Refining SCJ Mission Specifications into Parallel Handler Designs

Frank Zeyda
University of York, York, YO10 5GH, UK.
frank.zeyda@york.ac.uk

Ana Cavalcanti
University of York, York, YO10 5GH, UK.
ana.cavalcanti@york.ac.uk

Safety-Critical Java (SCJ) is a recent technology that restricts the execution and memory model of Java in such a way that applications can be statically analysed and certified for their real-time properties and safe use of memory. Our interest is in the development of comprehensive and sound techniques for the formal specification, refinement, design, and implementation of SCJ programs, using a correct-by-construction approach. As part of this work, we present here an account of laws and patterns that are of general use for the refinement of SCJ mission specifications into designs of parallel handlers used in the SCJ programming paradigm. Our notation is a combination of languages from the *Circus* family, supporting state-rich reactive models with the addition of class objects and real-time properties. Our work is a first step to elicit laws of programming for SCJ and fits into a refinement strategy that we have developed previously to derive SCJ programs.

Keywords: SCJ, models, refinement, laws, patterns, tactics, *Circus*.

1 Introduction

Java is indisputably one of the most popular programming languages. Despite this, its use in the safety-critical industry has been modest due to Java's generality and rich set of features. Significant issues are, for example, the use of garbage collection and problems related to thread prioritisation [24, 26], which render it inadequate for time-critical applications. Safety-Critical Java (SCJ) [16], a recent initiative, addresses these issues by introducing a restricted subset of Java; it is based on the Real-time Specification for Java (RTSJ) [27], but further restricts RTSJ's execution and memory model. This facilitates the formal analysis of SCJ applications, and thereby enables the application of formal methods to satisfy stringent criteria of certification standards like DO-178C.

SCJ is organised in three levels (Level 0 to Level 2) that define progressively more complex models of execution. Our focus is SCJ Level 1, which roughly corresponds to the Ravenscar profile for Ada [5]. At Level 1, applications are organised as a sequence of missions, and each mission consists of a set of handlers that are executed in parallel. Handlers can either be periodic, which means they are released at regular time intervals, or aperiodic implying that they are released sporadically by some external event or stimulus. When a handler is released, its `handleAsyncEvent()` method is scheduled for execution.

Our previous work has focused on complementing the informal account of SCJ [26] with a formal model of SCJ's mission-based execution paradigm [29] and memory model [11]. Our notation is a combination of languages from the *Circus* family [9, 10, 23], specifically tailored for the specification and development of state-rich reactive systems with the addition of discrete time, object-orientation, and object references. We have also proposed a refinement strategy [12] to transform abstract specifications of SCJ programs into models that directly correspond to SCJ programs. Such a strategy is inherently ambitious and complex, as it simultaneously addresses a multitude of concerns. Therefore, it is not

surprising that the existing work [12] only gives a broad description of the top-level approach; details of the application of this strategy to a specific example are available in [30].

Our contribution in this paper is to examine in detail the refinement of centralised and sequential specifications of missions into parallel handler designs. Our general starting point is a *Circus* process specification that supports all constructs of *Circus*, including Z data operations, classes, and Timed CSP constructs, except for parallelism and interleaving. We then show how decomposition at the level of data operations, time budgets, and process actions can be used to transform the model into a uniform shape that determines the structure and behaviour of handlers of an SCJ mission. Refinement laws directly reflect particular program designs that encapsulate the way in which data is shared and how the computational work is divided between the handlers of a mission.

The motivation for our work is to pave the way for automated tool support. Due to the novelty of SCJ, there are not many tools currently available that support the development of critical software in SCJ. The available tools mostly focus on isolated statically-checkable properties [25, 13, 15], but do not address the combination of concerns that characterise the SCJ paradigm. While we do address many concerns of SCJ simultaneously by using a highly expressive language, the practicalities of performing actual refinements are largely an open problem. It is, clearly, unrealistic to carry out such refinements entirely by hand, which is well illustrated by the complexity of the example in [30]. Some refinement steps are, however, inherently difficult to automate. Our work, most importantly, highlights where automation is feasible, and where human guidance is indispensable to guide the refinement process.

The results in this paper contribute towards elaborating the proposed refinement strategy for SCJ in [12], but they are also useful outside the context of that technique. Decomposition of centralised models is a general issue in refinement-based techniques [9], and the models we produce can, in principle, serve as a starting point for any form of parallel implementation. As the essence of the SCJ paradigm (its mission-based execution model) can be captured independently of the Java language, our account on mission decomposition is relevant for other languages that adopt a similar execution model, too.

The structure of this paper is as follows. In Section 2 we review preliminary material: Safety-Critical Java and the *Circus* family of languages. Section 3 then discusses our refinement laws, and Section 4 presents an example of their application. Finally, in Section 5 we conclude and suggest future work.

2 Preliminaries

We here discuss in more detail Level 1 SCJ and the *Circus* family of notations.

2.1 Level 1 SCJ

The execution model for SCJ Level 1 programs is based on four primary conceptual entities: safelet, mission sequencer, missions and handlers. They are realised by classes that derive either from an interface or abstract class of the SCJ API. Namely, these are `Safelet`, `MissionSequencer`, `Mission`, `PeriodicEventHandler`, and `AperiodicEventHandler`.

Fig. 1 illustrates the life-cycle of a Level 1 safelet, the top-level entity of an SCJ application. The SCJ infrastructure[1] first initialises the safelet. This is followed by a series of mission executions, each involving the initialisation, execution and termination of a particular mission of the safelet. Mission initialisation creates the mission's event handlers, which are released either periodically or by external events during mission execution. When there are no more missions to execute, the safelet terminates.

[1] By 'SCJ infrastructure' we mean an SCJ-compliant virtual machine.

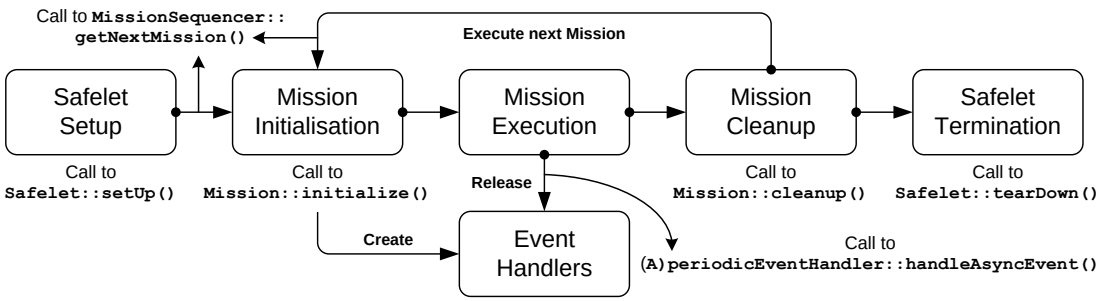

Figure 1: Life-cycle of a safelet during execution of a Level 1 application

In terms of the SCJ API, a class implementing `Safelet` has to provide the methods `setUp()` and `tearDown()`, which are called by the SCJ infrastructure to initialise and shutdown the safelet. Another method (not in Fig. 1) is called on the safelet object to obtain the mission sequencer of the application, which defines the sequence of missions to execute. In addition, various methods are called by the infrastructure on the mission sequencer, mission and handler objects during execution of the safelet. Most notably, these are `getNextMission()` to obtain the next mission to execute, `initialize()` to create the handlers of a mission, and `handleAsyncEvent()` when a handler is released. An SCJ program must provide implementations of these methods, and it thereby defines the architecture of the application in terms of missions and handlers. (We note that although the missions and handlers of a safelet are determined at run-time, we assume in our model that they are *a priori* fixed.)

When a mission terminates, `cleanup()` is called on the mission object to perform application-specific clean-up tasks. As already mentioned, the entire safelet terminates when there are no more missions to execute, signalled by `getNextMission()` returning a `null` reference. In summary, the safelet and the mission sequencer are control components that orchestrate the execution of the missions (and their handlers). The missions and the handlers, on the other hand, are the central components that implement the behaviour of the program, and the main focus of our work here.

2.2 The *Circus* family

Circus [9] is a language for specification and refinement of state-rich reactive systems. It combines notations from CSP [22], Z [28], and Morgan's refinement calculus [19]. As in CSP, the key elements of *Circus* models are processes that can interact with their environment through channels. Unlike CSP, *Circus* processes encapsulate a state that can be modified by actions and data operations of the process. *Circus* has a denotational semantics defined using the Unifying Theories of Programming [21].

An example of a *Circus* process is given in Fig. 2. It illustrates the general form of an SCJ handler design, and the laws we discuss in the next section transform (sequential) specifications of safelets into processes of this shape. The name of the process is *SCJDesign*, and its state is defined by the *State* schema, introducing the components c_i of type T_i (*Inv* is an optional state invariant). The T_i may be Z schema types or *OhCircus* class types, as it is also the case for the types in any of the laws. We then have local action definitions for *Init*, $Mission_i$, $Handler_i$ and *HdlControl*. The actual behaviour of the process is defined by the main action at the bottom after the '•'; it typically makes use of the local actions.

Local actions can be either specified by Z operation schemas or using a mixture of CSP constructs and guarded commands. Here, *Init* is a Z operation that initialises the state, and $Mission_i$ and $Handler_i$ are CSP actions that provide models of missions and handlers as they emerge during verification. Each

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

state $State == [c_1 : T_1;\ c_2 : T_2, \ldots, c_n : T_n \mid Inv(c_1, c_2, \ldots, c_n)]$

$Init == [State' \mid InitOp(c'_1, c'_2, \ldots, c'_n)]$

$$Mission_i \mathrel{\widehat=} \left(\begin{array}{l} \left(\begin{array}{l} Handler_1 \\ [\![ns_1 \mid cs_1 \cup \{\!| termMsn |\!\} \mid ns_2 \cup \ldots \cup ns_h]\!] \\ Handler_2 \\ [\![ns_2 \mid cs_2 \cup \{\!| termMsn |\!\} \mid ns_3 \cup \ldots \cup ns_h]\!] \\ \ldots \\ [\![ns_{h-1} \mid cs_{h-1} \cup \{\!| termMsn |\!\} \mid ns_h]\!] \\ Handler_h \end{array} \right) \\ [\![ns_1 \cup \ldots \cup ns_h \mid cs_c \cup \{\!| termReq, termMsn |\!\} \mid ns_c]\!]\ HdlControl \end{array} \right)$$

where $Handler_i$ may be either defined as

$Handler_i \mathrel{\widehat=}$ (for an aperiodic handler)

$$\left(\mu X \bullet \left(\left(\begin{array}{l} e_{(i,1)}\,?\,v \longrightarrow A(v)\ \Box \\ e_{(i,2)}\,?\,v \longrightarrow A(v)\ \Box \\ \ldots \end{array} \right) ; X \right) \right) \quad \text{or} \quad \left(\mu X \bullet \left(\begin{array}{l} (A \blacktriangleright T \,|||\, \mathbf{wait}\ T)\,;\ X \\ \Box\ termMsn \longrightarrow \mathbf{skip} \end{array} \right) \right)$$

$Handler_i \mathrel{\widehat=}$ (for a periodic handler)

$$HdlControl = (termReq \longrightarrow termMsn \longrightarrow \mathbf{skip}) \parallel \ldots$$

$\bullet\ Init\,;\ Mission_1\,;\ Mission_2\,;\ \ldots\,;\ Mission_m$

end

Figure 2: Target for refinement transforming mission models.

$Mission_i$ action is defined by a parallel composition of a mission-specific set of handler actions. In *Circus*, parallel composition of two actions A_1 and A_2 is written as $A_1\ [\![ns_1 \mid cs \mid ns_2]\!]\ A_2$, where cs is a set of interface channels that require synchronisation of the actions, and ns_1 and ns_2 are disjoint sets of variables that each action is allowed to write to. Hence, all handlers of a mission write to mutually disjoint parts of the state space, determined by the ns_i. This ensures that all *Circus* constructs (including parallel composition) are monotonic with respect to refinement due to intrinsic non-interference in shared data access by parallel processes and actions.

The action *HdlControl* is included to incorporate control mechanisms. It controls termination of the mission via the channels *termReq* (for a termination request raised by one of the handlers) and *termMsn* (to synchronously terminate the handlers). It also permits the definition of additional control actions (dots) whose design is not a concern for mission decomposition into handlers.

The handler models, captured by the actions $Handler_i$, take different shapes for aperiodic and periodic handlers. Both, however, have the form of a recursion $(\mu X \bullet A\,;\ X \,\Box\, termMsn \longrightarrow \mathbf{skip})$ that repetitively executes some action A and at the same time enables termination by *HdlControl*. Aperiodic handlers are modelled by an external choice that synchronises on a set of channels $e_{(i,1)}$, $e_{(i,2)}$, and so on, which correspond to SCJ events that are bound to the handler i and therefore cause its release. Potentially, each event provides an input v, and the `handleAsyncEvent()` method is specified by $A(v)$.

For periodic handlers, the repetitive behaviour is determined by the action $A \blacktriangleright T \,|||\, \mathbf{wait}\ T$, using additionally constructs from *Circus* Time. The $A \blacktriangleright T$ operator imposes a termination deadline T on A, **wait** T corresponds to a delay of T time units, and the interleaving with **wait** T prevents the action from terminating *before* T time units have elapsed. Hence, we obtain a cyclic behaviour that executes the

`handleAsyncEvent()` method A once every T time units. For clarification, we point out that all *Circus* Time constructs take relative times as their arguments.

We note that interleaving $(A_1 \,|||\, A_2)$ is a special case of parallelism where the synchronisation set cs is empty; termination only occurs when both parallel actions have terminated. Later on, we also make use of the **wait** $t_1 \,..\, t_2$ statement, which corresponds to a nondeterministic delay between t_1 and t_2 time units, and $A \blacktriangleleft T$ which is a deadline on A to interact via a visible event, such as a communication or synchronisation.

The main action of *SCJDesign* at the end first initialises the state and then executes all missions in sequence (operator $A_1 \,;\, A_2$). In Fig. 2, we use notations from both *Circus* and *Circus* Time [23], and generally also support the use of constructs from *OhCircus* [10] for class objects. The UTP [17] enables us to give a sound semantic foundation to this combination of languages.

3 Refinement Laws

Our starting point is a centralised mission specification that defines communication patterns, data operations, and timing restrictions using sequential *Circus* actions. We deal with three aspects of the verification of a mission implementation with respect to such a centralised specification. The first aspect is decomposition of data operations to introduce functional models of handlers. The second is distribution of time budgets between the handlers. And the third is parallelisation of handlers to match the architecture of Level 1 SCJ; this also addresses data flow and control mechanisms via communications. We present here collections of *Circus* refinement laws and tactics to address each of these verification issues. Although some of these laws have already been given in [8] and [9], the parallelisation laws in Fig. 5 and Fig. 10 are to our knowledge novel, and so are the *Circus* Time laws in Section 3.2.

3.1 Decomposition of data operations

Here we target data operations. We note that we do not generally require that the specification of a mission involves a single data operation. For missions with simple interaction patterns, such as reading an input, performing a computation, and writing an output, it is possible to capture the functional aspects of the mission in a single data operation. In the general case, however, where inputs and outputs may occur sporadically during mission execution, a functional mission model may be split into more than one data operation. We assume, on the other hand, that all data operations specify mission behaviour at a suitably high level of abstraction: this means they are centralised models of functionality, and hence do not already encapsulate any form of computational design.

Our goal is to decompose data operations so that the (functional) specifications of individual handlers emerge. We employ schema composition to model sequential execution of handlers, and schema conjunction to model parallel execution of handlers. All refinement is carried out at the level of Z. The Z Refinement Calculus [8, 14], whose laws are valid in *Circus* [21], provides the foundation for our laws here. The laws we present are therefore applicable and relevant for Z refinement in general.

Though [8, 14], for example, present a collection of laws that address issues of decomposition too, it is well understood that decomposition of data operations is overall difficult to automate. We propose a number of specialised laws that cover a broad spectrum of mission designs. Each law encapsulates either a sequential or parallel design that carries out a centralised computation by two or more handlers.

Law 1 *Let State* $== [x : T_1;\ y : T_2 \mid I_1(x) \land I_2(y)]$. *Then,*

$$
\begin{array}{|l}
Op \\
\hline
\Delta State \\
\hline
P(x,y,x') \land Q(y,y')
\end{array}
\equiv
\begin{array}{|l}
Op_1 \\
\hline
\Delta[x : T_1 \mid I_1(x)] \\
\Xi[y : T_2 \mid I_2(y)] \\
\hline
P(x,y,x')
\end{array}
\fatsemi
\begin{array}{|l}
Op_2 \\
\hline
\Delta[y : T_2 \mid I_2(y)] \\
\Xi[x : T_1 \mid I_1(x)] \\
\hline
Q(y,y')
\end{array}
$$

Figure 3: Sequential decomposition of independent data operations.

Law 2 *Let State* $== [x : T_1;\ y : T_2 \mid I_1(x) \land I_2(x,y)]$. *Then,*

$$
\begin{array}{|l}
Op \\
\hline
\Delta State \\
\hline
P(x,y,x') \land Q(x',y,y')
\end{array}
\equiv
\begin{array}{|l}
Op_1 \\
\hline
\Delta[x : T_1 \mid I_1(x)] \\
\Xi[y : T_2] \\
\hline
I_2(x,y) \land \\
P(x,y,x')
\end{array}
\fatsemi
\begin{array}{|l}
Op_2 \\
\hline
\Xi[x : T_1 \mid I_1(x)] \\
\Delta[y : T_2] \\
\hline
I_2(x',y') \land \\
Q(x,y,y')
\end{array}
$$

Figure 4: Sequential decomposition of dependent data operations.

Laws for sequential decomposition of data operations We distinguish two fundamental cases. The first one assumes no dependency between the data operations in terms of the computed results. The corresponding law is presented in Fig. 3. We assume the existence of a *State* schema that specifies the state on which the operations act. It is partitioned into two disjoint lists of variables, x and y, which are respectively constrained by the state invariants $I_1(x)$ and $I_2(y)$. The law decomposes Op into a sequence $Op_1 \fatsemi Op_2$, where Op_1 only modifies the components in x, and Op_2 only modifies the components in y and does not depend on x. Application of this law entails transforming the predicate of an operation schema into a form $P(x,y,x') \land Q(y,y')$. This may in general require intelligent decision making, but in some cases ought to be automatable using elementary laws and syntax-driven rewriting realised by generic tactics of proof for the Z mathematical notation.

The second case is where there exists a dependency between the data operations in terms of the result. That is, the second operation uses data that is computed by the first one. Here, we have the general law in Fig. 4. The crucial difference is in the shape of the predicate of the refined operation Op, where $Q(x',y,y')$ refers to the final value of x. The state invariant is decomposed as well, namely into a conjunct $I_1(x)$ that only considers constraints on x, and another conjunct $I_2(x,y)$ that relates x and y. The invariant $I_2(x,y)$ is not enforced by the Δ and Ξ schemas but moved into the predicates for technical reasons: we observe that I_2 may in fact not hold for the intermediate state of the decomposition.

The propagation of invariants proves to be especially important to facilitate further decomposition and later algorithmic refinement. Invariant decomposition once again requires guidance. It involves the transformation of a single invariant $I(x,y)$ into the conjunction $I_1(x) \land I_2(x,y)$ so that all relevant knowledge about the components in x is encoded by $I_1(x)$.

We have defined several variations of the previous two laws that moreover deal with inputs and outputs of operations. We omit their discussion as they are straightforward generalisations. They can, however, be found in the appendix of [12]. Next, we take a look at parallel decomposition.

Law 3 *Let State* $== [x : T_1;\ r : T_2 \mid I_1(x) \land I_2(x,r)]$. *Then,*

$$
\begin{array}{|l}
Op \\
\hline
\Xi[x : T_1 \mid I_1(x)] \\
\Delta[r : T_2 \mid I_2(x,r)] \\
\hline
\exists r_1, \ldots, r_n : T_2 \mid \\
\left(\begin{array}{l} Q(r_1, 1, x) \land \\ Q(r_2, 2, x) \land \\ \ldots \\ Q(r_n, n, x) \end{array} \right) \bullet \\
r' = r_1 \ op \ r_2 \ op \ \ldots \ op \ r_n
\end{array}
\quad \equiv \quad
\left(\begin{array}{l}
\mathbf{var}\, r_1, \ldots, r_n : T_2 \bullet \\
(\exists i? : \mathbb{Z} \bullet POp[r_1/r!] \land i? = 1) \land \\
(\exists i? : \mathbb{Z} \bullet POp[r_2/r!] \land i? = 2) \land \\
\ldots \\
(\exists i? : \mathbb{Z} \bullet POp[r_n/r!] \land i? = n); \\
MOp([\![r_1, \ldots, r_n]\!])
\end{array} \right)
$$

where

$$
\begin{array}{|l}
POp \\
\hline
\Xi[x : T_1 \mid I_1(x)] \\
r! : T_2 \\
i? : 1 \,..\, n \\
\hline
Q(r!, i?, x)
\end{array}
$$

and

$$
\begin{array}{|l}
MOp \\
\hline
\Xi[x : T_1 \mid I_1(x)] \\
\Delta[r : T_2 \mid I_2(x,r)] \\
rb? : \mathrm{bag}\ T_2 \\
\hline
\exists s : \mathrm{seq}\ T_2 \mid s = items\, rb? \bullet \\
r' = \mathbf{fold}\ op\ zero\ s
\end{array}
$$

provided that op is an associative and commutative binary operation. The function **fold** *is the standard folding operation over a sequence of values and zero a zero for op.*

Figure 5: Parallel decomposition of dependent data operations.

Laws for parallel decomposition of data operations As before, we have a pair of laws that consider the case of independent and dependent data operations. Dependency here means that the operations cumulatively participate in the computation of some result. For independent data operations, the law is similar to that in Fig. 3 with a small modification of the right-hand side: firstly, the sequence $Op_1 \fatsemi Op_2$ is replaced by a conjunction $Op_1 \land Op_2$, and secondly, we remove the Ξ schemas in the declaration part of Op_1 and Op_2. The fact that both laws have the same left-hand side illustrates that there is often more than one possible handler design, giving rise to different degrees of parallelisation.

A more interesting parallelisation law is presented in Fig. 5. There, we have n handlers participating in the computation of the result r and using the components x. The behaviour of the handlers is specified by the predicate $Q(r_i, i, x)$ for $1 \leq i \leq n$. Decomposition here yields a conjunction that includes a conjunct *POp* for each handler, as well as a merge operation *MOp* that collects the partial results r_i to compute the overall result of the refined operation. Following the Z convention, the symbols '?' and '!' in the declaration part of the schemas *POp* and *MOp* are used to identify input and output parameters. The merge operation is parametrised by a bag to enforce syntactically that the order in which the results are delivered is irrelevant. Hence, we require that the binary operation used in the merge is associative and commutative; the merge basically consists of folding this operation over the list of partial results.

It turns out that the application of the above decomposition laws, in comparison to subsequent sets of laws, is the most challenging to automate. The developer needs to determine the target of each law application, that is, the schema predicates on the right-hand side of the laws. With that, a verification condition can be generated to establish that the predicate of the schema being refined can be written in the form required for the application of the law. Specialised proof tactics will be useful in this context.

Law 4 $\mathbf{wait}\,0\,..\,t \equiv \mathbf{wait}\,0\,..\,t_1\,;\,\mathbf{wait}\,0\,..\,t_2 \quad \textit{where} \quad t = t_1 + t_2$
Law 5 $\mathbf{wait}\,0\,..\,t_1 \sqsubseteq \mathbf{wait}\,0\,..\,t_2 \quad \textit{where} \quad t_2 \leq t_1$
Law 6 *Assuming Op is a data operation and P is a Circus process, we have* $P(\mathbf{wait}\,t_1\,..\,t_2\,;\,Op) \equiv P(Op\,;\,\mathbf{wait}\,t_1\,..\,t_2)$

Figure 6: Laws for decomposition and distribution of time budgets.

3.2 Distribution of time budgets

Data operations in *Circus* are atomic and instantaneous. Hence, all timing behaviour has to be specified explicitly using timed action operators. Time budgets specify the permissible amount of time that an implementation may take to execute a data operation; in *Circus*, they can be captured by nondeterministic wait statements of the form $\mathbf{wait}\,0\,..\,t$ that precede a data operation. The laws in this section are hence essentially about **wait** statements modelling time budgets, and, therefore, are useful in any context where we want to reason about the timing of Z data operations in (*Oh*)*Circus*.

Our general assumption is that the specification of mission behaviour may utilise **wait** statements in arbitrary places. The laws in this section decompose and distribute those **wait**s in order to attach them to the data operations emerging from decomposition in the previous section. Using these laws, we can equip each decomposed data operation *Op* with an operation-specific time budget $\mathbf{wait}\,0\,..\,TB_{Op}$, where TB_{Op} determines the amount of time the operation may take to execute in the SCJ program.

The refinement laws needed can be divided into two classes. In the first class, we have two key laws (given in Fig. 6) for the decomposition and narrowing of time budgets. Whereas the first Law 4 replaces a single time budget by a sequence of two time budgets, the second Law 5 incurs a reduction of nondeterminism that narrows a time budget. A point of design in applying these laws is to decide on the values of t_1 and t_2, which subsequently determine the amount of time available to the underlying data operations. Decomposition may, of course, be applied iteratively, so that a single time budget can be split into several time budgets for any given number of operations.

The second class of laws addresses the issue of moving the decomposed time budgets to suitable locations to attach them to their respective data operations. For this, we first transform all Z schema compositions into *Circus* action sequences. The standard law for this is recaptured below from [8].

Law 7 $Op_1 \fatsemi Op_2 \equiv Op_1\,;\,Op_2$ *provided* $\mathbf{pre}(Op_1 \fatsemi Op_2) \land Op_1 \Rightarrow \mathbf{pre}'(Op_2)$

As usual, $\mathbf{pre}(Op)$ yields the precondition (domain) of a Z operation *Op*, and we use $\mathbf{pre}'(Op)$ to indicate that the variables in the result are primed. We note that the semicolon '$\fatsemi$' is used for composition of Z operations, as opposed to ';' which is used for composition of *Circus* actions.

We further require the specialised distribution Law 6 in Fig. 6. This law is in fact non-compositional: it is a law about processes rather than actions. Hence, it only holds if the underlying action $\mathbf{wait}\,t_1\,..\,t_2\,;\,Op$ is embedded in a process *P*. The justification for the law comes from the structure and semantics of processes that prevents observation of the precise time at which an (internal) state change takes place. A proof is possible by induction over the structure of processes.

We note that no distribution laws exist to move time budgets across prefixes, since such transformations would not be correct as they alter the observable behaviour. Consider, for example, $c \longrightarrow \mathbf{wait}\,t\,;\,A$. Refining this action by $\mathbf{wait}\,t\,;\,c \longrightarrow A$ would be wrong since the refining action refuses communication on the channel *c* for *t* time units, whereas the refined action offers it immediately. Some general laws for *Circus* refinement in [20] are useful, too, namely to distribute time budgets into and out of internal and external choice. Lastly, we have a fusion law for nondeterministic choice of time budgets:

Law 9 *Let A_1 and A_2 be actions and c a fresh typeless channel. Then,*
$A_1 ; A_2 \equiv ((A_1 ; c \longrightarrow \mathbf{skip}) \llbracket \mathrm{wrt}(A_1) \mid \{\!| c |\!\} \mid \mathrm{wrt}(A_2) \rrbracket (c \longrightarrow A_2)) \setminus \{\!| c |\!\}$
provided $\mathrm{wrt}(A_1) \cap \mathrm{wrt}(A_2) = \varnothing$ *and* $\mathrm{wrt}(A_1) \cap \mathrm{used}(A_2) = \varnothing$

Figure 7: Parallelisation of independent sequential data operations.

Law 10 *Let A_1 and A_2 be actions and c a fresh channel. Then,*
$A_1 ; A_2 \equiv ((A_1 ; c\,!\,x \longrightarrow \mathbf{skip}) \llbracket \mathrm{wrt}(A_1) \mid \{\!| c |\!\} \mid \mathrm{wrt}(A_2) \rrbracket (c\,?\,x \longrightarrow A_2)) \setminus \{\!| c |\!\}$
provided $\mathrm{wrt}(A_1) \cap \mathrm{wrt}(A_2) = \varnothing$ *and* $\mathrm{wrt}(A_1) \cap \mathrm{used}(A_2) = \{x\}$

Figure 8: Parallelisation of dependent sequential data operations.

Law 8 $\mathbf{wait}\, t_1 .. t_2 \sqcap \mathbf{wait}\, t_1' .. t_2' \equiv \mathbf{wait}\, \min(t_1, t_1') .. \max(t_2, t_2')$

This law is useful as it enables the combination of two budgets.

The laws we present here are evidently complete for mission specifications in which each abstract data operation is already associated with an (abstract) time budget. Automation of the refinement can be envisaged by annotating each data operation with the intended time budget, and using tactics to mechanically perform the decomposition and distribution steps. An overall caveat for the transformation is that we cannot distribute time budgets into parallel data operations which are represented by Z schema conjunctions. This is because the conjunction operator only applies to schemas and not to actions, and the schema calculus does not support timing constructs such as **wait** $t_1 .. t_2$. (In our strategy, we, therefore, distribute the budgets of parallel operations after the *Circus* parallel operators are introduced.)

The next section examines the refinement of sequential actions and schema conjunctions, as they emerge from the laws discussed so far, into parallel actions.

3.3 Introduction of parallel handler actions

In Section 3.1, we have presented laws to parallelise data operations using schema conjunction, but considered no laws to parallelise actions. The laws we discuss next can be used to parallelise mission actions. Like in Section 3.1, we divide the necessary laws into two classes: laws that account for sequential designs and laws that cater for parallel designs. The shapes we target are precisely those produced by earlier decomposition of data operations, which makes this aspect of the verification more susceptible to automation. In the sequel, we discuss both classes of laws.

Laws for sequential handler designs Two central laws for parallelisation of handlers are given in Fig. 7 and Fig. 8. The first one assumes that there exists no data dependency between the sequential handler actions A_1 and A_2, hence we have the proviso $\mathrm{wrt}(A_1) \cap \mathrm{used}(A_2) = \varnothing$, which states that the state components written by A_1 are disjoint from those read by A_2. A fresh typeless channel c is introduced to control the order of execution of the parallel actions: they both have to synchronise on it, so that the right parallel action $c \longrightarrow A_2$ blocks until the left parallel action is ready to execute the prefix $c \longrightarrow \mathbf{skip}$. The channel c models an SCJ event that is bound to the second handler and fired by the first handler.

The second law (Fig. 8) assumes that there is a data dependency between the sequential handlers. In that case, the channel c is parametrised by the type of the data that is passed between A_1 and A_2. Multiple data items can be passed by using product types, and, as mentioned earlier, class types are permissible,

Law 11 $Op_1 \wedge Op_2 \equiv Op_1 \,[\![\, \mathrm{wrt}(Op_1) \mid \varnothing \mid \mathrm{wrt}(Op_2) \,]\!]\, Op_2$
provided $\mathrm{wrt}(Op_1) \cap \mathrm{wrt}(Op_2) = \varnothing$

Figure 9: Low-level law for refining parallel data operations into actions.

too. An interesting observation at this point is that the channel c fulfils a dual purpose: it controls both the order of execution of handlers and makes available shared data. Further refinement is hence required to untangle these concerns, namely by way of encapsulating the shared data independently of the control aspect. This is, however, beyond the scope of parallelisation of handlers and a separate and orthogonal design issue, so we do not discuss it further here. The report [30] examines it in detail though.

We emphasise that the parallelisations performed by Law 9 and Law 10 are to align the model with the SCJ paradigm and architecture. In other words, they do not parallelise the computations of the respective handlers, which are still performed in sequence here. This reflects that any sequentialism in an SCJ design needs to be explicitly enforced, while parallel execution (of handlers) is the default.

For multiple applications of the two laws, we also require the application of several elementary *Circus* laws between each application of Law 9 or Law 10. Their purpose is firstly to extract the newly introduced channel c to the outer level of the mission action in which the targeted (refined) action is embedded, and secondly to distribute prefixes $c\,[?x] \longrightarrow A_2$ introduced in the right-hand parallel action into A_2, namely if A_2 is itself an action sequence or parallelism.

We conclude by observing that the first parallelisation Law 9 targets precisely the shape of models generated by earlier application of Law 1 (Fig. 3), and the second parallelisation Law 10 precisely the shape of models generated by earlier application of Law 2 (Fig. 4), subsequent to replacing Z compositions by action sequences, which is done collaterally as part of the distribution of time budgets.

Laws for parallel handler designs A key law for transforming parallel data operations modelled by conjunctions into parallel actions is presented in Fig. 9. It applies to data operations Op_1 and Op_2 that write to disjoint sets of variables, which is what we usually expect from a parallelism at that level.

Although this law permits us to replace parallel data operations by parallel actions, this might not immediately yield a top-level parallelism of handlers as present in our refinement target in Fig. 2. It shows, in general, that due to the fact that the conjunction might be embedded into action sequences (see Law 3), there is still a considerable number of refinement steps and specialised laws required to arrive at the desired shape. In particular, these refinements involved further decomposition of time budgets related to the particular parallel design adopted. Applying Law 11, for instance, to the result of Law 3 (Fig. 5), we observe that there still remains a sequential composition with MOp. We can parallelise it using Law 10 in the previous section, but this does not completely eliminate it due to a prefix emerging in the left parallel action. In [30], we precisely detail the basic refinement steps that are needed prior and subsequent to application of Law 9; they involve two specialised laws: one for channel decomposition and one for distribution of an interleaving of basic communications into a preceding parallelism.

We also consider high-level parallelisation laws. Namely, Law 12 in Fig. 10 directly targets shapes emerging from parallelising data operation via Law 3 and at the same time caters for further decomposition of time budgets. This shows in the time budgets POp_{TB}, Rec_{TB} and $Merge_{TB}$ replacing the global time budget Op_{TB}. We hence have a proviso $POp_{TB} + n * Rec_{TB} + Merge_{TB} \leq Op_{TB}$ that considers the time allowance of the parallelised operations to compute the partial results, the time to record them, and the time needed to merge them. The concrete value of these budgets has to be determined by the developer as part of the verification process.

Law 12 $\mathbf{wait}\ 0\,..\,Op_{TB}\,;$ *RHS of Law 3* $\sqsubseteq$

$$
\left(\begin{array}{l}
\left(\begin{array}{l}
(\mathbf{var}\, r_1 : T \bullet \mathbf{wait}\ 0\,..\,POp_{TB}\,;\ (\exists i? : \mathbb{Z} \bullet i? = 1)\,;\ rec\,!\,r_1 \longrightarrow \mathbf{skip}) \parallel \\
(\mathbf{var}\, r_2 : T \bullet \mathbf{wait}\ 0\,..\,POp_{TB}\,;\ (\exists i? : \mathbb{Z} \bullet i? = 2)\,;\ rec\,!\,r_2 \longrightarrow \mathbf{skip}) \parallel \\
\ldots \\
(\mathbf{var}\, r_n : T \bullet \mathbf{wait}\ 0\,..\,POp_{TB}\,;\ (\exists i? : \mathbb{Z} \bullet i? = n)\,;\ rec\,!\,r_n \longrightarrow \mathbf{skip})
\end{array}\right) \\
\quad [\![\varnothing \mid \{\!| rec |\!\} \mid \{r\}]\!] \\
\left(\begin{array}{l}
\mathbf{var}\, r_1, r_2, \ldots, r_n : T \bullet \\
\quad \left(\begin{array}{l}
(rec\,?x \longrightarrow \mathbf{wait}\ 0\,..\,Rec_{TB}\,;\ r_1 := x) \\
(rec\,?x \longrightarrow \mathbf{wait}\ 0\,..\,Rec_{TB}\,;\ r_2 := x) \\
\ldots \\
(rec\,?x \longrightarrow \mathbf{wait}\ 0\,..\,Rec_{TB}\,;\ r_n := x)
\end{array}\right)\,; \\
\quad \mathbf{wait}\ 0\,..\,Merge_{TB}\,;\ MOp([\![r_1, r_2, \ldots, r_n]\!])
\end{array}\right)
\end{array}\right)
$$

provided $POp_{TB} + n * Rec_{TB} + Merge_{TB} \leq Op_{TB}$

Figure 10: High-level law for refining parallel data operations into actions.

A design artifact of Law 12 is that it introduces a fresh typed channel *rec* that is used to communicate the partial results to a parallel operation that receives and merges them into the final result. From this, a control fragment emerges that is later refined into shared data to hold the partial result(s); it contributes to the *HdlControl* action in Fig. 2 and its refinement gives rise to further design of how partial results are stored and processed; this relies on its own set of laws which are omitted here.

To conclude this aspect of the refinement, we observe that we can either tackle it by way of applying the more general Law 11, or use specialised high-level laws like Law 12 that encapsulate particular designs. Since it is still an open issue how the general case can profit from further elementary laws and their automation, we recommend the use of high-level laws. Therefore, we assume that for every decomposition law into a parallel data operation, there exists at least one specialised action law that directly targets the emerging shape. So far, this appears to be the case, however, further experience needs to be gained to ascertain this. In [30], we sketch a proof of Law 12 which uses a few novel and interesting elementary laws. Beyond this, future work may propose alternative parallelisation laws with more sophisticated merge operations that can, for instance, deal with partial results of heterogeneous type. We next look at an example that illustrates the refinement of a realistic SCJ program.

4 Example

As an example, we consider the refinement of an action that models the behaviour of the collision detector (CD_x benchmark) in [18]. The CD_x SCJ program consists of a single mission that periodically carries out the following tasks: reading a set of aircraft positions from a radar device, calculating their predicted motions, and identifying the number of aircraft at risk of colliding due to their distances decreasing below a certain threshold. Whereas [18] provides a sequential implementation using a single handler, we have developed a parallel program by breaking down the mission design into seven handlers: (1) a cyclic input handler that reads the next radar frame; (2) a reducer handler that performs a voxel-hashing algorithm, which partitions the space; (3) four parallel detector handlers that carry out the detection work; and (4) an output handler that communicates the result.

Our starting point is the abstract operation *ComputeCycle* in Fig. 11. It is embedded into an action

$$
\begin{array}{l}
\textit{ComputeCycle} \\
\hline
\Delta[\textit{currentFrame} : \textit{RawFrame};\ \textit{state} : \textit{StateTable};\ \textit{work} : \textit{Partition};\ \textit{collisions} : \mathbb{Z}] \\
\textit{frame}? : \textit{Frame} \\
\hline
\exists \textit{posns}, \textit{posns}', \textit{motions}, \textit{motions}' : \textit{Frame} \mid \\
\quad \mathrm{dom}\,\textit{posns} = \mathrm{dom}\,\textit{motions} \land \mathrm{dom}\,\textit{posns}' = \mathrm{dom}\,\textit{motions}' \bullet \\
\exists \textit{voxel\_map} : \textit{HashMap}[\textit{Vector2d}, \textit{List}[\textit{Motion}]] \mid \textit{voxel\_map} \neq \mathbf{null} \bullet \\
\left(\begin{array}{l}
\textit{posns}' = \textit{frame}? \land \\
\textit{motions}' = (\lambda\, a : \mathrm{dom}\,\textit{posns}' \bullet \mathbf{if}\, a \in \mathrm{dom}\,\textit{posns}\, \mathbf{then}\, (\textit{posns}'\, a) -_V (\textit{posns}\, a)\, \mathbf{else}\, \textit{ZeroV}) \land \\
\textit{posns} = F(\textit{currentFrame}) \land \textit{motions} = G(\textit{currentFrame}, \textit{state}) \land \\
\textit{posns}' = F(\textit{currentFrame}') \land \textit{motions}' = G(\textit{currentFrame}', \textit{state}') \land \\
\left(\begin{array}{l}
\forall\, a_1, a_2 : \textit{Aircraft} \mid \{a_1, a_2\} \subseteq \mathrm{dom}\,\textit{posns}' \bullet \\
(a_1, a_2) \in \textit{CalcCollisionSet}(\textit{posns}', \textit{motions}') \Rightarrow \\
\quad \left(\begin{array}{l}
\exists\, l : \textit{List}[\textit{Motion}] \mid l \in \textit{voxel\_map}\,.\,\textit{values}()\,.\,\textit{elems}() \bullet \\
\text{“predicate that states the collision pair } (a_1, a_2) \text{ is in } l\text{”}
\end{array}\right)
\end{array}\right) \land \\
\textit{voxel\_map}\,.\,\textit{values}()\,.\,\textit{elems}() = \bigcup\{i : 1\,..\,4 \bullet \textit{work}'\,.\,\textit{getDetectorWork}(i)\,.\,\textit{elems}()\} \land \\
\exists\, \textit{collset} : \mathbb{F}\,(\textit{Aircraft} \times \textit{Aircraft}) \mid \textit{collset} = \textit{CalcCollisionSet}(\textit{posns}', \textit{motions}') \bullet \\
\quad (\#\textit{collset} = 0 \land \textit{collisions}' = 0) \lor (\#\textit{collset} > 0 \land \textit{collisions}' \geq (\#\textit{collset})\ \mathrm{div}\ 2)
\end{array}\right)
\end{array}
$$

Figure 11: Z operation specifying the cyclic mission behaviour of the CD_x.

that defines the cyclic mission behaviour, as specified below.

$$
\begin{array}{l}
\textit{CDxMission} \mathrel{\widehat=} \mu X \bullet \\
\quad \left(\left(\begin{array}{l}
\left(\begin{array}{l}
(\textit{next\_frame}\,?\textit{frame} \longrightarrow \textit{ComputeCycle}) \blacktriangleleft \textit{INP\_DL}; \\
\mathbf{wait}\, 0\,..\,(\textit{FRAME\_PERIOD} - \textit{INP\_DL} - \textit{OUT\_DL}); \\
(\textit{output\_collisions}\,!\textit{collisions} \longrightarrow \mathbf{skip}) \blacktriangleleft \textit{OUT\_DL}
\end{array} \right) \\
\interleave\ \mathbf{wait}\, \textit{FRAME\_PERIOD}
\end{array} \right); X \right)
\end{array}
$$

The channel *next_frame* (of a type *Frame* encoding radar frames) is used to read the next frame of aircraft positions, and *output_collisions* (of type $\mathbb{Z}$) to output the detected number of collisions. Collisions are computed by *ComputeCycle* and stored in a state component *collisions*. The constant *FRAME_PERIOD* determines the length of a cycle, and *INP_DL* and *OUT_DL* are deadlines on external communications. We observe that *ComputeCycle* is equipped with a time budget $\textit{FRAME\_PERIOD} - \textit{INP\_DL} - \textit{OUT\_DL}$, obtained by subtracting from the cycle time the maximal amount of time that the communications are permitted to take. Besides, *RawFrame*, *StateTable* and *Partition* are *OhCircus* classes.

We start by decomposing *ComputeCycle* into sequences and conjunctions of data operations. This is done by applying Law 2 three times, followed by an application of Law 3. This is not trivial, however, since the *ComputeCycle* operation contains further existentially quantified variables that either correspond to abstract model variables (*posns* and *motions*) here arising from earlier data refinement, or local variables like *voxel_map*, capturing the result of the voxel-hashing algorithm. These quantifiers either have to be eliminated using the one-point rule, or localised to predicates corresponding to single handlers.

Another issue that needs to be addressed is that the data flow is not always explicit in abstract operations specifying missions. In our SCJ program, for example, data is transmitted between the reducer handler that carries out the voxel-hashing, and the detector handlers that perform the detection. That is, the reducer handler writes to the component *work* which determines how the computational work is split,

and this variable is also read by the detector handlers. In the operation, the last existential conjunct

$$\exists collset : \mathbb{F}(Aircraft \times Aircraft) \mid collset = CalcCollisionSet(posns', motions') \bullet \\ (\#collset = 0 \land collisions' = 0) \lor (\#collset > 0 \land collisions' \geq (\#collset) \operatorname{div} 2)$$

models the detector handlers, and we notice that the new value of *collisions* is determined by the function $CalcCollisions(posns', motions')$ in terms of the abstract model variables. To reformulate it in terms of *work* requires some *ad hoc* refinements that appear to be difficult to automate by a machine. We skip further details and merely present the result of the decomposition.

$$CDxMission \mathrel{\widehat=} \mu X \bullet \\ \left(\left(\left(\left(\left(\begin{array}{l} next\_frame?frame \longrightarrow \\ \left(\begin{array}{l} RecordFrame; \\ ReduceAndPartitionWork; \\ DetectCollisions \end{array} \right) \end{array} \right) \blacktriangleleft INP\_DL; \right. \right. \right. \right. \\ \left. \left. \left. \left. \begin{array}{l} \mathbf{wait}\, 0 \,..\, (FRAME\_PERIOD - INP\_DL - OUT\_DL); \\ (output\_collisions\,!\,collisions \longrightarrow \mathbf{skip}) \blacktriangleleft OUT\_DL \end{array} \right) \right) \right) \right. \\ \left. \left. \left. \left. \;\|\!\| \, \mathbf{wait}\, FRAME\_PERIOD \right) \right) \right) \right) ; X \Bigg)$$

where the decomposed Z operations *CalcPartCollisions*, *SetCollisionsFromParts* and *DetectCollisions* can be found in Appendix A. We next decompose and distribute the time budget between the newly introduced sequential operations. For this, we introduce the handler-specific time budgets RF_{TB}, RPW_{TB} and DC_{TB}. This yields the following refinement.

$$CDxMission \mathrel{\widehat=} \mu X \bullet \\ \left(\left(\begin{array}{l} next\_frame?frame \longrightarrow \\ \left(\begin{array}{l} \mathbf{wait}\, 0 \,..\, RF_{TB}\,;\, RecordFrame; \\ \mathbf{wait}\, 0 \,..\, RPW_{TB}; \\ ReduceAndPartitionWork; \\ \mathbf{wait}\, 0 \,..\, DC_{TB}\,;\, DetectCollisions \end{array} \right) \blacktriangleleft INP\_DL \\ \ldots \end{array} \right) ; X \right)$$

The time budget DC_{TB} is further decomposed during the parallelisation of actions. For the last part of the refinement parallelising the action above, we refer to the detailed description in our technical report [30], which is available from `http://www.cs.york.ac.uk/circus/publications/techreports/`. It entails applying Law 10 and Law 12, and after a finalising transformation that uses elementary laws and can be automated, too, we obtain an action that has the shape in Fig. 2.

5 Conclusion

We have presented a collection of *Circus* refinement laws that can be used to refine sequential specifications of SCJ mission behaviour into parallel designs that match the SCJ Level 1 programming model. Our refined models are a suitable starting point for further refinement of shared data and control mechanisms. We have also highlighted challenges for automation: they are, primarily, in the decomposition of sequential and parallel data operations, and to provide a repository of parallelisation laws, both at the level of data operations and actions, that deal with a wide spectrum of recurring program designs. Due to the novelty of SCJ, there are still open issues related to the designs that ought to be supported, and hence

we do not claim completeness at this stage. On the other hand, our results showed that the decomposition of time budgets can largely be automated, and so can (the intermediate steps in) the refinement of data operations into parallel handler actions, which ultimately creates a positive outlook. Like in SCJ, our model and strategy also supports data being shared between missions. But this is less of an issue for the refinement laws because no write conflicts or race conditions can arise. The mission design in fact emerges where sequential actions of an abstract centralised model are retained during refinement.

In practical terms, we propose to facilitate the decomposition of data operations, the more difficult aspect of a refinement, by asking the developer to identify intermediate target models that permit the application of one of the decomposition laws. Each intermediate model generates a refinement proof obligation which can be tackled in isolation, and, as we hope, its resolution will be able to take some advantage of automatic refinement tactics. The development of useful tactics is still work in progress, however, their mechanisation may use a tool like [31] to ensure soundness of refinements and laws alike.

An open issue is the validation of our laws against a semantics for the particular combination of *Circus* languages that we use. Our recent work explores in detail the semantics of *Circus* Time, and this shall provide a platform to prove, for instance, the laws about time budgets in Section 3.2. Further work is, however, required to integrate that semantics with that of *OhCircus*. And importantly, we require a proof that the laws from either language (*OhCircus* and *Circus* Time) hold within the combined language. The Unifying Theories of Programming (UTP) [17], the common semantic foundation for all *Circus* dialects, ought to facilitate such a proof. It is an issue that is high on our agenda of research.

Related work includes action systems and their refinement [2, 3]. Action systems combine state and behaviour by away of atomic *actions* that operate on the state and that can be executed concurrently if there are no write conflicts to variables. Like *Circus*, action systems come with an extensive refinement calculus, supporting the refinement of centralised sequential specifications into distributed implementations [2, 4]. The computational paradigm is, however, more restrictive since actions have to adhere to a specific form, whereas *Circus* actions can, for instance, use all of CSP's constructs.

Event-B [1] is a practically-oriented formalism closely-related to action systems; it has been successfully used in the formal development of distributed systems in academia and industry. Research has been prompted to overcome initial restrictions of the method to deal with decomposition [6] and time [7]. It would be interesting to see whether Event-B would be expressive enough for SCJ handler models, and whether the refinement laws we propose can be formulated and perhaps validated.

SCJ is still a very recent technology, and, as far as we know, this is the first work that looks at refinement more specifically in the context of the SCJ programming model. Our results though contribute to a wider objective of proposing and proving refinement laws for *all* aspects of the verification of SCJ programs. These are, among others, data refinements in *Circus* Time and the introduction of class objects, the refinement of shared data and use of object references, and the transformation of models into *SCJCircus*, a new language sufficiently concrete to be directly translatable into code. They are all immediate areas for future work, each bringing its own set of challenges for refinement and automation.

Acknowledgements This work was funded by the EPSRC grant EP/H017461/1. We are grateful to Andy Wellings for many useful clarifications of SCJ, and we also thank the anonymous reviewers for their pertinent and useful suggestions.

References

[1] J. R. Abrial (2010): *Modeling in Event-B*. Cambridge University Press, Cambridge, CB2 8BS, UK, doi:10.1017/CBO9781139195881.

[2] R. J. R. Back (1990): *Refinement Calculus, Part II: Parallel and Reactive Programs*. In: *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, *LNCS* 430, Springer, pp. 67–93, doi:10.1007/3-540-52559-9_61.

[3] R. J. R. Back & R. Kurki-Suonio (1983): *Decentralization of Process Nets with Centralized Control*. In: *Proceedings of PODC '83*, ACM, pp. 131–142, doi:10.1145/800221.806716.

[4] R. J. R. Back & J. von Wright (2003): *Compositional Action System Refinement*. *Formal Aspects of Computing* 15, pp. 103–117, doi:10.1007/s00165-003-0005-6.

[5] A. Burns (1999): *The Ravenscar Profile*. *ACM SIGAda Ada Letters* XIX, pp. 49–52, doi:10.1145/340396.340450.

[6] M. Butler (2009): *Decomposition Structures for Event-B*. In: *Proceedings of IFM 2009*, *LNCS* 5423, Springer, Düsseldorf, Germany, pp. 20–38, doi:10.1007/978-3-642-00255-7_2.

[7] D. Cansell, D. Méry & J. Rehm (2006): *Time Constraint Patterns for Event B Development*. In: *Proceedings of B 2007*, *LNCS* 4355, Springer, Besançon, France, pp. 140–154, doi:10.1007/11955757_13.

[8] A. Cavalcanti (1997): *A Refinement Calculus for Z*. Ph.D. thesis, University of Oxford, UK, Oxford, OX1 3QD, UK. Available at `http://www.cs.ox.ac.uk/files/3451/PRG123.pdf`.

[9] A. Cavalcanti, A. Sampaio & J. Woodcock (2003): *A Refinement Strategy for Circus*. *Formal Aspects of Computing* 15, pp. 146–181, doi:10.1007/s00165-003-0006-5.

[10] A. Cavalcanti, A. Sampaio & J. Woodcock (2005): *Unifying classes and processes*. *Software and Systems Modeling* 4(3), pp. 277–296, doi:10.1007/s10270-005-0085-2.

[11] A. Cavalcanti, A. Wellings & J. Woodcock (2011): *The Safety-Critical Java Memory Model: A Formal Account*. In: *Proceedings of FM 2011*, *LNCS* 6664, Springer, Limerick, Ireland, pp. 246–261, doi:10.1007/978-3-642-21437-0_20.

[12] A. Cavalcanti, F. Zeyda, A. Wellings, J. Woodcock & K. Wei (2012): *Safety-Critical java programs from Circus models*. *Real-time Systems*, doi:10.1007/s11241-013-9182-4. Under publication.

[13] A. Dalsgaard, R. Hansen & M. Schoeberl (2012): *Private Memory Allocation Analysis for Safety-Critical Java*. In: *Proceedings of JTRES 2012*, ACM, pp. 9–17, doi:10.1145/2388936.2388939.

[14] L. Groves (2002): *Refinement and the Z Schema Calculus*. *Electronic Notes in Theoretical Computer Science* 70, pp. 70–93, doi:10.1016/S1571-0661(05)80486-4.

[15] G. Haddad & G. Leavens (2011): *Specifying Subtypes in SCJ Programs*. In: *Proceedings of JTRES 2011*, ACM, pp. 40–46, doi:10.1145/2043910.2043917.

[16] T. Henties, J. Hunt, D. Locke, K. Nilsen, M. Schoeberl & J. Vitek (2009): *Java for Safety-Critical Applications*. In: *Proceedings of SafeCert 2009*, York, UK, pp. 1–11. Available at `http://www.vmars.tuwien.ac.at/php/pserver/extern/download.php?fileid=1641`.

[17] C. A. R. Hoare & H. Jifeng (1998): *Unifying Theories of Programming*. Prentice Hall Series in Computer Science, Prentice Hall, Upper Saddle River, NJ, USA.

[18] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer & J. Vitek (2009): *CD_x: A Family of Real-time Java Benchmarks*. In: *Proceedings of JTRES 2009*, ACM, pp. 41–50, doi:10.1145/1620405.1620412.

[19] C. C. Morgan (1990): *Programming from Specifications*. Prentice Hall International Series in Computer Science, Prentice Hall, Upper Saddle River, NJ, USA.

[20] M. Oliveira (2005): *Formal Derivation of State-Rich Reactive Programs using Circus*. Ph.D. thesis, University of York, York, YO10 5GH, UK. Available at `http://www.cs.york.ac.uk/circus/publications/papers/06-oliveira.pdf`.

[21] M. Oliveira, A. Cavalcanti & J. Woodcock (2009): *A UTP semantics for Circus*. *Formal Aspects of Computing* 21, pp. 3–32, doi:10.1007/s00165-007-0052-5.

[22] A. W. Roscoe (1997): *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science, Prentice Hall, Upper Saddle River, NJ, USA.

[23] A. Sherif, A. Cavalcanti, H. Jifeng & A. Sampaio (2009): *A process algebraic framework for specification and validation of real-time systems*. Formal Aspects of Computing 22, pp. 153–191, doi:10.1007/s00165-009-0119-6.

[24] H. Søndergaard, B Thomsen & A. P. Ravn (2006): *A Ravenscar-Java Profile Implementation*. In: *Proceedings of JTRES 2006*, ACM, Paris, France, pp. 38–47, doi:10.1145/1167999.1168008.

[25] D. Tang, A. Plsek & J. Vitek (2010): *Static Checking of Safety Critical Java Annotations*. In: *Proceedings of JTRES 2010*, ACM, pp. 148–154, doi:10.1145/1850771.1850792.

[26] The Open Group (2011): *Safety Critical Java Technology Specification*. Technical Report JSR-302, Java Community Process. Available at http://jcp.org/en/jsr/detail?id=302.

[27] A. Wellings (2004): *Concurrent and Real-Time Programming in Java*. Wiley, West Sussex, PO19 8SQ, UK.

[28] J. Woodcock & J. Davies (1996): *Using Z: Specification, Refinement and Proof*. Prentice Hall International Series in Computer Science, Prentice Hall, Upper Saddle River, NJ, USA.

[29] F. Zeyda, A. Cavalcanti & A. Wellings (2011): *The Safety-Critical Java Mission Model: A Formal Account*. In: *Proceedings of ICFEM 2011*, *LNCS* 6991, Springer, Durham, UK, pp. 49–65, doi:10.1007/978-3-642-24559-6_6.

[30] F. Zeyda, A. Cavalcanti, A. Wellings, J. Woodcock & K. Wei (2012): *Refinement of the Parallel CD$_x$*. Technical Report, University of York, York, UK. Available at http://www.cs.york.ac.uk/circus/publications/techreports/.

[31] F. Zeyda, M. Oliveira & A. Cavalcanti (2012): *Mechanised support for sound refinement tactics*. Formal Aspects of Computing 24(1), pp. 127–160, doi:10.1007/s00165-011-0218-z.

A Decomposed data operations of the CD$_x$ example

CalcPartCollisions

$$\Xi[currentFrame : RawFrame;\ state : StateTable;\ work : Partition;\ collisions : \mathbb{Z}]$$
$$i? : 1\,..\,4$$
$$pcolls! : \mathbb{Z}$$
$$pcolls! = \#\{a_1 : Aircraft;\ a_2 : Aircraft \mid \exists l : work\,.\,getDetectorWork(i?).elems\,() \bullet \ldots\} \operatorname{div} 2$$

SetCollisionsFromParts

$$\Delta[currentFrame : RawFrame;\ state : StateTable;\ work : Partition;\ collisions : \mathbb{Z}]$$
$$collsbag? : \text{bag } int$$
$$currentFrame' = currentFrame \land state' = state \land voxel\_map' = voxel\_map \land work' = work$$
$$\exists s : \text{seq } int \mid s = items\, collsbag? \bullet collisions' = \Sigma s$$

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