Java in the Safety-Critical Domain

Ana Cavalcanti, Alvaro Miyazawa, Andy Wellings, Jim Woodcock, Shuai Zhao

Department of Computer Science, University of York, UK

Abstract. Safety-Critical Java (SCJ) is an Open Group standard that defines a novel version of Java suitable for programming systems with various levels of criticality. SCJ enables real-time programming and certification of safety-critical applications. This tutorial presents SCJ and an associated verification technique to prove correctness of programs based on refinement. For modelling, we use the *Circus* family of notations, which combine Z, CSP, Timed CSP, and object orientation. The technique caters for the specification of functional and timing requirements, and establishes the correctness of designs based on architectures that use the structure of missions and event handlers of SCJ. It also considers the integrated refinement of value-based specifications into class-based designs using SCJ scoped memory areas. As an example, we use an SCJ implementation of a widely used leadership-election protocol.

1 Introduction

Java needs no introductions: it has a wide base of programmers, an impressive collection of libraries, and continues to evolve with the backing of a very large number of companies. However, it lacks effective support for real-time application development, in particular it has poor facilities for real-time scheduling and unpredictable memory management. This has led to the creation of the Real-Time Specification for Java (RTSJ) [47], which augments the Java platform to provide a real-time virtual machine and support preemptive priority-based scheduling and a complementary region-based memory management mechanism.

Java augmented by the RTSJ provides a comprehensive set of facilities suitable for a wide range of real-time applications. Safety-critical applications, however, require the use of a controlled engineering approach, to ensure reliability, robustness, maintainability, and traceability. Many of them also require certification based on standards before they can be deployed. For these reasons, it is common to reduce complexity (and with it flexibility) via the adoption of language subsets. Examples are SPARK Ada [3] and MISRA C [37]. In this context, RTSJ is far too rich: it includes the whole of Java, and more.

SCJ has been designed under the Java Community Process: JSR 302. It defines a minimal set of capabilities required for safety-critical applications using Java implementations. As a result of this effort, we have an SCJ specification, a reference implementation, and a technology compatibility kit, which contains benchmark examples used to confirm that a particular implementation is compatible with the SCJ specification. The goal is to support certification under, for example, the DO-178 [42]. Nothing is said, however, about design techniques.

As opposed to the RTSJ, SCJ enforces a constrained execution model based on missions, event handlers, and memory areas [46]. SCJ restricts the RTSJ. It prohibits use of the heap and defines a policy for the use of memory areas, which are cleared at specific points of the program flow to avoid the unpredictable garbage collection of the heap. The SCJ design is organised in Levels (0, 1, and 2), with a decreasing amount of restrictions to the execution model.

In this tutorial, we give a detailed description of SCJ and its programming and memory models. For illustration, we use a Level 1 implementation of a leadership-election protocol, which is widely used for coordination of distributed systems. SCJ Level 1 corresponds roughly to the Ravenscar profile for Ada [6].

We also present here a technique for verification by refinement of SCJ Level 1 programs [12]. It uses the *Circus* family of notations [10], which combine constructs from Z [49] for data modelling, CSP [40] for behavioural specification, and standard imperative commands from Morgan's refinement calculus [34]. We cover *Circus Time* [45], with facilities for time modelling from Timed CSP [39], and *OhCircus* [11], based on the Java model of object-orientation. This tutorial gives an overview of *Circus* and its constructs relevant for modelling SCJ designs.

Our technique is based on the stepwise development of SCJ programs based on specification models that do not consider the details of either the SCJ mission or memory models. Development proceeds by model transformation justified by the application of algebraic laws that guarantee that the transformed model is a refinement of the original model. Before, presenting the SCJ refinement technique, we give an overview of algebraic refinement.

The verification technique is a refinement strategy: a procedure for application of algebraic refinement laws. Four *Circus* specifications characterise the major development steps: we call them anchors, as they identify the (intermediate) targets for model transformation and the design aspects treated in each step of development. Each anchor is written using a different combination of the *Circus* family of notations. The first anchor is the abstract specification written in *Circus Time*. The last is written in *SCJ-Circus*; it is so close to an SCJ program as to enable automatic code generation. This tutorial describes this technique using the verification of the leadership-election protocol as an example.

Next, we present the notations used in our work, namely, SCJ, in Section 2, and *Circus*, in Section 3. Algebraic refinement is the subject of Section 4. Finally, Section 5 presents our refinement strategy. We draw some conclusions, where we identify open problems on refinement for SCJ, in Section 6.

2 Safety-Critical Java

This section provides an introduction to the Safety-Critical Java programming model and gives an example of a simple program that can control several robots. The robots are shown in Figure 1 and they perform a coordinated dance. One of them is elected the leader robot and initiates the dance routine. The others are followers and perform the actions indicated by the leader. During the dance, robots can fail and, if necessary, a new leader can be elected. An identi-

cal SCJ program runs on each robot. We present the overall architecture of the application and then focus on the details of the election algorithm.

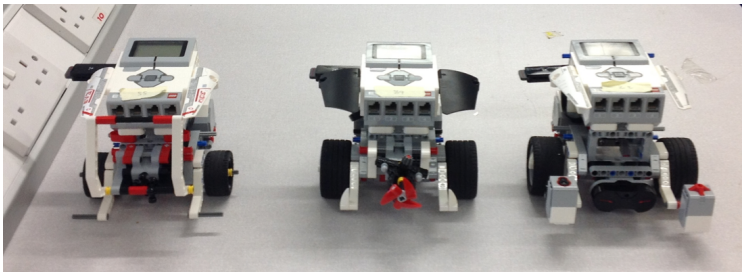

Fig. 1. Dancing Robots

An SCJ program is executed under the auspices of an SCJ virtual machine, which provides core Java services and an infrastructure to manage the life-cycle of safety-critical applications. The core services are those typically provided by a standard Java virtual machine and include support for bytecode execution and memory management. The infrastructure is provided in a Java extension library named`javax.safetycritical`. It supports the main programming abstractions defined by SCJ and requires specialised support from the core services, not found in standard Java virtual machines. Typically, an SCJ virtual machine is hosted on a high-integrity operating system (such as Green Hills Integrity real-time operating system), as illustrated in Figure 2.

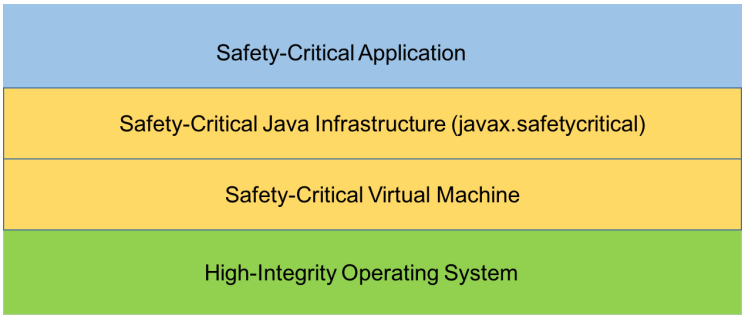

Fig. 2. Safety-Critical Java: VM and Infrastructure

In order to understand the SCJ programming model, there are three main topic areas that must be mastered:

1. applications, missions and mission sequencers;

2. concurrency and scheduling; and
3. memory management.

These topics are covered in the next three sections. Throughout, we use the robot leadership-election application as an illustrative example.

2.1 Applications, missions and mission sequencers

An SCJ program is started by invoking the SCJ virtual machine with a parameter that identifies the application's main program. This is called a *safelet* in SCJ, as it is analogous to an applet in which Java code executes in a constrained web-browser environment. The SCJ infrastructure defines the interface to a safelet, and the application must provide a class that implements this interface.

The application itself consists of the execution of a sequence of *missions,* where a mission represents an application's activity that must be completed. For example, a program that controls the flight of an aircraft might have three main missions: one that manages the take-off activity, one that maintains the flight at its cruising altitude, and one that oversees the landing procedures.

In our robot application, there are two missions. During the first mission a leader is elected. Once the election is completed, the robots perform their dance mission. If a failure occurs, the robots return to the election mission.

As illustrated in Figure 3, each mission has three phases of operation:

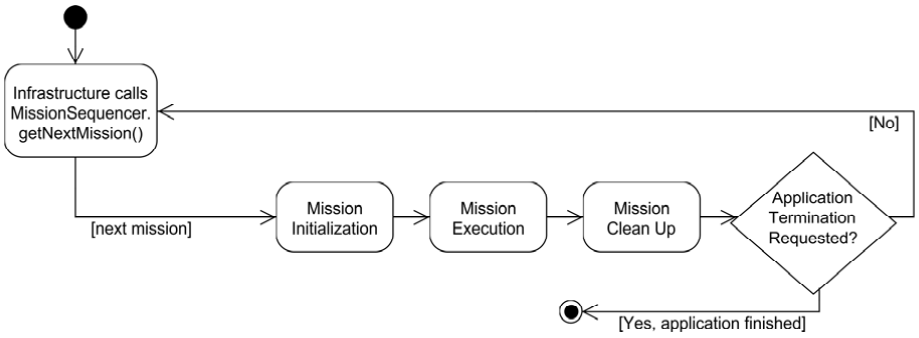

Fig. 3. Safety-Critical Application Phases[29]

1. Initialization – during which the resources needed to complete the mission are acquired and initialised. In our robot application, the initialisation of the election mission acquires access to a wireless network and establishes links with the other robots.
2. Execution – during which the activity of the mission is performed: it starts after the initialization phase has been completed. In our robot application, the execution phase of the election mission implements a communication protocol to elect a new leader robot.

3. Cleanup – starts after completion of the execution phase, and is responsible for returning any resources and performing any other needed finalization code. In our robot application, all resources are returned automatically to the operating system when the program terminates. Hence, there is no explicit application cleanup code.

The order of execution of missions is controlled by an application-defined *mission sequencer* as also illustrated in Figure 3.

Hence, the `Safelet` interface contains the following two methods:

```
1  package javax.safetycritical;
2
3  public interface Safelet<M extends Mission<M>> {
4   public void initializeApplication();
5   public MissionSequencer<M> getSequencer();
6   ...
7  }
```

The `initializeApplication` method is called by the SCJ infrastructure after the SCJ virtual machine has been initialised. Following this, it calls the `getSequencer` method to obtain the application mission sequencer that will oversee the sequence of execution of the missions.

As mentioned in the previous section, SCJ has three compliance levels. The SCJ uses Java generics to ensure that a mission sequencer and its missions have been designed for the same compliance level and are type safe.

The structure of the code for the robots example is shown below:

```
1  import javax.safetycritical.*;
2
3  class RobotApp implements Safelet<RobotMission> {
4
5    @Override
6    public MissionSequencer<RobotMission> getSequencer() {
7      return new RobotSequencer(...);
8    }
9
10   @Override
11   public void initializeApplication() {
12   ...
13   }
14 }
```

All missions that are scheduled by an application must have a common superclass. In the robots example, this is called `RobotMission` and appears as the generic parameter at line 2. The `getSequencer` method at line 5 now can only return a mission sequencer that schedules missions of type `RobotMission`.

For this tutorial, we ignore a mission sequencer's parameters and just consider one of its main methods: `getNextMission` on line 10 below. This method is called

by the infrastructure to select the initial mission to execute, and subsequently, each time a mission terminates, in order to determine the next mission to execute.

```
1  package javax.safetycritical;
2
3  public abstract class MissionSequencer<M extends Mission<M>>
4          extends ManagedEventHandler {
5
6    /** Construct a MissionSequencer object to oversee
7      * a sequence of mission executions
8      */
9    public MissionSequencer(...) {
10     ...
11   }
12
13   protected abstract M getNextMission() {
14     ...
15   }
16   ...
17 }
```

A mission sequencer is an *asynchronous event handler* (ASEH): it executes in its own thread of control. This is considered in depth in Section 2.2.

The structure of the robot mission sequencer can now be given:

```
1  import javax.safetycritical.*;
2
3  class RobotSequencer extends MissionSequencer<RobotMission> {
4
5      private Mission mission;
6      private boolean electing = true;
7
8      public RobotSequencer(...) {
9        super(...); . . .
10     }
11
12     @Override
13      public Mission getNextMission() {
14        if (electing) {
15            return new ElectionMission();
16        } else {
17            return new DanceMission();
18        }
19     }
20
21 }
```

The boolean variable `electing` on line 4 indicates whether a new leader needs to be elected. If it has value true, then the method `getNextMission` in line 12 returns a mission to perform this task: an instance of `ElectionMission`.

The `Mission` class encapsulates the direct infrastructure support for an SCJ mission; its main methods are shown below. The application extends this class and overrides its `initialize` and `cleanUp` methods.

```
1  package javax.safetycritical;
2  public abstract class Mission<M extends Mission<M>> {
3    public Mission() {}
4
5    protected abstract void initialize();
6    protected boolean cleanUp() {...}
7
8    /* Request that this mission be terminated */
9    public final boolean requestTermination() {...}
10
11   /* Is there an outstanding termination request for this mission */
12   public final boolean terminationPending() {...}
13
14   /* Obtain the controlling sequencer */
15   public MissionSequencer<M> getSequencer() {...}
16
17   /* Obtain the current mission.*/
18   public static <M extends Mission<M>> M getMission() {...}
19 }
```

A typical implementation of `initialize` instantiates and registers all the ASEHs that constitute the `Mission`. Besides, `initialize` may also instantiate and initialise mission-level data structures. The infrastructure ensures that ASEHs can only be instantiated and registered during the `initialize` method. The infrastructure also arranges to begin executing the registered ASEHs associated with a particular `Mission` upon return from its `initialize` method.

The `cleanUp` method is called by the infrastructure after all asynchronous event handlers registered with the mission have terminated.

The `requestTermination` method is called by the application to initiate mission termination. When it is called, the infrastructure invokes the method `signalTermination` (see Section 2.2) on each ASEH registered in the mission. Additionally, the infrastructure (1) disables all periodic event handlers (PEHs) associated with this `Mission`, so that they experience no further releases; (2) disables all aperiodic event handlers (APEHs), so that no further releases are honoured; (3) clears the pending event (if any) for each event handler (including any one-shot event handlers), so that the event handler can be effectively shut down following completion of any event handling that is currently active; (4) waits for all of the ASEH objects associated with this mission to terminate their execution; (5) invokes the `cleanUp` methods for each of the ASEHs associated with this mission; and (6) invokes the `cleanUp` method associated with this mission.

In our robot example, the `Election` and `Dance` missions have a common superclass: the `RobotMission` class sketched below. Irrespective of the mission's main functionality, it must manage communication between the robots across the wireless network. The common initialization code, therefore, creates and

registers two ASEHs. On line 4 below, the `Receiver` class is a PEH. Its goal is to receive communication from the robots.

```
1  public abstract class RobotMission extends Mission<RobotMission> {
2    ...
3    protected void initialize() {
4      Receiver receiver = new Receiver(...);
5      receiver.register();
6
7      Sender sender = new Sender(...);
8      sender.register();
9    }
10 }
```

Similarly on line 7, the `Sender` class is also a PEH. Its goal is to broadcast communication to the other robots. Both of the robot's missions extend this class; for instance, the election mission is given below.

```
1  class ElectionMission extends RobotMission {
2    @Override
3    protected void initialize() {
4      super();
5      Elector elector = new Elector(...);
6      elector.register();
7    }
8    ...
9  }
```

`ElectionMission` creates and registers an additional PEH. Its goal is to use the state of each robot to determine whether it should be a leader or a follower.

2.2 Concurrency and scheduling

In general, there are two models for creating concurrent programs. The first is a thread-based model in which each concurrent entity is represented by a thread of control. The second is an event-based model, where an event handler executes in direct response to the firing of its associated event. The RTSJ, upon which SCJ is based, supports a rich concurrency model allowing real-time threads and asynchronous events. The SCJ Level 1 concurrency model simplifies this and relies exclusively on asynchronous event handling.

An ASEH executes in response to invocation requests (known as *release events*); the resulting execution of the associated logic is a *release*. Release requests are categorised as follows: periodic, sporadic, or aperiodic. If R_i denote the time at which an ASEH has had the i^{th} release event occur, then:

1. an ASEH is periodic when there exists a value $T > 0$ such that, for all i, $R_{i+1} - R_i = T$, where T is called the period;
2. an ASEH that is not periodic is said to be aperiodic; and

3. an aperiodic ASEH is said to be sporadic when there is a known value $T > 0$ such that for all i, $R_{i+1} - R_i \geq T$. T is then called the minimum interarrival time (MIT).

PEHs are timed triggered in SCJ, which means that they are indirectly released via the passage of time (using a real-time clock). APEHs and sporadic (SEH) handlers can be both timed triggered or released directly from application code.

SCJ specifies a set of constraints on the RTSJ concurrency model. This constrained model is enforced by defining a new set of classes, all of which are implementable using the concurrency constructs defined by the RTSJ. As an example, the following shows the class for a PEH. This class permits the automatic periodic execution of code. The `handleAsyncEvent` method behaves as if the handler were attached to a periodic timer. This method is executed once for every release. The class is abstract; non-abstract sub-classes must override `handleAsyncEvent` and may override the default `cleanUp` method.

```
1  package javax.safetycritical;
2
3  public abstract class PeriodicEventHandler extends ManagedEventHandler
4  {
5    /* Constructs a periodic event handler.
6     * priority: specifies the priority parameters for this periodic event handler.
7     * release: specifies the periodic release parameters, in particular the
8     * start time, period and deadline miss handler.
9     */
10   public PeriodicEventHandler(PriorityParameters priority,
11                               PeriodicParameters release, ...) {...}
12
13   /* Applications override this method to provide
14      the code to be executed on each release */
15   public void handleAsyncEvent() {...}
16
17   /* Register this handler with its mission */
18   public void register() {...}
19
20   /* Called by the infrastructure during the mission cleanup phase *
21   public void cleanUp() {...}
22
23  /* Called by the infrastructure to indicate that the enclosing mission
24   * has been instructed to terminate. SS*/
25    public void signalTermination() {...}
26 }
```

The SCJ supports communication between ASEHs using shared variables, and so requires support for synchronisation and priority-inversion management protocols. On multiprocessor platforms, it is assumed that all processors can access all shared data and resources, although not necessarily with uniform access times. SCJ requires implementations to support priority-ceiling emulation, a particular protocol that allows the synchronisation to be analysed for its timing properties.

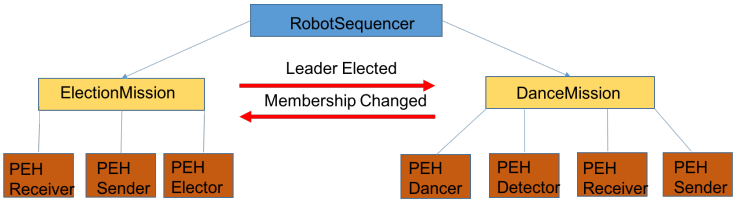

Fig. 4. The Architecture of the Robots Safety-Critical Application

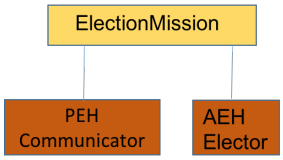

Fig. 5. Optimised Architecture of the Election Mission

Scheduling in SCJ is performed in the context of a *scheduling allocation domain*. The scheduling allocation domain of an ASEH consists of the set of processors on which that schedulable object may be executed. Each ASEH can be scheduled for execution in only one scheduling allocation domain. At Level 1, multiple allocation domains may be supported, but each domain must consist of a single processor. Hence, from a scheduling perspective, a Level 1 system is a fully partitioned system. Within a scheduling allocation domain, multiple ASEHs are scheduled for execution in priority order using a priority-based scheduler. If ASEHs have the same priority, then they are scheduled in a FIFO order, that is, the order in which they become schedulable.

In the robot example, there are several PEHs in each mission. Two handlers are responsible for robot-to-robot communication in each mission. The other ones focus on the main activity of the mission (electing a leader, detecting a change in leadership, and performing the dance). The full software architecture of the program is illustrated in Figure 4.

For small embedded systems, it is often required that we optimise the solution in order to reduce the scheduling overheads. One possibility is to combine the PEHs responsible for communication into one handler. The `Elector` can then be transformed into an APEH which is released on successful receipt of one round of communication. This is illustrated in Figure 5.

2.3 Memory Management

In standard Java all objects are allocated on a heap. Traditionally, dynamic memory allocation and the resulting heap management (garbage collection) has been vetoed by the authorities who certify safety-critical systems on the grounds

that it is too unpredictable. For this reason, the RTSJ introduces the notion of a memory area; this is a chunk of memory from where the memory for object allocation is taken. The Java heap is an example of a memory area.

The RTSJ supports two additional types of memory areas: immortal and scoped memory. Every object allocation is performed with an allocation context. It can change dynamically by a thread of control entering into and exiting from a memory area. The current allocation context at the time an object allocation is requested determines which memory area its space comes from.

Objects created in immortal memory are never collected: once created they exist for the lifetime of the application. Objects in scoped memory areas are automatically freed when no thread of control has an active allocation context for that memory area, that is, it has entered but not exited that memory area.

SCJ constrains the memory model of RTSJ by not allowing the heap memory area. It also distinguishes between scoped memory areas that can be entered by multiple ASEHs (called *mission memory*) and those that are private to an ASEH (called *private memory*). Each mission has a single mission memory. Each ASEH has a single private memory area (called *per-release memory area*), which is entered into automatically when the ASEH is released and exited automatically (and hence has all its objects collected) when the release completes. Each ASEH may also have nested private memories for ephemeral objects. All objects stored in mission memory are collected at the end of each mission.

In addition, all ASEHs have a thread stack where they can store references to objects created in the various memory areas. Figure 6 illustrates the memory hierarchy of an SCJ program. In order to maintain the referential integrity of all objects in SCJ programs, a reference to an object A cannot be assigned in a field of an object B if object A's lifetime is less than object B's lifetime. If allowed, object A could disappear and leave object B with a dangling pointer.

In our robot example, the system state is stored in immortal memory, data that must be communicated between handlers is stored in mission memory, and all other data is stored in private memory areas or on the handlers stack. This is a typical data design for valid and efficient SCJ programs.

2.4 The election details

For the election, each robot has the following associated information:

- Id: this uniquely identifies the robot and its IP address;
- Petition: a unique ranking among the robots that indicates how badly the robot wants to be the leader. The robot with the highest petition that is online is elected the leader;
- Status: an indication of whether the robot is the leader, a follower or undecided.

The application maintains in immortal memory an array with this information; it has one position for each robot. The array includes a logical timestamp that indicates the freshness of the state of the information received. The timestamp is incremented by the `Elector` and `Detector` handlers in every period.

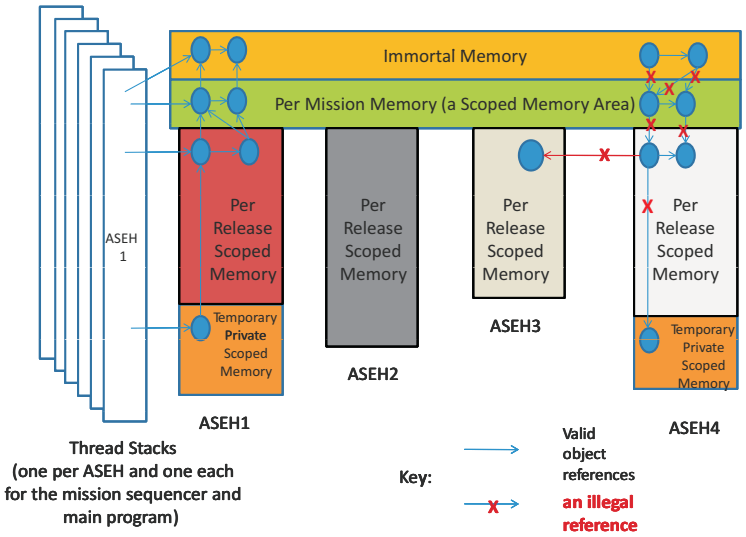

Fig. 6. Memory Hierarchy of an SCJ Program

The `Sender` PEH broadcasts its robot's state every 500 milliseconds. The `Receiver` PEH receives as many messages as are available every 500 milliseconds. The `Elector` reviews the global state each 3000 milliseconds and decides whether its robot should elect itself as leader or become a follower. When the global state shows that all robots have decided and that there is only one leader, the mission terminates, and the `Dancing` mission is executed. During this mission, the `Detector` continues to monitor the global state every 3000 milliseconds. If the status quo is changed, then the `Dance` mission terminates and a new `Election` mission is executed. The `Dancer` PEH sends and receives dancing commands every 3000 milliseconds depending on whether it is the leader or the follower.

The full code for the leadership-election example that is presented here is at `www.cs.york.ac.uk/circus/hijac/code/LeaderElectionPaper.zip`. The pseudo code shown Figure 7 summarises the overall optimized approach.

Essentially there are two parallel activities (the Communicator – lines 4 - 9 and the Elector – line 9-19). The Communicator is responsible for periodically sending out a robot's state to its neighbouring robots, and then acquiring their current states. The Elector is aperiodic and is released after the Communicator has acquired all its neighbours states. It analyses the global state and if a leader has been globally agreed, it requests termination of the current mission. Otherwise, if its petition is the highest among all the robots, then it makes a claim to be the leader, or then it settles for being a follower.

In the next section, we present a formal specification for the robot application. We develop the optimized approach.

```
1  For each robot:
2
3  IN PARALLEL
4    Communicator:
5      PERIODIC
6        IN SEQUENCE
7          broadcast my state to neighbours
8          get state from each of my neighbours
9    Elector:
10     APERIODIC
11       IN SEQUENCE
12         analyse global state
13         IF no leader agreed
14           IF I have highest petition
15             Claim leadership
16           ELSE
17             Claim follower
18         ELSE
19           Request mission termination as leader has been established
```

Fig. 7. Pseudo-code for the optimised leadership-election algorithm

3 *Circus*

The search for increasing levels of abstraction is a key feature in software engineering, and, particularly, in language design. For example, the concept of class embeds a notion of an abstract data type and allows a structured modelling of real-world entities, capturing both their static and dynamic properties. The notion of process abstracts from low-level control structures, allowing a system architecture to be decomposed into cooperative and active components.

Despite the complementary nature of constructs for describing data and control behaviour, most programming languages focus only on one or the other aspect. Java is no exception: it offers (abstract) classes, interfaces, and packages; in contrast, only the low-level notion of threads is available. There are exceptions like Ada [23], whose design has clearly addressed abstract data and control behaviour (with packages and tasks), but even so there are several limitations; for example, a package is not a first-class value.

The design of specification languages has followed a similar trend, with state-based and property-oriented formalisms concentrating on high-level data constructs [5], and process algebras exploring control mechanisms. A current and active research topic is the integration of notations to achieve the benefits of both abstract data and control behaviour [20, 44]. *Circus* is one of these integrated notations, whose focus is refinement (to code).

In this section, we present a combination of Z [49] and CSP [41], traditional languages for data modelling and a process algebra. Their combination in a language called *Circus* supports the specification of both data and behavioural

aspects of concurrent systems, and a development technique. Such a combination has obvious advantages: Z is good at describing rich information structures using a predicative style (based on invariants, and pre and postconditions), and CSP is good at describing behavioural patterns of communication and synchronisation.

In *Circus*, Z constructs can be combined with executable commands, like assignments, conditionals, and loops. Reactive behaviour, including communication, parallelism, and choice, is defined with the use of CSP constructs.

In this section, we give an overview of *Circus*: we describe the structure of the *Circus* models, and explain how Z and CSP are combined. Beforehand, we say a bit more about Z (Section 3.1) and CSP themselves (Section 3.2). As an example, we provide a model of the leadership-election protocol (Section 3.3).

3.1 Z

A system specification in Z consists basically of a definition for a state and a collection of operations. The state is composed by variables representing information used and recorded in the system. The operations take inputs and produce outputs, and possibly update the state. Both the state and the operations are defined by schemas, which group variable declarations and a predicate.

Example 1. As a very simple example, we consider a system presented in [35] that calculates the mean of a sequence of numbers. The state of this system has only one component: the sequence seq of integers input so far.

Calculator

$$seq : \operatorname{seq} \mathbb{Z}$$

The state definition gives it a name, *Calculator*, and declares its component.

This system has three operations. The first, *Init*, initialises the state.

Init

$$Calculator'$$

$$seq' = \langle\rangle$$

The reference to *Calculator* indicates that this is an operation over that state. In an operation definition, we can refer to seq and to seq'. The former refers to the value of the state component before the operation, and the latter, to the value after the operation. The dash decoration on *Calculator*, however, indicates that *Init*, as an initialisation, can refer to seq' only. The predicate of *Init* specifies that, after the initialisation, the value of seq is the empty sequence.

The second operation, *Enter*, records an input value in the sequence.

Enter

$$\Delta Calculator$$
$$n? : \mathbb{Z}$$

$$seq' = seq \frown \langle n? \rangle$$

The Δ in front of *Calculator* indicates that *Enter* changes the state. The variable

$n?$ represents an input of *Enter*: the number to be inserted. The predicate defines that the new sequence of numbers can be obtained by inserting the input at the end of the existing sequence; $\frown$ is the concatenation operator.

The last operation, *Mean*, calculates the mean of the numbers input so far.

Mean

$$\Xi Calculator$$
$$m! : \mathbb{Z}$$

$$seq \neq \langle\rangle$$
$$m! = (\Sigma seq)\ div\ (\#\, seq)$$

The Ξ indicates that *Mean* does not change the state. The output is represented by the variable $m!$. The specification requires that the sequence *seq* is non-empty. This is a precondition for this operation: even though *Mean* can be executed when this condition is not satisfied, its result is not predictable in such a situation. If, however, the precondition is satisfied, the specification requires the output to be the sum of the elements in the sequence divided by its size. The Σ operator is not directly available in Z, but can be easily specified. □

3.2 CSP

In CSP, a system and its components are modelled by processes that interact with their environment not via inputs and outputs, like in Z, but via synchronisations that characterise events. These events, however, can model exchange of data, as well as simple interactions of interest. In the description of a CSP process, a first element of interest is the set of events in which it can participate; the definition of an event simply gives it a name.

Example 2. A process that controls a revolving door can be characterised in terms of the events *step-in*, *revolve*, *step-out*, and *stop*; the first denotes the arrival of someone in the area around the door, the second represents the start of the revolving movement, *step-out* is the event that captures the exit of a person from the door area, and, finally, *stop* occurs when the door stops moving.

In the specification of the door, inputs and outputs are not a concern; the relevant issue is the form in which the door interacts with the environment: the people that use the door. Below, we present the definition of processes $Door(i)$, where i is the number of people already using the door.

If there are no people using the door, the only possible event is for someone to arrive; afterwards, the door starts revolving, and proceeds to behave as a door that is being used by one person. In the specification of $Door(0)$, we use the prefix operator $a \rightarrow P$, which gives the unique event a in which the process is prepared to engage, and a process P that characterises its behaviour afterwards.

$$Door(0) = \textit{step-in} \rightarrow \textit{revolve} \rightarrow Door(1)$$

We use prefixing twice: first, the door is prepared to record the event *step-in*, then the only possible event is *revolve*, before the door behaves as $Door(1)$.

If there is one person using the door, then either someone else arrives or that person leaves. We use the choice operator $P \Box Q$ to specify this behaviour: this process is prepared to behave as P or Q; the choice is made by the environment.

$$Door(1) = step\text{-}in \rightarrow Door(2) \Box step\text{-}out \rightarrow stop \rightarrow Door(0)$$

If the event *step-in* takes place, then the door behaves as a door used by two people. If *step-out* takes place, the only possible event is *stop*, and then we have the behaviour of $Door(0)$ again. The definitions of $Door(0)$ and $Door(1)$ are mutually recursive; the use of recursion is very common in CSP.

Doors with two or more people are similar; for $n > 1$, $Door(n)$ is below.

$$Door(n) = step\text{-}in \rightarrow Door(n+1) \Box step\text{-}out \rightarrow Door(n-1)$$

If someone steps in an door with n people, for n greater than 1, then we have the behaviour of a door with $n+1$ people. If someone steps out, then the behaviour is that of a door with $n-1$ people.

In a big building, we usually have a number of these doors. They work in parallel, but independently. We define a process entrance with m doors as follows.

$$Entrance = \left|\left|\left| i : 1 \,..\, m \bullet \right.\right.\right. \\ Door(0)[step\text{-}in.i, revolve.i, step\text{-}out.i, stop.i / step\text{-}in, revolve, step\text{-}out, stop]$$

In this process we have m copies of $Door(0)$ recording events *step-in.i*, *revolve.i*, *step-out.i*, *stop.i*, for i between 1 and m. The set of events of *Entrance* comprise all of the $m \times 4$ events: 4 for each of the m doors. The parallel operator $|||$ is for an interleaving composition, where the parallel processes do not interact with each other. Above we use the iterated form ($\left|\left|\left|\right.\right.\right.$) of this operator.

A polite door contains an additional component: a process that detects that someone has arrived and welcomes this person with a greeting message. This *Polite* process can be specified as follows.

$$Polite = step\text{-}in \rightarrow welcome \rightarrow Polite$$

The extra event *welcome* signals the play of the greeting. The polite door can be characterised by the parallel execution of the standard $Door(0)$ and *Polite*.

$$PDoor = Door(0) \,[\![\{step\text{-}in\}]\!]\, Polite$$

In this parallel process ($[\![...]\!]$), there is interaction between the two components; they are not independent as in the previous example. Since *step-in* is an event of both *Door* and *Polite*, they synchronise on this event. Every time someone steps in, $Door(0)$ and *Polite* act jointly; from the point of view of *PDoor*, just one event occurs. $\Box$

As already explained, *Circus* includes both Z and CSP constructs. We present *Circus* next via our running example: the leadership-election protocol.

3.3 Leadership election in *Circus*

A *Circus* model is formed by a sequence of paragraphs that specify types, constants, functions, and, crucially, processes. Like in CSP, processes define systems and their components. Their definitions use the types, constants and functions defined globally, as well Z and CSP constructs.

In our example, we first define two types: $DEVICEID$ and $STATUS$.

$$DEVICEID == \mathbb{N}$$
$$STATUS ::= leader \mid follower \mid undecided \mid off$$

These types are sets that contain the valid identifiers for devices, and constants *leader*, *follower*, *undecided*, and *off* that represent the status of a device. For simplicity, we define the identifiers of the devices to be natural numbers. We need to use an ordered set, because the election conditions use the order of the devices to resolve ties. We could make this more abstract by requiring only a set of identifiers with a total order, but it is simpler to use the natural numbers.

We also have some global constants. UP_LMT is the maximum value for the petition of a device. P is the period of the protocol. $TIMEOUT$ is how long a device waits for information from a neighbour before giving up, and marking it as offline. ID and OD are the input and output deadlines. The set *devices* contains all the identifiers of the devices in the network: a subset of $DEVICEID$.

$$UP\_LMT : \mathbb{N}$$
$$P, TIMEOUT, ID, OD : \mathbb{N}$$
$$devices : \mathbb{P}\, DEVICEID$$
$$\hline$$
$$TIMEOUT \leq P \land ID \leq P \land OD \leq P \land \# devices > 0$$

A constraint ensures that the timeout, input and output deadlines are all less than or equal to the period P. Moreover, there must be at least one device.

The process that defines the functional requirements of the protocol is called $ABReqsLE$. It is introduced below.

process $ABReqsLE \mathrel{\widehat=}$ **begin**

In its body, the first few paragraphs define the state space.

The state of a device includes its identifier id, *status*, and *petition*.

DeviceState

$$id : DEVICEID$$
$$status : STATUS$$
$$petition : \mathbb{N}$$
$$\hline$$
$$id \in devices$$
$$petition \leq UP\_LMT$$

Constraints on the type ensure that id is for a device in the network, and the *petition* is valid, that is, below the limit defined by UP_LMT.

To execute the election protocol, a device needs additional information, captured in records of the type *ElectionState*. In addition to the state components in *DeviceState* described above, *ElectionState* records the highest petition of a device claiming to be a leader as well as its identifier: *highest* and *highestid* in the schema, that is, record type, *Highest* below.

$$Highest == [highest : \mathbb{N};\ highestid : DEVICEID]$$

The schema *ElectionState* includes all the components of *DeviceState* and *Highest*. It also records the number *nLeaders* of leaders in the network, the index i of the device currently communicating (lines 7-8 in Figure 7) in a sequence *nodes* that records information about individual devices, and a function *next* that gives the index (in *nodes*) of the device considered in the next cycle.

ElectionState

$$\begin{array}{l} DeviceState \\ Highest \\ nLeaders : \mathbb{N} \\ i : \mathbb{N};\ nodes : \operatorname{seq} DeviceState;\ next : \mathbb{N} \to \mathbb{N}^+ \\ \hline i \in \operatorname{dom} nodes \\ \forall\, n : \mathbb{N}^+ \bullet next\ n = ((n-1) \bmod (\#\, nodes)) + 1 \\ devices = \{d : \operatorname{ran} nodes \bullet d.id\} \\ \#\, nodes = \#\, devices \\ \theta\, DeviceState \in \operatorname{ran} nodes \end{array}$$

The invariant states that the index i is an index for *nodes*. Moreover, the function *next* identifies indices of *nodes* in a way that iterates through this sequence, circling back to the beginning at the end.

The set *devices* includes the identifiers in the range of *nodes*. By requiring that the size of this set and the size of *nodes* are equal, we ensure that the range of *nodes* does not include two records for the same device identifier. Finally, the invariant establishes that the current device, identified by a record θ *DeviceState*, containing the fields of *DeviceState* in *ElectionState*, is also in the range of *nodes*.

Unlike CSP processes, a *Circus* process has a state defined by a schema. It *ElectionState* that defines the state of the process *ABREqsLE* being specified.

$$\textbf{state}\ st == ElectionState$$

The next few paragraphs define data operations. Initially, there is no leader, the highest petition is 0 and the index i is that for the device itself.

InitElectionState

$$\begin{array}{l} ElectionState' \\ \hline nLeaders' = 0 \land highest' = 0 \land (nodes'\ i').id = id \end{array}$$

This means that when all devices are in a network, in the first step of the protocol, they all broadcast their status to the others.

When a status $valC?$ and petition $valP?$ is received from a device whose identifier is $idDev?$, we can update the fields of *Highest* if the device claims to be a *leader* and the petition $valP?$ is higher than previously recorded, or if it is the same and the identifier $idDev?$ is greater than the previous identifier.

UpdateHighest

$$\Delta Highest$$
$$idDev? : ID;\ valC? : STATUS;\ valP? : \mathbb{N}$$

$$valC? = leader$$
$$valP? > highest \lor (valP? = highest \land idDev? > highestid)$$
$$valP? > highest \Rightarrow highest' = valP? \land highestid' = idDev?$$
$$valP? = highest \Rightarrow highest' = valP? \land highestid' = idDev?$$

This operation is partial; it should only be used when an update to *highest* or *highestid* is needed as indicated. This is ensured by its use in *UpdateDevice*, shown below, which also modifies the remaining components of *ElectionState*.

UpdateDevice

$$\Delta ElectionState$$
$$idDev? : ID;\ valC? : STATUS;\ valP? : \mathbb{N}$$

$$\textbf{let}\ d == (\mu\ x : \operatorname{ran} nodes \mid x.id = idDev? \bullet x) \bullet$$
$$\left(\begin{array}{l} d.status = leader \land valC? \neq leader \land nLeaders' = nLeaders - 1 \\ \lor \\ d.status \neq leader \land valC? = leader \land nLeaders' = nLeaders + 1 \\ \lor \\ d.status = leader \land valC? = leader \land nLeaders' = nLeaders \\ \lor \\ d.status \neq leader \land valC? \neq leader \land nLeaders' = nLeaders \end{array}\right)$$
$$nodes' = nodes \oplus \{(nodes\ ^\sim\ d)) \mapsto$$
$$\qquad \langle\!| id == idDev?, status == valC?, petition == valP? |\!\rangle\}$$
$$\theta\, DeviceState = \theta\, DeviceState' \land next' = next \land i' = i$$
$$UpdateHighest \lor [\Xi Highest \mid \neg\ (\textbf{pre}\ UpdateHighest)]$$

UpdateDevice takes the information d on $idDev?$ in *nodes* using the definite description operator μ and updates the number of leaders *nLeaders* depending on the previous $d.status$ and current value $valC?$ of its status. It also overrides ($\oplus$) *nodes* with the newly received information; with $nodes\ ^\sim\ d$, we get the index of d. *UpdateDevice* also leaves the components of *DeviceState*, the function *next* and index i unchanged, and updates the components of *Highest* using *UpdateHighest*, if necessary, as captured by the precondition **pre** *UpdateHighest* of this operation.

In a *Circus* process, the Z data operations can be combined to define actions. In the definition of actions, we can also use CSP constructs.

The action *BReq*1 specifies the communications the protocol. It identifies which device ($(nodes\ i).id$) is to be considered (recorded by i). If it is id itself,

it broadcasts its state using the action *Broacast*.

$$
\begin{array}{l}
BReq1 \mathrel{\widehat{=}} \\
\quad \textbf{if} (nodes\ i).id = id \longrightarrow Broadcast(id, status, petition) \\
\quad [] (nodes\ i).id \neq id \longrightarrow \\
\qquad \left(\begin{array}{l} receive.(nodes\ i).id?valC?valP \longrightarrow \\ \quad UpdateDevice((nodes\ i).id, valC, valP) \\ \Box \\ timeout \longrightarrow UpdateOff((nodes\ i).id) \end{array} \right); \\
\qquad \textbf{if}\ status = undecided \longrightarrow \\
\qquad\quad \left(\begin{array}{l} \textbf{if}\ nLeader > 0 \longrightarrow status := follower \\ [] nLeader \leq 0 \longrightarrow \\ \quad \left(\begin{array}{l} \textbf{if}\ id = next\ i \longrightarrow \\ \quad \left(\begin{array}{l} \textbf{if} \left(\begin{array}{l} (highest = petition \wedge highestid < id) \vee \\ highest < petition \end{array} \right) \longrightarrow \\ \quad\ status := leader \\ [] \neg \left(\begin{array}{l} (highest = petition \wedge highestid < id) \vee \\ highest < petition \end{array} \right) \longrightarrow \\ \quad\ status := follower \\ \textbf{fi} \end{array} \right) \\ [] id \neq next\ i \longrightarrow status := undecided \\ \textbf{fi} \end{array} \right) \\ \textbf{fi} \end{array} \right) \\
\qquad [] status = leader \longrightarrow \\
\qquad\quad \left(\begin{array}{l} \textbf{if}\ nLeader > 0 \longrightarrow status := undecided \\ [] nLeader \leq 0 \longrightarrow \\ \quad \left(\begin{array}{l} \textbf{if}\ id = next\ i \longrightarrow \\ \quad\ petition := min(UP\_LMT, petition + 1);\ status := leader \\ [] id \neq next\ i \longrightarrow status := leader \\ \textbf{fi} \end{array} \right) \\ \textbf{fi} \end{array} \right) \\
\qquad [] status = follower \longrightarrow \\
\qquad\quad \left(\begin{array}{l} \textbf{if}\ nLeader = 0 \longrightarrow status := undecided \\ [] nLeader \neq 0 \longrightarrow status := follower \\ \textbf{fi} \end{array} \right) \\
\qquad \textbf{fi} \\
\quad \textbf{fi}
\end{array}
$$

If the device under consideration ($(nodes\ i)$) is not *id* itself, the protocol waits for information about the device on the channel *receive* and updates the state using the operation *UpdateDevice* shown above, or for a timeout on the channel *timeout*, in which case it udpates the state using the action *UpdateOff*. The CSP operator $\Box$ offers an external choice between these actions. The next section gives a concise overview of basic features of the CSP notation. After updating the state with the received information, the device decides its own status based on its previous status (**if** $status = undecided \longrightarrow \ldots$ **fi**). We notice that the operation *UpdateDevice* does not change the device's own state.

If the device is undecided, its new status depends on the number of leaders. If there are leaders, it becomes a follower (assignment $status := follower$), otherwise, the protocols considers the device $next\ i$. If it is id itself, then it compares its petition to the highest petition and becomes a leader or follower depending on whether or not its petition (or identifier) is greater than the highest petition (or identifier) recorded. If the next device is not id, the status remains undecided.

If id's status is $leader$, and there are other leaders (besides itself as $nLeaders$ only refer to leaders among the neighbours), then it becomes undecided. Otherwise, it remains a leader and increments its own petition (up to the maximum UP_LMT) if it is the next device to be considered. Finally, if id is a follower and there are no leaders, it becomes undecided. Otherwise, it stays a follower.

The actions used in $BReq1$ above are defined next.

$$Broadcast \mathrel{\widehat=} \mathbf{val}\ id : DEVICEID;\ status : STATUS;\ petition : \mathbb{N} \bullet \Big|\Big|\Big|\ i : devices \setminus \{id\} \bullet send.id.i.status.petition \longrightarrow \mathbf{skip}$$

$Broadcast$ sends in interleaving ($|||$) the $status$ and $petition$ of the device to each of the neighbouring devices using the channel $send$. These are the devices d in nodes whose identifier is not id itself. The parameters of $send$ are the identifiers of the source and target devices, the status and petition values. The protocol assumes an asynchronous bus, so this communication does not deadlock even if the target device is unavailable. Since communications in *Circus* are synchronous, the model requires the definition of the bus (omitted here).

$$UpdateOff \mathrel{\widehat=} \mathbf{val}\ idDev : DEVICEID \bullet \mathbf{var}\ valC : STATUS;\ valP : \mathbb{N} \bullet valC, valP := off, 0;\ UpdateDevice$$

$UpdateOff$ uses the schema operation $UpdateDevice$ to update the state of the process. It sets the $status$ and $petition$ to off and 0, before updating the state.

As already said, the process $ABReqsLE$ describes the behavioural requirements for the protocol on a single device. Its behaviour is defined below by the main action. It initialises the state using the schema operation $InitElectionState$ and starts a recursive action ($\mu\, X \bullet ...X$), which at each step executes the action $BReq1$ and updates the index i using the function $next$.

$$\bullet\ InitElectionState;\ (\mu\, X \bullet BReq1;\ i := next\ i;\ X)$$
end

The timing requirements are specified in a separate process $ATReqsLE$ shown below. Its main behaviour is also defined by a recursion, but at each iteration it offers a choice between receiving information on $receive$, indicating a timeout using the channel $timeout$, or sending information to all neighbours in interleaving through the channel $send$. The particular values communicated through these channels are irrelevant here; they are defined in $ABReqsLE$ specified above. Here, on the other hand, the time in which these events occur is important.

Communications on $send$ and $receive$ must start within OD and ID time units as defined by the deadline operator $\blacktriangleleft$. OD and ID are global constants

previously defined. All communications lead to an action that potentially lets time pass until the end of a period P. The *Circus* statement $\mathbf{wait}\ 0\,..\,(P-t)$ is a nondeterministic choice of a delay of 0 up to $P-t$ time units. In the example, t is the time between the communication being offered and actually taking place. In each case, that is recorded via the $@$ operator, like in $timeout@t$.

$$
\begin{array}{l}
\mathbf{process}\ ATReqsLE \mathrel{\widehat=} \mathbf{begin} \\
\quad TReq1 \mathrel{\widehat=} (TReqCycle \blacktriangleright P \mid\mid\mid \mathbf{wait}\ P);\ \ TReq1 \\
\quad TReqCycle \mathrel{\widehat=} \\
\quad (\Big|\Big|\Big|\ i : 1\,..\,\#\,devices - 1 \bullet (send?x?y?z?w@t \longrightarrow \mathbf{wait}\ 0\,..\,(P-t)) \blacktriangleleft OD) \\
\quad \Box \\
\quad (receive?x?y?z?w@t \longrightarrow \mathbf{wait}\ 0\,..\,(P-t)) \blacktriangleleft ID \\
\quad \Box \\
\quad (timeout@t \longrightarrow \mathbf{wait}\ 0\,..\,(P-t)) \blacktriangleleft P \\
\quad \bullet\ TReq1 \\
\mathbf{end}
\end{array}
$$

Finally, the overall specification is given by the process *LeaderElection*.

$$
\begin{array}{l}
\mathbf{process}\ LeaderElection \mathrel{\widehat=} \\
\quad (ABReqsLE \llbracket \{\!|\, send, receive, timeout \,|\!\} \rrbracket ATReqsLE) \setminus \{\!|\, timeout \,|\!\}
\end{array}
$$

It is the parallel composition ($\llbracket ... \rrbracket$) of the behavioural and timing processes, synchronising on the external channels *send* and *receive*, and on *timeout*, which is hidden ($\setminus$) and, therefore, internal to *LeaderElection*. We note that parallelism is used not to define a parallel architecture for a design, but to define a conjunction of requirements: the behavioural requirements of *ABReqsLE* and the timing requirements of *ATReqsLE*. Synchronisation ensures that the communications transmit values as defined in *ABReqsLE* within the times defined by *ATReqsLE*.

In Section 5, we explain how we can refine this abstract specification of the leadership-election protocol to obtain a model of an SCJ program. Beforehand, in the next section, we say more about refinement and the *Circus* approach.

4 Algebraic refinement

Circus distinguishes itself in that it is aimed at the (calculational) refinement of specifications. Besides Z and CSP, *Circus* also includes specification constructs usually found in refinement calculi [34, 2, 36] and Dijkstra's language of guarded commands [16], a simple imperative language with nondeterminism. The extra constructs that we use here are familiar: assignments, conditionals, and so on.

As a refinement language, *Circus* is a unified programming language, in which we can write specifications (in a combination of Z, Morgan's specification statements, and CSP), designs (using choice and concurrency constructs of CSP, for instance), and programs, and can relate all these kinds of artefacts to each other via refinement. Data refinement, failures-divergences refinement, and refinement to code (as a special case of data refinement) can all be carried out using *Circus*.

The notion of refinement captures the essence of the daily tasks of software engineers, who design systems based on their specifications, and programmers, who implement these designs. In both cases, the main objective is the construction of systems and programs in accordance with their specifications. The final product, above all, should be, or has to be, correct.

Refinement is the relationship that holds between a specification and its correct designs and implementations. Formal methods of program development are based on this notion, as are all other methods in some way. A formal technique, however, goes further since refinement of an initial specification to obtain an acceptable implementation is the primary aim. Acceptability may be judged, for instance, in terms of performance, but the guarantee provided is that the specification and the implementation are related by refinement.

In this section, at first we present the classical notions of refinement. Initially, refinement was extensively studied in the context of sequential programs [26, 27, 4], where the concern is the relation between inputs and outputs. It was identified that there are basically two ways of refining a specification. The first is the introduction and transformation of programming and control structures, like assignments, conditions, and loops. This is called algorithmic refinement.

The second form of refinement is related to the data structures used in the program. Systems may be specified in terms of data types that are appropriate to describe properties of the application domain, without, for example, any considerations related to efficiency. During design, however, ingenious decisions usually involve the introduction of data structures that are available in the programming language and make the computation tasks easier or faster. The change of data representation involved in this task is called data refinement [21, 24, 25].

For an object-oriented language like Java, there are new concerns related to the presence of classes and their use as data types [28]. Refining a class is very much like refining a data structure in a traditional imperative setting. Nevertheless, due to the presence of, for instance, type tests, type casts, and dynamic binding, new techniques are needed. Type tests and casts may be used to distinguish objects of different classes. Even if we have two classes with the same fields and methods, but different names, type tests (and casts) can be used to distinguish objects of these classes. Dynamic binding means that a method call may lead to the execution of several different pieces of code. To ensure correctness, we need to consider all possibilities. Pointers are also a challenge.

For concurrent reactive systems like those that we can specify in *Circus* and program in SCJ, the main concern is their interactions with other systems and the environment [40]. Like we have discussed in the previous section, functionality is not characterised by a relation between inputs and outputs, but by the ways in which communications and synchronisations can take place; inputs and outputs are particular forms of communications. Termination is not a strong requirement as systems that run indefinitely, but continuously interact with their environments in a useful way, are very much of interest. Refinement, in this context, has to consider the behaviour of the systems in each of their interactions.

Refinement of imperative programs, including data and algorithmic refinement is the subject of Section 4.1; there we use Z as a concrete notation. Refinement of concurrent reactive systems is addressed in Section 4.2.

4.1 Basic Concepts

A formal specification is the starting point of any formal development method. Correctness is a relative notion: a program is correct or wrong depending on whether it implements its specification or not; the specification is the basis for the evaluation. To guarantee correctness, we need a formal specification.

Specifying a system is the first step to get its implementation right. A formal development method takes such a specification as a basis to produce a correct implementation: one that refines the specification.

Refinement is based on the idea that a specification is a contract between the client and the developer. The client cannot complain if, when executed in situations that satisfy their preconditions, the operations of the implementation produce outputs that satisfy the properties stated in the specification. In this case we have a correct implementation.

Data refinement Our first opportunity for refinement typically comes in the change of representation of state components. As said before, a Z specification describes the relation between inputs and outputs when the system is initialised and a sequence of operations is executed. The values of the state components, however, are not visible. Similarly, in *Circus*, the state of a process, which is defined in Z, is not visible. We can only observe the behaviour of a process via its interactions with its environment, which use the channels that are in scope.

In Example 1, for instance, we use a sequence to record the numbers input; this is a natural way of describing the system. It is less space-consuming, however, to record just the sum and the number of integers input. If the operations are updated accordingly, it is perfectly valid to change the representation of the state in this way. This sort of change is known as data refinement; the original specification is regarded as abstract and the new specification, as concrete.

The other opportunity for refinement is the development of implementations for the operations; this is the subject of the next section, where we discuss algorithmic refinement. Since these implementations are affected by changes in the state, we consider data refinement first. At this stage, we change the operations only to adapt them to the new data types. In Z, we write the concrete specification in the same style as that used for the abstract specification.

Example 3. The concrete state suggested above can be defined as follows.

CalculatorC

$$size, sum : \mathbb{Z}$$

There are two components: the *size* of the sequence input and its *sum*.

The new definition for the operations is as follows. The initialisation, $InitC$, records that no numbers have been input.

$$
\begin{array}{|l}
\hline
InitC \\
\hline
CalculatorC' \\
\hline
size' = 0 \land sum' = 0 \\
\hline
\end{array}
$$

The operation $EnterC$, which inputs a number, increments $size$ and updates sum by adding the input to it.

$$
\begin{array}{|l}
\hline
EnterC \\
\hline
\Delta CalculatorC' \\
n? : \mathbb{Z} \\
\hline
size' = size + 1 \land sum' = sum + n? \\
\hline
\end{array}
$$

The operation that calculates the mean has a much simpler specification.

$$
\begin{array}{|l}
\hline
MeanC \\
\hline
\Xi CalculatorC' \\
m! : \mathbb{Z} \\
\hline
size \neq 0 \\
m! = sum \ div \ size \\
\hline
\end{array}
$$

The needed values are readily available in sum and $size$. □

After providing the concrete specification, we have to prove that it satisfies the refinement property mentioned above: clients that agreed on the abstract specification cannot complain if they get an implementation of the concrete specification [17, 49, 38]. The most widely used technique to carry out such a proof is known as simulation. It involves the definition of a relation between the abstract and concrete states that specifies how the information in the abstract state is represented in the concrete state. In the context of Z, this relation is known as a retrieve relation and is specified using a schema.

There are, actually, two simulation techniques that can be applied: forwards (or downwards) simulation and backwards (or upwards) simulation. Here, we concentrate on the forwards simulation technique, as it is often enough in practice. Upwards simulation is a similar technique. (The difference lies in the way it handles nondeterminism in data operations.)

For our example, the appropriate retrieve relation can be specified as follows.

$$
\begin{array}{|l}
\hline
Retrieve \\
\hline
Calculator \\
CalculatorC \\
\hline
size = \# seq \land sum = \Sigma seq \\
\hline
\end{array}
$$

The inclusion of the abstract and of the concrete state definitions $Calculator$

and *CalculatorC* reflects the fact that we are specifying a relation between them. Basically, a concrete state is related to an abstract state when the value of *size* is indeed the size of *seq* and *sum* is the sum of the numbers in this sequence.

Given the retrieve relation, we need to check first that the initialisation is adequate: given an initial concrete state, there is a corresponding abstract initial state. In general, if A and C are the schemas that specify the abstract and concrete states, AI and CI are the corresponding initialisation operations, and R is the retrieve relation, then we have to prove the following.

$$\forall C' \bullet CI \Rightarrow \exists A' \bullet AI \land R' \qquad \text{(initialisation)}$$

The use of schemas in predicates is common in Z. We are required to prove that, for all values that the components of the concrete state may assume, if these values are those of an initial state, then there are initial values that can be assigned to the abstract state components that are related to those of the concrete initial state. The use of C', A', and R' is necessary because the predicates of CI and AI are written in terms of the dashed version of the state components.

For our data refinement, we are required to prove the following property.

$$\begin{array}{l} \forall size', sum' : \mathbb{Z} \bullet size' = 0 \land sum' = 0 \Rightarrow \\ \quad \exists seq' : \operatorname{seq} \mathbb{Z} \bullet seq' = \langle\rangle \land size' = \# seq' \land sum' = \Sigma seq' \end{array}$$

With two applications of a one-point rule we get $0 = \#\langle\rangle \land 0 = \Sigma\langle\rangle$, which is true as the size of and the sum of the elements of the empty sequence are 0. This reflects the fact that the initialisation of *CalculatorC* chooses values to *size* and *sum* that are in accordance with the initial value of *seq*. This is, of course, relative to the way in which we represent *seq* using *size* and *sum*.

We also need to prove that each of the operations CO of the concrete specification is in accordance with the specification of the corresponding operation AO of the abstract specification. We have to prove the following, where **pre** AO and **pre** CO refer to the precondition of the operations.

$$\forall A;\ C \bullet \mathbf{pre}\, AO \land R \Rightarrow \mathbf{pre}\, CO \qquad \text{(applicability)}$$
$$\forall A;\ C \bullet \mathbf{pre}\, AO \land R \Rightarrow (\forall C' \bullet CO \Rightarrow \exists A' \bullet AO \land R') \qquad \text{(correctness)}$$

When refining an operation, there are usually two separate concerns: its precondition and its effect, also known as postcondition. The precondition of an operation characterises the situations in which it behaves properly. The first proof obligation above, applicability, requires that, whenever the precondition of the abstract operation holds, the related concrete states satisfy the precondition of the concrete operation. So, this proof obligation requires that whenever the abstract operation behaves properly, so does the concrete operation.

In our example, the preconditions of *Enter* and *EnterC* are both true, therefore applicability is not interesting. For *Mean*, the precondition is $seq \neq \langle\rangle$. For *MeanC*, the precondition is $size \neq 0$. Applicability is as follows.

$$\begin{array}{l} \forall seq : \operatorname{seq} \mathbb{Z};\ size, sum : \mathbb{Z} \bullet \\ \quad seq \neq \langle\rangle \land size = \# seq \land sum = \Sigma seq \Rightarrow size \neq 0 \end{array}$$

This is also a simple proof-obligation: if *seq* is not empty, and *size* is its length,

then *size* is certainly different from 0.

The second proof-obligation, correctness, is related to the effect of the operations. First of all, we are only interested in the situations in which the precondition of the abstract operation holds; if it does not, then there are no requirements on the concrete operation. If it does, for all states resulting from the execution of the concrete operation in a related state, exists a related abstract state that could be obtained with the execution of the abstract operation.

For *Mean* and *MeanC*, correctness is as follows.

$$\forall\, Calculator;\ CalculatorC \bullet seq \neq \langle\rangle \land Retrieve \Rightarrow \\ (\forall\, CalculatorC' \bullet MeanC \Rightarrow \exists\, Calculator' \bullet Mean \land Retrieve')$$

Three applications of the one-point rule (and basic predicate calculus properties) reduces this predicate to true.

A special case of simulation that involves simpler proof obligations is that in which the retrieve relation is a total function from the concrete to the abstract state. Most proof-obligations generated in a refinement, however, are long, but simple, and a lot of help is provided by tools [31]. Data refinement can also be applied to variable blocks and to modules. As long as we have a structure for information hiding, this kind of change of representation is always possible.

Algorithmic refinement Once we have decided on the data types to be used in the program, we can proceed to work on the implementation of the operations. There are basically two approaches to refinement in general: verification and calculation. For data refinement, we have proposed a new specification and then proved that it is satisfactory: we verified the proposed refinement to be correct.

For algorithmic refinement, we can use a calculational approach [34, 2, 36]. In such techniques, the initial specification is the starting point for a sequence of transformations, each captured by a refinement law, to gradually transform the specification into a program. Because refinement is a transitive relation, this establishes that the initial specification is refined by the final program.

Each law captures a model transformation, which is the essence of the very popular model-based approach to design and programming. Distinctively, however, laws of refinement, guarantee that the transformations that they specify preserve the behaviour of the original program. For Z, such a refinement calculus has been presented in [7, 13, 9], and it is called ZRC. Its laws can also be used to transform Z operations defined in a *Circus* process.

The language of ZRC, as of all refinement calculi, can be used to write specifications, designs, which involve programming and specification constructs, and programs. Besides Z, this language includes assignments, conditionals, iterations, and procedures, among other constructs, like in *Circus*. In a design, we may have, for instance, a loop whose body is a schema. Specifications, designs, and programs are all regarded as programs; refinement is a relation between programs in this more general sense. The refinement relation is usually denoted by $\sqsubseteq$.

For a calculation, the differentiated roles of preconditions and postconditions are very relevant. Since schemas do not distinguish them, it can be convenient to

transform a schema into a so called specification statement. This is a construct that takes the form $w : [pre, post]$, where w is a list of variables, and pre and $post$ are predicates: the precondition and the postcondition. The list of variables is the frame, which determines the variables that can be changed.

For instance, *EnterC* can be specified by the specification statement $size, sum : [true, size' = size + 1 \land sum' = sum + n?]$, where the state components are explicitly listed as part of the frame. Similarly, *MeanC* can be specified as $m! : [size \neq 0, m! = sum \ div \ size]$. A refinement law in [13] explains how the conversion can take place. That work also includes laws that refine elaborate schema expressions to more refined programs; we have, for instance, a law to translate schema disjunctions into conditionals.

Refinement laws can be applied to transform a specification statement into a design or program; they embody common intuition about programming. We present *assigI*, a law that transforms a specification statement to an assignment.

Law *assigI* Assignment introduction
$w, v : [pre, post] \sqsubseteq v := e$ **provided** $pre \Rightarrow post[e/v'][\_/']$

Since the assignment $v := e$ potentially modifies the variable v, it must be in the frame of the specification statement. The proviso ensures that, when the precondition of the specification statement holds, its postcondition is satisfied if v' assumes the value e established by $v := e$. To put it more simply, it certifies that this assignment really implements the specification statement. The predicate $post[e/v'][\_/']$ is that obtained by substituting the expression e for v' and removing the dashes from the free variables of $post$.

With an application of *assigI*, we can transform the second specification statement presented above to $m! := sum \ div \ size$. The proviso generates the proof-obligation $size \neq 0 \Rightarrow sum \ div \ size = sum \ div \ size$, which follows by reflexivity of equality. The precondition is ignored; if it does not hold, the result of the assignment is not predictable. This is in accordance with the specification.

More interesting developments give rise to a sequence of law applications. Substantial examples can be found later on in Section 5. To give a flavour of the approach, we consider the law below, which splits a specification statement into another specification statement and an assignment.

Law *fassigI* Following assignment introduction
$w, v : [pre, post] \sqsubseteq w, v : [pre, post[e'/v']]; \ v := e$

This law introduces an assignment, which does not implement the specification statement by itself. We are still left with a specification before the assignment, which has the same precondition as the original one, but a modified postcondition. A substitution of e, with its free variables dashed, for v' records the fact that the assignment that follows makes the value of v to become e. With the substitution, the original postcondition is required to be established when v takes value e. This should be an easier task as illustrated next.

To refine $size, sum : [true, size' = size + 1 \land sum' = sum + n?]$, we can apply *fassigI* to introduce the assignment to *sum*. We are left with the program below.

$$size, sum : [true, size' = size + 1 \land sum' + n? = sum + n?];$$
$$sum := sum + n?$$

Since, the assignment already updates the *sum*, the new postcondition actually requires only that its value is not changed: $sum' + n? = sum + n?$ is equivalent to $sum' = sum$. This is an easier task. With an application of law *assigI* we can refine the remaining specification statement to $size := size + 1$.

Due to space restrictions, we cannot discuss ZRC or refinement calculi in more detail. Many interesting issues are involved in the development of code from specification using these techniques. An important point is that the sequence of laws applied defines the structure of the obtained program. In the simple examples above, we have just an assignment, or a sequence whose last component is an assignment. Conversely, if we have a program of a particular structure in mind, to a large extent, that defines the sequence of laws that need to be applied. So, we can use the calculational approach also to verify an existing program, by reconstructing the sequence of laws that can be used to generate it.

The refinement strategy presented in the next section can be applied in this spirit, to verify an existing SCJ program. As we discuss there, the constrained architecture of an SCJ program determines to some extent a particular sequence of *Circus* refinement laws that are useful to establish refinement. It is, therefore, possible to define a procedure (or strategy) to apply such laws.

We note, however, that the applications of the laws require additional information. For instance, if our target program has a sequence of statements ending in an assignment, we may decide to use the Law *fassigI* above to derive it. We, however, still need to define the particular variable that is to be assigned last, and the expression that is to be assigned to it. Specifically, in the application of *fassigI*, we need to define v and e; these are parameters of this law. If the target program is known in advance, it determines the right arguments for v and e. In this case, the application of the refinement law is fully determined.

Before presenting the refinement strategy for SCJ, we discuss refinement of processes, considering both CSP and *Circus* processes as examples.

4.2 Process refinement

Further challenges are present when we consider the development of concurrent programs: processes that interact with each other and an external environment. When developing a process, we are not only interested in the inputs and outputs, but also in each of the interactions in which the process may engage. As previously explained, inputs and outputs are forms of interaction in this context.

Specification, design, and implementation of processes has been carefully studied in the context of CSP [22, 40]. Like the languages of the refinement calculi discussed in Section 4.1, this is a unified language with an associated notion of refinement that can support the development of programs.

Refinement is based on the possible interactions of the processes. Basically, the interactions of the implementation process have to be interactions that could be performed by the specification process. For our example, we observe that it is not realistic to assume that an arbitrary number of people can use a door at the same time. A possible implementation of $Door(0)$ can be obtained if we consider that there is a limit max to this number of people and define $Door(max)$ as follows, where we assume $max > 2$.

$$Door(max) = step\text{-}out \rightarrow Door(max - 1)$$

When the maximum number of people is reached, the door is not prepared to accept the arrival of any further people. The only event enabled is *step-out*.

We observe that the number of people using a door is part of the state of *Door* and is not visible to the environment. In *Circus* and CSP, each process encapsulates its state information; interaction between the processes is achieved through events. Refinement, as said above, is concerned with these events.

On the other hand, since the state is hidden, we can consider data refinement. In the case of *Circus*, since the state and its data operations are defined using Z, the simulation technique adopted in Z can be used to data refine *Circus* processes. In CSP, the state is defined by parameters and the data model uses a functional language, so we have a simpler set up. For instance, we could use the negative integers to represent the number of people using a door, as shown below.

$$DoorN(0) = step\text{-}in \rightarrow revolve \rightarrow DoorN(-1)$$

$$DoorN(-1) = step\text{-}in \rightarrow DoorN(-2) \,\Box\, step\text{-}out \rightarrow stop \rightarrow DoorN(0)$$

$$\begin{aligned} DoorN(-n) = \; & step\text{-}in \rightarrow DoorN(-n-1) \\ & \Box \\ & step\text{-}out \rightarrow stop \rightarrow DoorN(-n+1) \quad \text{if } -n < -1 \end{aligned}$$

The processes $Door(0)$ and $DoorN(0)$ are equivalent. This sort of refinement, however, has not been the interest of the CSP community as the data language of CSP is very simple. The main concern is really interaction.

A further concern involved in the refinement of concurrent processes is related to the events in which a process may refuse to engage, and to the sequence of events that may lead to a divergent process. For instance, the specification of the door is a process that does not refuse the arrival of people in any circumstance; the implementation, on the other hand, may refuse this event if the door is full. From this point of view, it is not really a proper implementation.

Due to space restrictions, we do not discuss this any further. We note, however, that refinement in CSP and *Circus* ensures that safety and liveness 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, we note that we use *Circus Time* in our work for SCJ. Refinement in *Circus Time* also ensures preservation of time properties. This requires that the deadlines and budgets defined in the specification are enforced by the deadlines and budgets defined for the components of the program.

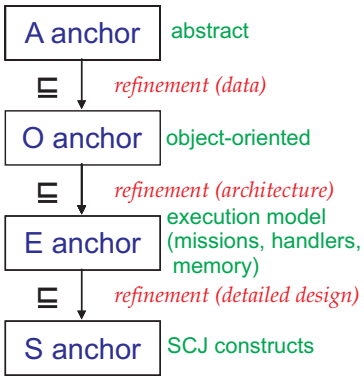

Fig. 8. Our approach to development and verification

5 Refining from *Circus* to SCJ

In this section, we describe the steps of our refinement approach.

In our strategy, refinement is carried out in three main steps, each characterised by an anchor: a *Circus* model written in a particular subset of *Circus* and following a particular pattern. Besides defining a target for a model transformation, an anchor captures a significant aspect of an SCJ program development: abstract specification, object-oriented design, missions, and SCJ infrastructure. Figure 8 shows the four Anchors: A, O, E, and S. The objective is to guarantee that the anchors are related by refinement.

The first refinement 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. Finally, the third step, produces the S anchor, and tackles the correctness of the algorithms implemented. It also describes the sequence of missions and parallelism of handlers in the E anchor in terms of SCJ constructs.

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.

For the leadership-election protocol, for example, the *Circus* model described in Section 3 is the A anchor. Below, Sections 5.1 to 5.3 describe the phases of each of the three refinement steps, and their stages.

5.1 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

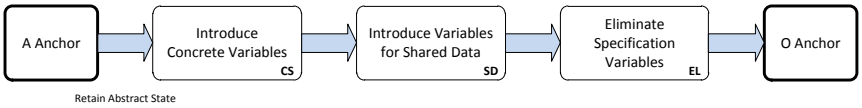

Fig. 9. Overview of the strategy for the Anchor O Step

and objects. The object-oriented constructs employed are those of *OhCircus*, which basically includes the possibility to define types via classes. The design of *OhCircus* is inspired by the Java approach to inheritance.

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. As explained in Section 4, data refinement only replaces or adds state components to the model. The types of the concrete components may be specified by *OhCircus* classes, but creation and allocation of objects are not considered yet. The structure of the actions is not changed.

Although in a data refinement particular algorithms are not considered, 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 9 describes our proposed strategy for this step. We take inspiration from Morgan's auxiliary variables technique [33] 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.

Automation is restricted here, since data refinement embeds design decisions related to the way in which data is to be efficiently represented and shared in the program. On the other hand, once that creative design is carried out, as discussed, it may be possible to calculate the specification of the concrete model, if there is a functional relation between the concrete and the abstract states.

The phases address the following concerns: (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). In all phases, including EL, we carry out a data refinement using simulation. 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.

For the leadership-election protocol, this step is not needed. In the case of a protocol, even the A anchor provides a data model that is already very concrete.

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

Fig. 10. Anchor E: sketch of its structure

So, in this case, the A and O anchors are the same. For an example of a refinement to an SCJ program that involves a substantial data refinement, we refer to [14].

5.2 Anchor E: execution model

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

The E anchor process for a non-terminating program takes the shape sketched in Figure 10, where we consider a process named *SCJsystem*. Other patterns are considered in [32]. The state components of the E anchor, in Figure 10, 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 `initialize` method of the `Safelet` subclass). *System* is a sequence of *Mission* actions; in Figure 10, 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 10 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 10, 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, $InitM1$ in Figure 10, specifies how the values of these variables are to be initialised.

The handler actions, which in Figure 10 are $Handler1$, $Handler2$, and so on, define the behaviour of the releases of the handlers. Their local variables are allocated in per-release memory. More elaborate algorithms may use temporary private memory areas to control allocation and deallocation of objects.

The *Handlers* action specifies the behaviour of the handler releases during the mission; in Figure 10, we show $HandlersM1$. In the parallelisms between the handler actions, the synchronisation sets (omitted in Figure 10) contain channels that represent the releases, if any, of aperiodic handlers by other 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 10, we sketch $Mission1$. 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 10) 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.

The E anchor for the optimised leadership-election protocol in Figure 7 is sketched in Figure 11. In this case, we have a single mission, which we model using the action *ElectionMission*. All variables are allocated in the mission memory, and so are all local to *MArea*. As indicated in Figure 7, we also have one periodic handler *CommunicatorH* and am aperiodic handler *ElectorH*.

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 10. Five phases define the refinement strategy in this step as depicted in Figure 12. 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.

As already mentioned, for the leadership-election protocol, for instance, we use parallelism in the A anchor to separate the behavioural and timing requirements. In *Circus* models automatically generated from domain-specific languages, typically, we have a parallelism between the components of the high-level model. It is the objective of our refinement strategy to change that architecture to that adopted by the mission paradigm of SCJ, without introducing errors.

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

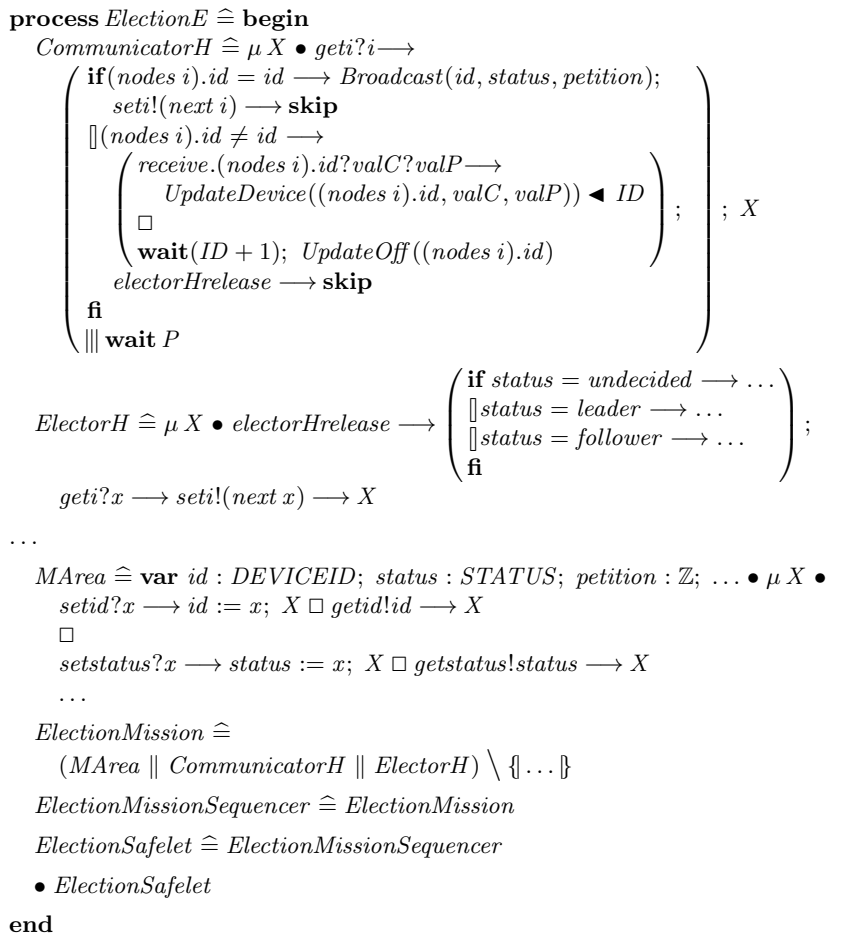

Fig. 11. E anchor for the leadership-election protocol

handlers releases, and the control mechanisms that orchestrate their execution. In SH, we define how variables are shared between handlers. The final phase AR uses algorithmic refinement to derive the implementation of the methods.

5.3 Anchor S: Safety-Critical Java

The S anchor is written using *SCJ-Circus*. As explained in Section 3, a *Circus* model is composed of a sequence of paragraphs: syntactic units that introduce types, constants, processes, and so on. *SCJ-Circus* is based on *Circus*, *OhCircus*, and *Circus Time*, but includes several new paragraphs [32]. We have paragraphs for the declaration of safelets, mission sequencers, missions, and handlers. Their semantics is defined by standard *Circus* processes and actions.

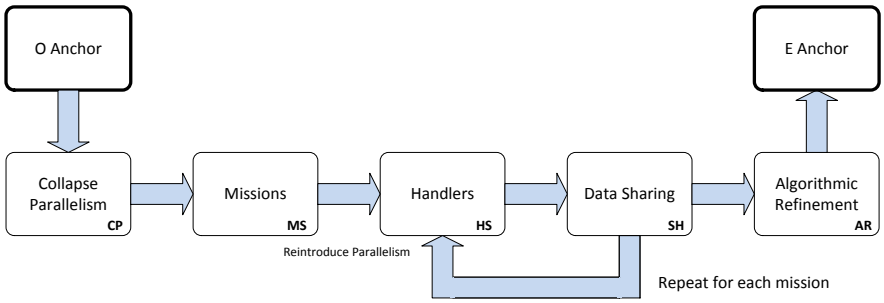

Fig. 12. Overview of the strategy for the Anchor E Step

In the last step of our refinement strategy, the process that defines the E anchor is split to yield the definition of these special 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 **initialize** paragraph. For statically defined sequences of missions, a simple **sequencer** paragraph 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 a refinement strategy detailed in [32]. 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 design suggested in the structure of the E anchor indeed matches the concurrency model of SCJ.

As an example, we present, the S anchor for the leadership-election protocol. The paragraph **safelet** defines the `initialize` and `cleanUp` methods. In our example, they are empty (**skip**), and so this paragraph is omitted.

The paragraph for our mission sequencer is shown below. We note that, since we are considering just the election mission, the **getNextMission** paragraph only ever returns the identifier *ElectionMission* of a single mission.

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** := *ElectionMission*
  $[]$ *mission_done* = *true* $\longrightarrow$ **ret** := *null*
  fi

end

The **state** paragraph of an *SCJ-Circus* component defines the fields of the corresponding SCJ class, and the **initial** paragraph defines its constructor. The other

paragraphs of these specialised components define methods of the SCJ class. Special paragraphs correspond to API methods. For instance, above we have **getNextMission** corresponding to the SCJ `getNextMission` method.

We use identifiers to refer to specific components: missions and handlers. For example, above, we use *ElectionMission* as an identifier for a mission. It is defined by the next *SCJ-Circus* paragraph.

The fields and constructor of the mission *Election* are defined by the schemas *ElectionState* and *InitElectionState* previously presented. In its API **initialize** method, it simply instantiates and registers the handlers *CommunicatorH* and *ElectionH*. As previously shown, the first is a periodic handler and the second, aperiodic. Creation of a handler H, specified by an *SCJ-Circus* paragraph **periodic** H or **aperiodic** H, is defined by the special expression **newHdlr** H.

$$
\begin{array}{l}
\textbf{mission}\ \mathit{ElectionMission} \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ \mathit{st} == \mathit{ElectionState} \\
\quad \textbf{initial} \mathrel{\widehat=} \mathit{InitElectionState} \\
\quad \textbf{initialize} \mathrel{\widehat=} \textbf{var}\ \mathit{ch}, \mathit{eh} : \mathit{ID} \bullet \\
\qquad \mathit{eh} = \textbf{newHdlr}\ \mathit{ElectorH}; \\
\qquad \mathit{ch} = \textbf{newHdlr}\ \mathit{CommunicatorH}(\mathit{eh}); \\
\qquad \mathit{eh}.\mathit{register}();\ \mathit{ch}.\mathit{register}() \\
\textbf{end}
\end{array}
$$

Like in SCJ, corresponding to handlers, we have objects, instances of an *OhCircus* class. Such objects, like *eh* and *ch* in the example above, respond to a *register* method. It identifies the handler as part of the mission.

We note how close the definition of *ElectionMission* above is to an SCJ class that implements a mission. On the other hand, *ElectionMission* defines a *Circus* process, as do *CommunicatorH* and *ElectorH* used there, although these processes use classes that model the data of the handlers and mission. The meaning of the special method calls, like the calls to *register*, for instance, is given by a (hidden) event. In the case of *register*, it triggers a data operation that enriches the (encapsulated) state of the mission process to record an instance of the relevant handler. So, what we have is a *Circus* semantics for the SCJ paradigm, (very much as explained in [50] for SCJ itself).

The periodic handler *CommunicatorH* is introduced as shown below.

$$
\textbf{periodic}\ \mathit{CommunicatorH} \mathrel{\widehat=} \textbf{begin}\ \textbf{start}\ 0\ \textbf{period}\ P
$$

This paragraph also defines the start time and the period of the protocol as P (as required in the timing specification given by *ATReqsLE* in the A anchor). It starts right at the beginning of the mission.

The state of *CommunicatorH* records the instance of *ElectionH* used in the mission. Its value is defined by the constructor.

$$
\begin{array}{l}
\textbf{state}\ \mathit{st} == [\mathit{electorH} : \mathit{ID}] \\
\textbf{initial} \mathrel{\widehat=} \textbf{val}\ \mathit{eh} : \mathit{ID} \bullet \mathit{electorH} := \mathit{eh}
\end{array}
$$

At each cycle, the release of *ElectionH* checks which device is being considered.

If the current device is itself, then it broadcasts its information just like in *ABReqsLE* and increments the index i to point to the next device. Otherwise it waits for either a communication on *receive* that must happen within *ID* time units and updates the state accordingly. If $ID+1$ time units pass without *receive* occurring, it updates the current device's information to indicate it is inactive. It then initiates an election by releasing the aperiodic handler *electorH*.

$$
\begin{array}{l}
\textbf{handleAsyncEvent} \mathrel{\widehat=} \\
\quad \textbf{if}\,(nodes\ i).id = id \longrightarrow Broadcast(id, status, petition);\ i := next\ i \\
\quad [\!]\,(nodes\ i).id \neq id \longrightarrow \\
\qquad \left(\begin{array}{l} receive.(nodes\ i).id?valC?valP \longrightarrow \\ \qquad UpdateDevice((nodes\ i).id, valC, valP)) \blacktriangleleft ID \\ \Box \\ \textbf{wait}(ID+1);\ UpdateOff((nodes\ i).id) \end{array}\right); \\
\qquad electorH.release() \\
\quad \textbf{fi} \\
\textbf{end}
\end{array}
$$

The actions used in **handleAsyncEvent** are also defined inside *ElectionH*. Their definitions are as presented previously.

The variables in the state of the mission *ElectionMission* are those to be allocated in mission memory. Accordingly, they are directly accessible in the handlers, like *nodes*, *id*, *status*, *petition*, and *next* above.

The *ElectorH* handler implements the conditional over *status* in *ABReqsLE*.

$$
\begin{array}{l}
\textbf{aperiodic}\ ElectorH \mathrel{\widehat=} \textbf{begin} \\
\textbf{handleAsyncEvent} \mathrel{\widehat=} \\
\quad \left(\begin{array}{l} \textbf{if}\ status = undecided \longrightarrow ... \\ [\!]\ status = leader \longrightarrow ... \\ [\!]\ status = follower \longrightarrow ... \\ \textbf{fi} \end{array}\right); \\
\quad i := next\ i \\
\textbf{end}
\end{array}
$$

It is not difficult to see that thee *SCJ-Circus* model can be automatically translated to SCJ code, actual Java code that can be compiled and executed.

6 Conclusions

In this tutorial, besides a didactic account of SCJ, we have given a brief introduction to *Circus*. In both cases, we have used the practical examples of a leadership-election protocol to illustrate the notations and concepts. Furthermore, we have reviewed the notion of refinement and formal techniques of program development in the context of both a traditional modelling language, like Z, and process algebra, namely, CSP and *Circus*.

To the best of our knowledge, all existing combinations of Z with a process algebra [19, 30, 44] model concurrent programs as communicating abstract data

types, where events are identified with operations that change the state. This is not the *Circus* approach. Events are just atomic instantaneous interactions like in CSP, and data operations have to be explicitly called, if needed. This is in keeping with the approach used in programming languages, and facilitates the use *Circus* to verify correctness of programs. Besides the general *Circus* refinement calculus [10], there are results for Ada [8].

We did not, of course, aim at a comprehensive view of all the issues and techniques available. We hope, however, to have given general pointers to the subject to support further reading. Most of all, we hope to have made it clear that refinement is about organised and clear justification of the intuition that millions of programmers use everyday to reassure themselves and others of the correctness of their designs and programs. The lack of such a framework has lead to poorly documented, intricated, and many times mistaken developments.

An understanding of refinement as the underpinning notion of all development methods can help good engineers or programmers to achieve their goal more successfully. Knowledge of the properties of the refinement relation, in the form, for example, of refinement laws, can lead to improved programming skills, even if a formal refinement technique is not really applied.

Other tasks involved in the construction of programs, like testing [1], refactoring [15], and compilation [43, 48, 18] have already been characterised as related to refinement. A lot has already been achieved by the formal methods community in the last two or three decades; there is a lot yet to be done.

Acknowledgments This work is funded by EPSRC grant EP/H017461/1. No primary data arises from the work reported here. We have benefitted from discussions with Frank Zeyda in the development of our case study.

References

1. B. Aichernig. Contract-based Mutation Testing in the Refinement Calculus. In *REFINE'2002*, Eletronic Notes in Theoretical Computer Science, 2002. Invited paper.
2. R. J. R. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
3. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
4. D. Bjorner and C. B. Jones. *Formal Specifications and Software Development*. Prentice-Hall, 1982.
5. T. Bolognesi. On State-oriented versus Object-oriented Thinking in Formal Behavioural Specifications. Technical Report 2003-TR-20, ISTI-Istituto di Scienza e Tecnologie della Informazione Alessandro Faedo, 2003.
6. A. Burns. The Ravenscar Profile. *Ada Letters*, XIX:49–52, 1999.
7. A. L. C. Cavalcanti. *A Refinement Calculus for Z*. PhD thesis, Oxford University Computing Laboratory, Oxford - UK, 1997. Technical Monograph TM-PRG-123, ISBN 00902928-97-X.
8. A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via *Circus*. *Formal Aspects of Computing*, 23(4):465–512, 2011.

9. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. An Inconsistency in Procedures, Parameters, and Substitution the Refinement Calculus. *Science of Computer Programming*, 33(1):87 – 96, 1999.
10. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146–181, 2003.
11. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *Software and System Modelling*, 4(3):277–296, 2005.
12. A. L. C. Cavalcanti, A. Wellings, J. C. P. Woodcock, K. Wei, and F. Zeyda. Safety-Critical Java in *Circus*. In A. P. Ravn, editor, *9th Workshop on Java Technologies for Real-Time and Embedded System*, ACM Digital Library. ACM, 2011.
13. A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC—A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267—289, 1999.
14. A. L. C. Cavalcanti, F. Zeyda, A. Wellings, J. C. P. Woodcock, and K. Wei. Safety-critical Java programs from *Circus* models. *Real-Time Systems*, 49(5):614–667, 2013.
15. M. L. Cornélio, A. L. C. Cavalcanti, and A. C. A. Sampaio. Refactoring by Transformation. In J. Derrick, E. Boiten, J. C. P. Woodcock, and J. Wright, editors, *REFINE'2002*, volume 70 of *Eletronic Notes in Theoretical Computer Science*. Elsevier, 2002. Invited paper.
16. E. W. Dijkstra. Guarded commands, nondeterminacy and the formal derivation of programs. *Communication of the ACM*, 18:453–457, 1975.
17. A. Diller. *Z : An Introduction to Formal Methods*. John Wiley & Sons, 2nd edition, 1994.
18. A. A. Duran, A. C. A. Sampaio, and A. L. C. Cavalcanti. Refinement Algebra for Fomal Bytecode Generation. In C. George and H. Miao, editors, *International Conference on Formal Engineering Methods*, volume 2495 of *Lecture Notes in Computer Science*, pages 347–358. Springer-Verlag, 2002.
19. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 2, pages 423–438. Chapman & Hall, 1997.
20. C. Fischer. How to Combine Z with a Process Algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *The Z Formal Specification Notation*. Springer-Verlag, 1998.
21. C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.
22. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
23. J. Ichbiah. Rationale for the Design of the Ada Programming Language. *ACM SIGPLAN Notices*, 14(6B (special issue)), 1979.
24. He Jifeng, C. A. R. Hoare, and J. W. Sanders. Data Refinement Refined. In G. Goos and H. Hartmants, editors, *ESOP'86 European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196, 1986.
25. He Jifeng, C. A. R. Hoare, and J. W. Sanders. Prespecification in Data Refinement. *Information Processing Letters*, 25(1), 1987.
26. C. B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall, 1980.
27. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.
28. B. H. Liskov and J. M. Wing. A Behavioural Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.
29. D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. *Safety*

Critical Java Specification, First Release 0.76. The Open Group, UK, 2010. `jcp.org/aboutJava/communityprocess/edr/jsr302/index.html`.
30. B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.
31. I. Meisels. *Software Manual for Windows Z/EVES Version 2.1*. ORA Canada, 2000. TR-97-5505-04g.
32. A. Miyazawa and A. L. C. Cavalcanti. Refinement strategies for Safety-Critical Java. In M. L. Cornélio, editor, *Brazilian Symposium on Formal Methods*, Lecture Notes in Computer Science. Springer, 2015.
33. C. C. Morgan. Auxiliary Variables in Data Refinement. *Information Processing Letters*, 29(6):293–296, 1988.
34. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
35. C. C. Morgan and P. H. B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, 27(6):481–503, 1990.
36. J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287–306, 1987.
37. Motor Industry Software Reliability Association Guidelines. Guidelines for Use of the C Language in Critical Systems. 2012.
38. B. F. Potter, J. E. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice-Hall, 2nd edition, 1996.
39. G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.
40. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
41. A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.
42. RTCA/DO-178C/ED-12C. Software Considerations in Airborne Systems and Equipment Certification, 2011.
43. A. C. A. Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing*. World Scientific, 1997.
44. S. Schneider and H. Treharne. Communicating B Machines. In D. Bert, J. Bowen, M. Henson, and K. Robinson, editors, *ZB'2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 416–435, 2002.
45. A. Sherif, A. L. C. Cavalcanti, J. He, and A. C. A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, 2010.
46. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
47. A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004.
48. L. Wildman. A Formal Basis for a Program Compilation Proof Tool. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods-Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 491–510. Springer-Verlag, 2002.
49. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.
50. F. Zeyda, L. Lalkhumsanga, A. L. C. Cavalcanti, and A. Wellings. *Circus* Models for Safety-Critical Java Programs. *The Computer Journal*, 57(7):1046–1091, 2014.