
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{end}_{\text{sequencer}_\text{app}} \rightarrow \text{Skip}
\end{align*}
\]

\[
\begin{align*}
\{!() | \{\text{getNextMissionCall, getNextMissionRet, end}_{\text{sequencer}_\text{app}}\} | ()\}
\end{align*}
\]

\[
F \begin{align*}
\begin{cases}
\text{sm} \rightarrow \begin{cases}
\text{SH}_1 \rightarrow r.id_1 \rightarrow \text{sh.id}_1 \rightarrow ah \rightarrow dh.id_1 \rightarrow \text{Skip} \quad \{r\}; \quad dm \rightarrow \text{Skip} \\
\text{SH}_2 \rightarrow r.id_2 \rightarrow \text{sh.id}_2 \rightarrow ah \rightarrow dh.id_2 \rightarrow \text{Skip}
\end{cases} \\
\{SH_1, SH_2\}
\end{cases}
\end{align*}
\]

\[
F \begin{align*}
\begin{cases}
\{!() | \{\text{sm, r, dm, miC, miR}\} | ()\}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{sm} \rightarrow \text{miC} \rightarrow \begin{cases}
\text{SH}_1 \rightarrow r.id_1 \rightarrow \text{Skip} \\
\text{SH}_2 \rightarrow r.id_2 \rightarrow \text{Skip}
\end{cases}; \quad \text{miR} \rightarrow dm \rightarrow \text{Skip} \\
\{SH_1, SH_2\}
\end{align*}
\]

\[
\begin{align*}
\text{sm} \rightarrow \text{miC} \rightarrow \begin{cases}
\text{sh.id}_1 \rightarrow ah \rightarrow dh.id_1 \rightarrow \text{Skip} \\
\text{sh.id}_2 \rightarrow ah \rightarrow dh.id_2 \rightarrow \text{Skip}
\end{cases}; \quad dm \rightarrow \text{Skip}
\end{align*}
\]

\[
\{\text{miC, miR}\} \cap \text{usedC(LHS)} = \emptyset
\]
Law [p-handler-split*]

\[ SH \rightarrow sh.id.s.p \rightarrow ah \rightarrow wait \; s; \; \mu X \bullet ((A \triangleright D || wait \; p); \; X \Box dh \rightarrow Skip) \]

\[ SH \rightarrow sh.id.s.p \rightarrow \mu X \bullet ((haeC \rightarrow A \triangleright D; \; haeR \rightarrow X) \Box dh \rightarrow Skip) \]

\[ sh.id?start?period \rightarrow ah \rightarrow wait \; start; \; \mu X \bullet \left( (\mu C \rightarrow haeR \rightarrow Skip) \blacktriangleright 0 \blacktriangleright period ; \; X \right) \]

\[ \Box dh \rightarrow Skip \]

\[ \{ haeC, haeR \} \]

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

Law [par-prefix-dist*]

\[ (A \mid ns_1 \mid cs \mid ns_2 \mid B); \; c \rightarrow Skip \]

\[ (A; \; c \rightarrow Skip \mid ns_1 \mid cs \cup \{ c \} \mid ns_2 \mid B; \; c \rightarrow Skip) \]

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

Lemma [periodic-handler-fw-cl*]

\[ SH \rightarrow start_peh?x!id!@p \rightarrow \mu X \bullet \left( \begin{array}{l}
handleAsyncEventCall?x!id \rightarrow (A \triangleright D; \; handleAsyncEventRet!x!id \rightarrow X) \\
\Box done_handler.id \rightarrow Skip
\end{array} \right) \]

\[ [\Box Vw(A) \mid [\Box start_peh, done_handler, handleAsyncEventCall, handleAsyncEventRet]\mid \{1\}] \]

\[ start_peh?x!id?start?period \rightarrow activate_handlers \rightarrow wait \; start; \; \mu X \bullet \left( \begin{array}{l}
(handleAsyncEventCall?x!id \rightarrow handleAsyncEventRet!x!id \rightarrow Skip) \blacktriangleright 0 \blacktriangleright period ; \; X \end{array} \right) \]

\[ \Box done_handler.id \rightarrow Skip \]

\[ SH \rightarrow start_peh?x!id!@p \rightarrow \mu X \bullet \left( \begin{array}{l}
handleAsyncEventCall?x!id \rightarrow (A \triangleright D; \; handleAsyncEventRet!x!id \rightarrow X) \\
\Box done_handler.id \rightarrow Skip
\end{array} \right) \]

\[ [\Box Vw(A) \mid [\Box start_peh, done_handler, handleAsyncEventCall, handleAsyncEventRet]\mid \{1\}] \]

\[ var \; start, \; period : \mathbb{R} \bullet \left( \begin{array}{l}
start_peh?x!id?x!id?x!id \rightarrow activate_handlers \rightarrow start := s; \; period := p; \; wait \; start; \\
(handleAsyncEventCall?x!id \rightarrow handleAsyncEventRet?x!id \rightarrow Skip) \blacktriangleright period ; \; X
\end{array} \right) \]

\[ \begin{array}{l}
\Box done_handler.id \rightarrow Skip \\
\{1\} \mid \{ handleAsyncEventCall?x!id, done_handler.id \mid \{1\} \}
\end{array} \]

Law [prefix-introduction]

\[ A = (c \rightarrow A) \setminus \{ c \} \]

provided \( c \not\in usedC(A). \)
Law [prefix-par-dist]

\[ c \rightarrow (A_1 \mid ns_1 \mid cs \mid ns_2) A_2 = (c \rightarrow A_1 \mid ns_1 \mid cs \cup \{c\} \mid ns_2) c \rightarrow A_2 \]

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

Law [rec-interleave*]

\[
\begin{align*}
&\text{sm} \rightarrow \text{miC} \rightarrow SH_1 \rightarrow SH_2 \rightarrow r.id_1 \rightarrow r.id_2 \rightarrow \text{miR} \rightarrow \text{dm} \rightarrow \text{Skip} \\
&\quad \{[0] \mid \{\text{sm}, r, dm, \text{miC}, \text{miR}\} \mid [0]\} \\
&\quad \text{sm} \rightarrow \text{miC} \rightarrow (r.id_1 \rightarrow \text{Skip} \parallel r.id_2 \rightarrow \text{Skip}); \text{miR} \rightarrow \text{Skip}; \\
&\quad \left(\text{sh.id}_1 \rightarrow \text{ah} \rightarrow \text{dh.id}_1 \rightarrow \text{Skip} \right); \text{dm} \rightarrow \text{Skip} \\
&\quad \left(\text{sh.id}_2 \rightarrow \text{ah} \rightarrow \text{dh.id}_2 \rightarrow \text{Skip} \right) \quad \{\text{miC}, \text{miR}, r\} \\
&\quad \text{sm} \rightarrow \text{miC} \rightarrow SH_1 \rightarrow SH_2 \rightarrow r.id_1 \rightarrow r.id_2 \rightarrow \text{miR} \rightarrow \text{dm} \rightarrow \text{Skip} \\
&\quad \{[0] \mid \{\text{sm}, r, dm, \text{miC}, \text{miR}\} \mid [0]\} \\
&\quad \text{var} \text{handlers} : \text{ID} \bullet \text{handlers} := \{\}; \\
&\quad \text{sm} \rightarrow \text{miC} \rightarrow \mu X \bullet \left( r^!h \rightarrow \text{handlers} := \text{handlers} \cup \{h\}; \ X \right); \\
&\quad \left(\text{sh.id} \rightarrow \text{ah} \rightarrow \text{dh.id} \rightarrow \text{Skip} \right); \text{dm} \rightarrow \text{Skip} \quad \{\text{miC}, \text{miR}, r\} \\
\end{align*}
\]

provided \text{handlers} is fresh.

Lemma [safelet-fw-cl*]

\[
\begin{align*}
&\text{getSequencerCall?x!id} \rightarrow \text{getSequencerRet!x!id!sid} \rightarrow \text{end_safelet_app} \rightarrow \text{Skip} \\
&\quad \{[0] \mid \{\text{getSequencerCall}, \text{getSequencerRet}, \text{end_safelet_app}\} \mid [0]\} \\
&\quad \text{getSequencerCall?x!id} \rightarrow \text{getSequencerRet!x!id?s} \rightarrow \text{start_sequencer} \rightarrow \text{done_sequencer} \\
&\quad \text{end_safelet_app} \rightarrow \text{Skip} \\
&\quad \text{getSequencerCall?x!id} \rightarrow \text{getSequencerRet?x!id?s} \rightarrow \text{end_safelet_app} \rightarrow \text{Skip} \quad \{\text{getSequencerCall}, \text{getSequencerRet}, \text{end_safelet_app}\} \\
&\quad \text{getSequencerCall?x!id} \rightarrow \text{safeletInitializeCall?x!id} \rightarrow \text{safeletInitializeRet?x!id} \rightarrow \text{end_safelet_app} \rightarrow \text{Skip} \\
&\quad \text{getSequencerCall?x!id} \rightarrow \text{safeletInitializeCall?x!id} \rightarrow \text{safeletInitializeRet?x!id}\rightarrow \text{end_safelet_app} \rightarrow \text{Skip} \\
&\quad \text{if } s \neq \text{null} \rightarrow \text{start_sequencer} \rightarrow \text{done_sequencer} \rightarrow \text{Skip} \\
&\quad \text{fi} \\
&\quad \text{end_safelet_app} \rightarrow \text{Skip} \\
&\quad \{\text{getSequencerCall}, \text{getSequencerRet}, \text{end_safelet_app}, \text{safeletInitializeCall}, \text{safeletInitializeRet}\} \\
\end{align*}
\]

provided \( \text{sid} \neq \text{null} \).
Law [seq-interleave*]

\[
\begin{align*}
\text{sm} &\rightarrow \text{miC} \rightarrow (\text{SH}_1 \rightarrow r.id_1 \rightarrow \text{Skip}) ; \text{miR} \rightarrow \text{dm} \rightarrow \text{Skip} \\
\text{FA} &\rightarrow (\text{SH}_1, \text{SH}_2) \\
\text{FA} &\rightarrow (\text{SH}_1, \text{SH}_2) \\
\text{FA} &\rightarrow (\text{SH}_1, \text{SH}_2)
\end{align*}
\]

where

\[
\begin{align*}
\text{FA} &\equiv \text{sm} \rightarrow \text{miC} \rightarrow (r.id_1 \rightarrow \text{Skip}) ; \text{miR} \rightarrow (sh.id_1 \rightarrow ah \rightarrow dh.id_1 \rightarrow \text{Skip}) ; \text{dm} \rightarrow \text{Skip}
\end{align*}
\]

Lemma [sequencer-split-mult-seq*]

\[
\begin{align*}
\text{start sequencer} &\rightarrow (\_ i : I \bullet \text{start mission} i \rightarrow \text{done mission} i \rightarrow \text{Skip}); \\
\text{done sequencer} &\rightarrow \text{Skip}
\end{align*}
\]

\[
\begin{align*}
\text{start sequencer} &\rightarrow \\
&\quad (\text{getNextMissionCall}!id \rightarrow \text{getNextMissionRet!id?m} \rightarrow \\
&\quad \mu X \bullet (\text{if } m = \text{null} \rightarrow \text{end sequencer app} \rightarrow \text{Skip} \\
&\quad \text{[} m \neq \text{null} \rightarrow \text{start mission} m \rightarrow \text{done mission} m \rightarrow X \text{]);} \\
\text{done sequencer} &\rightarrow \text{Skip} \\
\text{var } i : \text{nat} \bullet b := 1; \\
&\quad (\text{getNextMissionCall}?!x!id \rightarrow \\
&\quad \mu X \bullet (\text{if } i > \# I \rightarrow \text{getNextMissionRet}!x!id?null \rightarrow \text{Skip} \\
&\quad \text{[} n : \text{dom } I \bullet i = n \rightarrow \text{getNextMissionRet}!x!id!(I n) \rightarrow i := i + 1 \text{]);} \\
&\quad \text{X} \\
&\quad \text{end sequencer app} \rightarrow \text{Skip} \\
\text{provided } I \in \text{seq ID} \land \# I > 1 \land \forall i : I \bullet i \neq \text{null}
\end{align*}
\]
Law [sequencer-split-single*]

\[
\begin{align*}
\text{start\_sequencer} & \rightarrow \text{start\_mission}.
\end{align*}
\]

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

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

Law [sync-async-conv*]

\[
\begin{align*}
\mu \mathbb{X} \cdot (c \rightarrow G) \triangleright D_1 \triangleright | \text{wait} | P) \triangleright | \text{end} | \rightarrow \text{Skip} \triangleright | n_s_1 | c_s | n_s_2 | O\text{Handlers}
\end{align*}
\]

provided

\( B \) is not recursive and \( \text{LHS} \) is deadlock-free and feasible.

\( c_i \notin \text{usedC(LHS)} \land T < D_1 < P \land D_2 < P \)

\( O\text{Handlers} \triangleright | c | \text{TimeReq} = O\text{Handlers} \)

Law [termination-intro*]

\[
\begin{align*}
\mu \mathbb{X} \cdot F; X
\end{align*}
\]

provided \( | t, rt | \cap \text{usedC(F)} = \emptyset \).
Appendix B. Sample proof

Law [par-prefix-dist*]

\[ (A \parallel ns_1 \mid cs \mid ns_2 \parallel B); \ c \rightarrow Skip \]
\[ = \]
\[ (A; \ c \rightarrow Skip \parallel ns_1 \mid cs \cup \{c\} \mid ns_2 \parallel B; \ c \rightarrow Skip) \]

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

Proof

\[ (A \parallel ns_1 \mid cs \mid ns_2 \parallel B); \ c \rightarrow Skip \]
\[ = (A \parallel ns_1 \mid cs \mid ns_2 \parallel B); \ (c \rightarrow Skip \parallel ns_1 \mid \{c\} \mid ns_2 \parallel Skip) \] \( \text{(Law C.78[14])} \)
\[ = (A \parallel ns_1 \mid cs \cup \{c\} \mid ns_2 \parallel B); \ (c \rightarrow Skip \parallel ns_1 \mid \{cs \cup \{c\}\} \mid ns_2 \parallel Skip) \] \( \text{(Law C.80[14])} \)
\[ = (A; \ c \rightarrow Skip \parallel ns_1 \mid cs \cup \{c\} \mid ns_2 \parallel B; \ c \rightarrow Skip) \] \( \text{(Law C.88[14])} \)

\[ \square \]

Appendix C. Excerpts of the syntax of SCJ-Circus

\[
\text{SCJProgram} ::= \{\text{SCJParagraph}\}
\]

\[
\text{SCJParagraph} ::= \text{SCJCSafelet} | \text{SCJCMissionSequencer} | \text{SCJCMission} | \text{SCJCHandler} | \text{CircusParagraph}
\]

\[
\text{SCJCSafelet} ::= \text{safelet} N \equiv \text{begin} \]
\[ \{\text{SCJCSafeletProcessParagraph}\} \]
\[ \{\text{state Schema-Expression}\} \]
\[ \{\text{SCJCSafeletProcessParagraph}\} \]
\[ \{\text{initialize} \equiv \text{SCJCSafeletAction}\} \]
\[ \{\text{SCJCSafeletProcessParagraph}\} \]
\[ \text{getSequencer} \equiv \text{res return} : \text{sequencer} \bullet \text{SCJCSafeletAction} \]
\[ \text{SCJCSafeletProcessParagraph} \]
\[ \text{end} \]

\[
\text{SCJPeriodicHandler} ::= \text{periodic handler handler} N \equiv \text{begin} \]
\[ \{\text{SCJCHandlerProcessParagraph}\} \]
\[ \{\text{start Expression period Expression}\} \]
\[ \{\text{SCJCHandlerProcessParagraph}\} \]
\[ \{\text{state Schema-Expression}\} \]
\[ \{\text{SCJCHandlerProcessParagraph}\} \]
\[ \{\text{initialize} \equiv \text{SCJCHandlerAction}\} \]
\[ \{\text{SCJCHandlerProcessParagraph}\} \]
\[ \text{handleAsyncEvent} \equiv \text{SCJCHandlerAction} \]
\[ \{\text{SCJCHandlerProcessParagraph}\} \]
\[ \text{end} \]

\[
\text{SCJCHandlerProcessParagraph} ::= \text{Paragraph} \]
\[ | N \equiv \text{SCJCHandlerParametrisedAction} \]
\[ | \text{nameset} N \equiv \text{NameSetExpression} \]
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))