Noname manuscript No.
(will be inserted by the editor)

Safety-Critical Java Programs from *Circus* Models

Ana Cavalcanti · Frank Zeyda · Andy Wellings · Jim Woodcock · Kun Wei

the date of receipt and acceptance should be inserted later

Abstract Safety-Critical Java (SCJ) is a novel version of Java that addresses issues related to real-time programming and certification of safety-critical applications. In this paper, we propose a technique that reveals the issues involved in the formal verification of an SCJ program, and provides guidelines for tackling them in a refinement-based approach. It is based on *Circus*, a combination of well established notations: Z, CSP, Timed CSP, and object orientation. We cater for the specification of timing requirements and their decomposition towards the structure of missions and event handlers of SCJ. We also consider the integrated refinement of value-based specifications into class-based designs using SCJ scoped memory areas. We present a refinement strategy, a *Circus* variant that captures the essence of the SCJ paradigm, and a substantial example based approach on a concurrent version of a case study that has been used as a benchmark by the SCJ community: an aircraft collision detector.

Keywords SCJ; *Circus*; RTSJ; real-time systems; refinement; verification

1 Introduction

An international effort has recently produced an Open Group standard for a high-integrity real-time version of Java: Safety-Critical Java (SCJ) [21]. It is a subset of Java augmented by the Real-Time Specification for Java (RTSJ) [38], which supplements Java's garbage-collected heap memory model with support for memory regions [36] called memory areas.

The execution model of an SCJ program is based on missions and event handlers. Additionally, SCJ restricts the RTSJ memory model to prohibit use of the heap and defines a policy for the use of memory areas. The SCJ design facilitates certification. It is organised in Levels (0, 1, and 2), with a decreasing

Department of Computer Science, University of York, York, YO10 5GH, UK

amount of restrictions to the execution model. Our work is on SCJ Level 1, which corresponds roughly to the Ravenscar profile for Ada [4].

The standardisation work includes the production of a reference implementation, but no particular application-design technique. We address this using the *Circus* family of languages for refinement [8]. They are based on a flexible combination of elements from Z [41] for data modelling, CSP [32] for behavioural specification, and standard imperative commands from Morgan's calculus [26]. Variants and extensions of *Circus* include *Circus Time* [34], which provides facilities for time modelling from Timed CSP [31], and *OhCircus* [9], which is based on the Java model of object-orientation.

Circus has been used for modelling and verification of control systems specified in Simulink [7,24]. It is currently being used to verify aerospace applications, including virtualisation software by the US Naval Research Laboratory [14]. The semantics of the *Circus* family of languages is based on the Unifying Theories of Programming (UTP) [18]. This is a framework that supports refinement-based reasoning in the context of a variety of programming paradigms. It supports the independent treatment of programming theories, with associated techniques for their combination in a tractable way. In addition, it caters for the axiomatic, denotational, and operational styles of semantic definitions. This makes it possible for us to consider a rich language for refinement that supports the use of object-oriented and SCJ constructs as well as the modelling and verification of time properties.

The UTP has been used to define the semantics of object-oriented [33] and time [34] constructs. We have also presented a *Circus*-based formalisation of the SCJ execution model [42] and a UTP theory for the SCJ memory model [10]. What we present here is a refinement strategy for deriving SCJ programs from *Circus* specifications that builds on these results. We also rely on previous results on *Circus* variants and UTP theories for references [39,16,9].

We propose an approach for stepwise development of SCJ programs based on specification models that do not consider the details of either the SCJ mission or memory models. Four *Circus* specifications characterise the major development steps: we call them anchors, as they identify the (intermediate) targets for refinement and the design aspects treated in each step.

Each anchor is written using a different combination of the *Circus* family of notations. The first anchor is the abstract specification model. The last is so close to an SCJ program as to enable automatic code generation. It is written in *SCJ-Circus*, a new version of *Circus* extended with constructs that correspond to the components of the SCJ programming paradigm. They are syntactic abbreviations for definitions introduced in [42] to characterise the SCJ infrastructure and applications; they use a combination of the variants of *Circus* to cater for time, object-orientation, and the SCJ memory model.

By extending *Circus* with SCJ constructs, we can model SCJ programs accurately in the unified framework of a refinement language. Our final refinement target is, in a sense, an SCJ program, since the *SCJ-Circus* model is so low level as to allow direct translation to Java code. We are, however, tackling

this translation as separate work in line with what we have previously achieved for low-level *Circus* models and corresponding Java implementations [13].

A preliminary version of our refinement strategy is presented in [11], where it is applied to a very simple toy example: a communication line. Here, we describe our development technique in much more detail, and provide concrete guidance for design and verification. We also define modelling and refinement patterns that target specific design strategies. Finally, we apply our technique to a much more significant case study: the collision detector (CD_x) discussed in [19], which is a benchmark in the RTSJ and SCJ communities.

Our development strategy establishes, by construction, that the *SCJ-Circus* model is a refinement of the specification used as the first anchor. This means that safety, liveness, and timing properties are preserved. Safety requires that the sequences of interactions (traces) of the program are possible for the specification. Liveness requires that deadlock or divergence in the program can occur only if allowed in the specification. Finally, preservation of the timing properties requires that the deadlines and budgets defined in the specification are enforced by the deadlines and budgets defined for the components of the program. Our long-term goal is to provide for Safety-Critical Java at least the same level of support that the SPARK tools, for instance, provide for Ada.

Regarding time, our strategy makes use of decomposition via refinement. It is inspired by the work in [17], which introduces time into Morgan's refinement calculus so that derivation of code from specifications is similar to that for untimed specifications. In our approach, the requirements in the first anchor are localised in the SCJ components of the final target anchor. It is, in this way, annotated with the machine-independent timing requirements that every correct implementation (for a specific platform) needs to satisfy. Verifying that they do may require, for instance, schedulability analysis.

Mukherjee et al. [27] have developed a framework consisting of a real-time specification language based on VDM-SL, an implementation language designed for safety-critical systems, and a collection of rules for decomposing specifications into code. Specifications are expressed using pre and postconditions with a time clause. As in our approach, correctness of the decomposition is guaranteed by the rules, but their focus is on sequential systems.

Some aspects of resources are not covered in *Circus*. We argue informally that the patterns that we use ensure that the use of resources is adequate: for instance, there is no memory leak. More rigorous support for quantification of memory usage is proposed in complementary lines of work for RTSJ [3].

An additional contribution of the work presented here is an extension of the original CD_x to produce a concurrent version available at `www.cs.york.ac.uk/circus/hijac`; it exploits the features of SCJ Level 1. The complete description of the application of our refinement strategy to this example is in [43], where the refinement steps and the novel refinement laws needed are discussed in detail. Here, we describe the overall development strategy that we propose for SCJ programs, and use the concurrent CD_x to illustrate the ideas.

The goal of this paper is to present an accessible description of our technique. Our novel contributions are as follows. First, we present a detailed

strategy for development by refinement of SCJ programs. It is a solid basis for other more specialised techniques that can focus on particular aspects of the development, like parallelisation or sharing, for instance. Second, we propose a novel variant of *Circus* that captures the SCJ programming paradigm: *SCJ-Circus*. Finally, a concurrent version of an important benchmark for the RTSJ and SCJ communities has been developed and made available.

Next, we present the notations used in our work, namely, SCJ and *Circus*, and our case study, the CD_x. Section 3 presents our refinement strategy and Section 4, its application to the CD_x. We draw our conclusions in Section 5.

2 Preliminaries

We present now a brief overview of SCJ, of the CD_x, and of *Circus*.

2.1 Safety-Critical Java

The components of the programming paradigm adopted by SCJ are a safelet, a mission sequencer, missions, and event handlers. The control flow for an SCJ Level 1 program consists of a sequence of missions, during which a number of event handlers are executed concurrently. The safelet is the entry point of the application, and the mission sequencer defines the sequence of missions to be executed. During a mission, event handlers are released.

The handlers of a mission are either released periodically, or respond to aperiodic events. They are executed concurrently by a priority-based scheduler, and access to shared data has to be performed by `synchronized` methods (whose implementations support the priority-ceiling emulation protocol to bound priority inversions). A mission continues to execute until one of its handlers requests termination. A cleanup phase is performed after termination has occurred, and afterwards the next mission, if any, is prepared for execution.

An API defines interfaces and abstract classes to be used in an SCJ program. The safelet class is an implementation of the `Safelet` interface. Its methods are `setUp()` and `tearDown()`, which perform initialisation and finalisation tasks, and `getSequencer()`, which creates the mission sequencer. This is an object of a subclass of the `MissionSequencer` abstract class. It implements the `getNextMission()` method, which returns the next mission to be executed. Missions are implemented as subclasses of `Mission`; typically, they override methods like `initialize()`, which creates the mission's handlers.

The event handlers are instances of a subclass of `AperiodicEventHandler`, `AperiodicLongEventHandler`, or `PeriodicEventHandler`. An instance of a subclass of `AperiodicLongEventHandler` is an aperiodic handler that accepts simple input data at release time. The code executed when the handler is released is defined by the `handleAsyncEvent()` methods in these subclasses.

SCJ supports software events. These are in contrast with *Circus* events that represent inputs and outputs of the application, which are typically in

the form of an interaction with some hardware device. On the other hand, software events are instances of the `AperiodicEvent` SCJ class. Aperiodic handlers can be bound to them, and all bound handlers are released when a software event is fired (via a call to the `fire()` method of `AperiodicEvent`).

The program, the missions, and the handler releases have associated memory areas (where dynamically created objects are stored). The immortal area holds objects throughout the lifetime of the program: they are never deallocated. A mission area is cleared out at the end of each mission. Each handler has a per-release memory area, cleared out at the end of each release. During a release, a stack of temporary private memory areas can be created.

In the next section we present an SCJ program based on the case study described in [19], where the CD_x is presented as a benchmark to profile implementations of RTSJ virtual machines. It is later explored in [20] in order to adapt the code to comply with the architectural restrictions of SCJ. Here, we consider a novel concurrent implementation in SCJ Level 1.

2.2 CD_x: a concurrent collision detector

The purpose of the CD_x is to detect potential collisions of aircraft located by a radar device. We take the program discussed in [19] as a basis for the definition of our requirements. It uses a cyclic executive, and embeds the assumption that the radar collects (and buffers) a frame of aircraft positions that becomes available for input periodically. In each iteration, the CD_x: (1) reads a frame; (2) carries out a voxel-hashing step that maps aircraft to voxels; (3) checks for collisions in each voxel; and (4) records and reports the number of detected collisions. A voxel is a volumetric element; all voxels together subdivide the entire space. The voxels in the CD_x superimpose a coarse 2-dimensional grid on the x-y plane with the height of a voxel extending along the entire z-axis. Thus, the altitude of aircraft is abstracted away. This reduces the number of necessary collision tests: after mapping aircraft to the voxels that are intersected by their interpolated trajectories, it is sufficient to test for possible collisions within each voxel. Details of the algorithm can be found in [19].

Since the majority of the computation burden is in the checking for collisions in step (3), we propose a version of the CD_x where this task is parallelised. As a result, we obtain an SCJ program that illustrates the features of SCJ Level 1. Our aim with the concurrent CD_x is, most of all, to provide a genuine and more representative Level 1 application. Due to the novelty of the SCJ paradigm and technology, such applications are still difficult to come by in the public domain. On the other hand, even though we are not specifying a particular radar system, concurrent collision detection is a reasonable target to improve the performance of such an application.

Our program contains 27 classes, and almost 3000 lines of code. Its design is not overly complex, since it consists of a single mission, but it is not trivial. We have seven handlers that interact using shared variables and a barrier. Our first implementation had a mistake, a race condition, that we uncovered

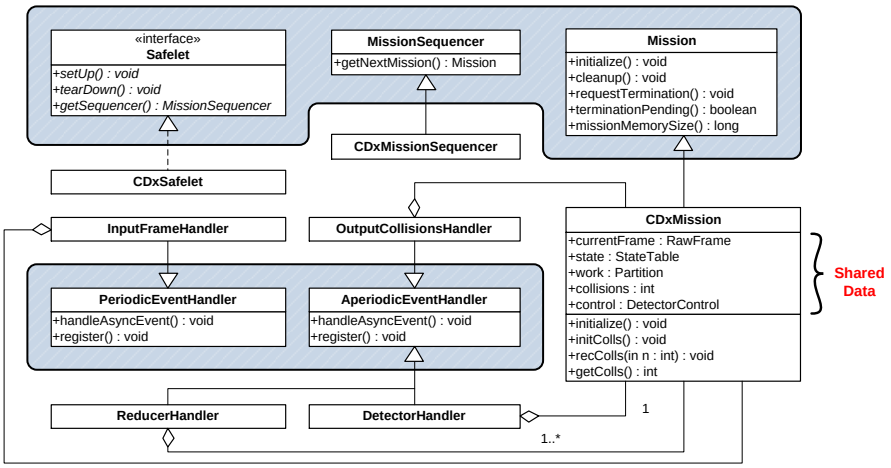

Fig. 1 UML diagram for the concurrent CD_x program

in conducting the case study discussed in Section 4. Unlike [19], we allow aircraft to enter or leave the radar frame.

Figure 1 presents a UML class diagram that illustrates the CD_x design. The classes shaded are part of the SCJ API. The classes named `CDxSafelet`, `CDxMissionSequencer` and `CDxMission` implement the safelet, the mission sequencer, and the mission. The behaviour of the `setUp()` and `tearDown()` methods of `CDxSafelet` is void; this is typically the case, and a new version of the SCJ standard might remove these methods. The method `getSequencer()` returns an instance of `CDxMissionSequencer`, and `getNextMission()` returns an instance of `CDxMission` when called for the first time. Since the mission does not terminate, `getNextMission()` is not called again.

For mission execution, first the `initialize()` method of `CDxMission` is called. It creates the mission's handler objects and shared data in mission memory. The handler classes are `InputFrameHandler`, `OutputCollisionsHandler`, `ReducerHandler`, and `DetectorHandler`. We choose to create four instances of `DetectorHandler`, possibly corresponding to a scenario in which we have four processors. The refinement in Section 3 can proceed without changes in the presence of two or more instances. A more general design for the CD_x could allow the configuration of the number of instances; our program, however, is enough to illustrate the main aspects of our technique.

The shared data is held by public fields of `CDxMission`. The `currentFrame` and `state` fields record the current and previous frame of aircraft positions; recording previous positions is important for calculating their predicted motions. As we divide and distribute the computational work, `work` holds the partitions of voxels to be checked by each of the detection handlers, and `collisions` is used to accumulate the result of the detection. We recall that the voxels partition the space in such a way that collision detection can be

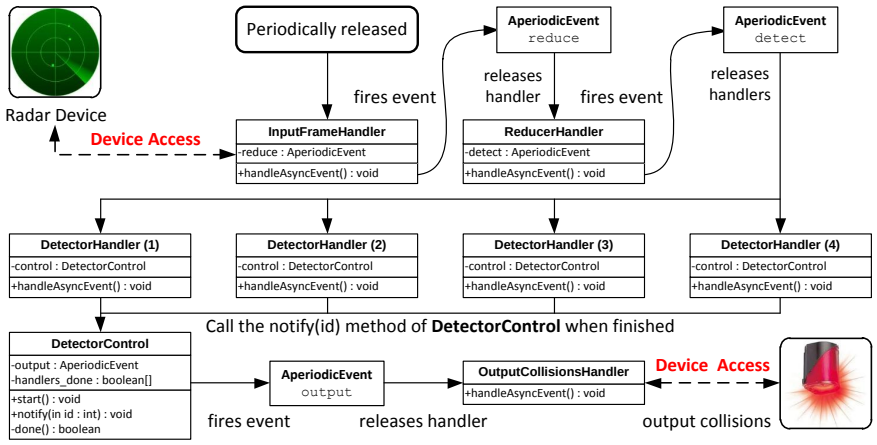

Fig. 2 Parallel CD_x control flow

carried out in each voxel independently. A further shared object `control` plays a crucial part in orchestrating the execution of handlers.

Figure 2 summarises the control mechanism of the SCJ application. The three software events, `reduce`, `detect` and `output`, are used to control execution of the handlers. The program design ensures that the handlers effectively execute sequentially in each cycle, apart from the four instances of `DetectorHandler`, which carry out their work concurrently.

The `InputFrameHandler` is the only periodic handler. It is released at the beginning of each cycle to interact with the hardware to read the frame into `currentFrame` and update `state` accordingly. Afterwards, it releases the `ReducerHandler`, via the `reduce` event, to carry out the voxel partitioning and distribute the work among the detector handlers by populating `work`. Once this is done, it concurrently releases all `DetectorHandler` instances by firing the `detect` event. These handlers carry out the detection work and store their result in `collisions`. The mechanism for releasing `OutputCollisionsHandler`, which outputs the number of collisions to an external device, uses the shared object `control`. `DetectorControl` provides a method `notify()`, which is called by the detector handlers at the end of each release. It fires the event `output` when all detection work is done. This illustrates that sharing may occur not only to exchange data, but also in the design of execution control.

Our program highlights various features of the SCJ mission framework: the subdivision of a mission into handlers, the control of handlers via software events, and the sharing of data for both data communication and control purposes. The verification of this program not only has to address functional correctness, but also must show that the flow of activities in Figure 2 can be executed within the duration of a cycle, which is treated as a hard deadline.

Next, we present *Circus*. In subsequent sections, we present our formal development strategy based on *Circus*, and revisit the CD_x as an example.

2.3 *Circus*

The key elements of *Circus* specifications are processes (like in CSP). They interact with the environment through atomic and instantaneous events: either simple synchronisations, or input and output communications. Unlike CSP, a *Circus* process also encapsulates local state, defined as in Z, and accessible by its local actions, but hidden to other processes.

A process specification defines its state as a Z schema (that is, a record type). Local actions operate on the state, while possibly interacting with the environment via events. The action notation is a mixture of the Z schema calculus for abstract specification of data operations, CSP constructs for specification of interaction patterns, and imperative commands (assignments, conditionals, and so on) from (Morgan's) refinement calculus. A main action at the end of the process specifies its visible behaviour.

An example of a process, $ABReqsCDx$, is in Figure 3; it specifies data and behavioural (but not timing) requirements of the CD_x as part of its first anchor. Its state is defined by the schema $AStateCDx$ with two components *posns* and *motions*; they hold the current frame of aircraft positions, and their motions in the radar field as 3-D vectors. The type *Frame* is that of (finite) functions from aircraft to vectors; it is defined in Appendix A along with all other constants and functions used in Figure 3. The state invariant is defined by a predicate in the schema: the domains of *posns* and *motions* are the same.

Init, *RecordFrame* and *CalcCollisions* are (local) actions of $ABReqsCDx$ specified by Z (data) operation schemas. *Init* defines the initial values of *posns* and *motions* to be the empty function $\varnothing$. *RecordFrame* changes the state (as indicated by $\Delta AStateCDx$) and takes an input *frame*?. It defines the new value $posns'$ of the *posns* state component to be equal to the input *frame*?. The new value $motions'$ of the state component *motions* is defined by a set of mappings. For each aircraft a in the domain of $posns'$, its associated motion depends on whether a was already in the radar frame ($a \in \mathrm{dom}\, posns$) or not. If it was, its motion is the difference between its current position $posns'\, a$ and its previous position $posns\, a$. Otherwise, it is the zero vector $ZeroV$.

CalcCollisions does not change the state (as indicated by $\Xi AStateCDx$) and has an output *colls*! that indicates the number of collisions. A function *CalcCollisionSet* determines the finite set *collset* of pairs of aircraft that are on a collision route. If it is empty, that is, its size given by $\#\, collset$ is 0, then *colls*! is 0. Otherwise, it is a number greater than or equal to the number of possible collisions: if the value is greater than 0, there is at least one possible collision. The division caters for the symmetry of collision detection: if a pair (a_1, a_2) of aircraft is in *collset*, so is (a_2, a_1). By dividing the result by 2, we obtain in *colls*! a better approximation of the number of collisions.

Additionally, we have a local action $BReq1$ that uses CSP constructs, namely an input prefix and an output prefix. In an input prefix $c?x \longrightarrow A$, we have that c is a channel, x is a new local variable that records an input taken from c for use in the associated action A. In an output prefix $c!e \longrightarrow A$, the value of an expression e is output through c, before A is exe-

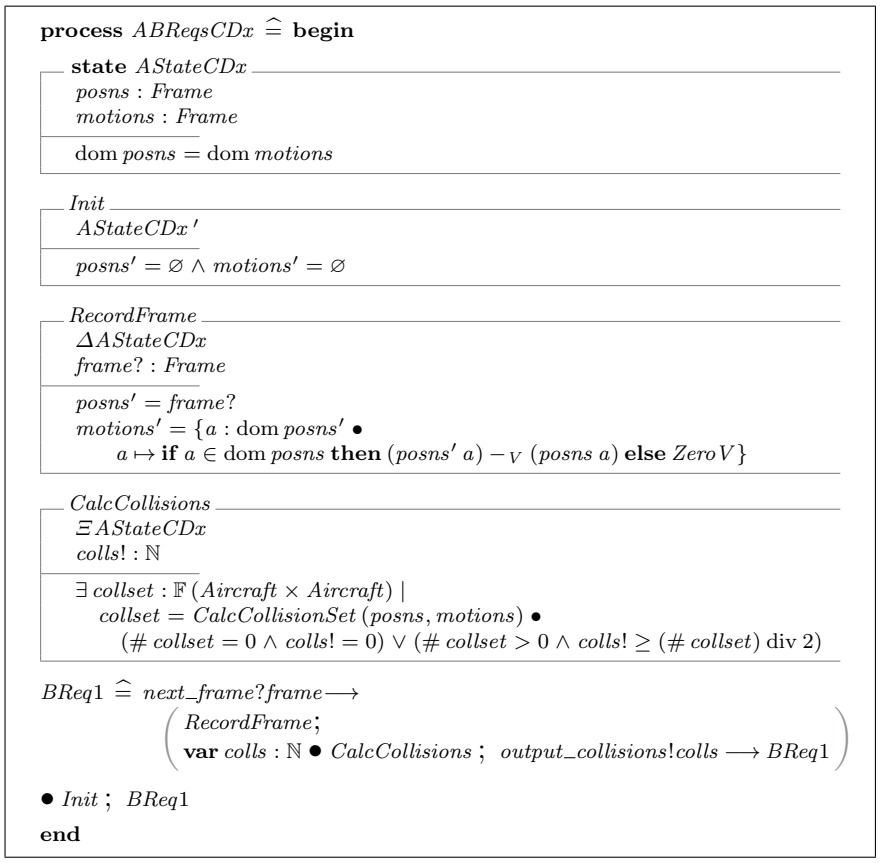

Fig. 3 Anchor A: Process for the behavioural requirements of the CD_x

cuted. For example, *BReq*1 takes an input *frame* on the *next_frame* channel, and calls *RecordFrame*, which uses this input. This is followed by the declaration of a local variable *colls* of type $\mathbb{N}$. The body of the variable block invokes *CalcCollisions* and then outputs on *output_collisions* the value of *colls*. At the end, we have a recursive call to *BReq*1 to establish a cyclic behaviour. The interactions on *next_frame* and *output_collisions* represent the interactions of the CD_x with the environment: the radar and display hardware.

The main action of a process, which defines its behaviour, is written at the end after the •. In our example, it is a call to *Init*, followed by a call to *BReq*1.

A parallel composition $(A_1 [\![ns_1 \mid cs \mid ns_2]\!] A_2)$ of actions is parametrised by a synchronisation set cs of channels and two disjoint sets ns_1 and ns_2 of variable names that the individual parallel actions may modify. The modifications made by each action only become visible after the parallelism has finished; during the parallelism there is no possibility of interference. The parallel actions must synchronise on interactions via the channels in cs.

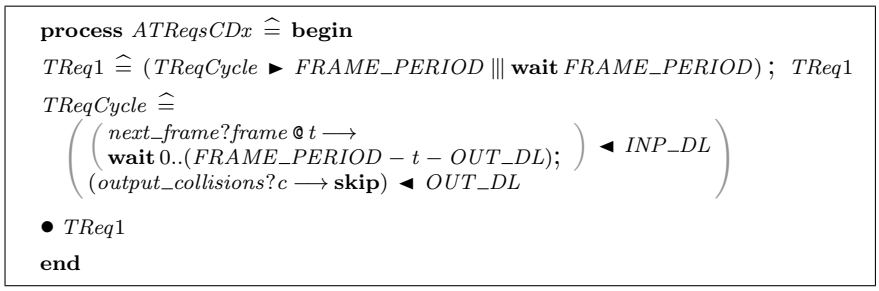

Fig. 4 Anchor A: Process for the timing requirements of the CD_x

Like actions, *Circus* processes can be combined using CSP operators: for example, sequentially $(P_1;\ P_2)$ or in interleaving $(P_1 \,|||\, P_2)$. In a parallelism $P_1 \,[\![\ cs\ ]\!]\, P_2$ of processes, the name sets ns_1 and ns_2 that are used in an action parallelism are omitted since processes encapsulate their states. In an interleaving, the processes (or actions) proceed in parallel, but independently without communicating with each other. In a parallelism, they synchronise on communications via the channels in the synchronisation set cs.

Process parallelism is used below in the definition of CDx, the process that specifies the CD_x. It combines $ABReqsCDx$ and the process $ATReqsCDx$ in Figure 4, which specifies timing requirements and is discussed next.

$$\textbf{process}\ CDx \mathrel{\widehat=} ABReqsCDx \,[\![\ \{\!|\ next\_frame, output\_collisions\ |\!\}\ ]\!]\, ATReqsCDx$$

The parallel processes synchronise on all communications via *next_frame* and *output_collisions*. (The fat brackets $\{\!|$ and $|\!\}$ are used to specify channel sets; in our example, we have a set containing *next_frame* and *output_collisions*.) In this way, the timing restrictions imposed by $ATReqsCDx$ on interactions via these channels are taken into account in the parallelism. Here, parallelism is used as a specification device to conjoin the data and behavioural requirements in $ABReqsCDx$ with the timing requirements in $ATReqsCDx$.

Circus Time also includes Timed CSP constructs and deadlines [34,39]. As opposed to Timed CSP, it has a discrete time model; this is enough to reason about software implementations. We still, however, have issues of robustness [22]; namely, it is possible to specify an infinitely fast system. On the other hand, time stops are not (necessarily) an indication of a faulty model; they are used to model deadlines. (Semantically, a deadline or time stop is a miracle [39], and is treated in the usual way in refinement calculi [26].)

The process $ATReqsCDx$ in Figure 4 makes use of *Circus Time* constructs. It is a process without state, whose main action $TReq1$ defines a periodic cyclic behaviour; the period is defined by a constant $FRAME\_PERIOD$.

There are two deadline operators in *Circus Time*: $A \blacktriangleright d$ asserts that A terminates within d times units, and $A \blacktriangleleft d$ asserts that A starts (via a visible interaction) within d time units. They typically are assumptions on the environment because if A requires synchronisation with the environment, the

deadlines enforce the fact that they take place in the established period. So, in *TReq*1, for example, a deadline ensures that *TReqCycle* does not take longer than *FRAME_PERIOD* time units, in spite of its reliance on the provision of inputs and acceptance of outputs by the environment.

An action **wait** t terminates after exactly t time units, and **wait** $t_1 \,..\, t_2$ is a nondeterministic choice of a wait period between t_1 and t_2. In *Circus Time*, data operations *Op* do not take time: any time properties need to be explicitly defined. In this context, the **wait** constructs can be used to define guarantees provided by the program, as opposed to assumptions on the environment.

For example, if a data operation *Op* is followed in sequence by an input, for instance, since *Op* terminates instantly, the program has to be ready to input instantaneously. No implementation can satisfy this restriction (unless *Op* is mostly trivial and can be regarded as instantaneous). So, a more realistic model is $Op \,;$ **wait** $0 \,..\, t$, which states that *Op* can take up to t times units to complete: t defines a time budget for *Op*. We observe that it is possible to use a loosely defined constant c, introduced using a Z axiomatic description that does not determine the value of c or even does not restrict it at all, to define t. The model does not need to specify a particular value for the budget.

In *TReq*1, the action $TReqCycle \blacktriangleright FRAME\_PERIOD$ is interleaved with a **wait** of *FRAME_PERIOD* time units. Because the interleaving finishes only when both these actions finish, if *TReqCycle* finishes before its deadline, that is, *FRAME_PERIOD* time units, the interleaving still waits until the end of the period. After *FRAME_PERIOD*, a recursive call restarts the cycle. This definition of *TReq*1 follows a pattern of specification for periodic tasks; we discuss it in more general terms in Section 3.1.

In a prefixing $c?t \longrightarrow A$, the variable t records the number of time units during which the communication c was available before it took place. In *TReqCycle*, for instance, $next\_frame?frame \,?\, t \longrightarrow \mathbf{skip}$ keeps in t the amount of time since the input is offered until it is taken. It is used to define the amount of time we can wait (for the calculation of collisions to finish), before we need to make an output available. This is given by *FRAME_PERIOD* minus t, minus the deadline *OUT_DL* for the environment to accept the output. We also have a deadline *INP_DL* for the input via *next_frame*. The values of *FRAME_PERIOD*, *OUT_DL*, and *INP_DL* are unspecified (that is, loosely defined), but we require $INP\_DL + OUT\_DL \leq FRAME\_PERIOD$ for feasibility of the model. If this does not hold, the *FRAME_PERIOD* deadline imposed on *TReqCycle* in *TReq*1 may become infeasible, since the input, the output, or their combination might take too long.

In *OhCircus*, we also have classes; an example is given in Appendix B.3. These, like processes, have a state (specified in Z). Behaviour, however, is not specified by actions, but methods: data operations over the state, expressed using either Z or the imperative command language adopted in *Circus*.

A complete account of *Circus*, *OhCircus*, and *Circus Time* can be found in [8,28,9,34]. What we have provided here is a brief overview; we explain a few more details of the notation as needed.

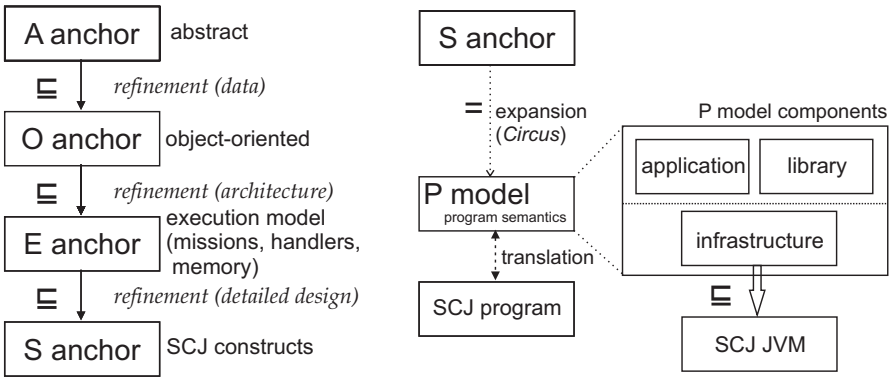

Fig. 5 Our approach to development and verification

3 Refinement strategy

In this section, we describe the steps of our refinement approach. Figure 5 shows the four Anchors A, O, E, and S, and other related artefacts. The anchors are all written using different subsets and versions of *Circus*. It is the objective of our refinement strategy to guarantee that the anchors are related by refinement ($\sqsubseteq$) as suggested in Figure 5. By transitivity of refinement, it establishes, therefore, that the A anchor, which defines an abstract model, is refined by the S anchor, which describes an SCJ program.

As already hinted, in our strategy, refinement is carried out in three main steps, each characterised by an anchor. The first step produces the O anchor, and tackles the object-oriented data model of the program. The second step introduces the E anchor, and tackles the correctness of the mission and handler decomposition and of the use of memory areas in the program. Finally, the third step, produces the S anchor, and tackles the correctness of the algorithms implemented. Importantly, it also describes the sequence of missions and parallelism of handlers in the E anchor directly in terms of SCJ constructs. This enables and facilitates later automatic generation of SCJ code.

Each of these refinement steps is divided into phases, which tackle individual aspects of the design of the target anchor. Typically, a refinement phase is realised in a series of stages, captured by the application of refinement laws. For some phases, specific refinement laws are always applicable. In other cases, there is a choice of laws depending on the design of the target anchor.

In Section 3.1, we discuss the construction of A anchors and present a few patterns that capture typical timing requirements. In Sections 3.2 to 3.4, we describe the phases of each of the three refinement steps, and their stages. Section 4 describes the application of the strategy to the CD_x, whose A anchor is in Section 2.3. It is not in the scope of this paper to provide a detailed presentation of refinement laws, but we present some examples.

3.1 Anchor A: abstract model

The A anchor is the *abstract* model of the system under development. In its specification, we use the basic *Circus* notation and *Circus Time*. Crucially, nothing is said about classes, or objects and their allocation. An A anchor defines an interaction pattern in the style of CSP. For that, it uses abstract data types in the style of Z. Parallelism is used as a way of combining (conjoining) requirements, rather than describing a concurrent design. The A anchor for our example is *CDx*, presented in the previous section and reproduced below.

$$\textbf{system}\ CDx \mathrel{\widehat=} ABReqsCDx \llbracket \{\!| next\_frame, output\_collisions |\!\} \rrbracket ATReqsCDx$$

We use **system** to emphasise the role of this process in specifying the behaviour of the overall system to be implemented. Semantically, there is no distinction to a **process** declaration. As already said, *ABReqsCDx* (Figure 3), specifies the behavioural requirements, and *ATReqsCDx* (Figure 4), the timing requirements. They are composed in parallel to define the system.

As shown in Figure 5, an A anchor is the starting point of our refinement strategy. We make no assumptions about the patterns of specification used in an A anchor, but recognise that providing a complete formal specification of a system from scratch can be a challenge. We, therefore, present here a few patterns that identify how widely used concepts in the specification and design of real-time systems can be captured; we consider periodic and sporadic tasks, and input and output jitters. They are particularly useful when we adopt the general pattern of modelling where we have a process that captures behavioural requirements and a separate process to capture timing requirements.

Periodic tasks In this case, we consider a computation executed periodically. In each period, there is a deadline d for completion of the computation, which must be less than or equal to the period p. Moreover, there is a budget b for the computation time, which is less than or equal to the deadline d.

The main action *TReq* of the process that specifies the timing requirements in this case can be written according to the following pattern, with $b \leq d \leq p$.

$$TReq \mathrel{\widehat=} (A(\mathbf{wait}\, 0 \,..\, b) \blacktriangleright d \,|||\, \mathbf{wait}\, p)\ ;\ TReq$$

We use $A(\mathbf{wait}\, 0 \,..\, b)$ to denote an action A whose computation time, defined in terms of *Circus Time* constructs, is between 0 and b. We observe that the action $\mathbf{wait}\, 0 \,..\, b$ does not necessarily occur in A and, in particular, it is not a parameter of A. In *Circus*, actions cannot be used as parameters of other actions. What this pattern requires is that the execution time defined for A (using any combination of *Circus Time* constructs) is bounded by b. The deadline ensures, in addition, that, even if there are synchronisations (with the environment) in A, its execution does not take more than d time units.

In the CD_x example, the main action *TReq*1 of the process *ATReqsCDx* that captures the timing requirements specifies a periodic task. In this case,

the deadline and the period are the same, *FRAME_PERIOD*, and *TReqCycle* defines the computation. It uses deadlines (*INP_DL* and *OUT_DL*) to add predictability to the communications, and a wait to define a budget for the computation characterised in the behavioural requirements.

If the above pattern is used for $d > p$, it is possible for the computation $A(\mathbf{wait}\, 0 \,..\, b) \blacktriangleright d$ to last longer than the period p, and in this case the next cycle is delayed. If $b > d$, the deadline on $A(\mathbf{wait}\, 0 \,..\, b)$ makes the choices of waiting times $d + 1 \,..\, b$ infeasible. As usual in refinement calculi, infeasible (miraculous) behaviours cannot be refined to code. For a timing requirement, this means that the deadlines cannot be met by any implementation. The restriction $b > d$ ensures that the specification is feasible, and therefore a useful starting point for a development by refinement.

Sporadic tasks In this case, a computation (with budget b and deadline d) is triggered by a release event e with a minimum inter-arrival time m.

$$TReq \mathrel{\widehat{=}} e \longrightarrow (A(\mathbf{wait}\, 0 \,..\, b) \blacktriangleright d \mid\mid\mid \mathbf{wait}\, m)\,;\ TReq$$

No relationship between d and m is required. If the computation finishes before the minimum inter-arrival time, the **wait** m forces the task to wait before a new trigger is accepted. Otherwise, the **wait** m has no effect. The budget b, however, should be smaller than the deadline d ($b \leq d$). Otherwise, as already explained, the possibilities of computations in $A(\mathbf{wait}\, 0 \,..\, b)$ that last longer than d time unities give rise to infeasible behaviours.

Input and output jitters Often, the computation involves an input, followed by a calculation and an output. An input jitter ij defines the maximum amount of time that is available for reading the input (before it becomes outdated). Similarly, an output jitter oj is the maximum amount of time available for writing the output. The computation deadline d is taken up by an input time less than or equal to ij, a calculation time, and an output time less than or equal to oj. We, therefore, require that $ij + oj$ is less than or equal to d.

If we take the input $in?x$ and the output $out!e$ as single events, we can specify the computation as follows. This pattern can be used in conjunction with either of the two previous patterns for tasks, with $(ij + oj) \leq d$.

$$(in?x@t \longrightarrow (A(\mathbf{wait}\ 0 \,..\, (d - t - oj)) \mid\mid\mid \mathbf{wait}(d - t - oj))) \blacktriangleleft ij;$$
$$(out!e \longrightarrow \mathbf{skip}) \blacktriangleright oj$$

In t, we record the amount of time taken to carry out the input: the amount of time since the program asked for the input x until it is actually provided. The computation budget is at most $d - t - oj$, but the output is not offered until $d - t - oj$ time units have passed. That leaves time for the output, which can take up to oj time units, and takes into account the fact that t time units have already been taken up. There is no flexibility in finishing the computation before, because the output should not be produced too soon.

If the pattern above is used for values of ij, oj, and d that do not satisfy the restriction $(ij + oj) \leq d$, then the value $d - t - oj$ may be negative. The action **wait** $t_1 \,..\, t_2$ is well defined even when t_2 is smaller than t_1. It is the infeasible action (miracle), and cannot be implemented. Again, the restriction associated with the use of the pattern ensures feasibility of the specification.

If abstracting the input and output as single events is not adequate, because we want to reason about the input and output mechanisms, then we need a different approach. In this case, we need *Circus* channels to represent the points of interaction where the input starts and finishes, and where the output starts and finishes, and we need to specify timing requirements using these events.

3.2 Anchor O: concrete state with objects

The first step of our refinement strategy is a data refinement: it introduces concrete data to represent the abstract data types of the A anchor, and the shared data. The target is an O anchor, which introduces the use of classes and objects. The object-oriented constructs employed are those of *OhCircus*.

Due to the nature of data refinement (in *Circus*), the structure of the O anchor, in terms of processes and actions, is the same as that of the corresponding A anchor. Data refinement only replaces and adds state components to the model. The types of the concrete components may be specified by *OhCircus* classes, but creation and allocation of objects is not considered yet.

It is standard for developments by refinement to start with a data refinement. The rationale is that architectural patterns and algorithms are typically associated with a particular data representation, which is, therefore, considered and captured in the first step of refinement.

In a data refinement, particular algorithms are not considered, but it is unrealistic to assume that the developer makes no consideration of how the concrete data types proposed can be efficiently used to realise the functionality of the program. In the case of our strategy, in this step we do not consider explicitly the structure of missions and handlers of the target program. On the other hand, it is only to be expected that a developer is aware of the need to provide the program functionality via missions and handlers, and of the sharing of data that might be required between them.

Figure 6 describes our proposed strategy for this step. We take inspiration from Morgan's auxiliary variables technique [25] to facilitate the specification of the concrete components. So, in the first two phases of this step, CS and SD, we introduce components of the concrete model, but eliminate those of the abstract model only in the third and final phase, EL.

The following concerns are addressed: (a) refinement of abstract (model) variables by concrete variables used by the program (in Phase CS); and (b) introduction of state components for data shared between handlers and missions (in Phase SD). We make a distinction between data and control objects. Whereas data objects are purely used to communicate data, control objects have as their primary purpose to control the flow of execution, for instance,

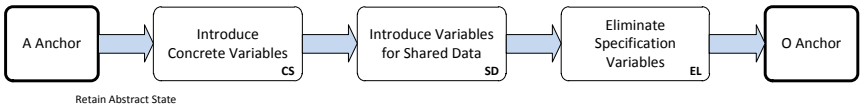

Fig. 6 Overview of the strategy for the Anchor O Step

$$
\begin{array}{l}
\textbf{process}\ SCJsystem \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ SCJstate == [x, y, z : \ldots \mid \ldots] \\
\quad Init \mathrel{\widehat=} \ldots \\
\quad Handler1 \mathrel{\widehat=} \ldots \textbf{var}\ a, b, c \bullet \ldots \\
\quad Handler2 \mathrel{\widehat=} \ldots \\
\quad \ldots \\
\quad InitM1 \mathrel{\widehat=} \ldots \\
\quad HandlersM1 \mathrel{\widehat=} (Handler1 \parallel Handler2 \parallel \ldots) \setminus swevts \\
\quad MArea1 \mathrel{\widehat=} \textbf{var}\ l, m, n \ldots \\
\quad Mission1 = InitM1 \,;\ (HandlersM1 \,[\![\, ns \mid mcs \mid \{\} \,]\!]\, MArea1) \setminus mcs \\
\quad \ldots \\
\quad System \mathrel{\widehat=} Mission1 \,;\ Mission2 \,;\ \ldots \\
\quad \bullet\ Init;\ System \\
\textbf{end}
\end{array}
$$

Fig. 7 Anchor E: sketch of its structure

by releasing handlers via software events. The shared objects we consider in the O anchor and, therefore, in Phase SD, are exclusively data objects.

In all phases, including EL, we carry out a data refinement by applying the *Circus* simulation laws in [8]. If any of the new components have a class type, it needs to be declared. Introduction of a new class definition is a trivial refinement; the only complexity comes from the specification of the class itself.

In our initial account of our refinement strategy [11], we do not consider specifically the issue of sharing in this step. This is, however, necessary, since the next step introduces the control structure of missions and handlers.

3.3 Anchor E: execution model

The second step of the refinement strategy introduces the architectural design of the program in accordance with the SCJ *execution model*. The E anchor embeds the structure of the missions and handlers. It is defined by a single process (and associated type and class definitions), and is still written using standard *Circus*, *OhCircus*, and *Circus Time* constructs.

The E anchor process takes the shape sketched in Figure 7, where we consider a process named *SCJsystem*. The state components of the E anchor, in Figure 7, x, y, and z, are the variables that should be allocated in immortal memory (since they can be referenced by all missions). In the SCJ program, they can become, for instance, static fields of the `Safelet` subclass.

In the main action of the E anchor process, we call the local actions *Init* and *System* in sequence. *Init* is the specification of the program initialisation (which can be implemented in the `setUp` method of the `Safelet` subclass). *System* is a sequence of *Mission* actions; in Figure 7, we have *Mission*1, *Mission*2, and so on. For applications in which the sequence of missions to be executed is defined dynamically (on the basis of values of variables in the immortal memory), the specification of *System* needs to be more elaborate.

For each mission, the E anchor process has a group of actions; in Figure 7 we show those for *Mission*1. The variables to be allocated in mission memory are defined as local variables of an action *MArea*. These are the variables that are shared between two or more handlers. In Figure 7, we show *MArea*1 with variables l, m, and n. Internal channels represent calls to data operations that use or change these variables. Typically, these are methods of the objects held in these variables. An initialisation action, *InitM*1 in Figure 7, specifies how the values of these variables are to be initialised.

The handler actions, which in Figure 7 are *Handler*1, *Handler*2, and so on, define the behaviour of the releases of the handlers. Any software events used to synchronise their behaviour are represented by internal channels; in Figure 7, we use *swevts* to denote the set of such channels for *Mission*1. It is the hiding of the set of channels *swevts* in the definition of *Handlers* that makes the channels that represent the software events, that is, the elements of *swevts*, internal. The hiding operator ($\setminus$) is available for both actions and processes. Local variables in the handler actions are allocated in per-release memory. More elaborate algorithms may need to use temporary private memory areas to control allocation and deallocation of objects.

The *Handlers* action specifies the concurrent behaviour of the handler releases during the mission; in Figure 7, we sketch *HandlersM*1. In the parallelisms between the handler actions, the synchronisation sets (omitted in Figure 7) contain channels (in *swevts*) representing software events. The particular synchronisations required reflect the needs for sequentialisation of the handlers. Access of handlers to objects in immortal memory is determined by the name sets in these parallelisms. Due to the restrictions on parallelism in *Circus*, we cannot have a race condition arising from handlers accessing the same state component (here, variable in immortal memory) at the same time.

As already said, the behaviour of the mission itself is given by a mission action; in Figure 7, we sketch *Mission*1. What we have is a parallelism between the *Handlers* and the *MArea* actions. The synchronisation set *mcs* in this parallelism contains all channels representing calls to methods of the objects in the mission memory (which are defined in the *MArea* action). The name set associated with the *Handlers* action (that is, *ns* in Figure 7) identifies the objects in immortal memory used by the handlers. The name set associated with the *MArea* action is always empty, since this action already encapsulates the data that it uses: the object variables to be allocated in mission memory.

It is the objective of the second step of our strategy to transform the O anchor to obtain a process in the shape of the E anchor identified in Figure 7. Five phases define the refinement strategy in this step as depicted in Fig-

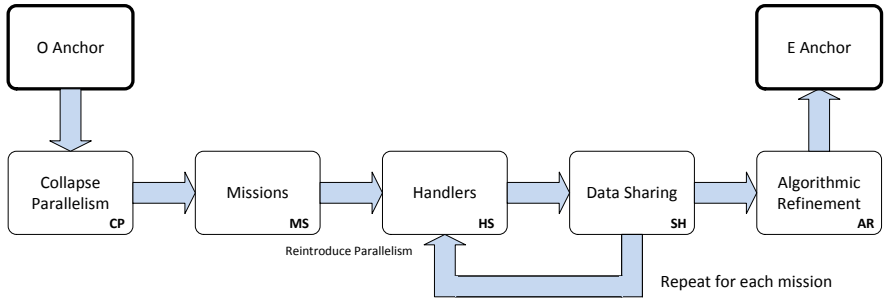

Fig. 8 Overview of the strategy for the Anchor E Step

ure 8. The first phase, CP, removes any parallelism used in the A anchor (and preserved in the O anchor) to specify requirements, since these parallelisms are typically not related to the concurrent design of the program. The second phase, MS, introduces the sequences that reflect the architecture of the missions. The next two phases, HS and SH are repeated for each of the missions. In HS, we introduce the parallelism that reflects the behaviour of the handlers releases, and the control mechanisms that orchestrate their execution. In SH, we define how variables are shared between handlers. The final phase HR uses algorithmic refinement to derive the implementation of the methods.

In the sequel we discuss the five phases of the anchor E construction.

3.3.1 Phase CP: collapse parallelism

In the CP phase, we remove the parallelism in the O (and A) anchor to produce a single process that gives a centralised and sequential account of the system. The resulting process is useful as a normalised starting point to introduce the concurrent program design in the next phases of this step.

The refinement to eliminate the parallelism can be carried out by judicious and exhaustive application of step laws that evaluate parallelism of processes and actions. They can be used to calculate (automatically) the definition of the process resulting from this phase and its local actions. A collection of step laws is available for *Circus* [28]; recasting them in the context of *Circus Time*, to deal with constructs such as **wait** actions and deadlines, is ongoing work.

It is the CP phase that makes the rest of this refinement step independent of the particular specification patterns used in the A anchor.

3.3.2 Phase MS: missions

In this phase, we transform the main action of the single process obtained in the CP phase to split it into a sequential composition of (named) actions that specify the requirements for the missions. We introduce, in particular, two new actions: *Init*, which specifies how variables to be held in immortal memory are

1. Definition of cycle timings.
2. Decomposition of data operations that are implemented across different handlers.
3. Distribution of time budget to sequential components.
4. Transformation of sequential data operations into parallel handler actions.
5. Transformation of parallel data operations into parallel handler actions.
6. Extraction of the handlers.

Fig. 9 Anchor E - Phase HS: Stages

to be initialised, and *System*, which specifies a sequence of mission actions. A final step writes the main action in terms of *Init* and *System*.

Since, as a result of the CP phase, the main action is already sequential, refinement in this phase is rather simple, and follows from the identification of the events and operations to be realised by each of the missions. Mainly, we apply an action-introduction law a few times to declare new local actions defined by actions already used in the main action, including the actions *Init* and *System* themselves. A standard copy rule is used to replace the occurrences of the new action definitions with calls to these actions.

3.3.3 Phase HS: handlers

In this phase, we tackle one of the mission actions. As a result, we obtain, as the specification of the behaviour of the mission after initialisation, an action that composes in parallel: (a) the handler actions for its handlers; (b) an action *HandlerController* that controls their interaction, if any; and (c) an action *Cycle* that captures timing requirements for a cyclic mission.

We have a number of refinement stages that address intermediate goals of the parallelisation; they are identified in Figure 9. Each stage is carried out by specialised refinement laws (reported in [43]) that capture design patterns for handlers and guide the refinement. Stage (2) refines Z data operations; all other stages in this phase focus on action refinement.

Stage (1) introduces a (cyclic) design that embeds the overall timing requirements. Typically, we use one of the patterns discussed in Section 3.1, with deadlines directly associated with the communications, and budgets with data operations. Elementary *Circus* refinement laws have to be applied that, for instance, isolate prefixing, and distribute deadlines. Specialised parallelisation laws introduce an interleaving with a **wait** action indicated in Section 3.1.

In Stage (2), decomposition of data operations is performed to reveal the sequential and concurrent activities carried out by the handlers. We observe that, even though handlers proceed concurrently, control between them via software events might establish some sequentiality. Decomposition into operations to be executed in sequence uses the *Circus* constructor for sequential (schema) actions (;). Decomposition into concurrent operations uses the conjunction operator of the Z schema calculus ($\wedge$).

The decomposition of data operations in Stage (2) is accompanied by the decomposition of their time budgets in Stage (3). We distribute the budget in

wait actions through the operations to their sequential components, including those that may have been just introduced in Stage (2). Distribution of time budgets to the parallel components is achieved in Stage (5).

In Stage (3), distribution uses the law $\mathbf{wait}\, 0..w = \mathbf{wait}\, 0..w_1 ; \mathbf{wait}\, 0..w_2$, provided $w = w_1 + w_2$, and distribution laws of **wait** (and nondeterminism). First of all, we decompose the **wait** actions into smaller time budgets, one for each sequential component. The decomposed **wait** actions are then moved through the structure of the operations and thereby attached to their respective components. This gives rise to timed definitions of the components.

Stage (4) introduces a concurrent design for handlers that embeds synchronisations to enforce any required sequential execution. More precisely, the timed sequential components that emerge in Stage (3) are now parallelised, if they are to be executed in different handlers (or group of handlers). This relies on (specialised) introduction laws for parallelism. Extra internal channels ensure that the original sequential flow of execution is preserved in the parallelism. Moreover, propagation between the handlers of (shared) data, if any, is achieved by communicating the data through these new channels.

Stage (5) mirrors the objectives of Stages (3) and (4), but is concerned with the parallel data operations. We tackle the parallelisation of data operations specified using schema conjunction. More precisely, the parallel actions identified in Stage (4) whose data operations are to be implemented concurrently in a group of handlers are now further decomposed themselves into parallelisms. The refinement again uses specialised laws for introduction of parallelism. They reflect design patterns used in handlers, and also distribute the time budget of the data operations to the new parallel actions. The distribution here, however, largely retains the original time budget, subtracting only the time needed for communicating and merging the result of each handler.

It is in Stages (4) and (5) that control actions may emerge. In the final Stage (6), we extract the behaviour of the mission cycle and handler releases. At the end of Stage (5), we have a recursion of parallel actions; the recursion captures a cyclic behaviour, and the parallelism can be directly traced to handlers in the program, and the actions *HandlerController* and *Cycle* mentioned above. In Stage (6), we distribute the recursion to the parallel actions to get models for the cyclic behaviour of the individual handlers. The distribution is justified by a law that introduces an extra internal channel *sync* that is used to ensure lock-step progress of the handlers after each cycle.

Many of the refinement laws to be used here are transformation laws that introduce parallelism. They are part of the basic *Circus* refinement strategy in [8,9]. The techniques presented in the setting of those works are, however, too general to allow for automation. In the case of the strategy that we propose, automation can be envisaged due to the restricted architecture of missions.

Specialised laws are needed in most stages, but they are all of general use across applications. The exceptions are the laws used in Stages (1) and (5), which embed particular design patterns. For the CD_x, for instance, as discussed in Section 4, we need, in particular, laws appropriate for a periodic mission and for handlers that execute in parallel several instances of a single handler

1. Encapsulate shared data of sequential handlers
2. Encapsulate shared data of concurrent handlers
3. Introduce data to realise control mechanisms
4. Collect specification of the memory area data

Fig. 10 Anchor E - Phase SH: Stages

class. For other designs, other laws are necessary. What we envisage is the development of a catalogue of laws that capture widely used designs.

3.3.4 Phase SH: sharing

In this phase, we model access to shared data and identify the atomic operations required to avoid data races. We distinguish between objects to be allocated in immortal, in mission, and in per-release or temporary private memory areas. The state components and local variables to be allocated in immortal memory remain or become state components. The variables to be allocated in mission memory become local to a new action *MArea*. Finally, those to be allocated in per-release and temporary memory areas remain or become local to the relevant handler actions.

This is performed in four stages identified in Figure 10. Stage (1) encapsulates data shared between handlers that execute in sequence. At this stage, a pair of (groups) handlers that execute in sequence and share data is combined in parallel, with some internal channel c used to (1) signal the end of one handler; (2) trigger the start of the other; and (3) communicate the shared data. We apply to each of these pairs a law that replaces the parametrised channel c by a new, typeless channel c, and two new *get* and *set* channels used to write and read the shared data. The resulting parallel actions still synchronise on the (fresh) c channel, but the data is now transferred between the handlers via the new *get* and *set* channels. The data sharing and control concerns are thus isolated, and whereas the synchronisations on the new channels are later refined to direct variable accesses, the synchronisation on c becomes a software event. A new parallel action encapsulates the shared data, and provides the get and set operations via synchronisations on the new channels. This new parallel action is incorporated in *MArea* in the final Stage (4).

Stage (2) addresses the introduction of shared data accessed by handlers concurrently. Unlike in Stage (1), this has to address synchronisation issues to avoid race conditions. Specialised laws that introduce data access at the right level of atomicity need to be applied in this stage. Precisely, this stage targets the refinement of the *HandlerController* action introduced in the previous phase; this is the action that defines the control and sharing of data between handlers executed concurrently. Specialised laws again introduce a new action to encapsulate the shared data, and provide (atomic) data operations via synchronisation on new fresh channels. Here, the specialised laws and strategy reflect design patterns for data sharing in a concurrent design.

Stage (3) is concerned with the introduction of additional shared data in order to realise control mechanisms. This shared data reflects particular control designs. Again, specialised laws can capture design patterns.

In the final Stage (4), we collect the shared data definitions to construct a new *MArea* action. It takes the shape of an interleaving of recursions; each interleaved action corresponds to a new data type that encapsules the variables declared locally and provides atomic operations for accessing the data. These are specified in a recursion that continuously offers synchronisations that correspond to calls to these operations.

The refinement in this stage is mostly a reordering of the parallel actions that declare local variables. After reordering, the parallelism between actions that provide methods of the same object is collapsed into a single parallel action since all their synchronisations correspond to methods in the same class: the mission class. Specialised laws are used to perform the merging of recursions where appropriate, and the refinement is guided by the program design. This stage is concluded by the introduction of the local action *MArea*.

The laws used in all stages are specialised, but most are useful across applications. It is in Stages (2) and (3) that we need laws that embed particular parallel designs. In the CD_x case study (see Section 4), for example, we use a law that supports the introduction of a barrier mechanism. As already mentioned, a catalogue of laws that capture widely used designs is needed.

3.3.5 Phase HR: handler refinement

In this final phase of the E anchor step, we mostly carry out algorithmic refinement using [12]. In addition, whenever shared variables are used, we need to explicitly introduce references and replace the organised access via channels with access via (synchronized) methods. This relies on the ability to relate values of variables to values stored in memory, as discussed in [16].

Any reference to logical methods is eliminated through algorithmic refinement. If extra classes are needed, the techniques in [9] can be used.

Any objects created during refinement in this phase should be assigned to local variables. This enforces their allocation in per-release memory, and avoids memory leaks (which happen when an object allocated in mission memory becomes unreachable because no variable in scope refers to it).

We can prove correct a program that leaks memory, because we do not distinguish programs on the basis of their use of resources. To avoid such designs, we propose the use of patterns. The allocation of local objects in per-release memory, as suggested, is such a pattern.

3.4 Anchor S: Safety-Critical Java

The S anchor is written using *SCJ-Circus*; it is based on *Circus*, *OhCircus*, and *Circus Time*, but includes several new constructs. Instead of paragraphs that define processes, we have paragraphs for the declaration of safelets, mission

sequencers, missions, and handlers. (We refer to Figures 29, 30, 32, and 33 for examples of paragraphs that define a mission sequencer, a mission, and handlers.) They highlight the main components of an SCJ program.

In the last step of our refinement strategy, the process that defines the E anchor is split to yield the definition of SCJ paragraphs that compose the S anchor. For example, the state components of the E anchor, if any, become state components of the **safelet** paragraph. The *Init* action gives rise to the definition of the safelet **setUp** paragraph. For statically defined sequences of missions, a simple **sequencer** paragraph (like that in Figure 29) is always adequate. Each *Mission* action gives rise to a **mission** paragraph, and so on.

The introduction of the new SCJ paragraphs in this last step is justified by their (new) refinement laws. They map the sequences and parallelisms of actions used in the E anchor to the new constructs representing the components of an SCJ program (handlers, missions, sequencers, and safelets).

The missions and handlers are already identified in the E anchor. What the transformations in this final step of the refinement strategy check is whether the use of memory is compatible with the SCJ paradigm.

For example, it is possible for development of the E anchor to introduce in the refinement of a *Handler* action, for instance, an assignment of a local variable to a state component. As previously explained, this would correspond to an assignment of an object in per-release memory to an object in immortal memory, which is forbidden in SCJ. Since refinement in *Circus* (and *Circus Time* and *OhCircus*) does not take into account the restrictions of the SCJ memory model, if the assignment achieves the specified functionality, it can be proved to be correct in the E anchor development. Such a refinement, however, does not correspond to a valid SCJ program. If adopted, the refinement in the final step of the strategy that generates the S anchor fails.

The actions and processes declared in the SCJ paragraphs are defined in terms of the P model described in [42] (see Figure 5). A safelet, for example, is described in the P model by the parallel composition of two *Circus* processes: one representing the SCJ framework and another the application class. The same applies to the other SCJ paragraphs. The paragraphs of an S anchor define the application components of these parallelisms.

3.5 Automation

In terms of automation of our technique, for the first O step, we have a challenge in the form of a data refinement. Results like those in [15] for Event-B and in [2] for Alloy can be helpful. They provide a route to automate the discovery of a retrieve relation between abstract and concrete states. This is the most challenging aspect of a data refinement proof.

For the second E step, a high degree of automation is possible. The removal of the structure of the O anchor in the CP phase can be automated (in a normalisation process). Afterwards, in Phases MS, HS and SH, automation can be achieved based on guidance provided by the definition of the parameters of

the design: the number of missions to be used, for each mission, the number of handlers, the external events and data operations under the responsibility of each handler, the allocation of the state components (in immortal or mission memory), and the time budgets and deadlines.

Besides, patterns of design are also necessary to guide the refinement in Stages (1) and (5) of Phase HS, and Stages (2) and (3) of Phase SH. These are stages that are related to two aspects of the design: timed (Stage (1) of Phase HS) and concurrent execution (all other stages just mentioned). It is for these stages that a catalogue of refinement laws catering for popular designs are envisaged. The wealth of literature and knowledge on patterns of design for timed concurrent systems [38,5,6] makes it feasible to propose that widely used architectures can be captured to provide a practical technique. A similar approach is adopted in [7] to define a technique for verification of implementations of Simulink diagrams; it is based on a widely adopted architectural pattern for safety-critical applications in avionics.

The final algorithmic refinement in Phase HR cannot, in general, be automated. For specialised models and programming architectures, however, the situation can be improved. We refer, for example, to the work in [1], which achieves a high level of automation in proofs of algorithmic refinement. This relies on the uniformity of the abstract models, which in the case of this work are generated automatically from Simulink diagrams.

Finally, the last S step can be automated, given the restricted structure of the E anchor. What must be noted, however, is that, as explained above, not every SCJ program matches the organised structure of an *SCJ-Circus* model. It is possible, for instance, to write an SCJ program using, for example, a single class to define the safelet and the mission sequencer. It remains to be investigated whether there is any practical advantage, perhaps in terms of use of resources, in pursuing such programs.

4 CD_x verification

The Anchor A for the CD_x is presented in Section 2.3. In this section, we apply our refinement strategy to derive a model of its implementation in *SCJ-Circus*.

4.1 Step to Anchor O

In the CD_x, the focus of the data refinement is the process *ABReqsCDx*, which, as already explained, has an abstract state with components *posns* and *motions* that need to be realised in a concrete data design. The process *ATReqsCDx* has no state and does not need to be changed. By data refining *ABReqsCDx*, we obtain a refined process, which we call *OBReqsCDx*. Compositionality of refinement guarantees that the new system specification given

Field	Type	Class	Purpose	Memory Area
`currentFrame`	`RawFrame`	`CDxMission`	data	mission memory
`state`	`StateTable`	`CDxMission`	data	mission memory
`work`	`Partition`	`CDxMission`	data	mission memory
`collisions`	`int`	`CDxMission`	data	mission memory
`control`	`DetectorControl`	`CDxMission`	control	mission memory

Table 1 Shared data objects of the concurrent CD_x

by the process *OCDx* below is a refinement of the process *CDx*.

$$\textbf{system}\ OCDx \mathrel{\widehat=} OBReqsCDx \llbracket \{\} \ldots \{\} \rrbracket ATReqsCDx$$

The parallel structure of the process *CDx* is preserved in *OCDx*, but we use the refined model of behavioural requirements given by *OBReqsCDx*, rather than the abstract *ABReqsCDx*. For conciseness, we omit the channel set used in the parallelism because it does not change. In the sequel, we explain how we construct *OBReqsCDx* by following each phase of our strategy in this step.

4.1.1 Phase CS

As already said, in this phase, we introduce the components of the concrete state that represent the information recorded in the abstract state. Table 1 lists the components of the concrete state of the CD_x. The variables *currentFame* and *state* represent the abstract *posns* and *motions*. Their types are the classes *RawFrame* and *StateTable*; they are defined in [43], where the complete refinement is presented, and reproduced in Appendices B.1 and B.2. We first of all introduce these class declarations. We can specify their fields and methods abstractly, and use algorithmic refinement to prove their implementations correct (in the last phase of the next step). Here, since this phase is not the focus of our strategy, we provide already very concrete models of these classes.

The schema that defines the state at the end of this phase is given below.

OCSStateCDx

$$posns_1 : Frame;\ motions_1 : Frame;$$
$$currentFrame_1 : RawFrame;\ state_1 : StateTable$$

$$\ldots$$

The variables $currentFrame_1$ and $state_1$ are new, whereas $posns_1$ and $motions_1$ correspond to the variables of the abstract specification. Figure 11 provides the retrieve relation that defines the abstract state *AStateCDx* of *ABReqsCDx* in terms of *OCSStateCDx*. When, like here, the retrieve relation is a function from the concrete to the abstract state, it is possible to calculate (automatically) the specification of the Z data operations in terms of the concrete state.

The predicate that defines the retrieve relation in Figure 11 first records that, as mentioned, $posns_1$ and $motions_1$ are just *posns* and *motions* themselves. Next, we assert that $currentFrame_1$ and $state_1$ are not **null**, and then

$$
\begin{array}{|l}
\hline
\mathit{OCSRetrCDx} \\
\quad \mathit{AStateCDx} \\
\quad \mathit{OCSStateCDx} \\
\hline
\mathit{posns}_1 = \mathit{posns} \land \mathit{motions}_1 = \mathit{motions} \\
\mathit{currentFrame}_1 \neq \mathbf{null} \land \mathit{state}_1 \neq \mathbf{null} \\
\mathit{posns} = \\
\quad \left(\lambda\, a : \mathit{Aircraft} \mid \mathit{currentFrame}_1.\mathit{find}(a) \neq -1 \bullet \right. \\
\qquad \left(\mathbf{let}\ i == \mathit{currentFrame}_1.\mathit{find}(a) \bullet \right. \\
\qquad\quad \left. \left. \mathit{MkVector} \left(\begin{array}{l} \mathit{currentFrame}_1.\mathit{positions}.\mathit{getA}(3 * i), \\ \mathit{currentFrame}_1.\mathit{positions}.\mathit{getA}(3 * i + 1), \\ \mathit{currentFrame}_1.\mathit{positions}.\mathit{getA}(3 * i + 2) \end{array} \right) \right) \right) \\
\mathit{motions} = \\
\quad \left(\lambda\, a : \mathit{Aircraft} \mid \mathit{currentFrame}_1.\mathit{find}(a) \neq -1 \bullet \right. \\
\qquad \left(\mathbf{let}\ \mathit{prev} == \mathit{state}_1.\mathit{position\_map}.\mathit{get}(\mathit{MkCallSign}(a)) \bullet \right. \\
\qquad\quad \mathbf{if}\ \mathit{prev} \neq \mathbf{null} \\
\qquad\quad \mathbf{then}\ \mathit{posns}(a) -_V \mathit{MkVector}(\mathit{prev}.x, \mathit{prev}.y, \mathit{prev}.z) \\
\qquad\quad \left. \left. \mathbf{else}\ \mathit{Zero}V \right) \right) \\
\hline
\end{array}
$$

Fig. 11 Anchor O - Phase CS: Retrieve relation for the first data refinement

we define *posns* and *motions* in terms of $\mathit{currentFrame}_1$ and state_1. In both cases, we use λ notation: $\lambda\, a : \mathit{Aircraft} \mid p \bullet e$ defines a function that associates an aircraft a that satisfies the predicate p to the value of the expression e.

The $\mathit{currentFrame}_1$ object of class *RawFrame* encodes aircraft positions in an array *positions* of coordinates. Arrays are objects with methods $\mathit{getA}(i)$ and $\mathit{setA}(i, v)$ to get and set (to v) the element at index i, and *length* to give its length. The x, y and z coordinates of an aircraft are stored in consecutive locations in *positions*. The class *RawFrame* establishes a mapping between aircraft identifiers (call signs) and their offsets in the array. A logical method $\mathit{find}(a : \mathit{Aircraft})$ returns the starting index of an aircraft a coordinates in *positions*, or -1 if no record for the aircraft exists. Logical methods are used in specifications, but do not need to be provided in an implementation. *MkVector* takes three coordinates and constructs a binding (record) of type *Vector*.

Motions are captured by the component state_1 of type *StateTable*. This class has a field *position_map* of a class type *HashMap*, which maps *CallSign* objects to *Vector*3*d* objects. *CallSign* encodes call signs to implement the abstract type *Aircraft*. The function *MkCallSign* converts *Aircraft* values into *CallSign* objects. For each aircraft a in *currentFrame* (that is, such that $\mathit{currentFrame}_1.\mathit{find}(a) \neq -1$), the *get* method of *HashMap* is used to get the motion vector *prev* of a in the *position_map* of state_1. If it is **null**, this is a new aircraft that entered the radar frame. Otherwise, the x, y, and z fields of *prev* are used to define a vector and calculate the new position.

The predicate that defines the concrete state invariant in *OCSStateCDx*, which is omitted above, as well as all Z data operations *Init*, *RecordFrame* and *CalcCollisions*, can be mechanically derived as described in [41].

OSDRetrCDx

$$
\begin{array}{l}
OCSStateCDx \\
OSDStateCDx \\
\hline
posns_2 = posns_1 \land motions_2 = motions_1 \\
currentFrame_2 = currentFrame_1 \land state_2 = state_1 \\
\exists\, collset : \mathbb{F}\,(Aircraft \times Aircraft) \mid \\
\quad collset = CalcCollisionSet\,(posns_2, motions_2) \bullet \\
\qquad \left(\begin{array}{l} (\#\, collset = 0 \land collisions_2 = 0) \lor \\ (\#\, collset > 0 \land collisions_2 \geq (\#\, collset) \operatorname{div} 2) \end{array} \right)
\end{array}
$$

Fig. 12 Anchor O - Phase SD: Retrieve relation for the second data refinement

4.1.2 Phase SD

In this phase, we introduce in the concrete state the variables that are to hold shared data. In our example, a shared variable `work` holds the partitioned lists of aircraft in each voxel, which are distributed between the detector handlers. Additionally, the shared integer `collisions` collects the detection result.

It can be useful to carry out an incremental data refinement and introduce a single variable at a time to leverage the proof effort. In our case study, we proceed in this way. So, we first introduce *collisions*, and then *work*, whose class type *Partition* is defined in Appendix B.3.

The state retains all existing components (subscripted for disambiguation). To introduce *collisions* we use the retrieve relation in Figure 12. The only significant constraint is on the new component $collisions_2$. Its specification resembles that of the *CalcCollisions* operation. The possibility of referencing the abstract state components $posns_2$ and $motions_2$ allows for this concise formulation of the retrieve relation. The retrieve relation for *work* is in [43].

4.1.3 Phase EL

To construct the final model, we have to eliminate from the state the abstract variables; in our example, those corresponding to *posns* and *motions*. The resulting process *OBReqsCDx* is sketched in Figure 13. The definitions of the data operations are those obtained after (significant) simplification of the predicates obtained by calculation based on the (functional) retrieve relations.

In the state invariant and in *RecordFrame* the abstract variables *posns* and *motions* are existentially quantified via local (**let**) definitions. For conciseness, we use F_1 and F_2 to denote the abstraction functions used in the retrieve relation *OCSRetrCDx* to define *posns* and *motions* (see Figure 11).

To define *work*, we use an existentially quantified variable *voxel_map* of type $HashMap[Vector2d, List[Motion]]$. This is a generic class parametrised by a key and a value type, here instantiated as objects that represent a pair (or a two-dimensional vector) and a list of motions. $Vector2d$, *List* and *Motion* are also *OhCircus* classes [43]. The existential quantification in the invariant specifies a fundamental correctness property of the voxel hashing: if two recorded

Fig. 13 Anchor O of the CD_x: behavioural requirements

aircraft collide, there is at least one voxel in *voxel_map* that contains both of them. This part of the invariant is omitted in Figure 13.

The method call *voxel_map.values.elems* gives all voxels in *voxel_map*. (The *values* method has the same meaning as the respective Java method of `HashMap`, and the logical method *elems* converts the resulting Java `List` into

$$
\begin{array}{l}
\textbf{system}\ ECPCDx \mathrel{\widehat=} \textbf{begin} \\
\ldots \\
StartCycle \mathrel{\widehat=} (next\_frame?frame \mathbin{@} t_1 \longrightarrow (RecordFrame \mathbin{;} CalcStep(t_1))) \blacktriangleleft INP\_DL \\
CalcStep(t_1) \mathrel{\widehat=} \textbf{wait}\ w : 0 \mathinner{..} (FRAME\_PERIOD - t_1 - OUT\_DL) \bullet \\
\quad \textbf{var}\ colls : \mathbb{N} \bullet CalcCollisions \mathbin{;}\ OutputStep(t_1, w, colls) \\
OutputStep(t_1, w, colls) \mathrel{\widehat=} \\
\quad \Big(\Big(output\_collisions!colls \mathbin{@} t_2 \longrightarrow \\
\qquad \textbf{wait}\ FRAME\_PERIOD - (t_1 + w + t_2) \Big) \blacktriangleleft OUT\_DL \Big) \mathbin{;}\ StartCycle \\
\bullet\ Init \mathbin{;}\ StartCycle \\
\textbf{end}
\end{array}
$$

Fig. 14 Anchor E - Phase CP: Result

a set.) The purpose of *work* is to partition *voxel_map.values.elems*. The last equality in the state invariant in Figure 13 establishes that the voxel list of *Motion* objects for detector i is obtained by $getDetectorWork(i)$. This does not force the partitions to be disjoint (which is not essential for correctness).

We note that the state components of the O anchor still contain object values rather than references, and so is not concerned with memory allocation.

4.2 Step to Anchor E

We describe here the application of each of the phases of the step that generates an E anchor in our case study. The starting point is the *OCDx* process, which, as explained in Section 4.1, is defined by the parallel composition of *OBReqsCDx* (in Figure 13) and *ATReqsCDx* (in Figure 4).

4.2.1 Phase CP

The process *ECPCDx* obtained in this phase is sketched in Figure 14; its state and data operations are similar to those of *OBReqsCDx* and omitted. Its behavioural and timing restrictions are exactly those of *OCDx* (and *CDx*) as defined in *ATReqsCDx*. It, however, combines in a single centralised specification the data operations, interactions, and time budgets and deadlines. The specification of *ECPCDx* is in a rather direct correspondence with an underlying state machine, and can be convoluted. This is not a problem, since this specification can, as mentioned in Section 3.5, be calculated automatically.

In more detail, the main action of *ECPCDx* (see Figure 14) defines that after the initialisation (*Init*), the behaviour of *ECPCDx* is given by *StartCycle*, which waits for the next frame to be communicated on *next_frame*. This has a deadline *INP_DL* as in the timing requirements in *ATReqsCDx*.

The subsequent data operation *RecordFrame* records the frame in the state. Next, a call to *CalcStep* takes as parameter the elapsed time t before the synchronisation on *next_frame* takes place. After a nondeterministically chosen

wait period w that captures the time needed for the computation, it (instantaneously) calculates and then passes as parameters to *OutputStep* the number *colls* of collisions, along with t and w. We use a construct **wait** $w : t_1 \,..\, t_2$, which introduces a variable w to record the nondeterministic choice of the waiting period (between t_1 and t_2). In this case, the wait period cannot be longer than the frame period *FRAME_PERIOD*, minus t_1, minus the maximum time *OUT_DL* that can be taken for an output. This reflects the time budget originally defined in *ATReqsCDx*.

OutputStep deals with outputting the collisions result via *output_collisions*. The time that has elapsed from the beginning of the cycle is given by $t_1 + w$. After imposing the respective deadline for the output, the final **wait** delays the recursion that initiates the next cycle until the cycle period has elapsed, considering that a further t_2 time units have been used in the output.

This flat structure is the basis for the introduction of the missions and handlers architecture, which is tackled next.

4.2.2 Phase MS

In our example, we have no variables in immortal memory, so no initialisation, and we obtain a call to *System* as the main action.

Since we have a single mission, we require no transformation to *StartCycle*. We introduce the *System* and mission actions below.

$$MisssionCDx \mathrel{\widehat{=}} Init;\ StartCycle$$
$$System \mathrel{\widehat{=}} MissionCDx$$

With these, we can define the main action to be *System* as required.

4.2.3 Phase HS

The definition for *StartCycle* that we obtain in this phase is presented in Figure 15. The parallel actions in *Handlers* correspond to the handlers; we have actions for `InputFrameHandler`, `ReducerHandler`, the four instances of `DetectorHandler`, and `OutputCollisionsHandler`. We introduce (again) a cyclic behaviour whose period is given by *FRAME_PERIOD*, but now with the deadlines and time budgets associated explicitly with the interactions and data operations, rather than separately via parallelism. In general, a cyclic behaviour may or may not be already explicitly specified in the abstract A anchor, and even if it is, it may need to be decomposed in an implementation.

In the sequel, we discuss the individual stages of this phase.

Stage (1) The action *StartCycle* (in Figure 14) is reproduced in Figure 16 with all action calls replaced by the actions themselves. There is no effort in calculating this definition of *StartCycle*; it is just the result of the previous phase, with some definitions expanded to make our discussions clearer.

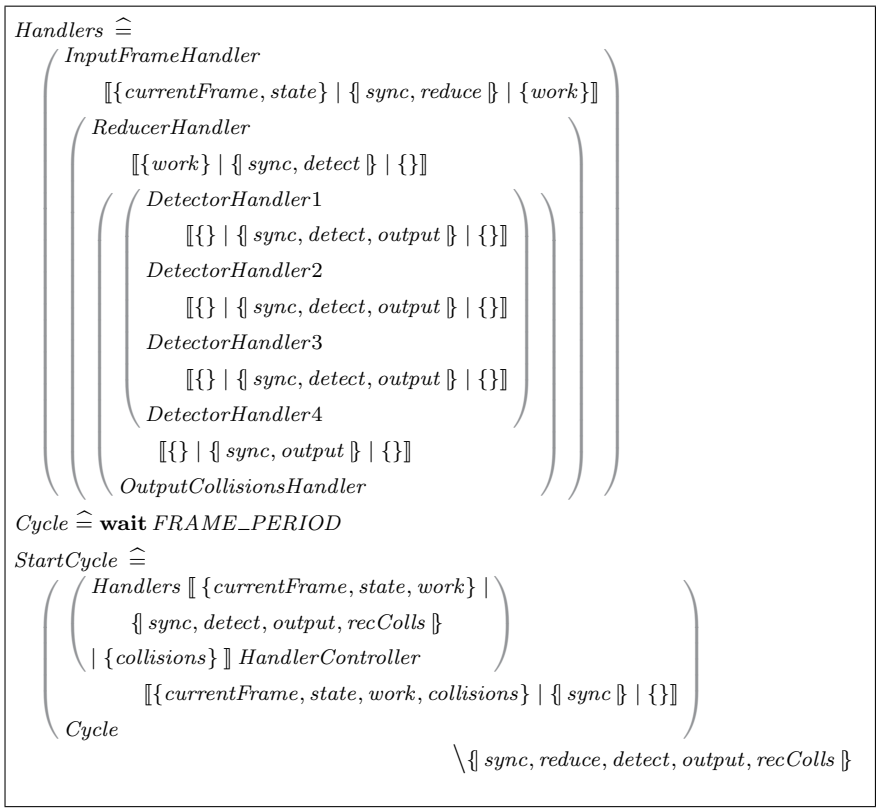

Fig. 15 Anchor E - Phase HS: Result

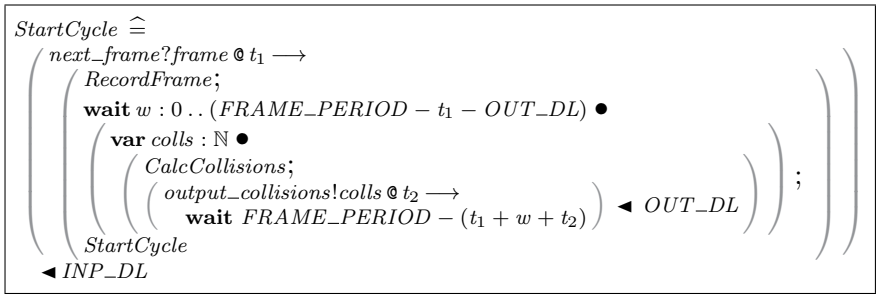

Fig. 16 Anchor E - Phase HS: *StartCycle* at the end of phase CP

Two modelling patterns, and associated refinement strategies, are used in this stage, whose result is in Figure 17. First, the mission is cyclic (with period *FRAME_PERIOD*), so we use an interleaving with a **wait** to specify the cycle. Second, in each cycle, we have an input (*next_frame*?*frame*) with a deadline (*INP_DL*), a calculation (specified by *RecordFrame* and *CalcCollisions*),

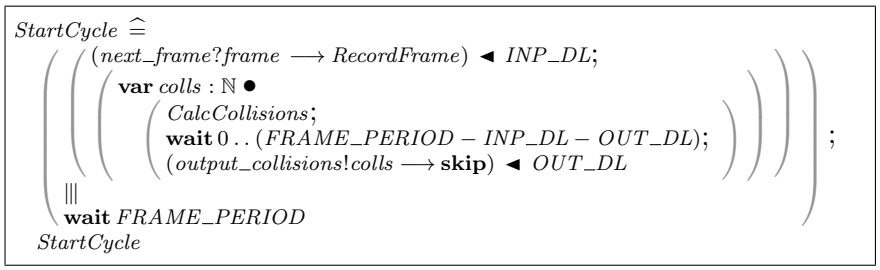

Fig. 17 Anchor E - Phase HS: *StartCycle* at the end of the Stage (1)

with a time budget ($FRAME\_PERIOD - INP\_DL - OUT\_DL$), and an output communication ($output\_collisions!colls$) also with a deadline (OUT_DL).

Here, refinement reduces the nondeterminism in the original definition of the budget for the calculations in *StartCycle*. In that definition (see Figure 16), it depends on the value of the variables t_1 and t_2, which record the time taken to input and output. In the design pattern that we consider, the budget is fixed to $FRAME\_PERIOD - INP\_DL - OUT\_DL$, which considers the greatest values that t_1 and t_2 can take: the deadlines INP_DL and OUT_DL. We, therefore, rule out implementations that can take advantage of any time saved in the input to take more time in the computation. Due to the unpredictability of the environment this gain in computation time can practically never be exploited.

As already explained, in this stage, we also localise the deadlines on the communications, and the budget on the calculations. Specifically, we isolate the communication on *next_frame* from the variable block and distribute the deadline INP_DL through the sequence. A specific parallelisation law introduces the interleaving that defines the cycle period.

Stage (2) Only *RecordFrame* requires decomposition, since it is jointly implemented by the `InputFrameHandler`, the `ReducerHandler`, and the (instances of) `DetectorHandler`. *CalcCollisions* just provides the result of the detection stored in the state; it is implemented in the `OutputCollisionsHandler`.

We first decompose *RecordFrame* into three sequential actions: *StoreFrame*, *ReducerAndPartitionWork*, and *DetectCollisions*. *StoreFrame* is implemented in the `InputFrameHandler`, and *ReducerAndPartitionWork* is implemented in the `ReducerHandler`. The implementation of *DetectCollisions* is distributed through the instances of `DetectorHandler`, and so in this stage we subsequently decompose *DetectCollisions* into parallel data operations.

The result of the decomposition of *RecordFrame* into a sequence shown in Figure 18. *StoreFrame* updates the state; it takes as input a new frame *frame*? of aircraft positions, which it records, and retains the previous frame of positions in *state* for computation of aircraft motions. *ReduceAndPartitionWork* is concerned with the voxel-hashing as well as the distribution to the detection handlers of voxel lists to be checked. It computes a new value for *work*.

The third operation *DetectCollisions* utilises the information in *work*. It is implemented concurrently by each of the detector handlers, so it is further

$$
\begin{array}{l}
RecordFrame(frame?) \mathrel{\widehat=} \\
\quad StoreFrame(frame?) \,;\; ReduceAndPartitionWork \,;\; DetectCollisions \\
DetectCollisions \mathrel{\widehat=} \\
\quad \left(\begin{array}{l}
\textbf{var}\ colls1, colls2, colls3, colls4 : \mathbb{Z} \bullet \\
\quad \left(\begin{array}{l}
(\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls1/pcolls!] \wedge i? = 1) \wedge \\
(\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls2/pcolls!] \wedge i? = 2) \wedge \\
(\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls3/pcolls!] \wedge i? = 3) \wedge \\
(\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls4/pcolls!] \wedge i? = 4)
\end{array} \right) ; \\
\quad SetCollisionsFromParts([\![\, colls1, colls2, colls3, colls4 \,]\!])
\end{array} \right)
\end{array}
$$

Fig. 18 Anchor E - Phase HS: *RecordFrame* at the end of the Stage (2)

decomposed into a conjunction of data operations also shown in Figure 18. The data operation *CalcPartCollisions* has an input $i?$; it defines the collisions *pcolls*! in the $i?$-th voxel. Above, each of the four uses of this operation assigns its partial result to a local variable *colls*1, *colls*2, *colls*3 or *colls*4. The operation *SetCollisionsFromParts* used in sequence merges those results by assigning a new value to *collisions*. It takes the bag of the local variables as a parameter, and gives as output the sum of their values.

Stage (3) In *StartCycle* (as shown in Figure 17), as already explained, the **wait** defines a period for the cycles; it is part of our pattern for modelling periodic tasks. On the other hand, the first **wait** captures a time budget for computations: *StoreFrame*, *ReduceAndPartitionWork*, *DetectCollisions* and *CalcCollisions*. In this stage, we distribute this budget to their sequential components: *CalcCollisions* and the newly introduced sequential components of *RecordFrame*, that is, *StoreFrame*, *ReduceAndPartitionWork*, and the conjunction and *SetCollisionsFromParts* in the definition of *DetectCollisions*.

The result is sketched in Figure 19. The constants SF_{TB}, RPW_{TB}, CPC_{TB}, SC_{TB}, and CC_{TB}, define the time budget for each data operation. Their sum is $FRAME\_PERIOD - INP\_DL - OUT\_DL$, which was the original overall budget of the calculation. In Figure 19, we partially expand the definition of *DetectCollisions* to show how the budget is associated with its sequential components. It is not distributed, however, to the parallel components in the conjunction: the *CalcPartCollisions* operations (omitted in Figure 19).

For clarity, in the sequel we use named local actions to denote the timed data operations. For instance, we define the timed specification of the data operation that stores a frame as follows.

$$StoreFrameT(frame) \mathrel{\widehat=} \mathbf{wait}\, 0 \,..\, SF_{TB} \,;\; StoreFrame(frame)$$

Similarly, we have *ReduceAndPartitionWorkT* and *CalcCollisionsT*. We name *DetectCollisionsT* the body of the first variable block in Figure 19, which uses *CalcPartCollisions* and *SetCollisionsFromParts*.

Stage (4) Some of our new timed sequential components are assigned for execution in different handlers or groups of handlers. Namely, *StoreFrameT*,

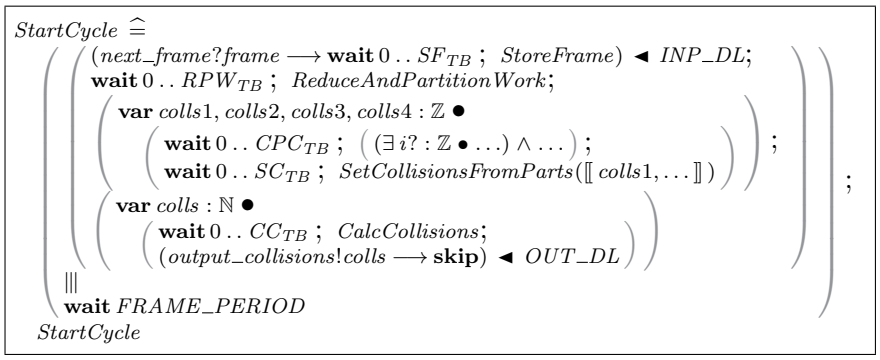

Fig. 19 Anchor E - Phase HS: *StartCycle* at the end of the Stage (3)

ReduceAndPartitionWorkT, and *CalcCollisionsT* are executed in individual separate handlers. Additionally, *DetectCollisionsT* embeds a concurrent design implemented in a (separate) group of handlers. We now introduce a parallelism of actions that correspond to these handlers and groups of handlers.

The result is in Figure 20. We have four sequential actions that are now executed concurrently, with new internal channels *reduce*, *detect* and *output* enforcing sequential execution. When *StoreFrameT* finishes, a communication on *reduce* starts *ReduceAndPartitionWorkT*. In addition, this communication outputs the values of *currentFrame* and *state* to *ReduceAndPartitionWorkT*. Similarly, the new internal channel *detect* is used to order the executions of *ReduceAndPartitionWorkT* and *DetectCollisionsT*, and share the value of *work*. Finally, a synchronisation on *output* signals the end of *DetectCollisionsT* and triggers the start of the output of the collisions.

Stage (5) We now tackle the action below from Figure 20.

$$detect?work \longrightarrow \\ \quad \mathbf{var}\ colls1, \ldots \bullet DetectCollisionsT\ ;\ output!collisions \longrightarrow \mathbf{skip}$$

It is refined to that shown in Figure 21. As explained in Section 2.2, we have four instances of a handler class `DetectorHandler`, whose calculations detect collisions as specified by *CalcPartCollisions* are merged as defined by *SetCollisionsFromParts*. We, therefore, use a law that introduces a parallelism between each of the calls to *CalcPartCollisions* and an extra action *HandlerController*, which embeds the behaviour of *SetCollisionsFromParts* and ultimately controls the execution of the concurrent handlers.

Each of the new parallel actions in Figure 21 synchronises on the original *detect* and *output* channels, so synchrony with the other actions is maintained. In addition, since there is still an element of sequentiality in *DetectCollisionsT*, we introduce another channel, *recColls*, to control the flow between the concurrent executions of *CalcPartCollisions* in *DetectorHandler*1, *DetectorHandler*2, *DetectorHandler*3, and *DetectorHandler*4, and the subsequent execution of *SetCollisionsFromParts* in *HandlerController*. Additionally, it communicates

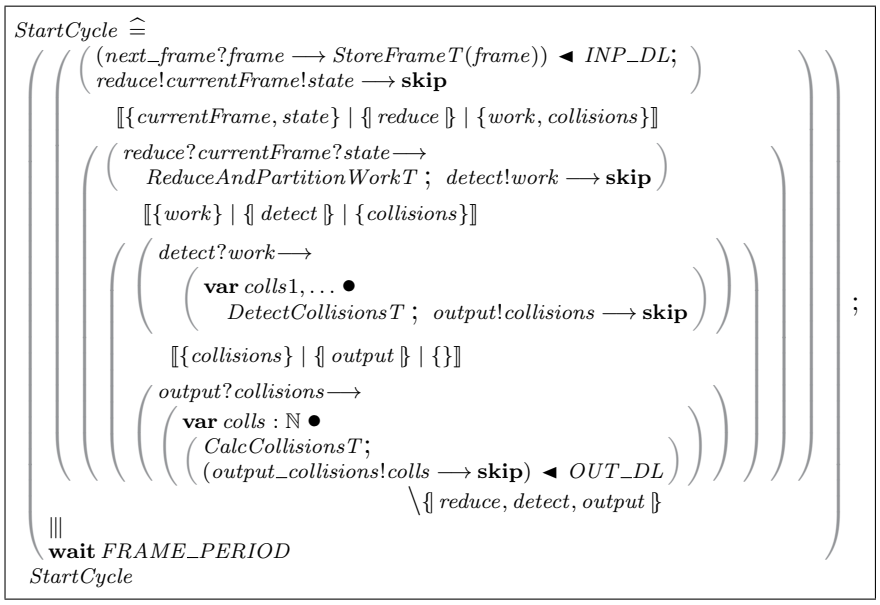

Fig. 20 Anchor E - Phase HS: *StartCycle* at the end of the Stage (4)

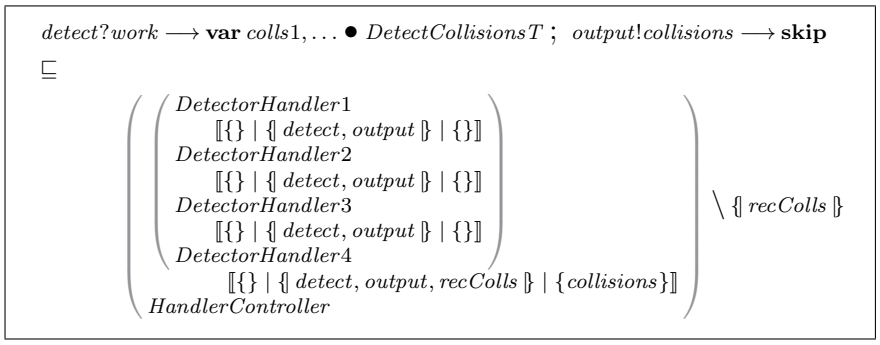

Fig. 21 Anchor E - Phase MH: Refinement in Stage (5)

the result obtained by each detection handler. In a later phase, *recColls* is refined to a call to a method that records the results of the detector handlers.

For example, the definition of *DetectorHandler*1 is as shown in Figure 22. The time budget CPC_{TB} for the conjunction in *DetectCollisionsT* is now available to each of the concurrent handlers that implement each of the operations. This relies on the fact that if a **wait** precedes a parallelism, then the time budget that it embeds can be passed on to all parallel actions. The final communication *output*?y ensures synchronisation at completion of each of the detector handlers, and triggers the start of the *OutputCollisionsHandler*. The actual value output is defined by *HandlerController*; in the detector handlers, *output*?y is used to indicate that any value y is acceptable.

$$
\begin{array}{l}
DetectorHandler1 \mathrel{\widehat=} detect?work \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{var}\ colls1 : \mathbb{Z} \bullet \\
\quad \left(\begin{array}{l}
\mathbf{wait}\ 0\,..\,CPC_{TB}; \\
(\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls1/pcolls!] \land i? = 1); \\
recColls!colls1 \longrightarrow \mathbf{skip}
\end{array} \right) ; \\
output?y \longrightarrow \mathbf{skip}
\end{array} \right)
\end{array}
$$

Fig. 22 Anchor E - Phase MH: *DetectorHandler*1 in Stage (5)

$$
\begin{array}{l}
HandlerController \mathrel{\widehat=} detect?work \longrightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\mathbf{var}\ colls1, colls2, colls3, colls4 : \mathbb{Z} \bullet \\
\quad \left(\begin{array}{l}
(recColls?x \longrightarrow \mathbf{wait}\ 0\,..\,RC_{TB}\ ;\ \ colls1 := x)\,; \\
(recColls?x \longrightarrow \mathbf{wait}\ 0\,..\,RC_{TB}\ ;\ \ colls2 := x)\,; \\
(recColls?x \longrightarrow \mathbf{wait}\ 0\,..\,RC_{TB}\ ;\ \ colls3 := x)\,; \\
(recColls?x \longrightarrow \mathbf{wait}\ 0\,..\,RC_{TB}\ ;\ \ colls4 := x)\,; \\
\mathbf{wait}\ 0\,..\,SCFP_{TB}\ ;\ \ SetCollisionsFromParts([\![\, colls1, \ldots]\!])
\end{array} \right)
\end{array} \right) ; \\
output!collisions \longrightarrow \mathbf{skip}
\end{array} \right)
\end{array}
$$

Fig. 23 Anchor E - Phase MH: *HandlerController* in Stage (5)

HandlerController is defined in Figure 23. It collects the results of the detection handlers via synchronisations on the *recColls* channel, records them in local variables, and then uses *SetCollisionsFromParts* to merge them and assign the result to *collisions*. The time budget SC_{TB} of *SetCollisionsFromParts* is now distributed, and $RC_{TB} \times 4 + SCFP_{TB}$ is equal to SC_{TB}.

A significant generalisation of our refinement strategy in this phase (in comparison to our previous work) follows from the handlers requiring synchronisation. As shown above, the parallelism of handlers is not merely an interleaving as in [11], but identifies synchronisation channels. This is to ensure that handlers are released and executed in the correct order. At the program level, this is achieved by software events and synchronised method calls.

Stage (6) This last stage eliminates the top-level recursion that defines the mission cycle by distributing it to the parallel actions in its body; Figure 20 shows the recursion. The result is a top-level parallelism of actions that can be directly traced to handlers in the program; it is shown in Figure 15.

4.2.4 Phase SH

In what follows, we discuss the refinements in each stage of SH.

Stage (1) In the CD_x, this first stage targets the parallelisms between the actions *InputFrameHandler* and *ReducerHandler*, and between *ReducerHandler* and the detector handler actions in Figure 15. These are the pairs of (groups of) handlers that are executed in sequence.

In the first parallelism, the values of *currentFrame* and *state* are communicated through the *reduce* channel. Refinement replaces it with a new typeless

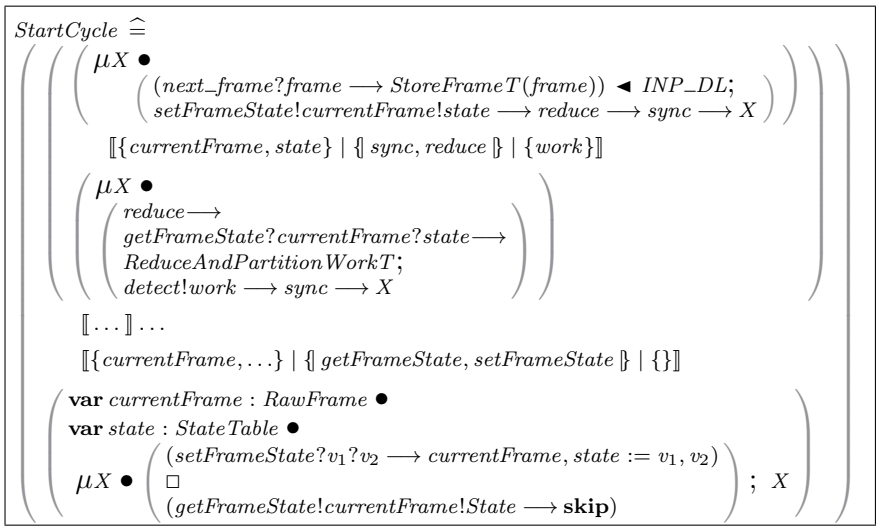

Fig. 24 Anchor E - Phase SH: Refinement of the detector handlers in Stage (1)

channel *reduce*, and two new channels, *getFrameState* and *setFrameState* used to write and read *currentFrame* and *state*. The result of refining the parallelism between *InputFrameHandler* and *ReducerHandler* is given in Figure 24. These actions still synchronise on the (fresh) *reduce* channel, but the data is now transferred between them via *getFrameState* and *setFrameState*. A new parallel action encapsulates *currentFrame* and *state*, and provides the get and set operations via synchronisations on these channels.

To give an example of a specialised refinement law used in our strategy, we present in Figure 25 Law seq-share-1, which captures the refinements to be carried out in this stage. It can be used to justify the transformation just described, for instance. This law applies to parallelisms of recursions of a particular form. In both recursions, the end of an iteration is marked by a synchronisation on a channel *end*. In addition, a channel c is used by one of the recursions to send the value of an expression e at the end of its iteration. In the other recursion, that value is read at the beginning of its iteration. The first proviso guarantees that the recursions do agree on communications over c and synchronisations on *end*, and the second that c is only used where explicitly shown. Therefore, in spite of being in parallel, each iteration of one recursion occurs before a corresponding iteration of the other recursion starts. In our example, the channel *end* matches *sync*, and the channel c matches *reduce*. The expression e is actually the values of *currentFrame* and *state*. (We have two expressions rather than one, which is a trivial generalisation.)

With an application of Law seq-share-1, we obtain a different parallelism. The original parallel actions are transformed to use channels c_1, c_2, and c_3, guaranteed to be fresh by a proviso. Synchronisation on c is replaced with synchronisation on c_3, output of e is carried out via c_1, and input via c_2. A

Law seq-share-1

$$
\left(\begin{array}{l}(\mu X \bullet A_1 \,;\; c\,!\,e \longrightarrow end \longrightarrow X)\\ \quad [\![ns_1 \mid cs \mid ns_2]\!]\\ (\mu X \bullet c\,?\,x \longrightarrow A_2 \,;\; end \longrightarrow X)\end{array}\right) \setminus \{\!| c |\!\}
$$

$=$

$$
\left(\begin{array}{l}\left(\begin{array}{l}(\mu X \bullet A_1 \,;\; c_1\,!\,e \longrightarrow c_3 \longrightarrow end \longrightarrow X)\\ \quad [\![ns_1 \mid (cs \setminus \{\!| c |\!\}) \cup \{\!| c_3 |\!\} \mid ns_2]\!]\\ (\mu X \bullet c_3 \longrightarrow c_2\,?\,x \longrightarrow A_2 \,;\; end \longrightarrow X)\end{array}\right) \setminus \{\!| c_3 |\!\}\\ \quad [\![ns_1 \cup ns_2 \mid \{\!| c_1, c_2 |\!\} \mid \varnothing]\!]\\ \left(\begin{array}{l}\mathbf{var}\ v : T \bullet \mu X \bullet (c_1\,?\,x \longrightarrow v := x \,\Box\, c_2\,!\,v \longrightarrow \mathbf{skip}) \,;\; X\end{array}\right)\end{array}\right) \setminus \{\!| c_1, c_2 |\!\}
$$

provided $\{\!| c, end |\!\} \subseteq cs$;
$c \notin usedC(A_1) \cup usedC(A_2)$; and
c_1, c_2 and c_3 are fresh channels.

Fig. 25 Law seq-share-1 for refinement of detector handlers in Stage (1)

new parallel action reads and keeps the value of e in a local variable x, and outputs it. In our example, c_3 is taken as a new typeless channel *reduce*, c_1 and c_2 are *setFrameState* and *getFrameState*. In addition, we have a pair of local variables *currentFrame* and *state*. A proof of Law seq-share-1 is in [43].

In a similar way we refine the parallelism between the new *ReducerHandler* action and the *DetectorHandler* actions. This uses a slightly generalised version of the sharing law, since not merely one, but four handlers concurrently synchronise on the channel through which the data is passed.

Stage (2) The definition of *HandlerController* is omitted in Figure 24, but remains as shown in Figure 23. We recall that this is an action that controls the interaction of the concurrent handlers, rather than a model of a handler itself. We use here another specialised law to obtain the result in Figure 26. As in the previous stage, in a parallel action we now encapsulate the shared *collisions* variable while identifying the atomic operations to access it, triggered by communication on the channels *initColls*, *recColls* and *output*.

In this refinement, we rely on the fact that *SetCollisionsFromParts* takes a bag of the values communicated through *recColls* as parameters. So, the sequence of communications in *HandlerController* is refined to an interleaving. The processing of the communicated values is isolated in the variable block that encapsulates *collisions* (which later becomes part of *MArea*).

The parallel action for control in Figure 26 that is still left is decomposed and collapsed with handler actions. Whereas in HS we admit residual control actions in the final model, in SH our aim is to eliminate them. This is carried out by specialised laws that refine the parallelism between the control action and each of the handlers under control. In our example, these are *ReducerHandler*, the subsequent *DetectorHandler* parallel actions, and the final *OutputCollisionsHandler*. The synchronisation on *initColls* moves

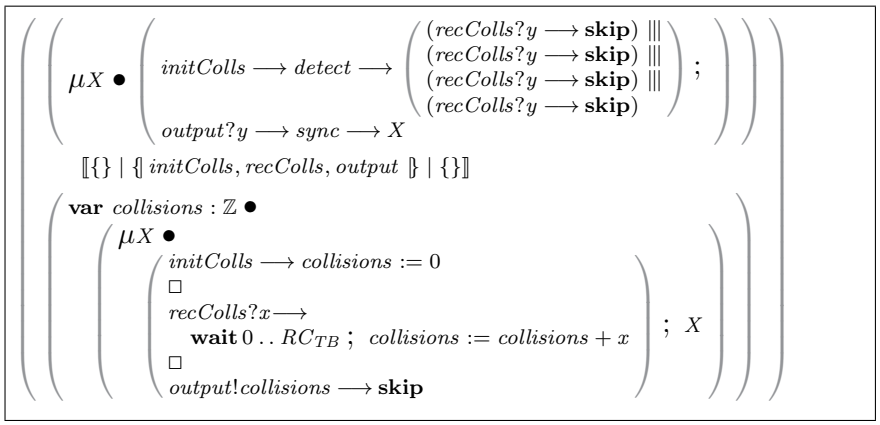

Fig. 26 Anchor E - Phase SH: Refinement of *HandlerController* in Stage (2)

to *ReducerHandler*. The following communications on *recColls* move to the *DetectorHandler* actions. Finally, since communication on *output* is used both for control and data communication, it is replaced with synchronisations on two channels like in Stage (1). A first communication on a new synchronisation channel *output* triggers the *OutputCollisionsHandler*, which then takes the value of *collisions* using a new channel *getColls*.

At the end of this stage, the handler actions in Figure 24 are nearly unchanged, except for the additional synchronisations to get and set data, and to trigger data operations. For example, the actions that model the detector handlers are similar. For the first handler, for instance, it is as follows.

$$\left(\mu X \bullet \left(\begin{array}{l} detect \longrightarrow \\ getWork?work \longrightarrow \mathbf{var}\ colls1 : \mathbb{Z} \bullet \ldots ; \\ recColls!colls1 \longrightarrow output?y \longrightarrow sync \longrightarrow X \end{array}\right)\right)$$

When triggered by *detect*, it gets *work* using a synchronisation, carries out its original calculation (omitted above), shares its result *colls*1 via *recColls*, and triggers *OutputCollistionsHandler* via *output* before synchronising on *sync*.

Stage (3) Further refinement is carried out in this stage to implement particular control mechanisms possibly embedded in the parallelism of handlers using SCJ constructs. In our case study, we have the barrier-like synchronisations on *output* performed between the detector handlers, which originates from the their parallelisation in Stage (5) of Phase MH.

Software events allow a handler to fire several other handlers at the same time. On the other hand, if we have a situation in which several handlers have to agree to fire another handler, we need a protocol. In *Circus*, both scenarios can be modelled using multi-synchronisation between handlers: using a single channel in a parallelism that requires all relevant handlers to synchronise. In a refinement to an implementation, different approaches are required.

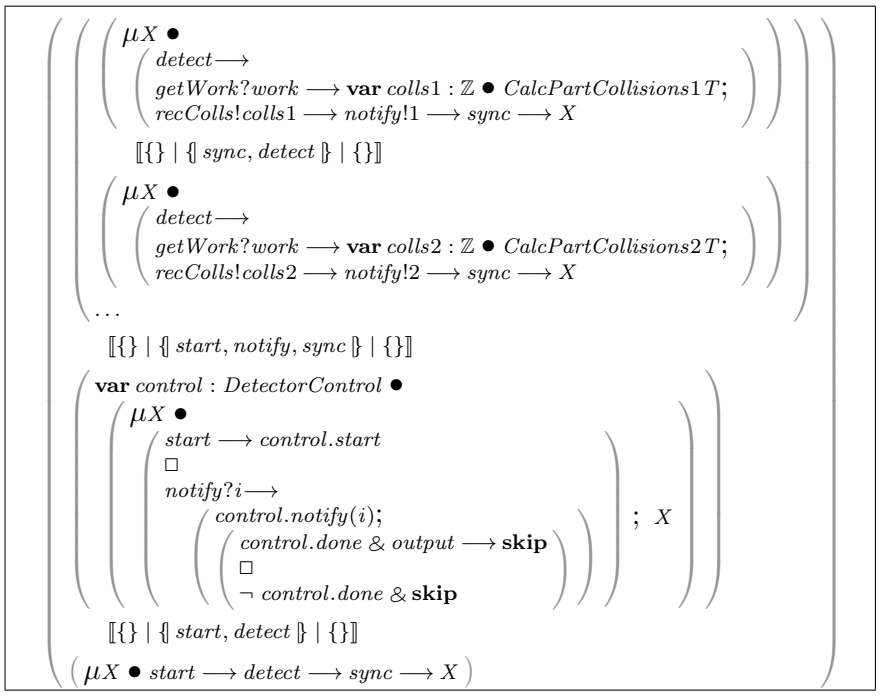

Fig. 27 Anchor E - Phase SH: Refinement of the detector handlers in Stage (3)

In our example, the internal channels *detect* and *output* are both used for multi-synchronisation (see Figure 15). To implement the multi-synchronisation on *detect*, we can use a software event to be used by the `ReducerHandler` to fire the detector handlers. As already discussed, this is tackled in Stage (1) above. On the other hand, the multi-synchronisation on *output* requires a barrier protocol, so that termination of all detection handlers lead to the release of the `OutputCollisionsHandler`. This is tackled here in Stage (3).

The result of the refinement is sketched in Figure 27 and again arises from the use of a specialised law; it is applied to the parallelism of the four detector handlers. Since there is no barrier mechanism in SCJ, the strategy embedded in the law is the use of a new control variable *control* that records the handlers still in execution. Synchronisations on *output* are replaced by the interleaved outputs on *notify*. They correspond to method calls to signal termination of each detector. The synchronisation on *output* is now carried out by the new control parallel action, and thus managed by the shared *control* object. It is not a multi-sychronisation anymore, and so models a software event now.

The channel *start* corresponds to a method call that resets the value of *control* to capture that none of the handlers has terminated yet. Just like in the previous stage, the new control action that enforces that *start* occurs before *detect* is distributed to the handlers. In this case, *start* is moved to *ReduceHandler* (and *detect* is already in the *DetectorHandler* actions).

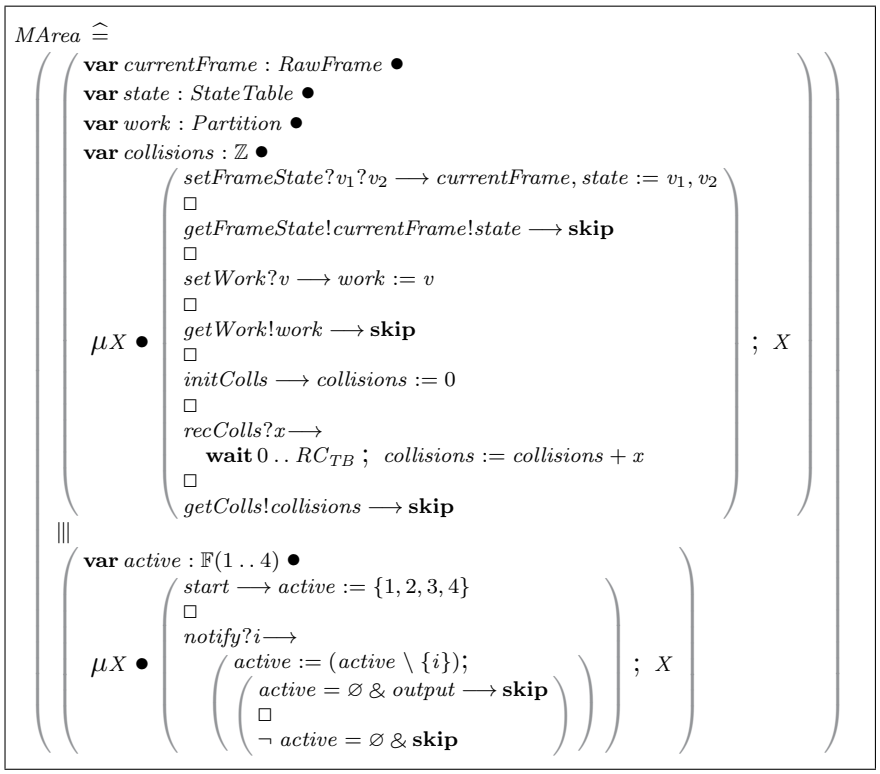

Fig. 28 Anchor E - Phase SH: *MArea*

There are different ways to achieve the barrier control of execution, hence the elimination of the synchronisation on *output* involves design. We can, however, provide refinement laws that embed a variety solutions.

Stage (4) The refinement in this stage is a reordering of the parallel actions that declare local variables; in our example, the variables are *currentFrame* and *state* (in Figure 24), *work*, which is also encapsulated in a parallel action (omitted here) in Stage (1), *collisions* (in Figure 26), and *control* (in Figure 27). They are brought together to specify *MArea* shown in Figure 28.

After reordering, the parallel actions that provide methods of the same object are collapsed. For instance, the parallel action that declares *currentFrame* and *state* synchronises on the channels *setFrameState* and *getFrameState*, the action for *work* synchronises on *setWork* and *getWork*, and that for *collisions* synchronises on *initColls*, *recColls* and *getColls*. They are collapsed into a single parallel action since all their synchronisations correspond to methods in the same class: the mission class. On the other hand, the action that models interaction with *control* is retained because in the program this is realised by a separate object (of the class `DetectorControl`).

sequencer *MainMissionSequencer* $\widehat{=}$ **begin**

state *MainMissionSequencerState* == [*mission_done* : *bool*]

initial $\widehat{=}$ *mission_done* := *false*

getNextMission $\widehat{=}$
  if *mission_done* = *false* $\longrightarrow$ *mission_done* := *true*; **ret** := *MissionCDx*
  [] *mission_done* = *true* $\longrightarrow$ **ret** := *null*
  fi

end

Fig. 29 Anchor S: Mission sequencer paragraph

4.2.5 Phase HR

In this phase, in principle we refine to code all remaining data operations. It is not the focus of this work or paper, so we do not discuss it any further.

4.3 Step to Anchor S

The components of the E anchor can be mapped to processes now defined by paragraphs of *SCJ-Circus* in direct correspondence with classes of a program.

Safelet The paragraph **safelet** defines the `setUp` and `tearDown` methods. In our example, they are empty (**skip**), and so this paragraph is omitted.

Mission sequencer Figure 29 presents the mission sequencer for the CD_x. In a **sequencer** paragraph, the **state** clause declares the fields of the SCJ mission-sequencer class, which determine the state components of the mission-sequencer application process. In the **initial** clause, we have the SCJ class constructor (and so the initialisation operation of the underlying process). Finally, **getNextMission** defines the body of the `getNextMission` method (and the corresponding *Circus* action in the mission-sequencer process). The assignment to the special variable **ret** defines the mission to be returned. (It is an implicit result parameter of methods that have a return type different from `void`.)

As illustrated, *SCJ-Circus* paragraphs establish a link via underlying *Circus* processes to the E anchor. The direct correspondence enables automation of refinement. On the other hand, *SCJ-Circus* paragraphs also establish a link to code, by identifying components directly implementable using an SCJ API.

Mission For each mission, we have a **mission** paragraph. Figure 30 shows the paragraph in our case study: it defines the behaviour of the single mission of our application. The shared variables in mission memory, which were clearly identified in the *MArea* action, become state components of the process, and references to objects. This is made explicit by the **ref** type constructor.

The **initial** action corresponds to the constructor of the SCJ class. We create shared objects using the special **newM** operator, which instantiates

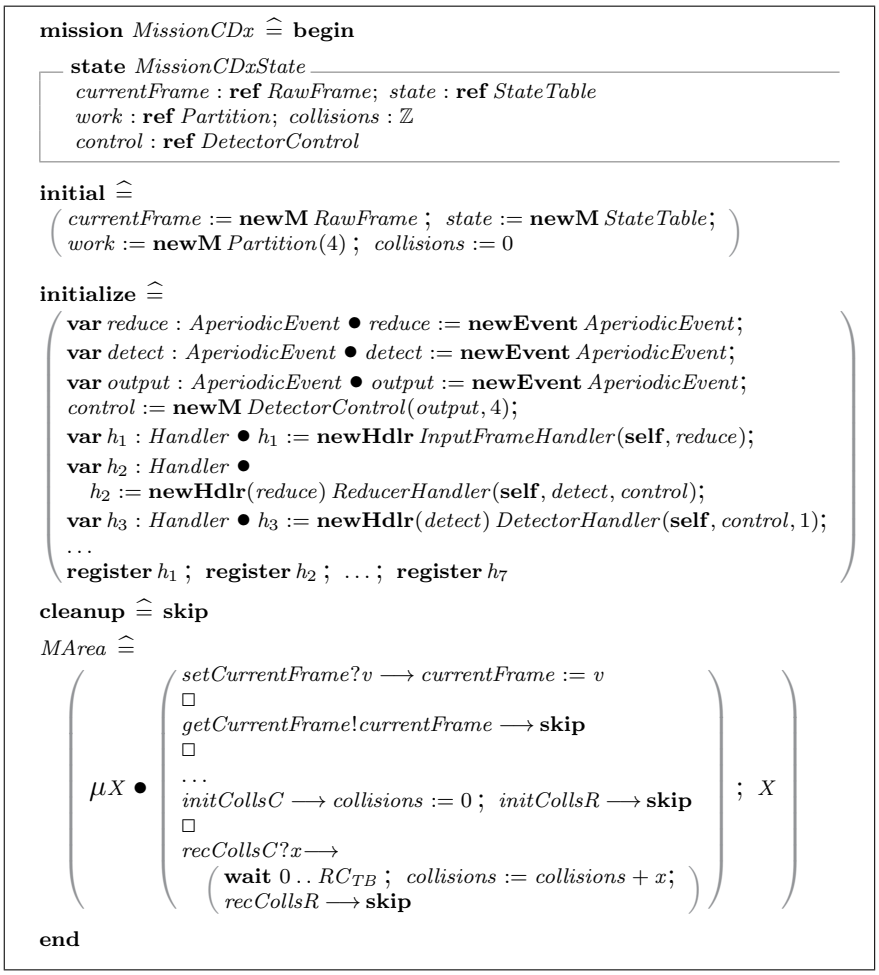

Fig. 30 Anchor S: Mission paragraph *MissionCDx*

them in mission memory. The **initialize** action models the `initialize()` method of the mission, creating its handlers and SCJ events.

Three events are created via the **newEvent** construct and assigned to local variables *reduce*, *detect* and *output* of type *AperiodicEvent*. We note that *AperiodicEvent* as well as *Handler* are not *OhCircus* class types, but sets of identifiers specifically introduced for each SCJ application. After creating the control objects that release the handlers, **initialize** instantiates the handlers of the mission. Here, we use new notation: **newHdlr** to create a handler, and **newHdlr** (*evt*) to specify that the new handler is bound to an aperiodic event *evt*. (In SCJ this corresponds to passing the event to the super constructor when extending `AperiodicEventHandler`.)

Two further *SCJ-Circus* constructs are introduced. The first, **register** h, corresponds to a call to the `register()` method of a handler; it registers the

handler h with the mission. The second one, **fire** *evt*, fires the event *evt* and thus corresponds to calling the `fire()` method of `AperiodicEvent`. These are not encoded in *OhCircus* because they model interactions with the framework.

The *MArea* action is also included in the mission paragraph. It includes the calls to the local methods of the mission. Methods that are to be implemented in other classes become part of their specifications.

Unlike in the E anchor, method calls in the S anchor are encoded by two channels, one for the call (with a name with suffix C) and one for the return (with suffix R). Direct variable access though is still modelled by a single synchronisation as we consider such operations atomic.

We observe the direct correspondence between the mission paragraph in Figure 30 and the *MissionCDx* action of the E anchor. We also observe the direct correspondence between the paragraph in Figure 30 and the SCJ code of the mission class, which we sketch in Figure 31. The state components become fields of the class. The **initial** action becomes the class constructor, and the **initialize** action becomes the `initialize()` method. We note that the bound event identified in **newHdlr** becomes the last parameter of the handler constructor. The **cleanup** action is just **skip**, so there is no need to (re)define the `cleanUp` method in the class, since what is inherited from `Mission` is appropriate in this case. Finally, the methods in the first interleaved action in *MArea* become local methods of the mission class.

Handlers The definition of a *Circus* process that models a handler depends on whether it is periodic or aperiodic. In our example, we have one periodic handler *InputFrameHandler*. The others are aperiodic. For conciseness, we omit the complete model of the handlers found in [43]. Instead, we give one example of a periodic and one example of an aperiodic handler.

A **periodic** paragraph defines a periodic handler, whose period is defined in the declaration of the paragraph. In our example in Figure 32, it is *FRAME_PERIOD*. The **state** paragraph defines the fields of the SCJ handler class. Here, we have just the mission and SCJ event that is used to release *ReducerHandler*. As already explained, these are identifiers (used in the framework model) that record the mission that uses the handler, and the release event. In the SCJ program, they are realised by objects.

The class constructor, defined in the **initial** paragraph, takes them as parameters. The handler method `handleAsyncEvent()` is defined by the paragraph of the same name. It has a direct correspondence with the handler specification as a parallel action of *MissionCDx* in the E anchor (see Figure 24, for example). The communication on *next_frame* corresponds to an external (synchronous) device access in the program. The subsequent **wait** captures the amount of time consumed by the data operation *StoreFrame*. The omitted definition of *StoreFrame* arises from algorithmic refinement during the HR phase of the previous step and uses methods of the *RawFrame* and *State* classes. The **dispatch** paragraph captures the behaviour in each release of the handler (typically using a call to **handleAsyncEvent**).

```
public class CDxMission extends Mission {
  public RawFrame currentFrame;
  public StateTable state;
  public Partition work;
  public int collisions;
  public DetectorControl control;

  public CDxMission() {
    currentFrame = new RawFrame();
    state = new StateTable();
    work = new Partition(4);
    collisions = 0;
  }

  public void initialize() {
    AperiodicEvent reduce = new AperiodicEvent();
    AperiodicEvent detect = new AperiodicEvent();
    AperiodicEvent output = new AperiodicEvent();

    control = new DetectorControl(output, 4);

    InputFrameHandler h1 = new InputFrameHandler(this, reduce);
    ReducerHandler h2 = new ReducerHandler(this, detect, control, reduce);
    DetectorHandler h3 = new DetectorHandler(this, control, 1, detect);
    ...

    h1.register();
    h2.register();
    h3.register();
    ...
  }

  public long missionMemorySize() {
    return Constants.MISSION_MEMORY_SIZE;
  }
  public RawFrame getFrame() {
    return currentFrame;
  }
  public void setFrame(RawFrame frame) {
    currentFrame = frame;
  }
  ...
}
```

Fig. 31 CD_x Mission class implementation

The **aperiodic** paragraph is similar in structure. The difference is in the description of the **dispatch** action, which now also captures the external or SCJ event by which the handler is released. An example of an aperiodic handler is included in Figure 33. The handler paragraph is parametrised by a handler identified *hdl* to permit instantiation: we require four instances of this handler.

In an aperiodic handler, the **dispatch** action is typically much more elaborate. In our example, a synchronisation is raised by the framework model of the SCJ event bound to the handler (*detect* in the example). More precisely, the

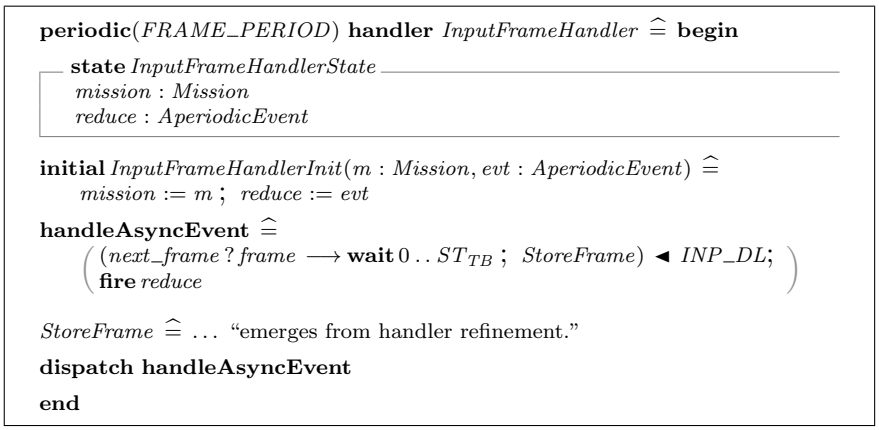

Fig. 32 Anchor S: Periodic handler paragraph *InputFrameHandler*

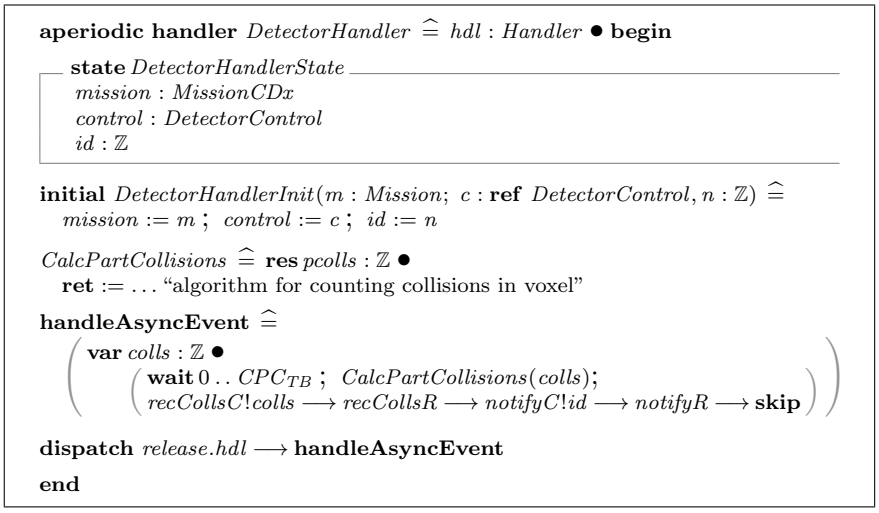

Fig. 33 Anchor S: Aperiodic handler paragraph *DetectorHandler*

channel *release* of type *Handler* is used to release aperiodic handlers bound to software events. The method in **handleAsyncEvent** is again in direct correspondence with the specification of the handler in the E anchor (see Figure 27).

Our refinement patterns and strategy lead to models for programs without memory leaks. In the **initial** action, only assignments to mission state components arise; these correspond to mission fields, so that, any objects referred by them do not become unreachable. In the **initialize** action, objects are associated with local variables, but these are passed as parameters to the handlers, whose constructors assign them to handler fields. The handler objects are part of the SCJ infrastructure. They are not even objects in *SCJ-Circus*. In addition, in the CD_x, in the refinement of *StoreFrame*, we ensure that allocated memory is reused when new frames arrive.

Our case study provides evidence that the CD_x, despite its complexity, can be essentially constructed using our refinement approach originally described in [11] with added support for SCJ events.

5 Conclusions

The SCJ specification enables the development of Java applications using a restricted infrastructure, in such a way that they are amenable to certification under, for instance, the stringent requirements of DO-178C, the civil aerospace guidance document for software development, and similar standards. The assumption is that a small and predictable virtual machine and libraries enhance certifiability and permit meeting tight performance requirements.

It is our view that the SCJ specification embeds two elements: a programming paradigm, and its implementation on a Java platform. One of the contributions of our work is to identify this novel paradigm in the design of *SCJ-Circus*. While the construction of an *SCJ-Circus* model may have many applications, our main goal is the definition of a programming theory for SCJ. This is embedded in the semantic basis of *SCJ-Circus*, and its laws.

An important product of such a theory of programming is a development and verification technique that reveals core laws (of refinement), and can also be used to justify more specialised techniques. Our main contribution in this paper is the definition of a general refinement technique for SCJ. We have presented in detail a strategy for refinement-based development of SCJ programs. It singles out the main issues involved in this task by identifying steps, phases, and stages of refinement that tackle them separately. Namely, we need to address data design using an object-oriented model; introduction of data for shared use; decomposition into a sequence of missions; for each mission, definition of a timed architecture, decomposition into concurrently executed handlers, and data sharing between handlers (both for those executed in sequence and for those truly executing concurrently); algorithmic refinement; and finally, implementation in terms of the SCJ API.

In relation to our own previous work [11], we provide a much more detailed account of the refinement steps related to the verification of missions, handlers, and sharing. We also now cover software events, handler synchronisation, barriers, and a few other patterns of programming. Finally, we validate the whole approach with a significant case study, which is in itself a contribution. Notably, the detailed work on the refinement of the case study revealed a race condition involving the control object: we have a faulty implementation of the barrier mechanism. It could cause the program to deadlock in a very specific execution that had never been observed during testing. We could trace this glitch directly to a failure of the refinement in the Stage (3) of the SH phase. The majority of the models that we have presented have been checked with the *Circus* tools. Tools for *SCJ-Circus* are under development.

Our technique, due to its reliance on algebraic laws of refinement (rather than on posit-and-prove techniques or on model checking) is suitable both

when we already have an implementation to verify and when we are developing an implementation from scratch. If an implementation exists, its design of missions and handlers and use of (shared) data can be used to guide the choice and application of refinement steps and laws. In this case, we start from the A anchor, and reconstruct the development of the existing implementation.

A proof of correctness of an existing implementation can be convoluted. The value in applying our technique comes in structuring and organising such a proof, by concentrating on particular aspects in isolation. It can also help to establish properties of the implementation that are not obvious from the code. For example, in the CD_x, the main data operation is *CalcCollisions* in the A anchor (see Figure 3). Its implementation is distributed over five handlers, and cannot be easily traced in the code. Traceability is established by the refinement: if we follow our example, we can see that the calculation in *CalcCollisions* becomes part of *RecordFrame* (O step, Figure 13), which is then decomposed and split to the reducer and detector handlers (Stages (2-5) of Phase HS in the E step, Figures 20 and 21).

We use several variants of *Circus*, and their underlying UTP theories. *SCJ-Circus* is built from the following items in the UTP framework: nondeterministic imperative programming with specifications (based on Z) [40]; reactive processes with concurrency and communication (based on CSP) [29]; object orientation, with classes and inheritance (based on *OhCircus*) [33]; discrete real-time (based on *Circus Time*) [34,37]; and an SCJ memory model [10]. Some of these theories need to be brought to maturity and linked together. Soundness of our laws relies on the fact that the combination of the theories is conservative. Proof and soundness of some laws is discussed in [43].

The general character of our technique is essential to fulfill its role in the SCJ programming theory. Since SCJ is a new technology, however, there is still much work to be done. Our technique can be used, for instance, to justify or suggest techniques that rely on A anchors generated from models written in more user-friendly notations. It can also be used to justify techniques that tackle specific aspects of the verification. Our technique provides a framework to guide the integration of other specialised techniques.

Future work As future work, we envisage the development of techniques for automatic generation of A anchors from domain-specific models, written in semi-formal or graphical notations like UML, for instance. In [23], for example, we describe a tool for automatic generation of *Circus* models from Simulink diagrams. In this scenario, where the starting specification is guaranteed (by construction) to have a particular architecture, it is possible to envisage more specialised refinement strategies. They can take advantage of the regularity of the model to improve automation of the development.

For certain tasks, successful application of the refinement strategy already requires the identification of design patterns. A catalogue of such patterns and their associated refinement laws and strategies is an important direction for future work. We observe, on the other hand, that our technique is already a

specialisation of the general *Circus* refinement technique. It is in this way that it can afford some level of automation.

In all our examples so far, the external events can be realised as atomic and virtually instantaneous interactions with the environment. For the CD_x, the input *next_frame*?*frame*, for instance, is realised by an interrupt that reads the frame into internal storage. This might take time, but it is a negligible amount of time, and so virtually instantaneous. When this does not hold, we need either to revise our event abstractions, or to provide a separate argument of noninterference, such as the inter-arrival times of frames is large enough to ensure that the device data will be available for the duration of the read. The latter is likely to be convoluted and not compositional. A catalogue of programming patterns that satisfy our assumption can avoid this complication.

We do not address the issue of resources formally. There is nothing in our refinement theory that favours one design or another based on their use of memory or their performance, and in our technique we embed good designs in specialised laws. For a more formal approach, we need resource-aware programming models. They can be accommodated in the UTP, as discussed in [18], but this topic has not been further developed yet.

We have not as yet come across SCJ programs (in the public domain) that make use of several missions. A case study to illustrate the development of programs with multiple missions is, therefore, left as future work. As a first step, we have developed ourselves a multi-mission pacemaker controller [35]. The indication of the Open Group is that missions can be used to cater for different modes of operation in control systems. In this case, the A anchor has to model the various modes of operations, and sequentialisation in the CP phase should bring out the individual sequential components that can be split across missions. In this case, the second step of our refinement approach, which generates the E anchor, needs to be adapted: we need a iterative strategy like that adopted for the handlers; we leave the details of this as future work.

The last Step S transforms a *Circus* model into an *SCJ-Circus* model, and in doing so, checks that the modelled program satisfies the restrictions of the SCJ paradigm. An alternative approach that can be explored consists in delaying any algorithmic refinement to the very end of the strategy. In this case, instead of refinement laws of *Circus*, we need to use refinement laws of *SCJ-Circus*. They would embed the restrictions of the SCJ memory model, and prevent the development of implementations that violate them.

SCJ-Circus programs are so close to actual SCJ code as to enable automatic code generation. A technique and tool to achieve this goal is part of our agenda for future work. In addition, the definition of the SCJ infrastructure embedded in the P model specifies the required functionality of an SCJ implementation: we plan to prove the correctness of an existing implementation.

As well as a target for our refinements, an S anchor can be used in isolation to model existing programs. It is possible to generate S anchors from SCJ programs that follow an organised pattern of programming. It requires, for example, the use of separate classes to define the safelet, the mission sequencer, the missions, and each of the handlers. As far as we can see, this does

not impose any serious restrictions. At the moment, however, example SCJ programs are few and far between, as the technology it yet to reach maturity.

In fact, additional case studies and tools are an important line of work to ascertain the practical relevance of SCJ and of our technique. Case studies are the best way forward in providing a usable catalogue of patterns, and associated laws and strategies. Automating different aspects of the strategy via, for instance, a specialised refinement editor for SCJ or a theorem prover like in [44] is a further goal that makes use of our strategy scalable.

Acknowledgments This work is funded by EPSRC grant EP/H017461/1. Chris Marriott and Neeraj Singh have contributed with useful discussions.

References

1. Adams, M.M., Clayton, P.B.: Cost-Effective Formal Verification for Control Systems. In: ICFEM 2005, *LNCS*, vol. 3785, pp. 465 – 479. Springer-Verlag (2005)
2. Bolton, C.: Using the Alloy Analyzer to Verify Data Refinement in Z. ENTCS **137**(2), 23 – 44 (2005)
3. Braberman, V., Fernandez, F., Garbervetsky, D., Yovine, S.: Parametric Prediction of Heap Memory Requirements. In: International Symposium on Memory Management, pp. 141 – 150. ACM Special Interest Group on Programming Language (2008)
4. Burns, A.: The Ravenscar Profile. Ada Letters **XIX**, 49 – 52 (1999)
5. Burns, A., Wellings, A.J.: Concurrent and Real-Time Programming in Ada. Cambridge University Press (2007)
6. Burns, A., Wellings, A.J.: Real-Time Systems and Programming Languages, 4th edn. Addison Wesley (2009)
7. Cavalcanti, A.L.C., Clayton, P., O'Halloran, C.: From Control Law Diagrams to Ada via *Circus*. FACJ **23**(4), 465 – 512 (2011)
8. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A Refinement Strategy for *Circus*. FACJ **15**(2 - 3), 146 – 181 (2003)
9. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: Unifying Classes and Processes. SoSyM **4**(3), 277 – 296 (2005)
10. Cavalcanti, A.L.C., Wellings, A., Woodcock, J.C.P.: The Safety-Critical Java memory model formalised. FACJ (2012). DOI: 10.1007/s00165-012-0253-4
11. Cavalcanti, A.L.C., Wellings, A., Woodcock, J.C.P., Wei, K., Zeyda, F.: Safety-Critical Java in *Circus*. In: A.P. Ravn (ed.) 9th JTRES, ACM Digital Library. ACM (2011)
12. Cavalcanti, A.L.C., Woodcock, J.C.P.: ZRC—A Refinement Calculus for Z. FACJ **10**(3), 267—289 (1999)
13. Freitas, A.F., Cavalcanti, A.L.C.: Automatic Translation from *Circus* to Java. In: FM 2006, *LNCS*, vol. 4085, pp. 115 – 130. Springer-Verlag (2006)
14. Freitas, L., McDermott, J.P.: Formal methods for security in the xenon hypervisor. STTT, **13**(5), 463 – 489 (2011)
15. Grov, G., Ireland, A., Llano, M.T.: Refinement Plans for Informed Formal Design. In: 3rd ABZ, *LNCS*, vol. 7316, pp. 208 – 222. Springer (2012)
16. Harwood, W., Cavalcanti, A.L.C., Woodcock, J.C.P.: A Theory of Pointers for the UTP. In: ICTAC, *LNCS*, vol. 5160, pp. 141 – 155. Springer-Verlag (2008)
17. Hayes, I.J., Utting, M.: A sequential real-time refinement calculus. Acta Informatica **37**(6), 385 – 448 (2001)
18. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
19. Kalibera, T., Hagelberg, J., Pizlo, F., Plsek, A., Titzer, B., Vitek, J.: CD_x: A Family of Real-time Java Benchmarks. In: Proceedings of the 7th JTRES, pp. 41 – 50. ACM (2009)
20. Kalibera, T., Parizek, P., Malohlava, M., Schoeberl, M.: Exhaustive Testing of Safety Critical Java. In: 8th JTRES, pp. 164 – 174. ACM (2010)

21. Locke, D., Andersen, B.S., Brosgol, B., Fulton, M., Henties, T., Hunt, J.J., Nielsen, J.O., Nilsen, K., Schoeberl, M., Tokar, J., Vitek, J., Wellings, A.: Safety Critical Java Specification, First Release 0.76. The Open Group, UK (2010). `jcp.org/aboutJava/communityprocess/edr/jsr302/index.html`
22. Markey, N.: Robustness in real-time systems. In: 6th IEEE International Symposium on Industrial Embedded Systems, pp. 28 – 34. IEEE (2011)
23. Marriott, C., Zeyda, F., Cavalcanti, A.L.C.: A Tool Chain for the Automatic Generation of *Circus* Specifications of Simulink Diagrams. In: ABZ, *LNCS*, vol. 7316, pp. 294 – 307. Springer (2012)
24. Miyazawa, A., Cavalcanti, A.L.C.: Refinement-oriented models of stateflow charts. SCP **77**(10 – 11), 1151 – 1177 (2012)
25. Morgan, C.C.: Auxiliary Variables in Data Refinement. IPL **29**(6), 293 – 296 (1988)
26. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice-Hall (1994)
27. Mukherjee, P., Stavridou, V.: Decomposition in Real-Time Safety-Critical Systems. Real-Time Systems **14**, 183 – 202 (1998)
28. Oliveira, M.V.M.: Formal Derivation of State-Rich Reactive Programs Using *Circus*. Ph.D. thesis, University of York (2006)
29. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: A UTP Semantics for *Circus*. FACJ **21**(1-2), 3 – 32 (2009)
30. Oliveira, W.R., Barros, R.S.M.: The Real Numbers in Z. In: Proceedings of the 2nd BCS-FACS Northern Formal Methods Workshop. British Computer Society (1997)
31. Reed, G.M., Roscoe, A.W.: A timed model for communicating sequential processes. ICTAC **58**, 249–261 (1988)
32. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall (1998)
33. Santos, T.L.V.L., Cavalcanti, A.L.C., Sampaio, A.C.A.: Object Orientation in the UTP. In: Unifying Theories of Programming, *LNCS*, vol. 4010, pp. 18 – 37. Springer-Verlag (2006)
34. Sherif, A., Cavalcanti, A.L.C., He, J., Sampaio, A.C.A.: A process algebraic framework for specification and validation of real-time systems. FACJ **22**(2), 153 – 191 (2010)
35. Singh, N.K., Wellings, A.J., Cavalcanti, A.L.C.: The cardiac pacemaker case study and its implementation in safety-critical Java and Ravenscar Ada. In: 10th JTRES, pp. 62 – 71. ACM (2012)
36. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation **132**(2), 109 – 176 (1997)
37. Wei, K., Woodcock, J.C.P., Burns, A.: A Timed Model of *Circus* with the Reactive Design Miracle. In: 8th SEFM, pp. 315 – 319. IEEE Computer Society (2010)
38. Wellings, A.: Concurrent and Real-Time Programming in Java. Wiley (2004)
39. Woodcock, J.C.P.: The miracle of reactive programming. In: Unifying Theories of Programming 2008, *LNCS*, pp. 202 – 217. Springer (2009)
40. Woodcock, J.C.P., Cavalcanti, A.L.C.: A Tutorial Introduction to Designs in Unifying Theories of Programming. In: IFM 2004, *LNCS*, vol. 2999, pp. 40 – 66. Springer-Verlag (2004). Invited tutorial.
41. Woodcock, J.C.P., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall (1996)
42. Zeyda, F., Cavalcanti, A.L.C., Wellings, A.: The Safety-critical Java Mission Model: a formal account. In: ICFEM, *LNCS* (2011)
43. Zeyda, F., Cavalcanti, A.L.C., Wellings, A., Woodcock, J.C.P., Wei, K.: Refinement of the Parallel CDx. Tech. rep., University of York, Department of Computer Science, York, UK (2012)
44. Zeyda, F., Oliveira, M.V.M., Cavalcanti, A.L.C.: Mechanised support for sound refinement tactics. FACJ **24**(1), 127 – 160 (2012)

A Appendix: Definitions in Anchor A of the CD_x

The type *Frame* is the set of partial functions from aircraft to (3d) vectors whose size is less than or equal to $MAX\_AIRCRAFT$, the maximum number of aircraft detected.

$$Frame \mathrel{\widehat=} \{ f : Aircraft \pfun Vector \mid \# f \leq MAX\_AIRCRAFT \}$$

The domain of the function determines the aircraft in view of the radar. We introduce

Aircraft as the set of non-empty sequences of byte values: $Aircraft \mathrel{\widehat=} \mathrm{seq}_1(byte)$ where *byte* is the set of integers from -128 to 127. The sequences represent unique call signs, mirroring the way aircraft are identified in aviation. The type *Vector* is defined by a schema whose components x, y and z correspond to the coordinates of a vector.

$$Vector \mathrel{\widehat=} [x : \mathbb{R};\ y : \mathbb{R};\ z : \mathbb{R}]$$

The work in [30] describes how real numbers can be axiomatised in Z.

We introduce common operators on vectors such as sum ($+_V$), difference ($-_V$), scalar produce ($*_V$), dot product ($\cdot_V$) and length ($|\_|$). Their Z definitions are omitted here as they are standard. We also define *ZeroV* and *UnitV* for the zero and unit vector. $MkVector(c_1, c_2, c_3)$ yields a record $\langle\!| x == c_1, y == c_2, z == c_3 |\!\rangle$ of type *Vector*.

The function *CalcCollisionSet* yields collisions as a set of aircraft pairs.

$$CalcCollisionSet : (Frame \times Frame) \to \mathbb{F}(Aircraft \times Aircraft)$$

$$\forall\, posns, motions : Frame \bullet CalcCollisionSet\,(posns, motions) = \left\{ \begin{array}{l} a_1 : Aircraft;\ a_2 : Aircraft \mid a_1 \in \mathrm{dom}\, posns \land a_2 \in \mathrm{dom}\, posns \land \\ \quad collide\,((posns\ a_1, motions\ a_1), (posns\ a_2, motions\ a_2)) \end{array} \right\}$$

The pairs (a_1, a_2) in the set of collisions are characterised by a set comprehension that uses of a relation *collide* that captures whether their trajectories $(posns\ a_1, motions\ a_1)$ and $(posns\ a_2, motions\ a_2)$ are at risk of colliding. A trajectory is a pair of vectors: the first gives the trajectory's position and the second its motion. We define $Trajectory \mathrel{\widehat=} Vector \times Vector$.

$$collide\ \_ : \mathbb{P}\,(Trajectory \times Trajectory)$$

$$\forall\, t_1, t_2 : Trajectory \bullet collide(t_1, t_2) \Leftrightarrow distance(t_1, t_2) \leq THRESHOLD$$

THRESHOLD is a constant that specifies the minimum acceptable distance between two trajectories; if it is less than or equal to that, we signal a potential collision.

The *distance* function carries out the actual distance calculation of aircraft trajectories.

$$distance : Trajectory \times Trajectory \to \mathbb{R}$$

$$\forall\, t_1, t_2 : Trajectory \bullet distance(t_1, t_2) = \left(\begin{array}{l} \mu\, d : \mathbb{R} \mid \\ \quad \left(\begin{array}{l} (\exists\, x : \mathbb{R} \mid 0 \leq x \leq 1 \bullet d = |(t_2.1 +_V x *_V t_2.2) -_V (t_1.1 +_V x *_V t_1.2)|) \land \\ (\forall\, x : \mathbb{R} \mid 0 \leq x \leq 1 \bullet d \leq |(t_2.1 +_V x *_V t_2.2) -_V (t_1.1 +_V x *_V t_1.2)|) \end{array} \right) \end{array} \right)$$

We determine the minimal distance between two traversing points. This may not be the minimal distance between any two points, but is consistent with the algorithm in [19].

B Appendix: some classes

B.1 Appendix: class *RawFrame*

The class *RawFrame* is used to encode radar frames as data objects in the program. It records the position of all aircraft, identified by their call sign, that are currently in view of the radar. Some constants capture static variables used in the program.

$$NUMBER\_OF\_PLANES : \mathbb{Z};$$
$$LENGTH\_OF\_CALLSIGN : \mathbb{Z}$$

class *RawFrame* $\mathrel{\widehat=}$ **begin**

The implementation uses two arrays: *callsigsn*, to record aircraft positions, and *positions*, to determine the respective call signs of the aircraft. It also includes a *planeCnt* integer

component to determine the number of valid entries in the array ensemble.

state *RawFrameState*

$$\begin{array}{l} \textbf{public}\ callsigns : byteArray \\ \textbf{public}\ positions : floatArray \\ \textbf{public}\ planeCnt : int \end{array}$$

The initialisation schema captures the initilisations of the state components in the code. Here, we create the data objects for both arrays and also define that, initially, there are no valid entries in the arrays; hence, no aircrafts are initially recorded in the frame.

initial *RawFrameInit*

$$\begin{array}{l} RawFrameState' \\ \hline callsigns' = \\ \quad \textbf{newM}\ byteArray(LENGTH\_OF\_CALLSIGN * NUMBER\_OF\_PLANES) \\ positions' = \textbf{newM}\ floatArray(3 * NUMBER\_OF\_PLANES) \\ planeCnt' = 0 \end{array}$$

The logical methods are used in the refinement for the definition of retrieve relations. The function *getCallSign* determines the call sign *result*! recorded in *callsigns* with index *plane*?.

logical function *getCallSign*

$$\begin{array}{l} \Xi RawFrameState \\ plane? : \mathbb{Z} \\ result! : \mathrm{seq}\ byte \\ \hline 0 \leq plane? < planeCnt \\ \#\, result! = LENGTH\_OF\_CALLSIGN \\ \forall\, i : 1 \,..\, LENGTH\_OF\_CALLSIGN \bullet \\ \quad result!(i) = callsigns.getA(plane? * LENGTH\_OF\_CALLSIGN + i - 1) \end{array}$$

The *find* logical function uses *getCallSign* to obtain the index *result*! for an aircraft given by its call sign *a*? within the array ensemble. This, in particular, allows us to determine the position of an aircraft with a given call sign.

logical function *find*

$$\begin{array}{l} \Xi RawFrameState \\ a? : Aircraft \\ result! : \mathbb{Z} \\ \hline result! = \left(\begin{array}{l} \textbf{if}\ (\exists_1\, i : 0 \,..\, (planeCnt - 1) \bullet RawFrame.getCallSign(i) = a?) \\ \textbf{then}\ (\mu\, i : 0 \,..\, (planeCnt - 1) \mid RawFrame.getCallSign(i) = a?) \\ \textbf{else} - 1 \end{array} \right) \end{array}$$

The following private method is overloaded. Although, strictly speaking, overloading is not allowed in *OhCircus*, a simple renaming can be used to give semantics to the class. To be faithful to the code, we use the overloading here. In the same vein, we also use **for** loops, whose meaning can be given by recursion in the usual way.

$$\textbf{private}\ copy(signs : byteArray, posns : floatArray) \mathrel{\widehat{=}} \textbf{var}\ i : \mathbb{Z} \bullet \left(\begin{array}{l} \textbf{for}\ i = 0\ \textbf{to}\ signs.length - 1 \bullet \textbf{self}.callsigns.setA(i, signs.getA(i)); \\ \textbf{for}\ i = 0\ \textbf{to}\ posns.length - 1 \bullet \textbf{self}.positions.setA(i, posns.getA(i)); \\ \textbf{self}.planeCnt = posns.length \ \mathrm{div}\ 3 \end{array} \right)$$

The only method in the interface of RawFrame is *copy* defined below.

$$\textbf{public}\ copy(frame : RawFrame) \mathrel{\widehat{=}} \textbf{self}.copy(frame.callsigns, frame.positions)$$

end

It uses the private *copy* methods above to copy the arrays of a given instance of the class *RawFrame* itself to the arrays of the current object.

B.2 Appendix: class *StateTable*

The class *StateTable* is used to record previous aircraft positions. This is important to calculate the predicted trajectories of aircraft and determine their potential collisions.

class *StateTable* $\widehat{=}$ **begin**

A hash map *positionMap* stores aircraft positions. For memory management, there is a store of pre-allocated objects for 3d vectors: the *allocatedVectors* and *usedVectors* fields.

StateTableState

private $position\_map : HashMap[CallSign, \textbf{ref}\ Vector3d]$
private $allocatedVectors : Vector3dArray$
private $usedVectors : int$

The initialisation creates the data objects for the position map and as allocates 3d vector objects for the object store. The allocation during initialisation ensures that the vector objects are created in mission memory; this is crucial since those objects are shared between the handlers. Initially, no objects are in use from the store.

initial *Init* $\widehat{=}$

$$\left(\begin{array}{l} position\_map := \textbf{newM}\ HashMap; \\ allocatedVectors := \textbf{newM}\ Vector3dArray(MAX\_VECTORS); \\ \left(\begin{array}{l}\textbf{for}\ index = 0\ \textbf{to}\ allocatedVectors.length - 1 \bullet \\ \quad allocatedVectors.setA(index, \textbf{newM}\ Vector3d)\end{array}\right); \\ usedVectors := 0 \end{array}\right)$$

The *put* method is used to insert an element into the *position_map*. This is essentially using the corresponding *put* method of *HashMap*, with added logic that ensures that 3d vector objects are not created anew, but recycled from the pre-allocated object store.

public $put(callsign : CallSign, x : float, y : float, z : float)$ $\widehat{=}$

$$\left(\begin{array}{l} \textbf{var}\ v : \textbf{ref}\ Vector3d \bullet v := position\_map.get(callsign); \\ \textbf{if}\ v = \textbf{null} \longrightarrow \\ \quad \left(\begin{array}{l} v := allocatedVectors.getA(usedVectors); \\ usedVectors := usedVectors + 1; \\ position\_map.put(callsign, v) \end{array}\right) \\ [\!]\ \neg\ v = \textbf{null} \longrightarrow \textbf{skip} \\ \textbf{fi}; \\ v.x = x\ ;\ v.y = y\ ;\ v.z = z \end{array}\right)$$

The *get* method infers the position of an aircraft from the position map.

public $get(callsign : CallSign)$ $\widehat{=}$ **ret** $:= position\_map.get(callsign)$

end

B.3 Appendix: class *Partition*

The class *Partition* holds the data for partitions of the voxel space.

class *Partition* $\widehat{=}$ **begin**

It records a list *parts* of arrays corresponding to the partitions as well as a cyclic *counter*

that facilitates recording voxels' aircrafts in the partitions.

state *PartitionState*

$$\begin{array}{l} \textbf{private}\ parts : ListArray[LinkedList] \\ \textbf{private}\ counter : \mathbb{Z} \\ \hline parts \neq \textbf{null} \land 0 \leq counter < parts.length \end{array}$$

The initialisation allocates the *parts* list and determines that all partitions are empty (this is achieved by a call to the method *clear* of this same class defined next).

$$\textbf{initial}\ Init(n : \mathbb{Z}) \mathrel{\widehat=} \left(\begin{array}{l} parts := \textbf{newM}\ ListArray(n); \\ (\textbf{for}\ i = 0\ \textbf{to}\ n - 1 \bullet parts.setA(i, newLinkedList)); \\ \textbf{self}.clear \end{array} \right)$$

The *clear* method clears the voxel lists for all partitions.

$$\textbf{public synchronized}\ clear \mathrel{\widehat=} (\textbf{for}\ index = 0\ \textbf{to}\ parts.length - 1 \bullet parts.getA(index).clear)\ ;\ \ counter := 0$$

The *recordMotionList* records the aircraft in a voxel in one of the partitions. The cyclic counter *counter* is used to ensure that partitions are populated in a balanced manner.

$$\textbf{public synchronized}\ recordMotionList(motions : List) \mathrel{\widehat=} \left(\begin{array}{l} parts.getA(counter).add(motions); \\ counter := (counter + 1) \bmod parts.length \end{array} \right)$$

The *getDetectorWork* method obtains the content of one of the partitions subsequent to all voxel motion lists being recorded. It simply returns the respective entry of the *parts* array.

$$\textbf{public synchronized}\ getDetectorWork(id : \mathbb{Z}) \mathrel{\widehat=} \textbf{ret} := parts.getA(id - 1)$$

end