Under consideration for publication in Formal Aspects of Computing

Laws of mission-based programming

Frank Zeyda and Ana Cavalcanti

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

Abstract. Safety-Critical Java (SCJ) is a recent technology that changes 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, as they are used in the SCJ programming paradigm. Our refinement 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. Starting from a sequential and centralised *Circus* specification, our laws permit refinement into *Circus* models of SCJ program designs. Automation and proof of the refinement laws is examined here, too. Our work is an important step towards eliciting laws of programming for SCJ and fits into a refinement strategy that we have developed previously to derive SCJ programs from specifications in a rigorous manner.

Keywords: SCJ; models; refinement; laws; patterns; automation; proof; *Circus*

1. Introduction

Java is indisputably one of the most popular programming languages nowadays. 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 [STR06, The11], which render it inadequate for time-critical applications. Safety-Critical Java (SCJ) [HHL$^+$09], a recent initiative, addresses these issues by introducing a restricted version of Java; it is based on the Real-Time Specification for Java (RTSJ) [Wel04], but further restricts RTSJ's execution and memory model. SCJ requires programs to conform with the SCJ execution paradigm, which is based on missions and handlers. 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 [RTC11].

SCJ is organised in three levels (Levels 0 to 2), which define progressively more complex models of execution. Our focus is SCJ Level 1, which roughly corresponds to the Ravenscar profile for Ada [Bur99]. 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. Missions and handlers are defined by application classes that either

Correspondence and offprint requests to: Frank Zeyda, Department of Computer Science, Deramore Lane, University of York, Heslington, York, YO10 5GH, UK. Email: frank.zeyda@york.ac.uk

extend or implement an abstract or interface class of the SCJ API; that API is defined as part of the SCJ technology [The11]. Handlers can either be periodic, which means they are released at regular time intervals, or aperiodic implying that they are released by some external event or stimulus. When a handler is released, its `handleAsyncEvent()` method is scheduled for execution by a priority-based scheduler that is part of a specialised virtual machine for executing SCJ programs.

Our previous work focused on complementing the informal account of SCJ [The11] with a formal model of SCJ's mission-based execution paradigm [ZCW11] and memory model [CWW11a]. Our modelling notation is a combination of languages from the *Circus* family [CSW03, CSW05, SCJS10], specifically tailored for state-rich reactive systems with the addition of discrete time, object orientation, and object references.

We have also proposed a refinement strategy [CZW$^+$13] to transform abstract specifications into models that directly correspond to SCJ programs. Such a strategy simultaneously addresses a multitude of concerns. Namely, we have to consider the preservation of real-time behaviour, the introduction of classes and object references, the SCJ memory model, the SCJ execution model, and the SCJ application interface. Therefore, it is not surprising that the existing work [CZW$^+$13] only gives a broad description of the general approach; details of the application of this strategy to a specific example are available in [ZCW$^+$12].

Our primary contribution in this paper is to examine in detail the refinement of centralised and sequential specifications of missions into parallel handler designs. In doing so, we elaborate and extend our account of laws in our previous work [ZC13]. Our starting point for the refinement is a *Circus* process specification that supports all constructs of *Circus*, including Z data operations, classes, and Timed CSP constructs, except for parallel composition and interleaving. We show how decomposition of data operations, time budgets, and process actions can be used to transform such a model into a uniform shape that determines the structure and behaviour of handlers of a mission. Refinement laws directly reflect particular program designs that encapsulate the way in which data is shared and safely accessed to avoid race conditions, how computational work is divided between the handlers of a mission, and how handler execution is controlled.

Another contribution is an account on automating the application of the laws. In particular, we discuss how different stages of the refinement may take advantage of automation and potential tactics for refinement, and what level of expertise is required in each aspect of the refinement to guide the formal development. We also present proofs of a few of the novel refinement laws, and thereby illustrate a strategy for validation of the laws with respect to a denotational semantics of our language. That semantics has been recently formulated in the context of Hoare and He's Unifying Theories of Programming (UTP) [HJ98].

The principal motivation for our work is to pave the way for automated tool support for the verification of SCJ programs. 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 [TPV10, DHS12, HL11], 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 [ZCW$^+$12]. Some refinement steps are, however, inherently difficult to automate. Our work highlights where automation is feasible, and where human guidance is indispensable to guide the refinement process.

The added contributions of this article with respect to our earlier account on refinement laws for SCJ in [ZC13] can be summarised as follows.

1. We refine the notion of an SCJ program design by considering termination of missions (Section 3).
2. We present laws for two aspects of the refinement that have not been considered so far. The first one is the introduction of cycle timings for periodic handlers (Section 4.1), and the second one is the design of shared data access as well as explicit mechanisms for handler control (Section 4.5).
3. We discuss in detail opportunities and ramifications for automating the application of the laws (Section 6).
4. We present proofs for some of the laws and thus illustrate the feasibility of validating the proposed refinement laws (Section 7). This is with respect to a UTP-based semantics of our language.
5. We elaborate our account of applying the refinement laws to our collision detector case study.

The results in this paper contribute towards elaborating the refinement strategy for SCJ in [CZW$^+$13], but they are also useful outside the context of that technique. Decomposition of centralised models is a general issue in refinement-based techniques [CSW03], and the models we produce can, in principle, serve as a starting point for any form of parallel implementation in languages other than SCJ. As the essence of

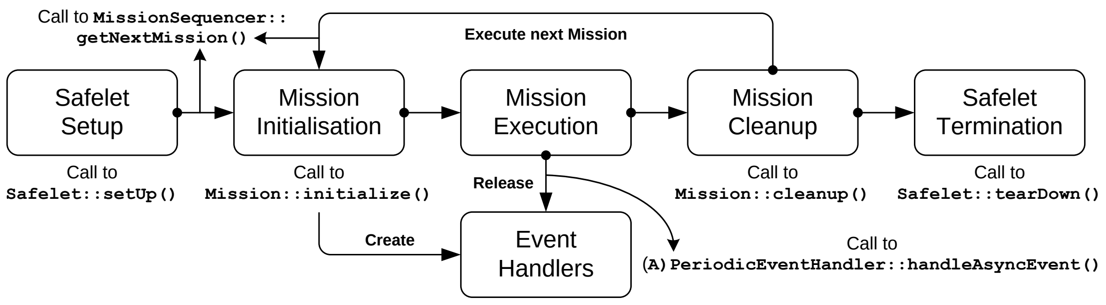
Call to MissionSequencer::getNextMission()
Execute next Mission
Safelet Setup
Mission Initialisation
Mission Execution
Mission Cleanup
Safelet Termination
Call to Safelet::setUp()
Call to Mission::initialize()
Release
Call to Mission::cleanup()
Call to Safelet::tearDown()
Create
Event Handlers
Call to (A) PeriodicEventHandler::handleAsyncEvent()

Fig. 1. Life-cycle of a safelet during execution of a Level 1 application.

the SCJ paradigm (its mission-based execution model) can be captured independently of the Java language, our account of mission decomposition is relevant for languages that adopt a similar execution model, too.

The structure of the article is as follows. In Section 2, we review preliminary material: Safety-Critical Java, the *Circus* family of languages, and a refinement strategy for development or verification of SCJ. Section 3 presents the *Circus* model of an SCJ program design targeted by the refinement. Section 4 then discusses our refinement laws by considering five independent design and verification aspects, and Section 5 presents an example that illustrates the application of the laws: a collision detector for aircraft. Next, Section 6 considers issues of automation, and in Section 7 we examine the proof of some of the laws. Finally, in Section 8, we conclude, and discuss related and future work.

2. Preliminaries

We next discuss in more detail SCJ Level 1 (Section 2.1), the *Circus* family of notations (Section 2.2), and our top-level refinement strategy (Section 2.3). We use a version of the collision detector for aircraft (CD_x) in [KHP+09] that has been adapted for SCJ Level 1 as a running example to explain the SCJ technology and our formal notation, and later on also to demonstrate the refinement laws. The executable SCJ program and additional documentation are available from `http://www.cs.york.ac.uk/circus/hijac/cdx.html`.

2.1. SCJ Level 1

Safety-Critical Java (SCJ) is a restriction of the standard Java language and JDK API. SCJ prohibits certain features of Java that are difficult to analyse for program certification or otherwise deemed unsafe in a safety-critical context, like garbage collection or unconstrained use of `synchronized` blocks. Moreover, SCJ requires program designs to adhere to particular structures, and therefore includes its own API that provides classes and interfaces that enable the user to write applications that conform to SCJ's execution paradigm. In detail, SCJ programs are distinguished by compliance with one of the three SCJ Levels 0, 1 and 2. These levels define progressively more complex application architectures and underlying execution models, and are each supported by a specific set of (abstract) classes and interfaces, provided by the SCJ technology [The11].

Level 0 applications define a set of sequential tasks that are periodically executed by a cyclic scheduler. SCJ Level 1 adopts the more elaborate execution model based on missions and handlers. SCJ Level 2 relaxes certain constraints on the use of synchronisation primitives in SCJ Level 1 and moreover supports nesting of missions. Our focus in this article is SCJ Level 1. The execution model for SCJ Level 1 programs is based on five primary conceptual entities: safelet, mission sequencer, missions, handlers and SCJ events. They are realised by the following abstract classes and interfaces: `Safelet`, `MissionSequencer`, `Mission`, `PeriodicEventHandler`, `AperiodicEventHandler` and `AperiodicEvent`. Concrete Level 1 programs implement these interfaces and classes and can then be executed by a specialised virtual machine for SCJ.

Figure 1 illustrates the life-cycle of a Level 1 safelet, the top-level entity of an SCJ application. The SCJ infrastructure, that is, an SCJ-compliant virtual machine, 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. Interaction of the infrastructure with the program to carry out these (and other) tasks

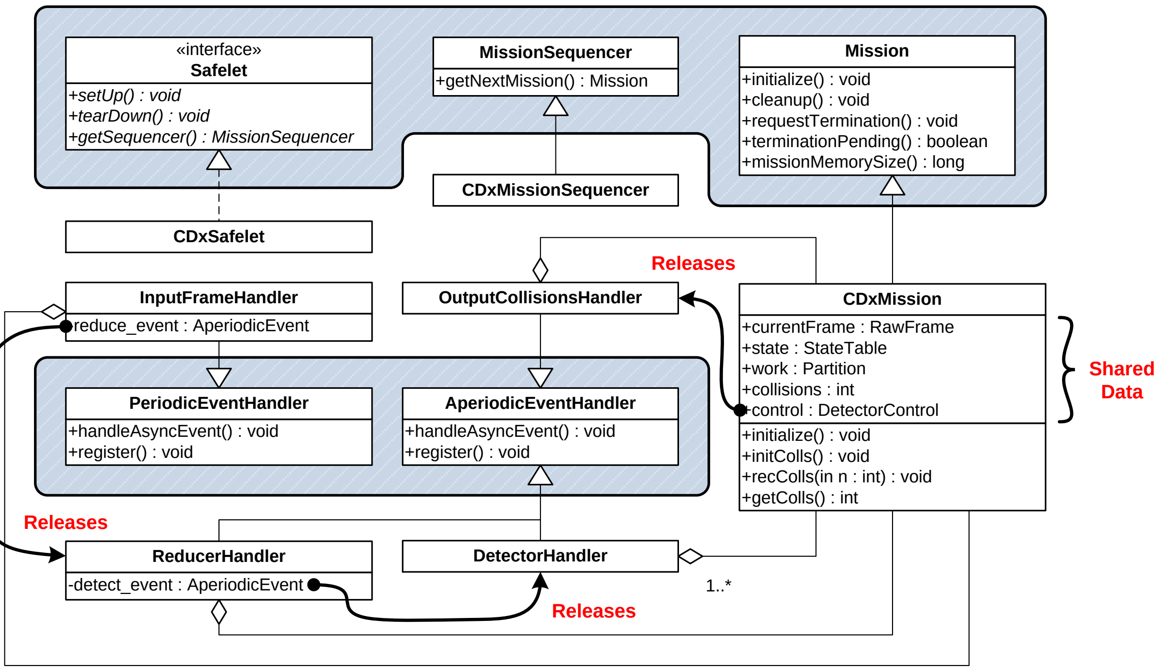

Fig. 2. Class diagram for an aircraft collision detector in Level 1 SCJ.

is done by method calls. Mission initialisation creates the mission's event handlers, which are released either periodically or by an external or SCJ event during mission execution. Whereas external events are raised by the environment, SCJ events are fired in software (by a method call) and enable applications to exercise explicit control over handler releases. SCJ events are modelled by instances of the `AperiodicEvent` class, which is also part of the SCJ API. When there are no more missions to execute, the safelet terminates.

Figure 2 includes the UML class diagram for a particular SCJ application: the collision detector (CD_x). Whereas the original CD_x [KHP+09] was designed for RTSJ, our version is a recast for SCJ Level 1 and takes advantage of multiple handlers that parallelise the detection of collisions. Classes surrounded by a blob belong to the SCJ API, and the remaining ones are application classes. The latter include `CDxSafelet`, `CDxMissionSequencer`, `CDxMission`, `InputFrameHandler`, `OutputCollisionsHandler`, `ReducerHandler` and `DetectorHandler`. We observe that they all implement an entity of the SCJ API.

An instance of the `CDxSafelet` class provides the safelet of the application and `CDxMissionSequencer` defines the mission sequencer, which here specifies the execution of a single mission `CDxMission` that also holds the data shared between the handlers. The remaining classes are handlers. `InputFrameHandler` is periodic whereas `ReducerHandler`, `DetectorHandler` and `OutputCollisionsHandler` are aperiodic and released by various SCJ events (of type `AperiodicEvent`). The purpose of `InputFrameHandler` is to periodically read radar frames of aircraft positions from an external device. Afterwards, it releases `ReducerHandler`, which reduces and divides the computational work using a spacial decomposition algorithm. The actual detection of collisions is performed by four instances of the `DetectorHandler` class; these instances record their partial results using the `void recColls(int)` method of the mission class. Once all detector handlers complete their work, `OutputCollisionsHandler` is released to output the result to a warning system. (`DetectorControl` is a utility class to control the release of `OutputCollisionsHandler`.) We note that the program depicted in the UML diagram has been *a priori* constructed, but in Section 5 we verify it by showing how its design emerges from the refinement laws that we discuss in Section 4.

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. The method `getSequencer()` (see Figure 2) is called on the safelet object to obtain the mission sequencer of the application, which defines the sequence of missions to execute (here it returns an instance of `CDxMissionSequencer`). In addition, various methods are called by the infrastructure on the mission sequencer, mission, and handler objects during execution of the safelet. Most notably, those shown in Figure 2 are `getNextMission()` to obtain the next mission to execute, `initialize()` to create the handlers of a mission, and `handleAsyncEvent()` when a handler is released. As mentioned before, an SCJ program must provide implementations of these methods and thereby define the structure of the application in terms of missions and handlers. The code for the `CDxMission` class is included in Appendix G and illustrates the creation of handlers, SCJ events and shared data, as well as methods to safely access the shared data. We note that although the missions and handlers of a safelet are generally determined at run-time, we shall assume that they are *a priori* fixed. This simplifies the model, and all SCJ programs we encountered so far adhere to it!

When a mission terminates, `cleanup()` is called on the mission object to perform application-specific cleanup tasks. The entire safelet terminates when there are no more missions to execute, signalled in the program by `getNextMission()` returning a `null` reference instead of a mission object.

Memory in SCJ is organised in scoped areas where each scope has a predefined life-span with respect to the safelet's life-cycle. Scoped memory eradicates the need for garbage collection while access rules on scopes alleviate problems of dangling references [PFHV04]. First, we have immortal memory, which is never released and thus may contain objects shared between missions. Further, each mission has its own mission memory area that remains valid for the duration of mission execution and is used for shared data between the handlers of a mission. Finally, handler methods execute in their own private scope when a handler is released; that scope is for temporary objects and reclaimed each time `handleAsyncEvent()` terminates.

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 key components that implement the behaviour of the program, and the main focus of our work here. SCJ events are moreover relevant for control mechanisms that release handlers explicitly in the program.

2.2. The *Circus* family

Circus [CSW03] is a language for specification and refinement of state-rich reactive systems. It combines notations from Z [WD96], CSP [Ros97], and Morgan's refinement calculus [Mor94]. 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 [OCW09] defined using Hoare and He's Unifying Theories (UTP) [HJ98].

An example of a *Circus* process is given in Figure 3. It is the specification of the CD_x program introduced in the previous section. As already noted, the collision detector exhibits a cyclic behaviour in which each cycle entails reading aircraft positions from a radar device, computing the number of pairs of aircraft at risk of colliding, and outputting the result to a warning system. The name of the process is $CDxSpec$, and its state is defined by the $CDxState$ Z schema, introducing the components $posns$ and $motions$ of type $Frame$. They are respectively used to record the positions and trajectories of the aircraft currently in view of the radar. The type $Frame$ is introduced as the set of partial and finite functions from aircraft identifiers to 3d vectors: $Frame \mathrel{\widehat=} Aircraft \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\, Vector$. The state invariant $\operatorname{dom} posns = \operatorname{dom} motions$ ensures that we record a motion trajectory for each visible aircraft. We note that in general, the state components of a process can either have Z (schema) types as in $CDxSpec$, or *OhCircus* [CSW05] class types as we use them later on.

Next, we have a sequence of local action definitions for the actions $Init$, $RecordFrame$, $CalcCollisions$ and $CDxCycle$. The actual behaviour of the process is specified by its main action after the '$\bullet$' at the bottom and, typically, makes use of the local actions. (Here, $Init$ to initialise the state and $CDxCycle$ to execute a single detection cycle.) Actions may either be specified using Z operation schemas, as in $Init$, $RecordFrame$ and $CalcCollisions$, or using a mixture of CSP constructs and guarded commands, as in $CDxCycle$. We also admit timed actions from *Circus Time* [SCJS10], which is based on a discrete-time version of Timed CSP [RR88, Ros11]. Our formal modelling notation is, therefore, a combination of *Circus*, *OhCircus* and *Circus Time*. The UTP enables us to give a sound semantic foundation to this combination of languages.

First, the $Init$ action of $CDxSpec$ initialises the state components to empty functions. This is by constraining primed state components only which, by convention, refer to their values after execution of an operation (unprimed variables refer to their initial values). We note that $CDxState\,'$, $\Delta CDxState$ and $\Xi CDxState$ in the declaration parts of the schema actions are all different ways of introducing primed state components. Whereas $CDxState\,'$ does not include initial (unprimed) variables, $\Delta CDxState$ enables us to refer to both, initial (unprimed) and final (primed) values of state components in the predicate of the operation schema. And $\Xi CDxState$ moreover incorporates an additional implicit constraint that the state components must not be changed by the operation. For example, $RecordFrame$ alters the value of the state components $posns$

$$\mathbf{process}\ CDxSpec \mathrel{\widehat=} \mathbf{begin}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathbf{state}\ CDxState} \\ \hline
posns : Frame;\ motions : Frame \quad (\text{where } Frame \mathrel{\widehat=} Aircraft \mathrel{+\!\!\!\!\twoheadrightarrow} Vector) \\ \hline
\operatorname{dom} posns = \operatorname{dom} motions \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{Init} \\ \hline
CDxState' \\ \hline
posns' = \varnothing \land motions' = \varnothing \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{RecordFrame} \\ \hline
\Delta CDxState;\ frame? : Frame \\ \hline
posns' = frame? \\
motions' = (\lambda\, a : \operatorname{dom} posns' \bullet \mathbf{if}\ a \in \operatorname{dom} posns\ \mathbf{then}\ (posns'\ a) -_V (posns\ a)\ \mathbf{else}\ ZeroV) \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{CalcCollisions} \\ \hline
\Xi CDxState;\ colls! : \mathbb{N} \\ \hline
\exists\, collset : \mathbb{F}\,(Aircraft \times Aircraft) \mid collset = CollSet(posns, motions) \bullet colls! = (\#\, collset)\ \operatorname{div} 2 \\ \hline
\end{array}$$

$$CDxCycle \mathrel{\widehat=} \left(\begin{array}{l}
next\_frame\,?\,frame\ @\ t_1 \longrightarrow \\
\left(\begin{array}{l}
RecordFrame; \\
\mathbf{wait}\ w : 0 \mathinner{..} FRAME\_PERIOD - OUT\_DL - t_1 \bullet \\
\mathbf{var}\ colls : \mathbb{N} \bullet CalcCollisions; \\
\left(\begin{array}{l}
output\_collisions\,!\,colls\ @\ t_2 \longrightarrow \\
\mathbf{wait}\ FRAME\_PERIOD - (t_1 + w + t_2)
\end{array}\right) \blacktriangleleft OUT\_DL
\end{array}\right)
\end{array}\right) \blacktriangleleft INP\_DL$$

$$\bullet\ Init\ ;\ (\mu X \bullet CDxCycle\ ;\ X)$$

$$\mathbf{end}$$

Fig. 3. Process example: specification of a collision detector.

and *motions*, and *CalcCollisions* implicitly leaves them unchanged. The decorations '?' and '!' are, as usual in Z, used to identify inputs and outputs of an operation; for instance, *frame*? is an input parameter of *RecordFrame* and *colls*! is an output of *CalcCollisions*.

The second action, *RecordFrame*, records a frame of aircraft positions, provided by the parameter *frame*?, in the state of the process. In addition, it records aircraft motions by calculating the differences of the current and previous aircraft positions. Where aircraft first appear on the radar, their motion is set to zero. The third action, *CalcCollisions*, as noted above, does not alter the state, but instead outputs the number of aircraft that are at risk of colliding as the distance between their predicted trajectories falls below a certain threshold. It uses an auxiliary function *CollSet* to compute a set of pairs containing all such aircraft; we divide its size by 2 to account for symmetry of the *collset* relation. The definition of *CollSet* is omitted here for brevity, but can be found in [ZCW+12].

Whereas Z operations are useful to specify computations, we require CSP actions to specify interactions with the environment as well as timing properties. As mentioned above, we introduce *CDxCycle* to define the behaviour of a single detection cycle. This action makes use of two communication channels, *next_frame* (of type *Frame*) to read the next frame of aircraft positions from the radar device, and *output_collisions* (of type $\mathbb{N}$) to output the number of collisions to a warning system. Generally, a prefixed action $c \longrightarrow A$ waits for communication on a channel c before proceeding with A. Special kinds of prefixes are inputs and outputs: we have that $c\,!\,e \longrightarrow A$ outputs a value e on the channel c while $c\,?\,x \longrightarrow A(x)$ reads and binds it to a local identifier x. The parallel composition $(c\,!\,e \longrightarrow A_1) \llbracket ns_1 \mid \{\!|\, c \,|\!\} \mid ns_2 \rrbracket (c\,?\,x \longrightarrow A_2(x))$, explained in more detail in Section 3, thus results in a value being communicated from the left to the right parallel action.

The timed prefix $next\_frame\,?\,frame\ @\ t \longrightarrow A$ is an input communication that binds *frame* to the value read, and moreover assigns to t the amount of time the communication was offered before it actually took place. The *Circus Time* action $(\ldots \blacktriangleleft INP\_DL)$ imposes a deadline on the communication to occur, reflecting the assumption that an environment makes the data available within INP_DL time units at the beginning of each cycle. INP_DL is a constant, whose declaration is omitted in Figure 3 and whose value is left unspecified. Whereas $A \blacktriangleleft t$, in general, is a deadline on some observable interaction of A with the environment, we also have an alternative construct $A \blacktriangleright t$, which is a deadline on A to terminate within t time units. For clarification, we point out that all *Circus Time* constructs take relative times as their arguments.

After reading the frame, we next invoke $RecordFrame$ to store the frame in the state of the process and calculate motion trajectories. This is followed by a nondeterministic delay: in general, $\mathbf{wait}\, w : t_1 \mathinner{..} t_2$ delays execution between t_1 and t_2 time units. Similar to the timed prefix, it binds the actual amount of time waited to a local variable, here w. If we are not interested in that time, we can use the plain and shortened form $\mathbf{wait}\, t_1 \mathinner{..} t_2$, which has the same effect but does not introduce w. Nondeterministic waits are typically used to define time budgets for an implementation to carry out some computational task, which here is the calculation of collisions via the $CalcCollisions$ operation. The time budget allocated to $CalcCollisions$ is the frame period ($FRAME\_PERIOD$) less the maximum time it may take to output the collisions (OUT_DL), and less the time t_1 already taken to read the radar frame. Fundamentally, data operations in *Circus Time* are always instantaneous, hence all timing behaviour has to be specified explicitly by deadlines and delays.

The number of detected collisions is stored in a local variable $colls$, declared by $\mathbf{var}\, colls : \mathbb{N} \bullet \ldots$; it is initialised by the call to $CalcCollisions$. Subsequently, collisions are output on the channel $output\_collisions$, via a timed output prefix that again records the amount of time that the communication was offered prior to being taken. The deadline ensures that it must, however, take place within OUT_DL time units, which is an imposition on the environment to accept the output in a set interval. In addition to nondeterministic delays, we may also have simple (deterministic) delays $\mathbf{wait}\, t$ where t defines the duration. The final $\mathbf{wait}\, FRAME\_PERIOD - (t_1 + w + t_2)$ delays execution so that each cycle takes exactly $FRAME\_PERIOD$ time units. For feasibility of the model, we moreover assume that $INP\_DL + OUT\_DL < FRAME\_PERIOD$ holds. This is formalised as part of the omitted loose specification of those three constants.

The overall behaviour of the CD_x is specified by the main action at the bottom, and consists of initialising the state (action $Init$) and then using a recursive action to repetitively invoke the cyclic behaviour specified by $CDxCycle$. The operator $\mu X \bullet F(X)$ denotes the weakest fixed point (with respect to refinement) of a function F on actions. It is used to define recursive actions since uses of X in F correspond to recursive calls. We observe that this process is entirely sequential and does not use any form of parallel composition. We shall return to it in Section 5 to illustrate the application of the laws we present in Section 4 in order to transform $CDxSpec$ into the *Circus* model of an SCJ program design.

2.3. Refinement strategy for SCJ

We next describe the SCJ refinement strategy, which can be used for development or verification of existing programs. It is a refinement procedure, organised in three steps, where each step is carried out by the application of refinement laws, some of which are the object of the work we present in this paper. Figure 4 presents the major models used in that refinement strategy. We refer to them as anchors in accordance with [CWW+11b], where the term 'anchor' was first introduced for the intermediate target models of our refinement strategy. A detailed discussion of each anchor can additionally be found in [CZW+13].

A anchor The A (abstract) anchor entails the abstract specification. In this model, nothing is said about objects yet, and the language that we use is a combination of *Circus* and *Circus Time*. Parallelism at this level is typically used to structure and conjoin requirements. Usually, models are expressed as a parallel composition of behavioural and timing requirements $BReqs \llbracket \ldots \rrbracket TReqs$. Such parallel compositions are later collapsed in the E anchor. We note that, for simplicity, we do not make use of parallel composition at all in the specification of the CD_x in Figure 3.

O anchor The O (object-oriented) anchor changes the way data is represented by introducing objects to record the abstract data in the A anchor; it, therefore, additionally uses constructs from *OhCircus* (classes, method calls, and so on). The refinement that is carried out in the construction of this anchor is a data refinement with added steps that introduce *OhCircus* class objects for schema types. For example, the abstract type $Frame$ in the process in Figure 3 is later refined in the O anchor into the *OhCircus* classes $RawFrame$ and $StateTable$ that are used to record aircraft positions and motions in the SCJ program. This aspect of the verification is not a concern for the laws we discuss in this paper. The report [ZCW+12], however, includes a detailed derivation of this anchor for the CD_x.

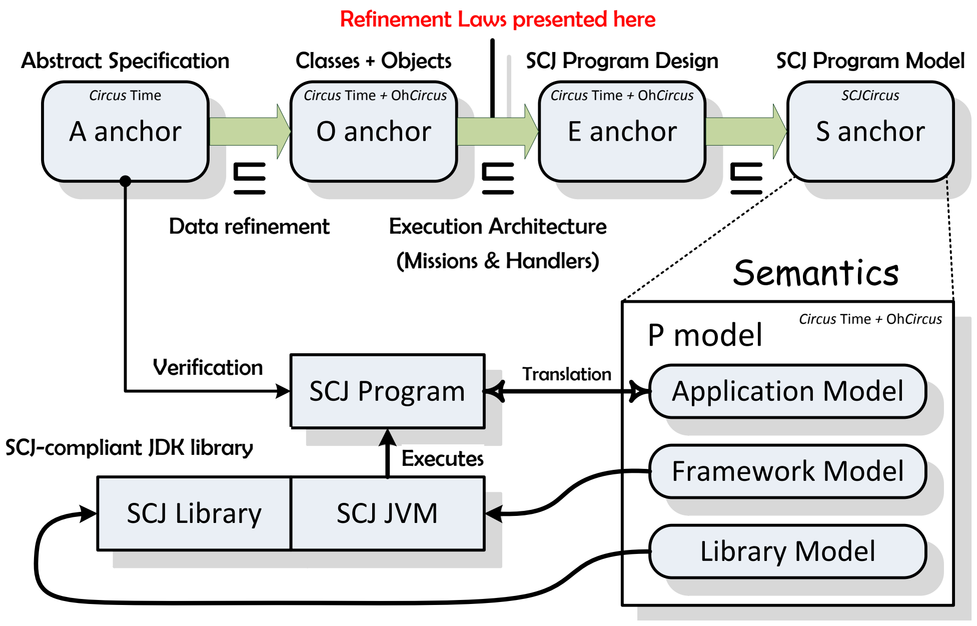

Fig. 4. Refinement strategy for verifying SCJ programs.

E anchor The E (execution) anchor introduces the SCJ program design in terms of missions and handlers. It is this step of the refinement strategy that the laws we present in Section 4 cater for, and the following Section 3 makes the structure of this anchor more precise. While an E anchor captures the essence of the mission-based execution paradigm and architecture, it is not the accurate model of an SCJ program yet and hence cannot be directly translated into code. It is the next and final anchor that yields such a model.

S anchor The S (SCJ) anchor factors the E anchor into two independent and concurrent parts: an application model that corresponds closely to an SCJ program and a predefined and fixed framework model that encapsulates the generic behaviour of the SCJ virtual machine to execute the program. The language of the S anchor is *SCJCircus*; it introduces special constructs that resemble the main entities of an SCJ program, namely the safelet, mission sequencer, missions, handlers, and SCJ events. There is a direct correspondence between S anchor models and SCJ programs, so that a translation can be performed automatically. The semantics of the S anchor is defined in terms of the P model, which gives meaning to all *SCJCircus* constructs and also determines the fixed *Circus* model of the SCJ framework.

While the S anchor is close to a concrete program, the E anchor concisely encapsulates the *design* of a program in terms of the missions and handlers. Our interest here are laws that can produce such designs. The subsequent refinement to the S anchor is challenging in its own right, but not particularly interesting in terms of the execution paradigm, which already emerges in the E anchor. From the sound construction of each anchor, we also obtain that the S anchor is a sound refinement of the A anchor, although the refinement techniques used vary across anchors. This is due to a unified semantic notion of refinement in our languages.

3. *Circus* model of an SCJ program design

Figure 5 presents another *Circus* process; it illustrates the general form of an E anchor, and, as already said, the laws we discuss in the next section transform (sequential) specifications of safelets into processes of precisely that shape. The name of the process here is *SCJDesign*, and its state is defined by the *IMState* schema, introducing the components c_i of type T_i. (*Inv* is an optional state invariant.)

The state components here are used to represent data objects in immortal memory. We then have local action definitions for $Setup$, $Mission_i$, $Init_i$, $Handlers_i$, $Handler_j$, $HdlTermCtrl$, $MArea_i$, $Cleanup_i$ and $Teardown$. The $Setup$ and $Teardown$ actions correspond directly to the `setUp()` and `tearDown()` methods

$$\begin{array}{l}
\textbf{process}\ SCJDesign \mathrel{\widehat=} \textbf{begin} \\
\textbf{state}\ IMState == [\, c_1 : T_1;\ c_2 : T_2;\ \ldots;\ c_n : T_n \mid Inv\,(c_1, c_2, \ldots, c_n)\,] \\
Setup \mathrel{\widehat=} [\, IMState' \mid P_{Init}(c_1', c_2', \ldots, c_n')\,]\ ;\ \ldots \\
Mission_i \mathrel{\widehat=} \left(\begin{array}{l} \left(\begin{array}{l} Init_i; \\ \left(\begin{array}{l} Handlers_i \\ \llbracket IMState \mid \{\!|\, termReq, termMsn \,|\!\} \mid \varnothing \rrbracket \\ HdlTermCtrl \end{array}\right) \setminus \{\!|\, termReq, termMsn \,|\!\}; \\ Cleanup_i \end{array}\right) \\ \llbracket IMState \mid cs_{sh} \mid \varnothing \rrbracket\ MArea_i \end{array}\right) \setminus cs_{sh} \\
Init_i = \ldots \\
Handlers_i \mathrel{\widehat=} \left(\begin{array}{l} Handler_1 \\ \llbracket ns_1 \mid cs_1 \cup \{\!|\, termMsn \,|\!\} \mid ns_2 \cup \ldots \cup ns_h \rrbracket \\ Handler_2 \\ \llbracket ns_2 \mid cs_2 \cup \{\!|\, termMsn \,|\!\} \mid ns_3 \cup \ldots \cup ns_h \rrbracket \\ \ldots \\ \llbracket ns_{h-1} \mid cs_{h-1} \cup \{\!|\, termMsn \,|\!\} \mid ns_h \rrbracket \\ Handler_h \end{array}\right) \setminus cs_1 \cup cs_2 \cup \ldots \cup cs_{h-1} \\
\text{where } Handler_j \text{ may be either defined as} \\
Handler_j \mathrel{\widehat=} \text{(for an aperiodic handler)} \qquad\qquad Handler_j \mathrel{\widehat=} \text{(for a periodic handler)} \\
\quad \left(\mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} e_{(j,1)}\,?\,v \longrightarrow A(v) \;\Box \\ e_{(j,2)}\,?\,v \longrightarrow A(v) \;\Box \\ \ldots \end{array}\right);\ X \\ \Box\ termMsn \longrightarrow \textbf{skip} \end{array}\right)\right) \quad \text{or} \quad \left(\mu X \bullet \left(\begin{array}{l} (A \blacktriangleright T \mathrel{|\!|\!|} \textbf{wait}\ T)\ ;\ X \\ \Box\ termMsn \longrightarrow \textbf{skip} \end{array}\right)\right) \\
HdlTermCtrl \mathrel{\widehat=} termReq \longrightarrow (\mu X \bullet termReq \longrightarrow X\ \Box\ termMsn \longrightarrow \textbf{skip}) \\
Cleanup_i = \ldots \\
MArea_i = \textbf{var}\ v_1 : T_1;\ v_2 : T_2;\ \ldots;\ v_m : T_m \bullet \ldots \\
Teardown \mathrel{\widehat=} \ldots \\
\bullet\ Setup\ ;\ Mission_1\ ;\ Mission_2\ ;\ \ldots\ ;\ Mission_m\ ;\ Teardown \\
\textbf{end}
\end{array}$$

Fig. 5. Target for the refinement that produces the `E` anchor.

of the class implementing the `Safelet` interface. Likewise, $Init_i$ and $Cleanup_i$ model the `initialize()` and `cleanup()` methods of concrete subclasses of the `Mission` class.

The behaviour of the process, as defined by its main action at the bottom, is first to invoke *Setup*, which initialises the state and thereby all variables in immortal memory. The '...' in the *Setup* action indicates that we may carry out other application-specific interactions here too, for instance, to reset or enable external devices. We then have a sequence of mission executions whose models are specified by the actions $Mission_i$. Lastly, *Teardown* is called to perform custom tasks, if applicable, for shutting down the safelet.

The model of a mission is in essence the parallel composition of its handlers. This composition is captured by the action $Handlers_i$ for some mission i. It uses a mission-specific set of $Handler_j$ actions that provide the model for individual event handlers. In *Circus*, the parallel composition of two actions A_1 and A_2 is written as $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, 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 modify. (We note that parallel composition is right-associative.) Hence, all handlers of a mission write to mutually disjoint parts of the state space, determined by the variable sets ns_i. This ensures that all *Circus* constructs (including parallel composition) are monotonic with respect to refinement by way of enforcing non-interference in shared data access. Monotonicity is crucial for piecewise development to ensure, for instance, that process actions can be refined individually to obtain a refinement of the process as a whole. We note that interleaving $(A_1 \mathrel{|\!|\!|} A_2)$

is a special case of parallel composition where the synchronisation set cs is empty. Termination in parallel compositions and interleavings only occurs when both parallel actions have terminated.

The complete model of a mission, as described by the actions $Mission_i$, composes in parallel a further action $HdlTermCtrl$ to incorporate a control mechanism for termination of the mission via the channels $termReq$ (for a termination request raised by one of the handlers) and $termMsn$ (to terminate the handlers). Termination requests can thus be issued asynchronously, whereas termination takes place synchronously. This corresponds to the protocol defined for the `requestTermination()` method of the `Mission` class in the SCJ technology specification [The11]. The $Init_i$ and $Cleanup_i$ actions are sequenced to perform mission initialisation and cleanup tasks. In addition, we have a further parallel action $MArea_i$ for each mission i. It encapsulates data that resides in mission memory and, therefore, is shared between the handlers. The shared data objects are introduced by way of local variables in $MArea_i$; they are read and modified by virtue of communications on a designated set of channels cs_{sh}. In terms of SCJ program design, $MArea_i$ defines protocols to access shared data safely so that race conditions cannot occur.

We observe that the channels $termReq$ and $termMsn$, as well as those in cs_{sh}, are hidden using the $A \setminus cs$ construct. Hidden channels cannot be observed by the environment anymore, and synchronisation on them takes place internally and as soon as possible. Hence, the only channels that are exposed by $SCJDesign$ are those that correspond to external events releasing one of the aperiodic handlers, as explained next.

The handler models, captured by the actions $Handler_j$, 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 via a synchronisation on $termMsn$. The event $termMsn$ is raised by the control action $HdlTermCtrl$ subsequent to a termination request, which can be issued by any of the handlers at any time through synchronising on $termReq$. The operator $A_1 \Box A_2$ is external choice: its resolution is done by the environment. For instance, in $(c \longrightarrow \mathbf{skip}) \,\Box\, (d \longrightarrow \mathbf{skip})$, both communications c and d are offered. This is in contrast with a nondeterministic (or internal) choice $A_1 \sqcap A_2$, where the environment has no control over the outcome of the choice. For example, the action $(c \longrightarrow \mathbf{skip}) \sqcap (d \longrightarrow \mathbf{skip})$ can arbitrarily choose to offer to the environment the communication c or d.

Termination of a handler is indeed enforced since, as explained above, $termMsn$ is hidden in the $Mission_i$ actions and takes place immediately when enabled. Aperiodic handlers are modelled by an external choice that synchronises on a set of channels $e_{(j,1)}$, $e_{(j,2)}$, and so on, which correspond to external or SCJ events bound to the handler j and, therefore, cause its release. Potentially, each event provides an input v; the handler's `handleAsyncEvent()` method is specified by $A(v)$.

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

The time t that is used in actions such as $A \blacktriangleright t$ and $\mathbf{wait}\ t$ is specification time: it abstractly captures requirements related to deadlines and delays. For instance, $A \blacktriangleright t$ and $A \blacktriangleleft t$ express the requirement that A terminates or interacts within t time units, respectively. When translating A into code, worst-case execution time (WCET) analysis is performed to verify that all deadlines are met, and only then do we consider the particular timing characteristics of a concrete hardware and execution environment. This is following Hayes' approach to refining real-time systems [HU01]. The main advantage of Hayes' technique is that, during refinement, we can ignore the real-time characteristics of an execution environment. WCET analysis can be subtle depending on the target architecture, but techniques for it have been well studied and are supported by various tools [W$^+$08]. This is hence not an issue we focus on in this article.

While Section 2.2 illustrated the starting point, in this section we have formally defined the target of our refinement technique and made precise the shape of models that we consider to be SCJ designs. The latter retain a deliberate degree of abstraction in not restricting the shape of the handler and memory area actions. Next, we discuss the refinement laws that enable the transformation between those two models.

4. Refinement Laws

Refinement laws are generally of the form $A \sqsubseteq B$, where A is the refined action, and B the refining action that replaces A when the law is applied. We also have equivalence laws $A \equiv B$, which imply refinement in both directions: $A \sqsubseteq B$ and $B \sqsubseteq A$. We consider five aspects of the verification of a mission implementation. Each aspect is dealt with by a collection of specialised refinement laws that we discuss in detail. The first aspect is the introduction of cycle timings by way of interleavings with **wait** T statements (Section 4.1). The second aspect is decomposition of data operations to introduce functional models of handlers (Section 4.2). The third is distribution of time budgets between the handlers (Section 4.3). The fourth is parallelisation of handlers to match the architecture of Level 1 SCJ, as formalised by the process in Figure 5; this also addresses the design of data flow and control mechanisms via communications (Section 4.4). And the fifth is the encapsulation of shared data and mechanisms to access it safely (Section 4.5). The *Circus* refinement laws we present address each of these verification issues in isolation and independently of each other.

Although some of the laws have already been given in [Cav97] and [CSW03], the majority of the laws here are novel. This includes, in particular, the *Circus Time* laws in Section 4.1 (Figure 7) and Section 4.3 (Figure 11), and the sharing laws in Section 4.5. The parallelisation laws in Figure 10 and Figure 15 are to our knowledge novel, too. Automation and proof issues are not discussed here, but separately in Sections 6 and 7.

4.1. Introduction of cycle timings

An important aspect of an SCJ program are its cycle timings. These are needed to model periodic handlers, as well as any handlers that have a cyclic behaviour constrained by deadlines and time budgets. This is a commonly found scenario in control applications.

In our approach, cycle timings are defined by recursions of the form $\mu X \bullet (A \blacktriangleright T_p \interleave \textbf{wait}\ T_p)\ ;\ X$, where A does not reference X, and T_p is the period of the cycle. We target the transformation of recursions to obtain recursions of this form. We observe that actions of this shape resemble the model for periodic handlers discussed in Section 3, though the body A here does not yet correspond to the model of a single handler, but instead abstractly specifies some cyclic activity, which may also be implemented by a mission, for example.

The refinement laws discussed in this section are applicable to actions that somehow define a cyclic behaviour with a fixed cycle length. We characterise the general form of such actions as given below, where the action A specifies the behaviour of a single cycle and does not include recursive calls to X. We use the notation $A(B)_1$ to denote an action A that includes a single occurrence of an action B.

$$\mu X \bullet A(\textbf{wait}\ T_p - t_A)_1\ ;\ X \tag{1}$$

The laws presented in this section are useful when the purpose of the **wait** statement is to fill the gap between the termination of A and the start of the next cycle. In this context, T_p is a constant that determines the cycle length, and t_A an expression that yields the elapsed execution time of A.

We note that not all actions of the above shape exhibit a cyclic behaviour with a fixed cycle length. For instance, there may be execution paths in A that do not execute the **wait** $T_p - t_A$ statement. In some cases, control flow analysis can establish if this is so, but generally it is not statically checkable. This is not a problem, though, since subsequent law applications fail if the targeted recursion does not have an intrinsic fixed cycle length. In that case, a design via a periodic handler is not appropriate and this step of the refinement strategy does not apply. The following steps, however, remain useful.

Our first goal is to transform such actions to the form $\mu X \bullet (A \interleave \textbf{wait}\ T_p)\ ;\ X$. For the subsequent introduction of the termination deadline on A, one possibility is the use of the simple law $A \sqsubseteq A \blacktriangleright T$ for any T, yielding an action $\mu X \bullet (A \blacktriangleright T_p \interleave \textbf{wait}\ T_p)\ ;\ X$ as described above. As explained in Section 3, the termination deadline T_p introduces an assumption for the worst-case execution time of A that has to be discharged when A is translated into program code, hence the law itself does not require a proviso. However, an issue already pointed out in [HU01] is the possibility of introducing so-called impossible deadlines, which cannot be met by any implementation on any machine due to contradictory model constraints on timing. Such deadlines do not affect the soundness of the technique, since the translation into code always catches them in stride, but preferably we would like to detect them earlier on in the verification.

We hence equip the law for deadline introduction with a supplementary proviso.

Law 1. $A \equiv A \blacktriangleright t$ provided $\mathit{TakesAtMost}(A) \leq t$

The function $\mathit{TakesAtMost}(A)$ yields an approximation (upper bound) for the specified execution time of A; it is defined in Figure 6 for all relevant action constructs. We note that the proviso is not concerned with the actual execution time of A on a concrete machine, which is, as before, an issue for code generation. We use $\mathit{TakesAtMost}(A)$ as a conservative oracle for the specified execution time of A.

We have, for instance, that data operations Op and **skip**, the action that terminates immediately without altering the state, take no time, but **stop**, the action that deadlocks, takes an infinite amount of time

Def 1. Calculation of an upper bound for the execution time of an action.

$$\begin{array}{|l}
\mathit{TakesAtMost} : \mathit{Action} \to (\mathit{TIME} \cup \{\infty\}) \\
\hline
\forall A, A_1, A_2 : \mathit{Action};\ d, t_1, t_2 : \mathit{TIME};\ \mathit{ns}_1, \mathit{ns}_2 : \mathit{NameSet};\ \mathit{cs} : \mathit{ChannelSet} \bullet \\
\mathit{TakesAtMost}(\mathit{Op}) = 0 \\
\mathit{TakesAtMost}(\mathbf{skip}) = 0 \\
\mathit{TakesAtMost}(\mathbf{stop}) = \infty \\
\mathit{TakesAtMost}(\mathbf{wait}\ d) = d \\
\mathit{TakesAtMost}(\mathbf{wait}\ t_1 \mathrel{..} t_2) = t_2 \\
\mathit{TakesAtMost}(c \longrightarrow A) = \infty \ \text{ where the prefix is not embedded in a synchronisation deadline.} \\
\mathit{TakesAtMost}((c \longrightarrow A) \blacktriangleleft d) = d + \mathit{TakesAtMost}(A) \\
\mathit{TakesAtMost}(A \blacktriangleright d) = \min(\mathit{TakesAtMost}(A), d) \\
\mathit{TakesAtMost}(A_1 ; A_2) = \mathit{TakesAtMost}(A_1) + \mathit{TakesAtMost}(A_2) \\
\mathit{TakesAtMost}(A_1 \Box A_2) = \max(\mathit{TakesAtMost}(A_1), \mathit{TakesAtMost}(A_2)) \\
\mathit{TakesAtMost}(A_1 \sqcap A_2) = \max(\mathit{TakesAtMost}(A_1), \mathit{TakesAtMost}(A_2)) \\
\mathit{TakesAtMost}(A_1 \mathbin{|||} A_2) = \max(\mathit{TakesAtMost}(A_1), \mathit{TakesAtMost}(A_2)) \\
\mathit{TakesAtMost}(\mathbf{var}\ v : T \bullet A) = \mathit{TakesAtMost}(A) \\
\mathit{TakesAtMost}(A \setminus \mathit{cs}) = \mathit{TakesAtMost}(A) \\
\mathit{TakesAtMost}(\mu X \bullet F(X)) = n * \mathit{TakesAtMost}(F(\mathbf{skip})) \\
\text{where } n \text{ is an upper bound for the number of recursive calls before the action terminates.}
\end{array}$$

Fig. 6. Definition of a function *TakesAtMost* to determine (upper) execution time bounds.

as it never terminates. A time budget **wait** $t_1 \mathrel{..} t_2$ models the fact that there is the possibility that the execution of an implementation may take between t_1 and t_2 time units, hence it takes at most t_2 time units. Communications $c \longrightarrow A$ are assumed to take infinite time as the environment is at liberty to postpone them. This is unless they are embedded in a synchronisation deadline; in that case, the deadline determines the execution time of the communication. For a termination deadline $A \blacktriangleright d$, we know that the action cannot take longer than d time units, but A may finish before that, hence the minimum construction, using $\mathit{TakesAtMost}(A)$ as an oracle for the execution time of A. The composition operators consider all possible execution paths using sums and maxima. For a recursive action $\mu X \bullet F(X)$, we require a means either to infer or predict the maximum number of recursive calls. The result of $\mathit{TakesAtMost}(A)$ is an overapproximation, since we, in a pessimistic approach, take maxima over branching execution paths.

Another refinement law we require is used to introduce an interleaving of **wait** $T_p - t_A$ with **skip**.

Law 2. $A \equiv \mathbf{skip} \mathbin{|||} A$

To do so, the action A in Law 2 above is matched against the **wait** $T_p - t_A$ when applying the law.

After the application of Law 2, the introduced interleaving is embedded in A as identified in (1). We next use a collection of laws to extract it from A. These laws are summarised in Figure 7 and need to be applied exhaustively. They are special distribution laws for interleaving with a time delay.

We note that the laws cannot extract such interleavings from arbitrarily-shaped actions. For instance, we cannot extract the **wait** T construct from an action $c \longrightarrow (A \mathbin{|||} \mathbf{wait}\ T)$ unless we have some information about the synchronisation time on c. Otherwise, $c \longrightarrow (A \mathbin{|||} \mathbf{wait}\ T)$ can take however much time c takes in addition to at least T time units, and in contrast, in $(c \longrightarrow A) \mathbin{|||} \mathbf{wait}\ T$ the waiting time is not added to the synchronisation time. So, just in cases where we have some knowledge about the synchronisation time (via an enclosing deadline), we can relate these actions, as it is captured by Law 4 in Figure 7.

Where extraction of the **wait** is not possible, it may be the case that the abstract model cannot be refined into a design that uses periodic handlers. Yet, the refinement may produce alternative designs that realise periodic activities, for instance, by way of aperiodic handlers that make use of explicit mechanisms for time control. So, this is not a limitation on refinement *per se* but rather on design.

Law 3 relies on the fact that data operations *Op* are instantaneous. A weaker version of the law may be specified in which *Op* is replaced by any action that (as a proviso) does not consume time. Law 4 deals with timed prefixes. Here, the prefix is a simple synchronisation, but the law can be easily generalised to input and output prefixes on the channel c. We observe that the argument of the **wait** statement in Law 4 needs to have a particular shape to enable application of the law. Moreover, the prefix has to be embedded

Law 3. $Op\ ;\ (A \mathbin{|||} \textbf{wait}\, t) \equiv (Op\ ;\ A) \mathbin{|||} \textbf{wait}\, t$

Law 4. $(c \mathbin{@} t \longrightarrow (A(t) \mathbin{|||} \textbf{wait}\ T - t)) \blacktriangleleft d \equiv (c \mathbin{@} t \longrightarrow A(t)) \blacktriangleleft d \mathbin{|||} \textbf{wait}\ T$ provided $d \leq T$

Law 5. $\textbf{wait}\, t : t_1 \mathinner{..} t_2 \bullet (A(t) \mathbin{|||} \textbf{wait}\ T - t) \equiv (\textbf{wait}\, t : t_1 \mathinner{..} t_2 \bullet A(t)) \mathbin{|||} \textbf{wait}\ T$ provided $t_2 \leq T$

Law 6. $(A \mathbin{|||} \textbf{wait}\, t) \blacktriangleleft d \equiv (A \blacktriangleleft d) \mathbin{|||} \textbf{wait}\, t$

Law 7. $(A \mathbin{|||} \textbf{wait}\, t) \blacktriangleright d \equiv (A \blacktriangleright d) \mathbin{|||} \textbf{wait}\, t$ provided $t \leq d$

Law 8. $(A_1 \mathbin{|||} \textbf{wait}\, t) \sqcap (A_2 \mathbin{|||} \textbf{wait}\, t) \equiv (A_1 \sqcap A_2) \mathbin{|||} \textbf{wait}\, t$

Law 9. $(A_1 \mathbin{|||} \textbf{wait}\, t) \mathbin{\Box} (A_2 \mathbin{|||} \textbf{wait}\, t) \equiv (A_1 \mathbin{\Box} A_2) \mathbin{|||} \textbf{wait}\, t$

Law 10. $(A \mathbin{|||} \textbf{wait}\, t) \setminus cs \equiv (A \setminus cs) \mathbin{|||} \textbf{wait}\, t$

Law 11. $\textbf{var}\, v : T \bullet (A \mathbin{|||} \textbf{wait}\, t) \equiv (\textbf{var}\, v : T \bullet A) \mathbin{|||} \textbf{wait}\, t$

Fig. 7. Laws for extraction of **wait** statements for cycle timings.

in a synchronisation deadline. Where this, initially, is not the case, transformation laws can be applied that distribute synchronisation deadlines through action operators; they are in Figure 24 in Appendix A.

Law 5 extracts an interleaving with **wait** $T - t$ from a time budget. Here, t is introduced by the budget to refer to the actual time waited. The remaining Laws 6–10 are straightforward distribution laws. Notably, Law 8 and Law 9 for internal and external choice require the delay to be the same in both actions. Again, supplementary transformations may be applied to rewrite actions into a shape that enables the application of these laws; the laws we discuss later on in Section 4.3 are useful for this purpose too.

To illustrate the use of the laws, we apply them to the recursive action given below.

$\mu X \bullet (c\,?\,x \mathbin{@} t \longrightarrow Op(x)\ ;\ \textbf{wait}\, 10 - t) \blacktriangleleft 5\ ;\ X$

This action (sequentially) describes a cyclic behaviour that repeats every 10 time units. An input communication on a channel c occurs within the first 5 times units, and is followed by a data operation $Op(x)$ that makes use of the input. For the refinement, we first identify that **wait** $10 - t$ fills the time gap between cycles, and we apply Law 2 to introduce an interleaving with **skip** there.

$\equiv$ "introduction of interleaving with **skip** (Law 2)"

$\mu X \bullet (c\,?\,x \mathbin{@} t \longrightarrow Op(x)\ ;\ (\textbf{skip} \mathbin{|||} \textbf{wait}\, 10 - t)) \blacktriangleleft 5\ ;\ X$

We proceed by applying the laws in Figure 7, as well as some trivial simplifications.

$\equiv$ "extraction of interleaving with a time delay from a sequence (Law 3)"

$\mu X \bullet (c\,?\,x \mathbin{@} t \longrightarrow ((Op(x)\ ;\ \textbf{skip}) \mathbin{|||} \textbf{wait}\, 10 - t)) \blacktriangleleft 5\ ;\ X$

$\equiv$ "simplification: $A\ ;\ \textbf{skip} \equiv A$"

$\mu X \bullet (c\,?\,x \mathbin{@} t \longrightarrow (Op(x) \mathbin{|||} \textbf{wait}\, 10 - t)) \blacktriangleleft 5\ ;\ X$

$\equiv$ "extraction of interleaving with a time delay from a prefix (Law 4)"

$\mu X \bullet (c\,?\,x \mathbin{@} t \longrightarrow Op(x) \blacktriangleleft 5 \mathbin{|||} \textbf{wait}\, 10)\ ;\ X$

The application of Law 4 above raises a proviso $5 \leq 10$, which is trivially discharged. We complete the refinement by applying Law 1 to introduce the termination deadline on the body of the recursion.

$\equiv$ "simplification: $c \mathbin{@} t \longrightarrow A \equiv c \longrightarrow A$ if A does not mention t"

$\mu X \bullet (c\,?\,x \longrightarrow Op(x) \blacktriangleleft 5 \mathbin{|||} \textbf{wait}\, 10)\ ;\ X$

$\equiv$ "introduction of termination deadline (Law 1)"

$\mu X \bullet ((c\,?\,x \longrightarrow Op(x) \blacktriangleleft 5) \blacktriangleright 10 \mathbin{|||} \textbf{wait}\, 10)\ ;\ X$

This raises a proviso $\mathit{TakesAtMost}(c\,?\,x \longrightarrow Op(x) \blacktriangleleft 5) \leq 10$. Applying the definition of *TakesAtMost* in Figure 6, we calculate that $\mathit{TakesAtMost}(c\,?\,x \longrightarrow Op(x) \blacktriangleleft 5) = 5 + \mathit{TakesAtMost}(Op(x)) = 5 + 0 = 5$ which discharges the proviso. We next examine laws for decomposition of data operations.

Law 12. Let $State == [\, x : T_1;\ y : T_2 \mid I_1(x) \land I_2(y)\,]$. Then,

$$\begin{array}{|l} \multicolumn{1}{l}{Op} \\ \hline \Delta\, State \\ \hline P(x, x', y) \land Q(y, y') \\ \hline \end{array} \equiv \begin{array}{|l} \multicolumn{1}{l}{Op_1} \\ \hline \Delta\,[\, x : T_1 \mid I_1(x)\,] \\ \Xi\,[\, y : T_2 \mid I_2(y)\,] \\ \hline P(x, x', y) \\ \hline \end{array} \ ⨾\ \begin{array}{|l} \multicolumn{1}{l}{Op_2} \\ \hline \Xi\,[\, x : T_1 \mid I_1(x)\,] \\ \Delta\,[\, y : T_2 \mid I_2(y)\,] \\ \hline Q(y, y') \\ \hline \end{array}$$

Fig. 8. Sequential decomposition of independent data operations.

Law 13. Let $State == [\, x : T_1;\ y : T_2 \mid I_1(x) \land I_2(x, y)\,]$. Then,

$$\begin{array}{|l} \multicolumn{1}{l}{Op} \\ \hline \Delta\, State \\ \hline P(x, x', y) \land Q(x', y, y') \\ \hline \end{array} \equiv \begin{array}{|l} \multicolumn{1}{l}{Op_1} \\ \hline \Delta\,[\, x : T_1 \mid I_1(x)\,] \\ \Xi\,[\, y : T_2\,] \\ \hline I_2(x, y) \land \\ P(x, x', y) \\ \hline \end{array} \ ⨾\ \begin{array}{|l} \multicolumn{1}{l}{Op_2} \\ \hline \Xi\,[\, x : T_1 \mid I_1(x)\,] \\ \Delta\,[\, y : T_2\,] \\ \hline I_2(x', y') \land \\ Q(x, y, y') \\ \hline \end{array}$$

Fig. 9. Sequential decomposition of dependent data operations.

4.2. Decomposition of data operations

Here we target data operations that specify the behaviour of a mission. We note that we do not generally assume 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 or algorithmic 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 (ZRC) [Cav97, CW98], whose laws are valid in *Circus* [OCW09], provides the foundation for our laws here. The laws we present are, therefore, applicable and relevant for Z refinement in general.

Though [Cav97, Gro02], 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.

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 Figure 8. The *State* schema that specifies the state on which the operations act is partitioned into two disjoint lists of variables, x and y, which are respectively constrained by the invariants $I_1(x)$ and $I_2(y)$. The law decomposes Op into a sequence $Op_1 ⨾ Op_2$, where Op_1 only modifies the components in x, and Op_2 only modifies those in y and does not depend on x. Application of this law entails transforming the predicate of an operation schema into a form $P(x, x', y) \land Q(y, y')$.

The second case is where there exists a data dependency between the operations, that is, the second operation uses data that is computed by the first one. Here, we have the general law in Figure 9. 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.

Law 14. Let $State == [\, x : T_1;\ r : T_2 \mid I_1(x) \wedge I_2(x, r)\,]$. Then,

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

$$
\text{where} \quad
\begin{array}{|l}
\multicolumn{1}{l}{POp} \\
\Xi\,[\, x : T_1 \mid I_1(x)\,] \\
r! : T_2 \\
i? : 1 \mathinner{\ldotp\ldotp} n \\
\hline
Q(r!, i?, x) \\
\hline
\end{array}
\quad \text{and} \quad
\begin{array}{|l}
\multicolumn{1}{l}{MOp} \\
\Xi\,[\, x : T_1 \mid I_1(x)\,] \\
\Delta\,[\, r : T_2 \mid I_2(x, r)\,] \\
rb? : \mathrm{bag}\ T_2 \\
\hline
\exists\, s : \mathrm{seq}\ T_2 \mid rb? = items\ s \bullet r' = \mathbf{fold}\ op\ zero\ s \\
\hline
\end{array}
$$

provided that op is an associative and commutative binary infix operator. The function **fold** is the standard folding operation over a sequence of values and $zero$ a zero for op, hence $zero\ op\ x = x$.

Fig. 10. Parallel decomposition of dependent data operations.

The decomposition and propagation of invariants proves to be especially important to facilitate further decomposition and later algorithmic refinement. Invariant decomposition involves the transformation of a single invariant $I(x, y)$ into the conjunction $I_1(x) \wedge I_2(x, y)$ 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 [CZW+13]. Next, we take a look at parallel decomposition.

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 Figure 8 with a small modification of the right-hand side: firstly, the sequence $Op_1 \mathbin{⨟} Op_2$ is replaced by a conjunction $Op_1 \wedge Op_2$, 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 Figure 10. 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. We use renamings $POp[r_i/r!]$ in the right-hand side of the law to replace in POp the schema component $r!$ by the local variables r_i to which the partial results are assigned. The existential quantifications $(\exists\, i? : \mathbb{Z} \bullet POp[\ldots] \wedge i? = n)$ are necessary to define the input $i?$ of POp accordingly for a particular invocation of POp. The merge operation is parametrised by a bag to enforce syntactically that the order in which the results are delivered is irrelevant. The notation $\lbrack\!\lbrack\, e_1, e_2, \ldots \rbrack\!\rbrack$ is used to construct a bag for a given set of elements and $items$ converts a sequence into a bag. We moreover require that the binary operation used in the merge is associative and commutative; the merge then basically consists of folding this operation over the list of partial results.

Law 15. $\textbf{wait}\,0\,..\,t \;\equiv\; \textbf{wait}\,0\,..\,t_1\,;\;\textbf{wait}\,0\,..\,t_2$ **provided** $t = t_1 + t_2$

Law 16. $\textbf{wait}\,0\,..\,t_1 \;\sqsubseteq\; \textbf{wait}\,0\,..\,t_2$ **provided** $t_2 \leq t_1$

Law 17. Assuming Op is a data operation and P is a *Circus* process, we have $P(\textbf{wait}\,t_1\,..\,t_2\,;\;Op) \;\equiv\; P(Op\,;\;\textbf{wait}\,t_1\,..\,t_2)$

Fig. 11. Laws for decomposition, narrowing and distribution of time budgets.

4.3. 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 $\textbf{wait}\,0\,..\,t$ that precede or follow 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 *Circus Time*.

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** statements in order to attach them to the data operations emerging from the decomposition in the previous step. Using these laws, we can equip each decomposed data operation Op with an operation-specific time budget $\textbf{wait}\,0\,..\,Op_{TB}$, where Op_{TB} determines the amount of time the operation may take to execute in an SCJ program.

The refinement laws needed can be divided into two classes. In the first class, we have two key laws given in Figure 11 for the decomposition and narrowing of time budgets. Law 15 replaces a single time budget by a sequence of two time budgets, and Law 16 reduces nondeterminism to narrow a time budget. Decomposition may be applied iteratively, so that a single budget can be split into several budgets.

The second class of laws addresses the issue of moving the decomposed time budgets to suitable locations in order to attach them to their respective data operations. For this, we first transform all Z schema compositions $(Op_1 \fatsemi Op_2)$ into *Circus* action sequences $(Op_1\,;\;Op_2)$. The distinction between these two operators for composition is mostly technical. Intuitively, they both capture the notion of sequential execution, namely of data operations via relational composition of schema predicates in Z, and actions within the UTP-based semantics of *Circus*. The Z schema composition, however, implicitly constrains nondeterminism in the first data operation to satisfy the precondition of the second data operation. This angelic behaviour is not present in the sequential composition of actions. The standard law for rewriting schema compositions is in [Cav97]. The motivation for this transformation is to enable the subsequent steps, which can only be carried out at the level of actions but not data operations, due to the latter not supporting timed constructs.

We further require the specialised distribution Law 17 in Figure 11. This law is in fact noncompositional: it is a law about processes rather than actions. Hence, it only holds if the underlying action $\textbf{wait}\,t_1\,..\,t_2\,;\,Op$ 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 of this law may, for example, proceed 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, $c \longrightarrow (\textbf{wait}\,t\,;\;A)$. Refining this action by $\textbf{wait}\,t\,;\;c \longrightarrow A$ would be unsound 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 [Oli05] 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 18. $\textbf{wait}\,t_1\,..\,t_2 \sqcap \textbf{wait}\,t_1'\,..\,t_2' \;\equiv\; \textbf{wait}\,\min(t_1, t_1')\,..\,\max(t_2, t_2')$

This law is useful as it enables the combination of two budgets, in addition to their decomposition.

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. An overall caveat for the transformation is that we cannot distribute time budgets between parallel data operations that are represented by Z schema conjunctions. This is because the conjunction operator only applies to schemas and not to actions, and the schema calculus, as already noted, does not support timing constructs such as $\textbf{wait}\,t_1\,..\,t_2$. Distribution of the budgets of parallel operations can, therefore, only be done after the *Circus* parallel operators are introduced.

Law 19. Let A_1 and A_2 be actions and c a fresh typeless channel. Then,
$A_1 \; ; \; A_2 \equiv ((A_1 \; ; \; c \longrightarrow \mathbf{skip}) \llbracket \operatorname{wrt}(A_1) \mid \{\!| c |\!\} \mid \operatorname{wrt}(A_2) \rrbracket (c \longrightarrow A_2)) \setminus \{\!| c |\!\}$
provided $\operatorname{wrt}(A_1) \cap \operatorname{wrt}(A_2) = \varnothing$ and $\operatorname{wrt}(A_1) \cap \operatorname{used}(A_2) = \varnothing$

Fig. 12. Parallelisation of independent sequential data operations.

Law 20. Let A_1 and A_2 be actions and c a fresh channel. Then,
$A_1 \; ; \; A_2 \equiv ((A_1 \; ; \; c \,!\, x \longrightarrow \mathbf{skip}) \llbracket \operatorname{wrt}(A_1) \mid \{\!| c |\!\} \mid \operatorname{wrt}(A_2) \rrbracket (c \,?\, x \longrightarrow A_2)) \setminus \{\!| c |\!\}$
provided $\operatorname{wrt}(A_1) \cap \operatorname{wrt}(A_2) = \varnothing$ and $\operatorname{wrt}(A_1) \cap \operatorname{used}(A_2) = \{x\}$

Fig. 13. Parallelisation of dependent sequential data operations.

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

4.4. Introduction of parallel handler actions

In Section 4.2, 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 4.2, we divide the necessary laws into two classes: those that account for sequential designs and those that cater for parallel designs. In the sequel, we discuss both classes of laws.

Laws for sequential handler designs The parallelisations achieved by the first class of laws given in Figures 12 and 13 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 sequential execution in an SCJ design needs to be explicitly enforced, while parallel execution (of handlers) is the default. The first law assumes that there exists no data dependency between the sequential handler actions A_1 and A_2, hence we have the proviso $\operatorname{wrt}(A_1) \cap \operatorname{used}(A_2) = \varnothing$, 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 (Figure 13) 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, *OhCircus* class types are permissible. 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 a separate and independent design issue that we address in Section 4.5.

Laws for parallel handler designs A key law for transforming parallel data operations modelled by conjunctions into parallel actions is presented in Figure 14. 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.

Law 22 (Figure 15) applies to the result of the earlier parallelisation Law 14 for data operations, and, beyond parallelisation into actions, also 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

Law 21. $Op_1 \wedge Op_2 \equiv Op_1 \llbracket \operatorname{wrt}(Op_1) \mid \varnothing \mid \operatorname{wrt}(Op_2) \rrbracket \, Op_2$
provided $\operatorname{wrt}(Op_1) \cap \operatorname{wrt}(Op_2) = \varnothing$

Fig. 14. Low-level law for refining parallel data operations into actions.

Law 22. $\mathbf{wait}\,0\,..\,Op_{TB}\ ;\ \boxed{\text{RHS of Law 14}}\ \sqsubseteq$

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

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

Fig. 15. High-level law for refining parallel data operations into actions.

to compute the partial results, the time to record them, and the time needed to merge them. The concrete values of these freshly introduced budgets do not have to be specified when applying the law, and are not an issue for the refinement since later schedulability analysis can determine them as part of translating the S anchor into a program for a concrete SCJ execution platform with known timing characteristics.

A design artifact of Law 22 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 action emerges (the right-hand parallel action of the law) that is later refined into shared data that aggregates the partial results as they arrive. Its refinement is treated separately, in the next section, and entails the design of the storage and processing of the partial results.

To conclude this aspect of the refinement, we observe that we can either tackle it by way of applying the more general Law 21, or by using specialised high-level laws like Law 22 that encapsulate particular designs.

4.5. Encapsulation of shared data

The purpose of the laws we discuss last is to isolate control mechanisms and shared data access, so that all control is modelled by designated, typeless channels, which may later be refined into models of SCJ events. Similarly, designated channels are introduced and used for shared data access, namely to read and modify shared data in a safe way, so that no data races occur; those channels model calls to `synchronized` methods.

The verification here relies on a set of highly specialised laws that encapsulate shared data into a separate action *MArea* (see Section 3). We show that shared data may not only arise from input and output communications as in Law 20, but also as a consequence of refining more sophisticated mechanisms of control, such as barrier synchronisations, or the control fragment emerging from high-level parallelisation laws like Law 22 above. We provide laws for each of those three cases and discuss them separately in the sequel.

Laws for channel communications The first law we discuss is Law 23 in Figure 16. It is a general channel decomposition law that, throughout some action A, replaces all occurrences of input and output prefixes involving a communication on a local channel c by a sequence of communications: two for control, namely on fresh typeless channels c_{sync} and c_{pivot}, and another one to read or write to a shared variable introduced to hold the data communicated through c. Reading and writing of the shared variable is via a pair of new channels c_{read} and c_{write} of the same type as c. To read from the shared variable, we use an input prefix $c_{read}\,?\,x \longrightarrow A(x)$, and to write a value e to it, an output prefix $c_{write}\,!\,e \longrightarrow \mathbf{skip}$.

To model the shared variable, the right recursive action of the parallel composition in Law 23 synchronises on these channels while the block $(\mathbf{var}\ v : T \bullet \ldots)$ introduces a local variable v of type T to hold the shared data. Another channel c_{term} is introduced to control termination of the recursive action, when it is no longer needed. This ensures that the parallel composition in Law 23 altogether terminates when the action on the left-hand side of the law does so (after synchronising on c_{term}).

Law 23. Extraction of shared data communicated through a typed channel.

$$A \setminus \{\!|\, c \,|\!\} \sqsubseteq \left(\begin{array}{l} \mathit{ChanDecomp}(c)(A) \setminus \{\!|\, c_{sync}, c_{pivot} \,|\!\} \, ; \; c_{term} \longrightarrow \mathbf{skip} \\ \quad [\![\mathrm{wrt}(A) \mid \{\!|\, c_{read}, c_{write}, c_{term} \,|\!\} \mid \varnothing]\!] \\ \left(\begin{array}{l} \mathbf{var}\, v : T \bullet \\ \quad \mu X \bullet \left(\begin{array}{l} (c_{read}\,!\,v \longrightarrow X) \,\Box \\ (c_{write}\,?\,x \longrightarrow v := x \,;\; X) \,\Box \\ (c_{term} \longrightarrow \mathbf{skip}) \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{\!|\, c_{write}, c_{read}, c_{term} \,|\!\}$$

provided $\{c_{read}, c_{write}, c_{term}, c_{sync}, c_{pivot}\} \cap \mathrm{usedC}(A) = \varnothing$ and $\mathit{WWConfFree}(c)(A)$

Fig. 16. Sharing law for encapsulation of data passed through a channel.

Whereas communications on the channels c_{read} and c_{write} model nonblocking variable access, the purpose of the channels c_{sync} and c_{pivot} is to ensure synchroneity and the absence of race conditions. The law also makes use of a function $\mathit{ChanDecomp}(c)(A)$, parametrised by the channel c (to be decomposed) and an action A to be transformed. The formal definition of this function is sketched below.

Def 2. Action transformation function for channel decomposition used by Law 23 (Figure 16).

$$\begin{array}{|l} \mathit{ChanDecomp} : \mathit{Channel} \to \mathit{Action} \to \mathit{Action} \\ \hline \forall A, A_1, A_2 : \mathit{Action};\ c, d : \mathit{Channel};\ cs : \mathit{ChannelSet};\ e : \mathit{Expr};\ v : \mathit{Var} \mid c \neq d \bullet \\ \mathit{ChanDecomp}(c)(c\,!\,e \longrightarrow A) \;=\; c_{sync} \longrightarrow \mathbf{skip} \,;\; c_{write}\,!\,e \longrightarrow c_{pivot} \longrightarrow \mathit{ChanDecomp}(c)(A) \\ \mathit{ChanDecomp}(c)(c\,?\,v \longrightarrow A) \;=\; c_{sync} \longrightarrow \mathbf{skip} \,;\; c_{pivot} \longrightarrow c_{read}\,?\,v \longrightarrow \mathit{ChanDecomp}(c)(A) \\ \mathit{ChanDecomp}(c)(d\,!\,e \longrightarrow A) \;=\; d\,!\,e \longrightarrow \mathit{ChanDecomp}(c)(A) \;\; \text{(for channels other than } c) \\ \mathit{ChanDecomp}(c)(d\,?\,v \longrightarrow A) \;=\; d\,?\,v \longrightarrow \mathit{ChanDecomp}(c)(A) \;\; \text{(for channels other than } c) \\ \mathit{ChanDecomp}(c)(A \setminus cs) \;=\; (\mathbf{if}\ c \in cs\ \mathbf{then}\ A\ \mathbf{else}\ \mathit{ChanDecomp}(c)(A)) \setminus cs \\ \mathit{ChanDecomp}(c)(A_1 [\![ns_1 \mid cs \mid ns_2]\!] A_2) = \mathit{ChanDecomp}(c)(A_1) [\![ns_1 \mid cs_r \mid ns_2]\!] \mathit{ChanDecomp}(c)(A_2) \\ \text{where } cs_r \mathrel{\widehat=} \mathbf{if}\ c \in cs\ \mathbf{then}\ (cs - \{\!|\, c \,|\!\}) \cup \{\!|\, c_{sync}, c_{pivot} \,|\!\}\ \mathbf{else}\ cs \\ \mathit{ChanDecomp}(c)(A_1 \,\Box\, A_2) \;=\; \mathit{ChanDecomp}(c)(A_1) \,\Box\, \mathit{ChanDecomp}(c)(A_2) \\ \mathit{ChanDecomp}(c)(A_1 \,;\; A_2) \;=\; \ldots \end{array}$$

The only action constructs that are affected are input and output prefixes on c, channel hiding, and parallel composition. The dots in the last line of the definition above indicate that $\mathit{ChanDecomp}(c)$ distributes through all other action operands, applying itself recursively. For communications on channels other than c, the transformation carried out by $\mathit{ChanDecomp}(c)$ simply distributes through the prefix. It also distributes through parallel actions, while replacing c in the synchronisation sets of parallel actions if present, namely by $\{\!|\, c_{sync}, c_{pivot} \,|\!\}$. If c is captured by a hiding $A \setminus cs$ such that $c \in cs$, the scope of the transformation ends.

Output prefixes $c\,!\,e \longrightarrow A$ are replaced by three synchronisations: the first one on c_{sync} initiates the communication and thereby captures its control aspect, since both the transformed input and output prefix synchronise on it initially. The second one, $c_{write}\,!\,e$, records the data that is communicated through the channel in a shared variable. And the third one on c_{pivot} is needed to avoid race conditions: it prevents progress of the reading action(s) prior to the data having been written by the writing action. (We note that the three communications could equivalently be specified in *ChanDecomp* as a single chain of prefixed actions $c_{sync} \longrightarrow c_{write}\,!\,e \longrightarrow c_{pivot} \longrightarrow \ldots$ because of the law $(c \longrightarrow \mathbf{skip} \,;\, A) \equiv c \longrightarrow A$; choosing one or the other form is a question of style.) Similarly, input prefixes $c\,?\,v \longrightarrow A(v)$ are translated into a sequence where the first action, as before, synchronises on c_{sync}, the second action waits for the acknowledgement c_{pivot} raised after the data has been written to the shared variable, and the third prefix performs a (nonblocking) read access to the shared variable to obtain the data sent through the channel c in the original communication. We note that this channel replacement principle is only valid if the replaced channel is hidden as indicated in the left-hand side of Law 23, chiefly as we cannot alter the way that the environment interacts with A.

Typically, we have matching input and output prefixes in parallel actions: this means that for every input prefix on a channel c, a communication that outputs on c is performed when the input is reached. To illustrate the validity of the law under this assumption, we consider the simple communication below.

$$((c\,!\,e \longrightarrow A_1) [\![\mathrm{wrt}(A_1) \mid cs \mid \mathrm{wrt}(A_2)]\!] (c\,?\,x \longrightarrow A_2(x))) \setminus \{\!|\, c \,|\!\}$$

where $c \in cs$ and $c \notin \mathrm{usedC}(A_1) \cup \mathrm{usedC}(A_2)$. Thus, we assume the channel c is not used by A_1 and A_2. The left-hand parallel action outputs a value on c, defined by the expression e, and the right-hand parallel action inputs that value. Hence, the above is equivalent to $(A_1 \llbracket \operatorname{wrt}(A_1) \mid cs \mid \operatorname{wrt}(A_2) \rrbracket A_2(e)) \setminus \{\!| c |\!\}$. Formally, this can be proved using step laws for parallel actions. (Since A_1 and A_2 do not use the channel c, this can indeed be further simplified to $A_1 \llbracket \operatorname{wrt}(A_1) \mid cs - \{\!| c |\!\} \mid \operatorname{wrt}(A_2) \rrbracket A_2(e)$.)

Applying Law 23 to the above action, we obtain the following refined action.

$$\left(\begin{array}{l} \left(\begin{array}{l} (c_{sync} \longrightarrow \mathbf{skip} \; ; \; c_{write} \,!\, e \longrightarrow c_{pivot} \longrightarrow A_1) \\ \llbracket \operatorname{wrt}(A_1) \mid (cs - \{\!| c |\!\}) \cup \{\!| c_{sync}, c_{pivot} |\!\} \mid \operatorname{wrt}(A_2) \rrbracket \\ (c_{sync} \longrightarrow \mathbf{skip} \; ; \; c_{pivot} \longrightarrow c_{read} \,?\, x \longrightarrow A_2(x)) \end{array}\right) \setminus \{\!| c_{sync}, c_{pivot} |\!\} \; ; \; \ldots \\ \llbracket \operatorname{wrt}(A_1) \cup \operatorname{wrt}(A_2) \mid \{\!| c_{read}, c_{write}, c_{term} |\!\} \mid \varnothing \rrbracket \\ \left(\begin{array}{l} \mathbf{var}\, v : T \bullet \\ \mu X \bullet \left(\begin{array}{l} (c_{read} \,!\, v \longrightarrow X) \;\Box \\ (c_{write} \,?\, x \longrightarrow v := x \; ; \; X) \;\Box\; \ldots \end{array}\right) \end{array}\right) \end{array}\right) \setminus \{\!| c_{write}, c_{read}, c_{term} |\!\}$$

Intuitively, the prefix $c_{sync} \longrightarrow \mathbf{skip}$ in the left-hand parallel composition of actions captures the original synchronisation on c, albeit without considering the communication of data (we note that c_{sync} is in the interface of that composition). Data communication is achieved in a separate step, by using the channels c_{read}, c_{write} and c_{pivot}. Whereas c_{write} writes the data communicated through c into the shared variable v, the channel c_{pivot} inhibits progress of the right (inner) parallel action until the data has been written by the left (inner) parallel action. In this way, c_{pivot} avoids a potential race condition: it ensures that reads cannot overtake writes of matching inputs and outputs. Because the channels c_{read} and c_{write} are hidden, and the corresponding synchronisations are never blocked by the action that models the shared data, the original behaviour of the data communication on the typed channel c is retained. Formally, this can be proved, as before, using step laws. We can also easily convince ourselves that the law remains valid when multiple actions simultaneously input on the channel c — in that case we have multiple communications on c_{read}.

In the above example, we assumed that A_1 and A_2 do not use the channel c. But generally, it does not invalidate the law if they do. To illustrate this, we consider two outputs on the channel c being performed in sequence. We thus alter the previously refined action by replacing the left parallel action with $c \,!\, e_1 \longrightarrow c \,!\, e_2 \longrightarrow A_1$ and the right parallel action with $c \,?\, x \longrightarrow c \,?\, y \longrightarrow A_2(x, y)$. This yields the following action fragment after application of the law (we omit the parallel composition with the shared data).

$$\left(\begin{array}{l} (c_{sync} \longrightarrow \mathbf{skip} \; ; \; c_{write} \,!\, e_1 \longrightarrow c_{pivot} \longrightarrow (\boxed{c_{sync} \longrightarrow \mathbf{skip}} \; ; \; c_{write} \,!\, e_2 \longrightarrow c_{pivot} \longrightarrow A_1)) \\ \llbracket \operatorname{wrt}(A_1) \mid (cs - \{\!| c |\!\}) \cup \{\!| c_{sync}, c_{pivot} |\!\} \mid \operatorname{wrt}(A_2) \rrbracket \\ (c_{sync} \longrightarrow \mathbf{skip} \; ; \; c_{pivot} \longrightarrow \boxed{c_{read} \,?\, x \longrightarrow} (c_{sync} \longrightarrow \mathbf{skip} \; ; \; c_{pivot} \longrightarrow c_{read} \,?\, y \longrightarrow A_2(x, y))) \end{array}\right) \setminus \ldots$$

We observe that the boxed c_{sync} event arising from the second output $c \,!\, e_2 \longrightarrow A_1$ can only occur once the $c_{read} \,?\, x \longrightarrow \ldots$ communication (boxed) of the first input has taken place. This ensures that the value of the shared variable is kept as long as there are pending reads. The initial synchronisation on c_{sync} in this way ensures that subsequent writes cannot overtake pending reads. This example moreover elucidates the need for two control channels as c_{pivot} alone turns out to be insufficient to avoid this kind of race condition.

Another case arises when we have one (or more) inputs, but no matching output in the refined action. In that case, either the initial or last value written to the shared variable is implicitly communicated. For the validity of the law, this is again not a problem. The reason for this is that in an action $(c \,?\, x \longrightarrow A(x)) \setminus \{\!| c |\!\}$, we have a nondeterminism as to what value is input on the channel. So, we are at liberty to refine such a prefix by communicating any value we like.

An issue arises though due to write conflicts. To illustrate this, we consider the action

$$A \mathrel{\widehat=} (c \,!\, 1 \longrightarrow \mathbf{skip} \llbracket \varnothing \mid \{\!| c |\!\} \mid \varnothing \rrbracket\, c \,!\, 2 \longrightarrow \mathbf{skip}) \setminus \{\!| c |\!\} \qquad (2)$$

where c is a channel of type $\mathbb{N}$. This action is equivalent to the deadlocked action **stop** since the output communications do not agree on the value output on the channel. Applying Law 23 here yields the action

$$\left(\begin{array}{l}\left(\begin{array}{l}(c_{sync} \longrightarrow \mathbf{skip} \; ; \; c_{write} \, ! \, 1 \longrightarrow c_{pivot} \longrightarrow \mathbf{skip}) \\ \quad [\![\varnothing \mid \{\!|\, c_{sync}, c_{pivot} \,|\!\} \mid \varnothing]\!] \\ (c_{sync} \longrightarrow \mathbf{skip} \; ; \; c_{write} \, ! \, 2 \longrightarrow c_{pivot} \longrightarrow \mathbf{skip})\end{array}\right) \setminus \{\!|\, c_{sync}, c_{pivot} \,|\!\} \; ; \; c_{term} \longrightarrow \mathbf{skip} \\ [\![\varnothing \mid \{\!|\, c_{read}, c_{write}, c_{term} \,|\!\} \mid \varnothing]\!] \\ \left(\begin{array}{l}\mathbf{var}\, v : \mathbb{N} \bullet \\ \mu X \bullet \left(\begin{array}{l}(c_{read} \, ! \, v \longrightarrow X) \;\Box \\ (c_{write} \, ? \, x \longrightarrow v := x \; ; \; X) \;\Box \\ (c_{term} \longrightarrow \mathbf{skip}\end{array}\right)\end{array}\right)\end{array}\right) \setminus \{\!| \ldots |\!\}$$

which can be proved equivalent to **skip** by unfolding the recursion and collapsing the parallel actions using step laws. Hence, we cannot apply the law to actions that contain conflicting outputs on the channel c.

To determine that such outputs potentially cannot arise, we make use of the predicate $\mathit{WWConfFree}(c)(A)$ in the proviso of the law. An extract of its definition is included in Figure 17. As specified by the first conjunct, we can trivially infer the absence of write conflicts if A does not mention the channel c. This, for instance, enables us to infer $\mathit{WWConfFree}(c)(\mathbf{skip})$ and $\mathit{WWConfFree}(c)(\mathbf{wait}\, t)$, among other cases. For most other operators (of which some are omitted for brevity), $\mathit{WWConfFree}(c)(A)$ merely has to be shown for their constituent action operands. The three notable exceptions to this are channel hiding, recursion and parallel actions. We discuss them in more detail as they require special treatment.

To establish the absence of write conflicts in a channel hiding $A \setminus cs$, we consider two cases. If $c \in cs$, there are no conflicts since the channel c is captured by the hiding. Otherwise, we have to show the absence of write conflicts in the action A. For a recursion $\mu X \bullet F(X)$, we show the absence of write conflicts in $F(X)$ under the assumption that there are no write conflicts in X.

For actions involving parallel compositions, we require further defining rules (not included in Figure 17) in order to evaluate $\mathit{WWConfFree}(c)(A)$. They are summarised below.

Def 3. Definition of $\mathit{WWConfFree}$ for actions involving parallel composition.

$$\begin{array}{|l}\mathit{WWConfFree} : \mathit{Channel} \rightarrow \mathit{Action} \rightarrow \mathbb{B} \\ \hline \ldots \; c \in cs \Rightarrow \\ \mathit{WWConfFree}(c)(A_1 \parallel A_2 \parallel \ldots \parallel A_n) \Leftrightarrow (\forall\, i, j : 1 \mathinner{..} n \mid i \neq j \bullet \mathit{WWConfFree}(c)(A_i \parallel A_j)) \\ \mathit{WWConfFree}(c)((c \, ! \, e \longrightarrow A_1 \; ; \; A_2) \; [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \; (c \, ? \, x \longrightarrow A_3 \; ; \; A_4)) \Leftrightarrow \\ \quad \mathit{WWConfFree}(c)((A_1 \; ; \; A_2) \; [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \; (A_3 \; ; \; A_4)) \\ \mathit{WWConfFree}(c)((c \, ! \, e \longrightarrow A_1 \; ; \; A_2) \; [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \; (c \, ! \, f \longrightarrow A_3 \; ; \; A_4)) \Leftrightarrow \mathit{false} \\ \mathit{WWConfFree}(c)((A_1 \; ; \; A_2) \; [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \; A_3) \Leftrightarrow \\ \quad \mathit{WWConfFree}(c)(A_2 \; [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \; A_3) \;\; \text{provided } c \notin \text{usedC}(A_1) \\ \mathit{WWConfFree}(c)((A_1 \,\Box\, A_2) \; [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \; A_3) \Leftrightarrow \\ \quad \mathit{WWConfFree}(c)(A_1 \; [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \; A_3) \wedge \mathit{WWConfFree}(c)(A_2 \; [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \; A_3) \\ \ldots\end{array}$$

The first conjunct deals with multiple parallel actions. To establish absence of write conflicts there, it is enough to show that pairwise parallel actions are free of such conflicts. The remaining cases address parallel compositions where the parallel actions have various shapes. Notable are the laws for prefixes: they are step deductions that allow us to remove a prefix from both sides of a parallel composition if there is no clash between two output communications (if there is, the result is *false* as we would expect). For the remaining operators, like external choice above, $\mathit{WWConfFree}(c)((A_1 \, \mathbf{op} \, A_2) \, [\![\ldots]\!] \, A_3)$ distributes through the operator.

To illustrate the use of $\mathit{WWConfFree}$, we apply it to the action used in the counterexample above (2). Using the definition of $\mathit{WWConfFree}(c)$, we obtain that

$$\mathit{WWConfFree}(c)(c \, ! \, 1 \longrightarrow \mathbf{skip} \; [\![\, \varnothing \mid \{\!|\, c \,|\!\} \mid \varnothing \,]\!] \; c \, ! \, 2 \longrightarrow \mathbf{skip}) \Leftrightarrow \mathit{false}$$

which violates the proviso of the sharing law.

The pattern that is used to model the shared variable in the right-hand side of Law 23 via a local variable and recursion is well-known from languages like CSP. In an SCJ program, it is implemented using plain get and set methods, and calls to those methods correspond to synchronisations on the read and write channels.

Def 4. The predicate *WWConfFree* is used to infer the absence of write conflicts.

$$
\begin{array}{|l}
WWConfFree : Channel \rightarrow Action \rightarrow \mathbb{B} \\
\hline
\forall A, A_1, A_2 : Action;\ c, d : Channel;\ cs : ChannelSet;\ e, f : Expr;\ x : Var \bullet \\
c \notin \mathrm{usedC}(A) \Rightarrow WWConfFree(c)(A) \\
WWConfFree(c)(d\,[\,!\,e]\,[\,?\,x] \longrightarrow A) \Leftrightarrow WWConfFree(c)(A) \\
WWConfFree(c)(A_1\ ;\ A_2) \Leftrightarrow WWConfFree(c)(A_1) \land WWConfFree(c)(A_2) \\
WWConfFree(c)(A_1 \,\square\, A_2) \Leftrightarrow WWConfFree(c)(A_1) \land WWConfFree(c)(A_2) \\
WWConfFree(c)(A_1 \sqcap A_2) \Leftrightarrow WWConfFree(c)(A_1) \land WWConfFree(c)(A_2) \\
WWConfFree(c)(A_1 \,|||\, A_2) \Leftrightarrow WWConfFree(c)(A_1) \land WWConfFree(c)(A_2) \\
WWConfFree(c)(A \setminus cs) \Leftrightarrow (c \notin cs \Rightarrow WWConfFree(c)(A)) \\
WWConfFree(c)(\mu X \bullet F(X)) \Leftrightarrow (WWConfFree(c)(X) \Rightarrow WWConfFree(c)(F(X))) \\
\ldots
\end{array}
$$

Fig. 17. Extract of the definition of *WWConfFree* used in Law 23.

As already explained, we also provide a mechanism that caters for termination of this action after termination of A, using the channel c_{term}; otherwise, the parallel action would deadlock even when A terminates.

We conclude by pointing out that Law 23 is very general and its application only requires the developer to identify typed channels for which shared data components have to be introduced. The parallel action that arises in the right-hand of the law directly contributes to the *MArea* action in Figure 5, which encapsulates all shared data of an SCJ design. In cases where we can show that the action A never terminates, a simpler version of the law can be used that does not require the channel c_{term}.

Synchronisation barrier refinement As mentioned earlier on, shared data can also arise from refining control mechanisms. A common control mechanism is a synchronisation barrier: a number of processes suspend execution until all processes have reached the barrier. At an abstract level, this is typically modelled by multiple actions synchronising on a channel c_{bsync}, which models the barrier. To illustrate the refinement of this control mechanism, we consider actions of the following shape.

$$
A_{barrier} \mathrel{\widehat=} \left(\begin{array}{l}
(\mu X \bullet c_{start} \longrightarrow A_1\ ;\ c_{bsync} \longrightarrow \mathbf{skip}\ ;\ X) \\
\quad [\![ns_1 \mid cs_1 \mid ns_2 \cup \ldots \cup ns_n]\!] \\
(\mu X \bullet c_{start} \longrightarrow A_2\ ;\ c_{bsync} \longrightarrow \mathbf{skip}\ ;\ X) \\
\quad [\![ns_2 \mid cs_2 \mid ns_3 \cup \ldots ns_n]\!] \\
\ldots \\
\quad [\![ns_{n-1} \mid cs_{n-1} \mid ns_n]\!] \\
(\mu X \bullet c_{start} \longrightarrow A_n\ ;\ c_{bsync} \longrightarrow \mathbf{skip}\ ;\ X)
\end{array}\right) \tag{3}
$$

where $\{\!|\, c_{start}, c_{bsync} \,|\!\} \subseteq cs_i$ and $\{\!|\, c_{start}, c_{bsync} \,|\!\} \cap usedC(A_i) = \varnothing$ for $1 \leq i \leq n$ so that all parallel actions are recursions whose bodies start synchronously as determined by the channel c_{start} and end synchronously as ensured by the synchronisation on the channel c_{bsync}. The channel c_{start} models an SCJ event that is bound to several handlers (modelled by the parallel actions) and concurrently releases them. In each handler action, A_i defines the behaviour before the barrier is reached. We assume that the A_i do not mention c_{start} and c_{bsync}, so that c_{bsync} is used only once per cycle. Moreover, the synchronisation on c_{bsync} has to be the last action before the handlers repeat their cycle (recurse into X).

It is important to note that in the context where $A_{barrier}$ occurs, other (handler) actions may synchronise on the channel c_{bsync} too, without being conceptually part of the barrier. For instance, in another handler action, c_{bsync} may be used to trigger the release of that handler, and, in that context, we think of it rather as modelling an SCJ event that is fired in response to the barrier having been reached by all actions. The problem of deciding which actions are part of a barrier in general requires human insight. The objective of the refinement here is thus not to remove c_{bsync} from the model, but to eradicate it from the handler parallelism that uses it as a barrier. This is done by replacing it with a mechanism that does not require synchronisation between the handler actions, and is realised by shared data.

Law 24. Design law for a synchronisation barrier.

$$A_{barrier}\,(3) \sqsubseteq$$

$$\left(\begin{array}{l}\left(\begin{array}{l}(\mu X \bullet c_{start} \longrightarrow A_1\,;\ c_{notify}\,!\,1 \longrightarrow \mathbf{skip}\,;\ X)\\ \quad [\![ns_1 \mid cs_1 - \{\!|\, c_{bsync} \,|\!\} \mid ns_2 \cup \ldots \cup ns_n]\!]\\ (\mu X \bullet c_{start} \longrightarrow A_2\,;\ c_{notify}\,!\,2 \longrightarrow \mathbf{skip}\,;\ X)\\ \quad [\![ns_2 \mid cs_2 - \{\!|\, c_{bsync} \,|\!\} \mid ns_3 \cup \ldots \cup ns_n]\!]\\ \qquad \ldots\\ \quad [\![ns_{n-1} \mid cs_{n-1} - \{\!|\, c_{bsync} \,|\!\} \mid ns_n]\!]\\ (\mu X \bullet c_{start} \longrightarrow A_n\,;\ c_{notify}\,!\,n \longrightarrow \mathbf{skip}\,;\ X)\end{array}\right)\\ \quad [\![ns_1 \cup \ldots \cup ns_n \mid \{\!|\, c_{start}, c_{notify} \,|\!\} \mid \varnothing]\!]\\ \left(\begin{array}{l}\left(\begin{array}{l}\mathbf{var}\ active : \mathbb{P}\,(1\,..\,n) \bullet\\ \mu X \bullet \left(\begin{array}{l}(c_{reset} \longrightarrow active := 1\,..\,n\,;\ X)\\ \quad \Box\\ \left(c_{notify}\,?\,x \longrightarrow \left(\begin{array}{l}active := active - \{x\};\\ \mathbf{if}\ active = \varnothing \longrightarrow c_{bsync} \longrightarrow \mathbf{skip}\\ [\!]\ \neg\ active = \varnothing \longrightarrow \mathbf{skip}\\ \mathbf{fi}\end{array}\right);\ X\right)\end{array}\right)\end{array}\right)\\ [\![\varnothing \mid \{\!|\, c_{reset}, c_{start} \,|\!\} \mid \varnothing]\!]\ (\mu X \bullet c_{reset} \longrightarrow c_{start} \longrightarrow X)\end{array}\right)\end{array}\right)$$

$$\quad \setminus \{\!|\, c_{reset}, c_{notify} \,|\!\}$$

provided $\{\!|\, c_{start}, c_{bsync} \,|\!\} \subseteq cs_i \wedge \{\!|\, c_{start}, c_{bsync} \,|\!\} \cap \mathrm{usedC}(A_i) = \varnothing$ for $1 \leq i \leq n$

and c_{reset} and c_{notify} are fresh channels where c_{reset} is typeless and c_{notify} is of type $\mathbb{N}$.

Fig. 18. Design law for a synchronisation barrier.

Law 24 in Figure 18 replaces the synchronisations on c_{bsync} by outputs of the form $c_{notify}\,!\,i \longrightarrow \mathbf{skip}$ where i identifies the handler. The c_{notify} channel of type $\mathbb{N}$ is introduced to signal that a handler has reached the barrier. The purpose of the shared data here is to record the handlers i that have not reached the barrier yet; once all handlers have reached it, a communication on c_{bsync} is raised. The model for the shared data fragment that becomes part of *MArea* is recaptured below from Law 24.

$$\left(\begin{array}{l}\mathbf{var}\ active : \mathbb{P}\,(1\,..\,n) \bullet\\ \mu X \bullet \left(\begin{array}{l}(c_{reset} \longrightarrow active := 1\,..\,n\,;\ X)\\ \quad \Box\\ \left(c_{notify}\,?\,x \longrightarrow \left(\begin{array}{l}active := active - \{x\};\\ \mathbf{if}\ active = \varnothing \longrightarrow c_{bsync} \longrightarrow \mathbf{skip}\\ [\!]\ \neg\ active = \varnothing \longrightarrow \mathbf{skip}\\ \mathbf{fi}\end{array}\right);\ X\right)\end{array}\right)\end{array}\right)$$

The shared variable *active* holds a set of handler identifiers and determines the handlers that have not reached the barrier yet. A synchronisation on c_{reset} establishes its initial value $1\,..\,n$, which corresponds to the set $\{1, 2, \ldots, n\}$. Synchronisation on $notify\,?\,x$ causes x to be removed from *active*, and when there are no more elements in *active*, the event c_{bsync} is raised. In an SCJ program, c_{reset} and c_{notify} are typically implemented by `synchronized` methods, reflecting their atomic execution in the model. The shared variable *active*, of abstract type $\mathbb{P}(\mathbb{N})$, has to be further refined into a data structure that is directly available in SCJ like a `List` or an array. The latter is an independent verification issue that requires further design laws.

A salient aspect of Law 24 is the parallel control fragment $\mu X \bullet c_{reset} \longrightarrow c_{start} \longrightarrow X$. Its purpose is to raise the c_{reset} event in order to initialise the shared data. In an SCJ program design, there are usually multiple possibilities where this initialisation could be performed. We require the developer to eliminate this parallel action in a separate step. This is achieved by collapsing the action with a handler that is identified to perform the initialisation, using parallel step laws, and gives rise to further design decisions.

In summary, Law 24 replaces the common synchronisation on c_{bsync} between the handlers by interleaved synchronisations on c_{notify} between individual handler actions and the shared data model. Although the handlers do not block when raising their c_{notify} event, correctness of the refinement is guaranteed by lock-step progress due to the initial synchronisation on c_{start}. The channel c_{bsync} fulfils a different purpose after the refinement: it is turned into the model of an SCJ event that, in the underlying SCJ program, is fired by the method that implements *notify* and controls those handler actions that synchronise on c_{bsync}, but are conceptually not part of the barrier. The barrier sharing law is clearly more specialised than the previous sharing Law 23. It, nevertheless, retains abstraction in two ways: firstly, in terms of the representation of the shared data, and secondly, as to where the shared data is initialised.

Refinement of control actions The last class of laws we discuss refine the way that shared data is realised rather than introducing it from scratch. For instance, Law 22 introduces shared data via a control action

$$A_{control} \mathrel{\widehat=} \left(\begin{array}{l} \mathbf{var}\, r_1, r_2, \ldots, r_n : T \bullet \\ \left(\begin{array}{l} (rec\,?\,x \longrightarrow \mathbf{wait}\,0\,..\,Rec_{TB}\ ;\ r_1 := x); \\ (rec\,?\,x \longrightarrow \mathbf{wait}\,0\,..\,Rec_{TB}\ ;\ r_2 := x); \\ \ldots \\ (rec\,?\,x \longrightarrow \mathbf{wait}\,0\,..\,Rec_{TB}\ ;\ r_n := x) \end{array}\right); \\ \mathbf{wait}\,0\,..\,Merge_{TB}\ ;\ MOp(\llbracket r_1, r_2, \ldots, r_n \rrbracket) \end{array}\right)$$

that constitutes the right-hand action of the resulting parallel composition. Above, we have a local variable r_i for each partial result communicated by a parallel handler action, and the computation of r via *MOp* takes place only when all r_i have been received. Law 25 permits the refinement of this action into an action that only uses a single variable r, namely to assimilate the results communicated by the concurrent computations *as they arrive*. The refined shared data design may, for instance, be preferable in situations where limited resources for storage are available. It is conceivable that control actions of a similar shape as above arise from other parallelisation laws too; hence Law 25 is likely to be useful beyond earlier application of Law 22.

For Law 25 to be applicable, the control fragment $A_{control}$ has to be embedded into an action

$$\mathbf{var}\, r : T \bullet (\mu X \bullet start \longrightarrow (\mathbf{wait}\,0\,..\,Init_{TB}\ ;\ InitOp\ ;\ A_{control}\ ;\ out\,!\,r \longrightarrow \mathbf{skip})\ ;\ lockstep \longrightarrow X)$$

We first have a local block ($\mathbf{var}\, r : T \bullet \ldots$) that introduces the variable r of type T that holds the result of the merge operation. The body of the recursion synchronises on a channel *start* and then performs a data operation to initialise the local variable r. This is followed by execution of the control action $A_{control}$ which updates the value of r using *MOp*, and a finalising synchronisation on *lockstep*. The channel *start* is needed to determine when $A_{control}$ should start, and *lockstep* signals termination of $A_{control}$ after which the recursive behaviour repeats and thus $A_{control}$ may be used again.

An important proviso of Law 25 is that the merge operation *MOp* must distribute through bag union, namely $MOp(b_1 \uplus b_2) = MOp(b_1);\ MOp(b_2)$. This establishes that the combination of partial results, which is done in one shot by the call $MOp(\llbracket r_1, r_2, \ldots, r_n \rrbracket)$ in $A_{control}$, can be decomposed into multiple incremental merge operations $MOp(\llbracket r_i \rrbracket)$. Each incremental merge operation takes into account the current value of r, whilst combining it with the next partial result r_i. The action modelling the shared data in the right-hand side of Law 25 (left parallel action) here supports three interactions: *init* to initialise the value of the shared data, output $rec\,?\,x$ to record a partial result, and input $out\,?\,y$ to read the aggregated result so far.

We notice that the law introduces a control fragment of its own, which is recaptured below.

$$\left(\mu X \bullet start \longrightarrow init \longrightarrow \left(\begin{array}{c} (rec\,?\,y \longrightarrow \mathbf{skip})\ ||| \\ (rec\,?\,y \longrightarrow \mathbf{skip})\ ||| \\ \ldots \\ (rec\,?\,y \longrightarrow \mathbf{skip}) \end{array}\right)\ ;\ out\,?\,y \longrightarrow \mathbf{skip}\ ;\ lockstep \longrightarrow X\right)$$

The purpose of this control action is to determine the order of interactions with the shared data. This is essential to ensure the validity of the law because the parallel action that models the shared data *per se* does not constrain that order, whereas the refined action clearly does. Essentially, we have a communication on *init*, followed by n communications on *rec*, and a communication on *out* and *lockstep* to finalise the cycle. Here, we are not concerned with the actual values communicated on the channels *rec* and *out* as y is not used. We recall that the interleaving does not cause synchronisation between the $rec\,?\,y \longrightarrow \mathbf{skip}$ actions and

Law 25. Design law for refining shared data arising during parallelisation.

$$\left(\begin{array}{l}\mathbf{var}\ r : T \bullet \\ \mu X \bullet start \longrightarrow \left(\begin{array}{l}\mathbf{wait}\ 0\,..\,Init_{TB}\ ;\ InitOp; \\ \mathbf{var}\ r_1, r_2, \ldots, r_n : T \bullet \\ \left(\begin{array}{l}(rec\,?\,x \longrightarrow \mathbf{wait}\ 0\,..\,Rec_{TB}\ ;\ r_1 := x); \\ (rec\,?\,x \longrightarrow \mathbf{wait}\ 0\,..\,Rec_{TB}\ ;\ r_2 := x); \\ \quad\ldots \\ (rec\,?\,x \longrightarrow \mathbf{wait}\ 0\,..\,Rec_{TB}\ ;\ r_n := x)\end{array}\right); \\ \mathbf{wait}\ 0\,..\,Merge_{TB}\ ;\ MOp([\![\, r_1, r_2, \ldots, r_n \,]\!]); \\ out\,!\,r \longrightarrow \mathbf{skip}\end{array}\right);\ lockstep \longrightarrow X\end{array}\right) \sqsubseteq$$

$$\left(\begin{array}{l}\left(\begin{array}{l}\mathbf{var}\ r : T \bullet \\ \mu X \bullet \left(\begin{array}{l}(init \longrightarrow \mathbf{wait}\ 0\,..\,Init_{TB}\ ;\ InitOp\ ;\ X) \\ \quad\Box \\ (rec\,?\,x \longrightarrow \mathbf{wait}\ 0\,..\,Rec_{TB}\ ;\ MOp([\![\, x \,]\!])\ ;\ X) \\ \quad\Box \\ (out\,!\,r \longrightarrow X)\end{array}\right)\end{array}\right) \\ \quad [\![\varnothing \mid \{\!|\ init, rec, out\ |\!\} \mid \varnothing]\!] \\ \left(\mu X \bullet start \longrightarrow init \longrightarrow \left(\begin{array}{l}(rec\,?\,y \longrightarrow \mathbf{skip})\ |||\\ (rec\,?\,y \longrightarrow \mathbf{skip})\ |||\\ \quad\ldots \\ (rec\,?\,y \longrightarrow \mathbf{skip})\end{array}\right);\ out\,?\,y \longrightarrow \mathbf{skip}\,;\ lockstep \longrightarrow X\right) \\ \ \backslash\ \{\!|\ init\ |\!\}\end{array}\right)$$

provided $InitOp$ and MOp are data operations and

$\mathrm{wrt}(InitOp) = \mathrm{wrt}(MOp) = \{r\}$ and $MOp(b_1 \uplus b_2) = MOp(b_1)\ ;\ MOp(b_2)$

Fig. 19. Design law for a control action.

terminates only when the environment has communicated n times on *rec*. Further refinement is needed to eliminate this control action by decomposing and distributing it into the handlers.

The refinement of shared data is perhaps the most novel and interesting part of the verification laws. Our experience shows that we can define specialised laws that deal with common design patterns, such as inputs and outputs, barrier synchronisations, and control fragments. The main challenges here are to identify the channels or actions that ought to be refined using a particular sharing law, and to eliminate emerging control fragments. The latter may be achieved by further laws (or tactics) that decompose and distribute those control fragments into handlers, subject to guidance by the developer. We next look at the CD_x example in order to demonstrate the refinement for a realistic and non-trivial SCJ program.

5. Refinement of the CD_x

To illustrate the refinement laws, we consider the refinement of the CD_x specification in Figure 3. This corresponds to an O anchor. The structure of this section mirrors that of Section 4, discussing the application of the laws for each of the five verification aspects in a separate section. A more detailed account of the refinement, including all elementary steps, can be found in [ZCW$^+$12] and the complete SCJ program code is available on `http://www.cs.york.ac.uk/circus/hijac/cdx.html` for inspection.

5.1. Introduction of cycle timings

Our starting point is the recursion in the main action of the process *CDxSpec* in Figure 3. It is recaptured below after applying the copy rule to eliminate the reference to *CDxCycle*. (The copy rule for actions permits us to replace the invocation of a local action by its definition within a process.)

$$\mu X \bullet \left(\begin{array}{l} \mathit{next\_frame}\,?\,\mathit{frame}\ @\ t_1 \longrightarrow \\ \left(\begin{array}{l} \mathit{RecordFrame}; \\ \mathbf{wait}\ w : 0\,..\,\mathit{FRAME\_PERIOD} - \mathit{OUT\_DL} - t_1 \bullet \\ \mathbf{var}\ \mathit{colls} : \mathbb{N} \bullet \mathit{CalcCollisions}; \\ \left(\begin{array}{l} \mathit{output\_collisions}\,!\,\mathit{colls}\ @\ t_2 \longrightarrow \\ \mathbf{wait}\ \mathit{FRAME\_PERIOD} - (t_1 + w + t_2) \end{array} \right) \blacktriangleleft \mathit{OUT\_DL} \end{array} \right) \end{array} \right) \blacktriangleleft \mathit{INP\_DL}\,;\ X$$

We begin by introducing cycle timings into the model. For that, we first identify that in the above recursion, **wait** $\mathit{FRAME\_PERIOD} - (t_1 + w + t_2)$ fills the time gap between cycles. We then apply Law 2 in order to introduce an interleaving with **skip** there. The extraction laws we require to move the **wait** action to the outside of the body of the recursion are, specifically, Law 3 – 5 and Law 11. The application of Law 3 and Law 4 raises proof obligations. For instance, Law 4 generates a proof obligation to show that $\mathit{OUT\_DL} \leq \mathit{FRAME\_PERIOD} - (t_1 + w)$. We can prove it by making use of local assumptions about the value of w. Generally, for actions **wait** $w : t_1\,..\,t_2 \bullet A(w)$, a valid inference is to introduce an assumption $t_1 \leq w \leq t_2$ into A (a similar law exists for timed prefixes embedded in a synchronisation deadline). In the above case, the local assumption implies that $w \leq \mathit{FRAME\_PERIOD} - \mathit{OUT\_DL} - t_1$, which can be rewritten into the proviso using elementary laws of arithmetic.

The result of the interleaving extraction is given by the action below.

$$\mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} \mathit{next\_frame}\,?\,\mathit{frame}\ @\ t_1 \longrightarrow \\ \left(\begin{array}{l} \mathit{RecordFrame}; \\ \mathbf{wait}\ 0\,..\,\mathit{FRAME\_PERIOD} - \mathit{OUT\_DL} - t_1; \\ \mathbf{var}\ \mathit{colls} : \mathbb{N} \bullet \mathit{CalcCollisions}; \\ (\mathit{output\_collisions}\,!\,\mathit{colls} \longrightarrow \mathbf{skip}) \blacktriangleleft \mathit{OUT\_DL} \end{array} \right) \end{array} \right) \blacktriangleleft \mathit{INP\_DL} \\ \boxed{|||\ \mathbf{wait}\ \mathit{FRAME\_PERIOD}} \end{array} \right); X$$

At this point, we apply a further refinement to narrow the time budget specified by the nondeterministic delay **wait** $0\,..\,\mathit{FRAME\_PERIOD} - \mathit{OUT\_DL} - t_1 \bullet \ldots$ using Law 16. Above, this budget, in theory, enables us to make use of additional time gained when an environment synchronises on *next_frame* sooner than the deadline *INP_DL*; more precisely, we then gain $\mathit{INP\_DL} - t_1$ time units. In practice, however, we cannot rely on the environment acting in a benevolent manner, hence that additional time is difficult (if not impossible) to utilise by an implementation. The motivation for narrowing the time budget is to remove the reference to t_1 so that we can subsequently remove the timed prefix on *next_frame*. We thus refine the budget into

$$\mathbf{wait}\ 0\,..\,\mathit{FRAME\_PERIOD} - \mathit{OUT\_DL} - \mathit{INP\_DL}$$

whereby we ignore the value of t_1. Finally, we also use the auxiliary distribution laws in Figure 24 (Appendix A) to localise the outer synchronisation deadline $(\ldots) \blacktriangleleft \mathit{INP\_DL}$ to the relevant prefix. We conclude the introduction of cycle timings by introducing a termination deadline $(\ldots) \blacktriangleright \mathit{FRAME\_PERIOD}$ on the body of the recursion. Below we have the action that results from the aforementioned refinement steps.

$$\mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} (\mathit{next\_frame}\,?\,\mathit{frame} \longrightarrow \mathit{RecordFrame}) \blacktriangleleft \mathit{INP\_DL}; \\ \mathbf{wait}\ 0\,..\,\mathit{FRAME\_PERIOD} - \mathit{OUT\_DL} - \mathit{INP\_DL}; \\ \mathbf{var}\ \mathit{colls} : \mathbb{N} \bullet \mathit{CalcCollisions}; \\ (\mathit{output\_collisions}\,!\,\mathit{colls} \longrightarrow \mathbf{skip}) \blacktriangleleft \mathit{OUT\_DL} \end{array} \right) \blacktriangleright \mathit{FRAME\_PERIOD} \\ |||\ \mathbf{wait}\ \mathit{FRAME\_PERIOD} \end{array} \right); X$$

The introduction of the termination deadline raises a proof obligation $\mathit{TakesAtMost}(B) \leq \mathit{FRAME\_PERIOD}$ where B is the interleaving above in the body of the recursion. This is to show that B does not overrun the cycle time. Appendix B contains the calculation proving that $\mathit{TakesAtMost}(B) = \mathit{FRAME\_PERIOD}$.

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

Fig. 20. Refined Z operation specifying the cyclic mission behaviour of the CD_x.

5.2. Decomposition of data operations

The next aspect of the verification consists of data refinement. This alters the state and data operations of the *CDxSpec* process. The laws for data refinement in Z [WD96, Cav97] and *Circus* [CSW03, Oli05] have been explored elsewhere and thus are not subject of this paper; we hence only present the relevant results. That is, in particular, the refined *RecordFrame* operation, given in Figure 20. A detailed account of the refinement steps is discussed in [ZCW+12], and the article [CZW+13] describes the overall strategy that is used here. We emphasise that data refinement is inherently a non-trivial activity that, in many cases, requires human ingenuity. It is a classical problem that has been extensively studied.

We observe that in Figure 20, the state components *posns* and *motions* of the abstract CD_x specification *CDxSpec* have been replaced by the concrete data objects *currentFrame* and *state*, which are, respectively, instances of the *OhCircus* classes *RawFrame* and *StateTable*. Their class definitions can be found in [ZCW+12]. Whereas *currentFrame* stores the current frame of aircraft positions by virtue of arrays, *state* records their previous positions in a Java `Map`; from this, it is possible to reconstruct the motion information. We note that the functions $F(\_)$ and $G(\_,\_)$ are abstraction functions mapping concrete to abstract states. For instance, given an object *currentFrame*, the function F calculates the corresponding value of *posns* based on the fields of the underlying *RawFrame* class that encodes aircraft positions by multiple arrays of type *float*. We omit the definitions of F and G for brevity, but likewise they can be found in the report [ZCW+12].

A further state component *work* of class type *Partition* (defined in Appendix C) is introduced: it records partitions of the computational work for collision detection, calculated by a voxel-based reduction algorithm. The local variable *voxel_map* models a Java `HashMap` that maps voxels — volumetric elements that subdivide the 3d space — to `Lists` of aircraft in that voxel. Finally, the *collision* component records the result of the detection. Hence, after the data refinement, *RecordFrame* not only records aircraft positions, but, also detects collisions, so that the refined *CalcCollisions* operation, in contrast to the abstract one in Figure 3, merely has to output the value of the *collisions* component.

We start by decomposing *RecordFrame* into sequences and conjunctions of data operations. This is done by applying Law 13 three times, followed by an application of Law 14. The refinement here is not trivial, since the *RecordFrame* operation contains further existentially-quantified variables that either correspond to abstract model variables (*posns* and *motions*) arising from earlier data refinement, or local variables like *voxel_map* (here 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. In particular, *voxel_map* can be localised to the inner predicate that models the handler for voxel-hashing.

$$\exists\, voxel\_map : HashMap[Vector2d, List[Motion]] \mid voxel\_map \neq \mathbf{null} \bullet \left(\left(\begin{array}{l} \forall\, a_1, a_2 : Aircraft \mid \{a_1, a_2\} \subseteq \mathrm{dom}\ posns' \bullet \\ (a_1, a_2) \in CollSet(posns', motions') \Rightarrow \\ \quad \left(\begin{array}{l} \exists\, l : List[Motion] \mid l \in voxel\_map.values().elems() \bullet \\ MkMotion(a_1, posns'\ a_1 -_V motions'\ a_1, posns'\ a_1) \in l.elems() \land \\ MkMotion(a_2, posns'\ a_2 -_V motions'\ a_2, posns'\ a_2) \in l.elems() \end{array} \right) \end{array} \right) \land \atop voxel\_map.values().elems() = \bigcup \{ i : 1\,..\,4 \bullet work'.getDetectorWork(i).elems() \} \right)$$

Another issue that needs to be addressed is that the flow of data is not always explicit in abstract operations specifying missions. In our SCJ program (see Figure 2), for example, data is transmitted between the `ReducerHandler` (captured by the predicate above) 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 records information of how the computational work is split, and this variable is also read by the detector handlers. In the data-refined *RecordFrame* operation in Figure 20, the last existential conjunct

$$\begin{array}{l} \exists\, collset : \mathbb{F}(Aircraft \times Aircraft) \mid collset = CollSet(posns', motions') \bullet \\ \quad (\#\, collset = 0 \land collisions' = 0) \lor (\#\, collset > 0 \land collisions' \geq (\#\, collset)\ \mathrm{div}\ 2) \end{array}$$

specifies the behaviour of the detector handlers, and we notice that the new value of *collisions* is determined by the function $CollSet(posns', motions')$ in terms of the abstract model variables. To reformulate it in terms of *work*, as it is needed to align the model to the data flow in the SCJ design that is verified, we require further rewriting that appears to necessitate essential human guidance. We skip further details of the refinement of *RecordFrame* and just present the result of the decomposition using Law 12 in Figure 8.

$$\mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} next\_frame\,?\,frame \longrightarrow \\ \quad \left(\begin{array}{l} StoreFrame \fatsemi \\ PartitionWork \fatsemi \\ DetectCollisions \end{array} \right) \end{array} \right) \blacktriangleleft INP\_DL; \\ \mathbf{wait}\ 0\,..\,FRAME\_PERIOD - OUT\_DL - INP\_DL; \\ \mathbf{var}\ colls : \mathbb{N} \bullet CalcCollisions; \\ (output\_collisions\,!\,colls \longrightarrow \mathbf{skip}) \blacktriangleleft OUT\_DL \end{array} \right) \blacktriangleright FRAME\_PERIOD \\ |||\ \mathbf{wait}\ FRAME\_PERIOD \end{array} \right) ;\ X$$

We observe that the *RecordFrame* operation has been replaced by a sequence of three data operations: that is, $StoreFrame \fatsemi PartitionWork \fatsemi DetectCollisions$. Law 13 has therefore been applied three times. The definition of *StoreFrame* and *PartitionWork* is in Appendix D. The report [ZCW+12] discusses in more detail the transformation of the refined *RecordFrame* operation that enables the application of the laws. We omit a further discussion of their application here as this is not the most interesting aspect of the refinement.

The *DetectCollisions* operation is further decomposed into a conjunction, using Law 14. Its definition at this stage of the refinement is included in Figure 21. The local variables $collset_1$, $collset_2$, and so on, have been subsequently introduced in auxiliary rewriting steps to prepare the application of the law; this again relies on human expertise. After application of Law 14, *DetectCollision* in Figure 21 is refined into the action

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

where the decomposed Z operations *CalcPartCollisions* and *SetCollisionsFromParts* can also be found in Appendix D. This completes the decomposition of data operations. Each data operation at this stage can be traced to one of the seven handlers of the design that we verify. Table 1 in Appendix D illustrates the relationship between the data operations and handlers of the SCJ program.

$$
\begin{array}{|l}
\textit{DetectCollisions} \\
\hline
\textit{work}, \textit{work}' : \textit{Partition};\ \textit{collisions}, \textit{collisions}' : \textit{int} \\
\hline
\textit{work}' = \textit{work} \land \\
\exists\, \textit{collset}_1, \textit{collset}_2, \textit{collset}_3, \textit{collset}_4 : \mathbb{F}\,(\textit{Aircraft} \times \textit{Aircraft}) \mid \\
\textit{collset}_1 = \left\{ \begin{array}{l} a_1, a_2 : \textit{Aircraft} \mid \\ \quad \left(\begin{array}{l} \exists\, l : \textit{List}[\textit{Motion}] \mid l \in \textit{work} \,.\, \textit{getDetectorWork}(1) \,.\, \textit{elems}() \bullet \\ \text{“Predicate that states that } (a_1, a_2) \text{ collide and are in } l\text{”} \end{array} \right) \end{array} \right\} \land \\
\textit{collset}_2 = \left\{ \begin{array}{l} a_1, a_2 : \textit{Aircraft} \mid \\ \quad \left(\begin{array}{l} \exists\, l : \textit{List}[\textit{Motion}] \mid l \in \textit{work} \,.\, \textit{getDetectorWork}(2) \,.\, \textit{elems}() \bullet \\ \text{“Predicate that states that } (a_1, a_2) \text{ collide and are in } l\text{”} \end{array} \right) \end{array} \right\} \land \\
\textit{collset}_3 = \ldots \land \textit{collset}_4 = \ldots \bullet \\
\textit{collisions}' = (\#\, \textit{collset}_1 \operatorname{div} 2) + (\#\, \textit{collset}_2 \operatorname{div} 2) + \#(\textit{collset}_3 \operatorname{div} 2) + (\#\, \textit{collset}_4 \operatorname{div} 2) \\
\hline
\end{array}
$$

Fig. 21. Shape of *DetectCollisions* before applying the decomposition law.

5.3. Distribution of time budgets

We proceed with the decomposition and distribution of time budget between the newly introduced sequential operations. For this, we introduce the handler-specific time budgets $\textit{StoreFrame}_{TB}$, $\textit{PartitionWork}_{TB}$ and $\textit{DetectCollisions}_{TB}$. Multiple applications of Law 15 in Figure 11 to the abstract time budget

$$\textbf{wait}\ 0 \,..\, \textit{FRAME\_PERIOD} - \textit{OUT\_DL} - \textit{INP\_DL}$$

produces the following action where the above budget has been split twice.

$$
\mu X \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} \textit{next\_frame}\,?\,\textit{frame} \longrightarrow \\ \quad \left(\begin{array}{l} \textit{StoreFrame} \,⨟ \\ \textit{PartitionWork} \,⨟ \\ \textit{DetectCollisions} \end{array} \right) \end{array} \right) \blacktriangleleft \textit{INP\_DL}; \\
\textbf{wait}\ 0 \,..\, \textit{StoreFrame}_{TB}; \\
\textbf{wait}\ 0 \,..\, \textit{PartitionWork}_{TB}; \\
\textbf{wait}\ 0 \,..\, \textit{DetectCollisions}_{TB}; \\
\ldots
\end{array} \right) \blacktriangleright \textit{FRAME\_PERIOD} \\
\interleave \textbf{wait}\ \textit{FRAME\_PERIOD}
\end{array} \right) ;\ X
$$

The law applications raise provisos that, in conjunction, require us to establish that

$$\textit{StoreFrame}_{TB} + \textit{PartitionWork}_{TB} + \textit{DetectCollisions}_{TB} \leq \textit{FRAME\_PERIOD} - \textit{OUT\_DL} - \textit{INP\_DL}$$

As we introduce the new budgets as abstract constants whose precise definition is deferred until we carry out the translation into code, we add the above as an axiomatic constraint to our *Circus* model.

Next, the time budgets are moved backwards through the sequence of data operations. Trivial elementary refinement steps are applied to distribute the prefix with an input via *next_frame* through the sequence and localise the deadline $(\ldots) \blacktriangleleft \textit{INP\_DL}$. The localisation of time budgets to the relevant Z operations essentially makes use of Law 17 in Figure 11. A related issue that arises is that Z compositions $(\textit{Op}_1 \,⨟\, \textit{Op}_2)$ have to be turned into *Circus* action compositions $(\textit{Op}_1\ ;\ \textit{Op}_2)$. Law *scompC* in [Cav97] achieves this. The result of localising budgets is illustrated by the action below.

$$
\mu X \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} \textit{next\_frame}\,?\,\textit{frame} \longrightarrow \\ \textbf{wait}\ 0 \,..\, \textit{StoreFrame}_{TB}\ ;\ \textit{StoreFrame} \end{array} \right) \blacktriangleleft \textit{INP\_DL}; \\
\textbf{wait}\ 0 \,..\, \textit{PartitionWork}_{TB}\ ;\ \textit{PartitionWork}; \\
\textbf{wait}\ 0 \,..\, \textit{DetectCollisions}_{TB}\ ;\ \textit{DetectCollisions}; \\
\ldots
\end{array} \right) \blacktriangleright \textit{FRAME\_PERIOD} \\
\interleave \textbf{wait}\ \textit{FRAME\_PERIOD}
\end{array} \right) ;\ X
$$

This last refinement step completes the allocation and distribution of time budgets. We note that further decomposition and distribution of the time budget $\mathit{DetectCollisions}_{TB}$ takes place implicitly during the subsequent parallelisation of actions. The reason we cannot perform this decomposition here is due to the fact that time budgets cannot be moved into schema conjunctions as present in the definition of *DetectCollisions*. Moreover, there is no obvious way to turn those conjunctions into actions at this stage as this would involve design via parallelisation laws, which is an orthogonal aspect of the verification, and discussed next.

5.4. Introduction of parallel handler actions

The parallelisation into handler actions relies on Law 20 (Figure 13) and Law 22 (Figure 22). First, Law 20 is applied three times to parallelise the sequential actions above, and then Law 22 is used to parallelise the conjunction in *DetectCollisions*. To illustrate the application of Law 20, we consider the sequential fragment

$$\left(\begin{array}{l} \left(\begin{array}{l} \mathit{next\_frame}\,?\,\mathit{frame} \longrightarrow \\ \mathbf{wait}\,0\,..\,\mathit{StoreFrame}_{TB}\ ;\ \mathit{StoreFrame} \end{array}\right) \blacktriangleleft \mathit{INP\_DL}; \\ \left(\begin{array}{l} \mathbf{wait}\,0\,..\,\mathit{PartitionWork}_{TB}\ ;\ \mathit{PartitionWork}; \\ \mathbf{wait}\,0\,..\,\mathit{DetectCollisions}_{TB}\ ;\ \mathit{DetectCollisions}; \\ \dots \end{array}\right) \end{array}\right)$$

Application of Law 20 transforms this into the action

$$\left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} \mathit{next\_frame}\,?\,\mathit{frame} \longrightarrow \\ \mathbf{wait}\,0\,..\,\mathit{StoreFrame}_{TB}\ ;\ \mathit{StoreFrame} \end{array}\right) \blacktriangleleft \mathit{INP\_DL}; \\ \mathit{reduce}\,!\,(\mathit{currentFrame}, \mathit{state}) \longrightarrow \mathbf{skip} \end{array}\right) \\ \quad [\![\{\mathit{currentFrame}, \mathit{state}\} \mid \{\!|\ \mathit{reduce}\ |\!\} \mid \{\mathit{work}, \mathit{collisions}\}]\!] \\ \left(\begin{array}{l} \mathit{reduce}\,?\,(\mathit{currentFrame}, \mathit{state}) \longrightarrow \\ \left(\begin{array}{l} \mathbf{wait}\,0\,..\,\mathit{PartitionWork}_{TB}\ ;\ \mathit{PartitionWork}; \\ \mathbf{wait}\,0\,..\,\mathit{DetectCollisions}_{TB}\ ;\ \mathit{DetectCollisions}; \\ \dots \end{array}\right) \end{array}\right) \end{array}\right) \setminus \{\!|\ \mathit{reduce}\ |\!\}$$

A new channel *reduce* of type $\mathit{RawFrame} \times \mathit{StateTable}$ is introduced by the law. It communicates the relevant shared data between the handler that inputs the next radar frame and records current and previous aircraft positions (*StoreFrame*), and the handler that performs the voxel hashing and partitioning of the computational work (*PartitionWork*). The channel moreover enforces sequential execution of the handlers.

Before proceeding with applying the law again, namely to parallelise the sequence between *PartitionWork* and *DetectCollisions*, a few elementary refinement steps are needed. These are to extract the hiding on *reduce* to the outer level of the recursion, and to distribute the prefix in the right-hand parallel action.

$$\left(\mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} (\ \dots\ ) \\ \quad [\![\{\mathit{currentFrame}, \mathit{state}\} \mid \{\!|\ \mathit{reduce}\ |\!\} \mid \{\mathit{work}, \mathit{collisions}\}]\!] \\ \left(\begin{array}{l} \left(\begin{array}{l} \mathit{reduce}\,?\,(\mathit{currentFrame}, \mathit{state}) \longrightarrow \\ \mathbf{wait}\,0\,..\,\mathit{PartitionWork}_{TB}\ ;\ \mathit{PartitionWork} \end{array}\right); \\ \left(\begin{array}{l} \mathbf{wait}\,0\,..\,\mathit{DetectCollisions}_{TB}\ ;\ \mathit{DetectCollisions}; \\ \dots \end{array}\right) \end{array}\right) \end{array}\right) \blacktriangleright \dots \\ |\!|\!|\ \mathbf{wait}\,\mathit{FRAME\_PERIOD} \end{array}\right) ; X \right) \setminus \{\!|\ \mathit{reduce}\ |\!\}$$

We omit a detailed discussion here of the two pending applications of Law 20 as this does not add anything new. Instead, we consider the parallelisation of *DetectCollisions* via Law 22. After the parallelisation of sequential actions, the action that carries out the detection of collisions has the following shape.

$$\left(\begin{array}{l} \mathit{detect}\,?\,\mathit{work} \longrightarrow \\ \mathbf{wait}\,0\,..\,\mathit{DetectCollisions}_{TB}\ ;\ \mathit{DetectCollisions}; \\ \mathit{output}\,!\,\mathit{collisions} \longrightarrow \mathbf{skip} \end{array}\right)$$

The channels *detect* and *output* have been introduced by two further applications of Law 20 and, as before, encapsulate the communication of shared data. For instance, *detect* propagates the shared data that determines voxel partitions from the reducer handler to the detector handlers, and *output* propagates the shared data used to hold the detection result to the handler that outputs it. Expanding the definition of *DetectCollisions* using the copy rule yields the following action.

$$\left(\begin{array}{l} detect\,?\,work \longrightarrow \\ \mathbf{wait}\,0\,..\,DetectCollisions_{TB}; \\ \left(\begin{array}{l} \mathbf{var}\,colls_1, colls_2, colls_3, colls_4 : \mathbb{Z} \bullet \\ \left(\begin{array}{l} (\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls_1/pcolls!] \wedge i? = 1) \wedge \\ (\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls_2/pcolls!] \wedge i? = 2) \wedge \\ (\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls_3/pcolls!] \wedge i? = 3) \wedge \\ (\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls_4/pcolls!] \wedge i? = 4) \end{array}\right); \\ \quad SetCollisionsFromParts(\llbracket colls_1, colls_2, colls_3, colls_4 \rrbracket) \end{array}\right); \\ output\,!\,collisions \longrightarrow \mathbf{skip} \end{array}\right)$$

Application of Law 22 refines it into the action below.

$$\left(\begin{array}{l} detect\,?\,work \longrightarrow \\ \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} \mathbf{var}\,colls_1 : \mathbb{Z} \bullet \mathbf{wait}\,0\,..\,CPC_{TB}; \\ (\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls_1/pcolls!] \wedge i? = 1); \\ rec\,!\,colls_1 \longrightarrow \mathbf{skip} \end{array}\right) \\ \parallel \\ \left(\begin{array}{l} \mathbf{var}\,colls_2 : \mathbb{Z} \bullet \mathbf{wait}\,0\,..\,CPC_{TB}; \\ (\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls_2/pcolls!] \wedge i? = 2); \\ rec\,!\,colls_2 \longrightarrow \mathbf{skip} \end{array}\right) \\ \parallel \\ \left(\begin{array}{l} \mathbf{var}\,colls_3 : \mathbb{Z} \bullet \mathbf{wait}\,0\,..\,CPC_{TB}; \\ (\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls_3/pcolls!] \wedge i? = 3); \\ rec\,!\,colls_3 \longrightarrow \mathbf{skip} \end{array}\right) \\ \parallel \\ \left(\begin{array}{l} \mathbf{var}\,colls_4 : \mathbb{Z} \bullet \mathbf{wait}\,0\,..\,CPC_{TB}; \\ (\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls_4/pcolls!] \wedge i? = 4); \\ rec\,!\,colls_4 \longrightarrow \mathbf{skip} \end{array}\right) \end{array}\right) \\ \quad \llbracket \varnothing \mid \{\!| rec |\!\} \mid \{colls\} \rrbracket \\ \left(\begin{array}{l} \mathbf{var}\,colls_1, colls_2, colls_3, colls_4 : \mathbb{Z} \bullet \\ \left(\begin{array}{l} (rec\,?\,x \longrightarrow \mathbf{wait}\,0\,..\,Rec_{TB}\ ;\ colls_1 := x); \\ (rec\,?\,x \longrightarrow \mathbf{wait}\,0\,..\,Rec_{TB}\ ;\ colls_2 := x); \\ (rec\,?\,x \longrightarrow \mathbf{wait}\,0\,..\,Rec_{TB}\ ;\ colls_3 := x); \\ (rec\,?\,x \longrightarrow \mathbf{wait}\,0\,..\,Rec_{TB}\ ;\ colls_4 := x) \end{array}\right); \\ \mathbf{wait}\,0\,..\,SCFP_{TB}\ ;\ SetCollisionsFromParts(\llbracket colls_1, colls_2, colls_3, colls_4 \rrbracket) \end{array}\right) \end{array}\right); \\ output\,!\,collisions \longrightarrow \mathbf{skip} \end{array}\right)$$

The result here illustrates that we require additional elementary law applications to turn this into a parallel composition for four detector handlers. In particular, the initial synchronisation on *detect* and the final synchronisation on *output* have to be moved into each detector handler and the auxiliary control action. For this, all detector handlers and the control action have to synchronise on *detect* and *output*. Whereas the synchronisation on *detect* is later refined into an SCJ event that releases multiple (detector) handlers, the synchronisation on *output* is a barrier mechanism that requires further refinement as shown below. After the aforementioned finalising steps each detector handler now has the following shape.

$$\left(\begin{array}{l} detect\,?\,work \longrightarrow \\ \left(\begin{array}{l} \mathbf{var}\,colls_k : \mathbb{Z} \bullet \mathbf{wait}\,0\,..\,CPC_{TB}; \\ (\exists\, i? : \mathbb{Z} \bullet CalcPartCollisions[colls_k/pcolls!] \wedge i? = k); \\ rec\,!\,colls_k \longrightarrow \mathbf{skip} \end{array}\right);\ output\,?\,y \longrightarrow \mathbf{skip} \end{array}\right) \text{where } k : 1\,..\,4$$

As noted earlier on, the refinement has resulted in further decomposition of a time budget. Namely, the budget $DetectCollisions_{TB}$ has been split into two budgets, CPC_{TB} and $SCFP_{TB}$, which encapsulate the time allowances for the parallel detectors as well as the operation that merges the results.

5.5. Encapsulation of shared data

The last verification issue is the refinement of shared data. First of all, the application of the parallelisation law for sequential actions (Law 20) has introduced three typed channels: *reduce*, *detect* and *output*. Each of these channels is decomposed now using the sharing Law 35 in Appendix A, after introducing a channel *lockstep* to ensure lock-step progress of the recursive handler actions. It is a specialisation of Law 23 presented in Section 4.5 that considers unidirectional communication between handlers that execute sequentially and do not terminate. Unlike Law 23, this law does not introduce the channel c_{sync} due to the particular structure of actions, and thus yields a simplified refinement result.

To illustrate the application of Law 35, we consider the parallel action

$$\left(\begin{array}{l}
\left(\mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} next\_frame\,?\,frame \longrightarrow \\ \mathbf{wait}\ 0\mathinner{..}StoreFrame_{TB}\ ;\ StoreFrame \end{array}\right) \blacktriangleleft INP\_DL; \\ reduce\,!\,(currentFrame, state) \longrightarrow \mathbf{skip} \end{array}\right) ;\ lockstep \longrightarrow X \right) \\
[\![\{currentFrame, state\} \mid \{\!|\, reduce, lockstep \,|\!\} \mid \{voxel\_map, work\}]\!] \\
\left(\mu X \bullet \left(\begin{array}{l} reduce\,?\,(currentFrame, state) \longrightarrow \\ \mathbf{wait}\ 0\mathinner{..}PartitionWork_{TB}\ ;\ PartitionWork; \\ detect\,!\,work \longrightarrow \mathbf{skip} \end{array}\right) ;\ lockstep \longrightarrow X \right) \\
[\![\{voxel\_map, work\} \mid \{\!|\, detect, lockstep \,|\!\} \mid \varnothing]\!] \\
\ldots
\end{array}\right)$$

This corresponds to the result of applying Law 20 on page 30 after exhaustive application of the parallelisation laws. The application of Law 35 (Appendix A) refines this into the action fragment below

$$\left(\begin{array}{l}
\left(\left(\begin{array}{l}
\left(\begin{array}{l}
\mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} next\_frame\,?\,frame \longrightarrow \\ \mathbf{wait}\ 0\mathinner{..}StoreFrame_{TB}\ ;\ StoreFrame \end{array}\right) \blacktriangleleft INP\_DL; \\ reduce_{write}\,!\,(currentFrame, state) \longrightarrow reduce_{pivot} \longrightarrow \mathbf{skip} \end{array}\right) ; \\
lockstep \longrightarrow X
\end{array}\right) \\
[\![\{currentFrame, state\} \mid \{\!|\, reduce_{pivot}, lockstep \,|\!\} \mid \{voxel\_map, work\}]\!] \\
\left(\begin{array}{l}
\mu X \bullet \left(\begin{array}{l} reduce_{pivot} \longrightarrow reduce_{read}\,?\,(currentFrame, state) \longrightarrow \\ \mathbf{wait}\ 0\mathinner{..}PartitionWork_{TB}\ ;\ PartitionWork; \\ detect\,!\,work \longrightarrow \mathbf{skip} \end{array}\right) ; \\
lockstep \longrightarrow X
\end{array}\right)
\end{array}\right) \setminus \{\!|\, reduce_{pivot} \,|\!\}\right) \\
[\![\{currentFrame, state, work, collisions\} \mid \{\!|\, reduce_{read}, reduce_{write} \,|\!\} \mid \varnothing]\!] \\
\left(\begin{array}{l}
\mathbf{var}\ v : RawFrame \times StateTable \bullet \\
\quad \mu X \bullet \left(\begin{array}{l} (reduce_{read}\,!\,v \longrightarrow \mathbf{skip}) \;\Box \\ (reduce_{write}\,?\,x \longrightarrow v := x) \end{array}\right) ;\ X
\end{array}\right) \\
\quad \setminus \{\!|\, reduce_{read}, reduce_{write} \,|\!\}
\end{array}\right)$$

where the third parallel action encapsulates the shared data and thus corresponds to (a part of) *MArea* in our target model in Figure 5. The typeless channel $reduce_{pivot}$ is freshly introduced and models an SCJ event that in the CD_x program releases `ReducerHandler`. The channels $reduce_{read}$ and $reduce_{write}$ model variable accesses to write to the shared variables `currentFrame` and `state` in the `CDxMission` class. To emphasise their rôle, we can, of course, subsequently rename these channels. Here, in particular, it is sensible to rename them to *getCurrentFrameState* and *setCurrentFrameState*, and also $reduce_{pivot}$ to $reduce_{fire}$.

As mentioned before, the sharing law for channel decomposition is applied two more times, which gives rise to two more parallel actions that contribute to *MArea*. For reasons of space, we omit a discussion of the application of the barrier law (Law 24) and parallel design law (Law 25) though both are needed for the CD_x refinement too. For a detailed account, we refer again to [ZCW⁺12]. Applying those laws produces a *Circus* process whose shape corresponds precisely to *SCJDesign* in Figure 5. The refinement here produces a model that reflects the design of the CD_x program that we introduced in Section 2.1. The aim of the refinement was indeed to establish this correspondence in order to verify an *a priori* given SCJ program. If no program is given, we are at liberty to apply the laws based on preference whenever alternative design choices emerge.

Having examined the application of the laws in the context of a specific example, in the next section, we look more generally at the possibility of automating the law applications and refinement.

6. Automation

In this section, we discuss ways of automating the application of the refinement laws. Again, we look at each of the five verification issues as they were described in Sections 4.1 to 4.5, and examine how the laws for that aspect may be automated by tactics and decision procedures. Our objective is to determine the parts of the verification that do not require expert knowledge in refinement techniques and, in relation to this, what guidance has to be provided by a (non-expert) user of our technique to enable the automation of those parts. This is based on our experience with automation of refinement in *Circus* [OZC11, ZOC12] using theorem provers and an embedding of *Circus* and its semantics.

6.1. Introduction of cycle timings

Human guidance is required to identify recursions that model cyclic activities, as well as **wait** statements inside those recursions that fill the time gap between cycles. Once this is done, the application of Law 2 (to introduce an interleaving with **skip**) and the extraction laws in Figure 7 can be automated, subject to automating simple transformations that align the shape of time expressions in **wait** statements to facilitate the matching of laws such as Law 4 and Law 5 in Figure 7.

Some of the extraction laws have provisos whose proofs, in most cases, merely require arithmetic rewriting and laws for solving inequalities. Ample work has been done elsewhere on automating such proofs [Nor03]. In some cases, it turns out that we also require local assumptions about bound variables introduced by timed prefixes and nondeterministic delays. For example, in $(c \mathbin{@} t \longrightarrow A(t)\ ;\ \ldots) \blacktriangleleft d$ we have a local assumption in A that $t \leq d$. Likewise, in $\mathbf{wait}\ t : t_1 \mathinner{..} t_2 \bullet A(t)$ we can assume that $t_1 \leq t \leq t_2$ in A. The propagation of such assumptions can be automated by a proof tool in a fairly straightforward manner.

An automatic procedure can decide when the **wait** statement has been fully extracted from the body of the recursion, as this terminates the application of the extraction laws. Where automation cannot proceed due to no extraction law being applicable, assistance is required either to perform elementary refinement steps to enable further application of extraction laws, or to concede that the cycle length of the recursive action is not fixed and thus not a valid target for the refinement.

The finalising application of Law 1 for the introduction of a termination deadline raises a proviso $\mathit{TakesAtMost}(A) \leq T$ where A is the body of the recursion and T its cycle. We expect that this proviso can be discharged automatically after rewriting the application of $\mathit{TakesAtMost}$ using the rules in Figure 6. This is again contingent on support for solving arithmetic inequalities and simplifying arithmetic expressions.

Overall, we conclude that the prospect of automating this part of the verification is altogether positive: with special tactics for solving arithmetic inequalities there ought to be no need for manual refinement. The only potential obstacle for automation is that $\mathit{TakesAtMost}(A)$ may yield too coarse an approximation of the execution time of A, and as a consequence the proviso of Law 1 becomes unprovable. This is not necessarily a show-stopper as we can introduce the termination deadline anyway, though it may render the model unimplementable. So far, we have not encountered specifications where we had to resort to this solution.

6.2. Decomposition of data operations

From our experience gained working on the CD_x case study, decomposition of data operations seems amongst the most difficult aspects of the refinement and presumably the most challenging to automate. This is not surprising; research in refinement techniques [AH07] identifies decomposition as being inherently a difficult problem. Furthermore, the shape of the operations that we target in this aspect of the verification is largely influenced by earlier data refinement, hence we can make little to no assumptions about the actual structure of models that we target with our laws in this aspect of the verification.

For automation, the main challenge is to conduct preliminary transformations that enable the application of one of the decomposition laws. For simple laws such as Laws 12 and 13, tailored refinement tactics could perhaps be used to decompose schema predicates and invariants automatically, so that the laws can be directly applied. Those tactics are expected to apply logic rewrites to isolate predicates that reference certain components of a schema, and guidance as to what components to isolate is likely to be needed.

For more complex patterns such as parallel decomposition via Law 13, the developer needs to determine the target of each law application, that is, the schema predicates on the right-hand side of the law. 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 by the application of the law. Specialised proof tactics may again be useful in trying to discharge those verification conditions. The advantage of this approach is that less expertise is needed because the developer just needs to postulate the result of the law application rather than carry out the refinement. If the automatic tactic fails to prove the verification condition, we envisage that the refinement verification requires more expertise and perhaps involvement of a domain expert.

We can in any case profit from a large body of existing work on decomposition. The main issue of the intermediate refinements here is to make the data flow in the program explicit in the predicates of operation schemas. Usually, the developer will have an understanding of how data is used in the SCJ program to be designed; this already determines which laws need to be applied.

6.3. Distribution of time budgets

This aspect involves two conceptual designs: the decomposition (splitting) of time budgets and the localisation of budgets to their respective actions. Both of these aspects require guidance from the developer, but the refinement itself is easy to automate. In a first step, the developer has to identify time budgets $\mathbf{wait}\, 0 \ldots TB$ in actions that have to be split. For this, it is convenient to specify a number of constants TB_1, TB_2, and so on, that constitute the split budgets where $(\Sigma i : 1 \ldots n \bullet TB_i) \leq TB$. We can then automate the sequenced application of the Laws 15 and 16 in order to refine the original budget into an action sequence

$$\mathbf{wait}\, 0 \ldots TB_1 \,;\, \mathbf{wait}\, 0 \ldots TB_1 \,;\, \ldots \,;\, \mathbf{wait}\, 0 \ldots TB_n \quad .$$

Further automation of the refinement can be envisaged by annotating each data operation with the intended time budget TB_i, and using tactics to mechanically perform the localisation of the budgets to the respective data operations. Such tactics merely have to decide whether a budget needs to be moved forward or backward through a sequence, and depending on this, either to use Law 17 directly or its symmetric version.

More difficult to automate is the intentional reduction of time budgets using Law 16 in order to obtain a more tractable model for further refinement. This is illustrated in Section 5.1, where the budget $\mathbf{wait}\, w : FRAME\_PERIOD - OUT\_DL - t_1$ is narrowed to simplify the shape of the cycle action. As with splitting budgets, the developer has to determine the narrowed budget as this is a design issue.

To conclude, we observe that the splitting and narrowing laws (Law 15 and Law 16) for budgets give rise to provisos that have to be discharged. The technique for doing so is similar to the one described earlier on in proving the provisos for cycle-timing laws. After application of the narrowing law, it is sensible to assume that all budgets are determined by plain constants rather than complex time expressions. The only notable proof effort is hence to discharge the proviso of Law 16. Overall, this verification issue appears to be the easiest to automate, subject to the developer providing the above information.

6.4. Introduction of parallel handler actions

The shapes we target here are the ones produced by earlier decomposition of data operations; this facilitates automation of this aspect of the verification. We observe, for instance, that the first parallelisation Law 19 targets precisely the shape of models generated by earlier applications of Law 12 (Figure 8). Similarly, the second parallelisation Law 20 targets the shape of models generated by earlier applications of Law 13 (Figure 9), subsequent to replacing Z compositions by action sequences (which is done collaterally as part of the distribution of time budgets). It is hence possible to suggest applicable laws automatically so that the developer merely has to make a choice of which law to apply if there is more than one.

As with the decomposition of data operations, there turn out to be situations where intermediate refinement steps are required between the law applications, as we illustrated in Section 5. Here, however, those steps are much more straightforward and susceptible to automation, although it is still an open issue how a collection of tactics can be defined. Heuristics are deemed to be useful in that context to normalise actions into shapes that make subsequent law applications more likely to be feasible.

6.5. Encapsulation of shared data

The encapsulation of shared data requires a more diverse set of strategies for automation. As already mentioned, we have three classes of laws here: firstly, laws for channel decomposition; secondly, laws for high-level control mechanisms such as a barrier; and thirdly, laws for the refinement of residual control actions that emerge during refinement and are expected to (mostly) disappear. Regarding the first class, we have the generic decomposition Law 23 that can be applied to arbitrary actions to decompose a channel c hidden in the model. Here, we merely require the developer to identify channels to be decomposed. These are typically channels that arise from earlier applications of Law 35 for sequential transfer of data. Although the definition of $\mathit{WWConfFree}$ is on the whole elaborate in having to deal with a lot of cases, fundamentally its evaluation can be automated in a straightforward manner. Establishing the proviso $\mathit{WWConfFree}(c)(A)$ is therefore trivial where information flow is static. Manual proof effort may only be required in cases where more sophisticated mechanisms determine the direction in which information flows.

We also require some intermediate refinements that, as before, are mostly concerned with reordering parallel actions and extracting channel hidings to tease out the MArea action; this is because the parallel fragments contributing to MArea are typically embedded inside the parallel handler actions after application of sharing laws like Law 23 and Law 35. These manipulations are not a significant challenge for automation. The channel replacement principle defined by $\mathit{ChanDecomp}(c)(A)$ is automatable, too.

For the second class of laws, the refinement is a simple matching of control design laws, modulo some preliminary transformations for reordering parallel actions and extraction of channel hidings; a refinement tool can make suggestions here, giving the developer a set of choices.

More interesting is the third class of laws. We recall that both the parallelisation Law 22 and sharing Law 24 give rise to an auxiliary control action. To complete the refinement, those control actions have to be eliminated so that we are left with only a parallel composition of handlers. The elimination proceeds by sometimes decomposing the control fragments further into parallel actions, and then collapsing those parallel actions with existing handler actions. This can give rise to further design, namely when the control fragments do not entirely disappear. We envisage that an engineer merely has to provide information about which actions ought to be collapsed, causing the relevant step laws to be applied automatically.

To conclude our account on automation, we observe that the refinement of time-related designs is the easiest to automate. Most difficult to tackle is the decomposition of data operations, but, as mentioned earlier, we can take advantage of extensive previous work on this subject. The parallelisation and sharing laws pose a twofold challenge in that they require more subtle guidance by the user as well as preliminary refinements that, however, seem feasible to be carried out by automatic tactics. If the laws can be determined *a priori*, the fundamental proof effort can be factored into verification conditions.

7. Proofs of Laws

In this section, we examine the proofs of a few of the laws in Section 4. In particular, we look at the novel *Circus Time* laws in Figure 11. We first present the Unifying Theories of Programming framework (UTP) which is our semantic framework, next give a brief account of the semantics of *Circus Time* in UTP, and afterwards present two examples of proofs. Further proofs can be found in [ZCW+12].

7.1. Unifying Theories of Programming

The UTP, at the core, is an algebra of relations described by alphabetised predicates. The latter are pairs formed by an alphabet of variables and a predicate over these variables. The alphabet of a predicate P is obtained by αP. Alphabets include observable quantities of interest, such as program variables or special (auxiliary) variables that capture particular aspects of a model of computation. Alphabetised predicates describe the observations we can make about program behaviour. For instance, the predicate $ok \land y \neq 0 \Rightarrow ok' \land z' = x/y$ specifies a computation that, if started in a state where $y \neq 0$, assigns to the program variable z the quotient of the program variables x and y. The variables ok and ok' (of boolean type) capture the observations that the program has started and terminated. Primed variables, as in Z, are used to refer to final (or sometimes intermediate) observations, and unprimed variables to initial ones.

More generally, the predicate $ok \land P \Rightarrow ok' \land Q$ encodes a computation that, if started in a state where its precondition P holds, terminates in a state where its postcondition Q holds. In UTP, this form is called a 'design' and the notation $P \vdash Q$ is used to abbreviate it.

UTP theories are sets of predicates over predefined alphabets. However, not every predicate describes a valid model of a computation. To delineate predicates that do from those that are meaningless, each theory defines a set of healthiness conditions. These are expressed as idempotent and monotonic functions. Valid models of computation in a UTP theory are the cumulative fixed points of those functions. For instance, for designs we have a healthiness condition $\mathbf{H1}(P) = ok \Rightarrow P$ whose fixed points are the predicates that are equivalent to *true* when $\neg\, ok$. This rules out any assumption about program behaviour before the program has started. Further healthiness conditions (**H2-H4**) for designs can be found in [HJ98].

Each UTP theory defines a collection of operators under which the predicates of the theory have to be closed. Certain operators have a uniform characterisation across theories. These operators are:

1. Sequential composition, which is modelled by relational composition.
2. Nondeterminism $P \sqcap Q$, which is modelled by disjunction: $P \sqcap Q \mathrel{\widehat{=}} P \vee Q$.
3. Parallel composition, which is modelled by a form of conjunction (parallel by merge).
4. The conditional $P \lhd b \rhd Q$, which is defined by $(b \wedge P) \vee (\neg\, b \wedge Q)$.
5. Refinement, which is modelled by (universal) reverse implication: $P \sqsubseteq Q \;\mathrel{\widehat{=}}\; [P \Leftarrow Q]$.

Above, $[P]$ denotes the universal closure over the variables in αP.

The definition of sequential composition as a predicate is recaptured below.

Def 5. $P\,;\; Q \;\mathrel{\widehat{=}}\; \exists\, v'' \bullet P[v' \setminus v''] \wedge Q[v \setminus v'']$ **provided** $out(\alpha P) = in(\alpha Q)'$

Above, v are the variables at the interface of P and Q. We note that the operator *in* yields the undashed (input) variables, and *out* the dashed (output) variables of an alphabet. The proviso establishes that the predicates are composable: the output variables of P have to correspond to the input variables of Q.

UTP designs taken by themselves are sufficient to model sequential programs, but they are not powerful enough to capture reactive and timing behaviour. For this, we next discuss the UTP theory of *Circus Time*.

7.2. *Circus Time*

The UTP theory of *Circus Time* is a derivative of the theory of reactive processes [OCW09]. It includes four auxiliary variables ok, $wait$, tr and $state$, as well as their dashed counterparts. The variables ok and ok' of boolean type record the observation that the predecessor or current action is in a stable state, and thus has not diverged. We note that this is different from the theory of designs where ok and ok' model program termination. Program termination here is captured by the boolean variables $wait$ and $wait'$. Specifically, $wait$ records that the predecessor has terminated, and $wait'$ records termination of the current action.

While termination is one possible observation of a reactive process, we may also observe interactions (prior to termination) with the environment. The variables tr and tr' of type $\mathrm{seq}^{+}(\mathrm{seq}\, Event \times \mathbb{P}\, Event)$ record time traces of interactions. In detail, tr records the interactions that have already taken place when execution starts, and tr' additionally includes the interactions of the current action, extending tr. Each element of the outer (non-empty) sequence represents an observation in one time unit. These observations are pairs where the first component of the pair is a sequence of events that have occurred within the time unit, and the second component is a refusal set containing the events that are refused at the end of the time unit. Lastly, the variables $state$ and $state'$ are used to record the initial and subsequent values of program variables.

The healthiness conditions of *Circus Time* are a recast of the healthiness conditions for reactive processes; Figure 22 summarises their definitions. $\mathbf{R1_A}(A)$ establishes that an action A cannot alter the previous history of interactions; the binary operator $\leq_A$ here is a specialised prefix operator for time traces, defined in Appendix E. $\mathbf{R2_A}(A)$ enforces insensitivity of A to interactions that took place before it started. Namely, we can replace tr by a trace with no interactions, and tr' by the difference $tr' -_A tr$ without changing the behaviour of A. Again, $-_A$ is a specialised sequence subtraction for time traces (also in Appendix E). $\mathbf{R3_A}(A)$ masks out any behaviours of A until the predecessor action has terminated ($wait$ is true). We note that in $\mathbf{R3_A}$, $\mathbf{I\!I}_A$ (also called skip) corresponds to the relational identity on the alphabet A; where $\mathbf{I\!I}$ does not have a subscript, it is the identity on the theory alphabet.

We define $\mathbf{R_A}$ as the composition $\mathbf{R1_A} \circ \mathbf{R2_A} \circ \mathbf{R3_A}$. The form $\mathbf{R_A}(P \vdash Q)$ is called a reactive design. It

Healthiness condition	Definition	Caveat
$\mathbf{R1_A}$	$\mathbf{R1}_A(A) \mathrel{\widehat=} A \wedge tr \leq_A tr'$	
$\mathbf{R2_A}$	$\mathbf{R2_A}(A) \mathrel{\widehat=} A[\langle(\langle\rangle, last(tr).2)\rangle, tr' -_A tr) / tr, tr']$	$tr \leq_A tr'$
$\mathbf{R3_A}$	$\mathbf{R3_A}(A) \mathrel{\widehat=} \mathbf{II}_A \lhd wait \rhd A$ where $\mathbf{II}_A \mathrel{\widehat=} (\neg\, ok \wedge tr \leq_A tr') \vee (ok' \wedge \mathbf{II}_{\{wait,tr,state\}})$	
$\mathbf{CSP1_A}$	$\mathbf{CSP1_A}(A) \mathrel{\widehat=} A \vee (\neg\, ok \wedge tr \leq_A tr')$	
$\mathbf{CSP2_A}$	$\mathbf{CSP2_A}(A) \mathrel{\widehat=} A\,;\; J_A$ where $J_A \mathrel{\widehat=} (ok \Rightarrow ok') \wedge \mathbf{II}_{\{wait,tr,state\}}$	

Fig. 22. Healthiness conditions for the theory of *Circus Time* actions.

can be shown that all predicates of the theory of *Circus Time* can be expressed in this form. This representation allows us to reduce proofs about *Circus Time* actions to (simpler) proofs about designs. In particular, since $\mathbf{R_A}$ is monotonic, $P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2$ establishes that $\mathbf{R_A}(P_1 \vdash Q_1) \sqsubseteq \mathbf{R_A}(P_2 \vdash Q_2)$, and the former can be further reduced to $(P_1 \Rightarrow P_2) \wedge (Q_2 \wedge P_1 \Rightarrow Q_1)$; that is, proofs about pre- and postconditions. This yields an effective strategy for proving our refinement laws.

The additional healthiness conditions **CSP1** and **CSP2** in Figure 22 are recast from the UTP theory of CSP: **CSP1** requires that upon divergence of an action, no assumptions can be made other than that the trace is extended. And **CSP2** captures that we cannot require a program to diverge.

We next present the definitions of several action constructs that are used later on in the proofs. For a complete account, the reader is referred to [Woo13]. The first definition is for a simple delay.

Def 6. $\textbf{wait}\, t \mathrel{\widehat=} \mathbf{R_A}(true \vdash delay(t) \wedge trace(tr') - trace(tr) = \langle\rangle)$

Above, $delay(t) \mathrel{\widehat=} (wait' \wedge \#\, tr' - \#\, tr < t) \vee (\neg\, wait' \wedge \#\, tr' - \#\, tr = t \wedge state = state')$ and the function *trace* converts a time trace into a conventional trace by concatenating the sequences of events in each time unit. For example, $trace(\langle(\langle a, b\rangle, r_1), (\langle c\rangle, r_2)\rangle) = \langle a, b, c\rangle$. The refusal sets r_1, r_2, and so on, are discarded by the *trace* function. While the *true* precondition of the reactive design implies the absence of divergence, the postcondition captures two essential behaviours: before t time units have elapsed ($\#\, tr' - \#\, tr < t$), we are in an intermediate (waiting) state. After t time units ($\#\, tr' - \#\, tr = t$), the action terminates ($\neg\, wait'$ holds) while leaving the state unchanged ($state' = state$). The conjunct $trace(tr') - trace(tr) = \langle\rangle$ in the definition captures that while $\textbf{wait}\, t$ executes, no interaction with the environment takes place.

The definition of a time budget $\textbf{wait}\, t_1 \mathinner{..} t_2$ is a nondeterministic choice of simple **wait** statements.

Def 7. $\textbf{wait}\, S \mathrel{\widehat=} \bigsqcap t : S \bullet \textbf{wait}\, t$

Here, we make use of the generalised choice $\bigsqcap x : S \bullet P(x)$, which is a nondeterminism of all behaviours $P(x)$ where x ranges over some set S. It is defined in UTP as follows: $\bigsqcap x : S \bullet P(x) \mathrel{\widehat=} \exists\, x : S \bullet P(x)$.

Having briefly introduced the semantics of *Circus Time*, we next present two examples of proofs of laws from Section 4 involving *Circus Time* actions.

7.3. Laws: example proofs

We present the proofs of two laws: Law 16 for narrowing budgets and Law 15 for splitting budgets.

7.3.1. Proof of the budget narrowing law

The first law we prove is Law 16 for narrowing a time budget. We recapture it below.

$\textbf{wait}\, 0 \mathinner{..} t_1 \sqsubseteq \textbf{wait}\, 0 \mathinner{..} t_2$ **provided** $t_2 \leq t_1$ (**Law 16**)

The proof of this law is facilitated by a more general lemma about generalised nondeterminism.

Lemma 1. $\bigsqcap x : S \bullet P(x) \sqsubseteq \bigsqcap x : T \bullet P(x)$ **provided** $T \subseteq S$

Law 26. Decomposition of simple time delays.

$$\textbf{wait}\ t \ \equiv\ \textbf{wait}\ t_1\ ;\ \textbf{wait}\ t_2 \ \ \textbf{provided}\ \ t = t_1 + t_2$$

Fig. 23. *Circus Time* law for the decomposition of simple delays.

This law is easily shown by rewriting the definition of $\bigsqcap x : S \bullet P(x)$ and using elementary logic deductions.

$\bigsqcap x : S \bullet P(x) \sqsubseteq \bigsqcap x : T \bullet P(x)$

$\equiv$ "unfolding definition of generalised choice"

$(\exists x : S \bullet P(x)) \sqsubseteq (\exists x : T \bullet P(x))$

$\equiv$ "unfolding definition of refinement"

$[\exists x : S \bullet P(x)] \Leftarrow [\exists x : T \bullet P(x)]$

$\Leftarrow$ "elementary logic"

$T \subseteq S \quad \square$

Because of $0 \mathinner{..} t_1 \subseteq 0 \mathinner{..} t_2$ under the assumption $t_2 \leq t_1$, we immediately obtain a proof of Law 16 after unfolding the time budgets by virtue of Def. 7 and subsequent application of Lemma 1.

7.3.2. Proof of the budget splitting law

More challenging is the proof of Law 15 for time budget splitting. For this, we have to show that

$$\textbf{wait}\ 0 \mathinner{..} t \ \equiv\ \textbf{wait}\ 0 \mathinner{..} t_1\ ;\ \textbf{wait}\ 0 \mathinner{..} t_2 \ \ \textbf{provided}\ \ t = t_1 + t_2 \ \ (\textbf{Law 15})$$

To prove this law, we first establish the validity of an analogous property for simple delays formulated by Law 26 in Figure 23. To prove it, we start by rewriting the right-hand side of that law:

$\textbf{wait}\ t_1\ ;\ \textbf{wait}\ t_2$

$\equiv$ "unfolding definition of **wait** statements (Def. 6)"

$\mathbf{R_A}(\mathit{true} \vdash \mathit{delay}(t_1) \land \mathit{trace}(tr') - \mathit{trace}(tr) = \langle\rangle)\ ;\ \mathbf{R_A}(\mathit{true} \vdash \mathit{delay}(t_2) \land \mathit{trace}(tr') - \mathit{trace}(tr) = \langle\rangle)$

In order to proceed, we require a specialised law for sequential composition of reactive designs. Two relevant laws for this are included in Appendix F. Law 36 is a general law for sequential composition of arbitrary *Circus Time* reactive designs and proved in [Woo13]; here, we use its specialisation given by Law 37, which applies to terminating designs only, but produces a simpler result with a *true* precondition.

$\equiv$ "application of Law 37 for reactive design composition"

$$\mathbf{R_A}\left(\mathit{true} \vdash \begin{array}{l} \mathbf{R1_A}(\mathit{delay}(t_1) \land \mathit{trace}(tr') - \mathit{trace}(tr) = \langle\rangle)\ ; \\ \mathbf{R1_A}(\mathbf{II} \lhd \mathit{wait} \rhd \mathbf{R2_A}(\mathit{delay}(t_2) \land \mathit{trace}(tr') - \mathit{trace}(tr) = \langle\rangle)) \end{array}\right)$$

$\equiv$ "unfolding definition of $\mathbf{R1_A}$ (Figure 22)"

$$\mathbf{R_A}\left(\mathit{true} \vdash \begin{array}{l} (\mathit{delay}(t_1) \land \mathit{trace}(tr') - \mathit{trace}(tr) = \langle\rangle \land tr \leq_A tr')\ ; \\ (\mathbf{II} \lhd \mathit{wait} \rhd \mathbf{R2_A}(\mathit{delay}(t_2) \land \mathit{trace}(tr') - \mathit{trace}(tr) = \langle\rangle)) \land tr \leq_A tr' \end{array}\right)$$

$\equiv$ "unfolding definition of *delay* and removal of the application of $\mathbf{R2_A}$ (Figure 22)"

$$\mathbf{R_A}\left(\mathit{true} \vdash \begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} (\mathit{wait}' \land \# tr' - \# tr < t_1) \lor \\ (\neg\ \mathit{wait}' \land \# tr' - \# tr = t_1 \land \mathit{state} = \mathit{state}') \end{array}\right) \land \\ \mathit{trace}(tr') - \mathit{trace}(tr) = \langle\rangle \land tr \leq_A tr' \end{array}\right)\ ; \\ \left(\mathbf{II} \lhd \mathit{wait} \rhd \left(\begin{array}{l} \left(\begin{array}{l} (\mathit{wait}' \land \# tr' - \# tr < t_2) \lor \\ (\neg\ \mathit{wait}' \land \# tr' - \# tr = t_2 \land \mathit{state} = \mathit{state}') \end{array}\right) \land \\ \mathit{trace}(tr') - \mathit{trace}(tr) = \langle\rangle \end{array}\right)\right) \land tr \leq_A tr' \end{array}\right)$$

The removal of the application of $\mathbf{R2_A}(\ldots)$ above is justified by a property of substitution. We first note that we can express the predicate $delay(t_2) \land trace(tr') - trace(tr) = \langle\rangle$ as a function of the trace difference $tr' -_A tr$; we omit details of this for brevity. For a predicate of the form $P(tr' -_A tr)$, it is easy to show that application of $\mathbf{R2_A}(\ldots)$ has no effect: $\mathbf{R2_A}(P(tr' -_A tr)) \equiv P((tr' -_A tr) -_A \langle\rangle) \equiv P(tr' -_A tr)$.
We next unfold the definition of $\mathbf{II}$ and sequential composition. This yields the following predicate.

$\equiv$ "unfolding definition of $\mathbf{II}$ and sequential composition (Def. 5)"

$$\mathbf{R_A}\left(true \vdash \left(\begin{array}{l} \exists\, ok'';\ wait'';\ tr'';\ state'' \bullet \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} (wait'' \land \#\, tr'' - \#\, tr < t_1) \lor \\ (\neg\, wait'' \land \#\, tr'' - \#\, tr = t_1 \land state = state'') \end{array}\right) \land \\ trace(tr'') - trace(tr) = \langle\rangle \land tr \le_A tr'' \end{array}\right) \land \\ \quad \left(\begin{array}{l} (ok' = ok'' \land wait' = wait'' \land tr' = tr'' \land state' = state'') \\ \qquad \lhd\, wait'' \rhd \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} (wait' \land \#\, tr' - \#\, tr'' < t_2) \lor \\ (\neg\, wait' \land \#\, tr' - \#\, tr'' = t_2 \land state'' = state') \end{array}\right) \land \\ trace(tr') - trace(tr'') = \langle\rangle \end{array}\right) \end{array}\right) \land \\ \quad tr'' \le_A tr' \end{array}\right)\right) \tag{4}$$

To proceed with the proof, our aim is to eliminate $wait''$ from the existential quantifier by splitting the predicate into two disjuncts: one where $wait'' = true$ and one where $wait'' = false$. For conciseness, we abbreviate the above reactive design by $\mathbf{R_A}(true \vdash \exists\, wait'' \bullet Q)$. By elementary logic, we then obtain $\mathbf{R_A}(true \vdash \exists\, wait'' \bullet Q) \equiv \mathbf{R_A}(true \vdash Q[wait'' \setminus true] \lor Q[wait'' \setminus false])$. We next provide lemmas that evaluate and simplify $Q[wait'' \setminus true]$ and $Q[wait'' \setminus false]$, respectively.

Lemma 2. Evaluation of postcondition Q where $wait'' = true$.

$Q[wait'' \setminus true] \equiv (wait' \land \#\, tr' - \#\, tr < t_1 \land trace(tr') - trace(tr) = \langle\rangle \land tr \le_A tr')$

The above lemma is proved by a few elementary steps shown below.

$Q[wait'' \setminus true]$

$\equiv$ "definition of Q and substitution of $wait''$"

$$\left(\begin{array}{l} \exists\, ok'';\ tr'';\ state'' \bullet \\ \quad (\#\, tr'' - \#\, tr < t_1 \land trace(tr'') - trace(tr) = \langle\rangle \land tr \le_A tr'') \land \\ \quad (ok' = ok'' \land wait' \land tr' = tr'' \land state' = state'') \land tr'' \le_A tr' \end{array}\right)$$

$\equiv$ "application of the one-point rule for existential quantifiers"

$(wait' \land \#\, tr' - \#\, tr < t_1 \land trace(tr') - trace(tr) = \langle\rangle \land tr \le_A tr') \land tr' \le_A tr'$

$\equiv$ "reflexivity of $\le_A$"

$(wait' \land \#\, tr' - \#\, tr < t_1 \land trace(tr') - trace(tr) = \langle\rangle \land tr \le_A tr')$ $\square$

An analogue lemma is provided next for the case where $wait''$ does not hold.

Lemma 3. Evaluation of postcondition Q where $wait'' = false$.

$Q[wait'' \setminus false] \equiv$

$$\left(\left(\begin{array}{l} (wait' \land t_1 \le \#\, tr' - \#\, tr < t_1 + t_2) \lor \\ (\neg\, wait' \land \#\, tr' - \#\, tr = t_1 + t_2 \land state = state') \end{array}\right) \land trace(tr') - trace(tr) = \langle\rangle \land tr \le_A tr'\right)$$

The proof is slightly more elaborate here and presented in the sequel.

$Q[wait'' \setminus false]$

$\equiv$ "definition of Q and substitution of $wait''$"

$$\left(\begin{array}{l} \exists\, ok''; \ tr''; \ state'' \bullet \\ \quad (\#\, tr'' - \#\, tr = t_1 \land state = state'' \land trace(tr'') - trace(tr) = \langle\rangle \land tr \leq_A tr'') \land \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} (wait' \land \#\, tr' - \#\, tr'' < t_2) \lor \\ (\neg\, wait' \land \#\, tr' - \#\, tr'' = t_2 \land state'' = state') \end{array}\right) \land \\ trace(tr') - trace(tr'') = \langle\rangle \land tr'' \leq_A tr' \end{array}\right) \end{array}\right)$$

$\equiv$ "application of the one-point rule and removing the unused ok''"

$$\left(\begin{array}{l} \exists\, tr'' \bullet \\ \quad (\#\, tr'' - \#\, tr = t_1 \land trace(tr'') - trace(tr) = \langle\rangle \land tr \leq_A tr'') \land \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} (wait' \land \#\, tr' - \#\, tr'' < t_2) \lor \\ (\neg\, wait' \land \#\, tr' - \#\, tr'' = t_2 \land state = state') \end{array}\right) \land \\ trace(tr') - trace(tr'') = \langle\rangle \land tr'' \leq_A tr' \end{array}\right) \end{array}\right)$$

$\equiv$ "reordering of conjuncts for readability"

$$\left(\begin{array}{l} \exists\, tr'' \bullet (tr \leq_A tr'' \land tr'' \leq_A tr') \land \\ (trace(tr'') - trace(tr) = \langle\rangle \land trace(tr') - trace(tr'') = \langle\rangle) \land \\ (\#\, tr'' - \#\, tr = t_1) \land \\ \left(\begin{array}{l} (wait' \land \#\, tr' - \#\, tr'' < t_2) \lor \\ (\neg\, wait' \land \#\, tr' - \#\, tr'' = t_2 \land state = state') \end{array}\right) \end{array}\right)$$

We next remove the existential quantification over tr''. Here, however, this is not so easy as we cannot apply the one-point rule. Instead, we use the trivial law $(\exists\, x \bullet P(x)) \equiv Q$ for some Q that does not mention x, and where we can show that $Q \Leftrightarrow (\exists\, x \bullet P(x))$. The most difficult part of this step is to find the correct Q as well as the witness x in proving the forward implication. We omit the details of finding Q and proving the proviso, but just present the result of applying this deduction.

$\equiv$ "removing existential quantification $\exists\, tr' \bullet P(tr')$"

$$\left(\begin{array}{l} (wait' \land t_1 \leq \#\, tr' - \#\, tr < t_1 + t_2) \lor \\ (\neg\, wait' \land \#\, tr' - \#\, tr = t_1 + t_2 \land state = state') \end{array}\right) \land trace(tr') - trace(tr) = \langle\rangle \land tr \leq_A tr'$$

We next put the two disjuncts resulting from the Lemmas 2 and 3 together to obtain the overall result of the elimination of $wait''$ from (4), and continue the proof of the law from there onwards.

(4) $\equiv$ "splitting existential quantifier over $wait''$ via Lemma 2 and Lemma 3"

$$\mathbf{R_A}\left(true \vdash \left(\begin{array}{l} (wait' \land \#\, tr' - \#\, tr < t_1 \land trace(tr') - trace(tr) = \langle\rangle \land tr \leq_A tr') \\ \quad \lor \\ \left(\begin{array}{l} \left(\begin{array}{l} (wait' \land t_1 \leq \#\, tr' - \#\, tr < t_1 + t_2) \lor \\ (\neg\, wait' \land \#\, tr' - \#\, tr = t_1 + t_2 \land state = state') \end{array}\right) \land \\ trace(tr') - trace(tr) = \langle\rangle \land tr \leq_A tr' \end{array}\right) \end{array}\right)\right)$$

$\equiv$ "application of distributivity laws to merge disjuncts"

$$\mathbf{R_A}\left(true \vdash \left(\begin{array}{l} \left(\begin{array}{l} (wait' \land \#\, tr' - \#\, tr < t_1 + t_2) \lor \\ (\neg\, wait' \land \#\, tr' - \#\, tr = t_1 + t_2 \land state = state') \end{array}\right) \land \\ trace(tr') - trace(tr) = \langle\rangle \land tr \leq_A tr' \end{array}\right)\right)$$

$\equiv$ "exploiting the assumption $tr \leq_A tr'$ in A; this holds due to application of $\mathbf{R1}_A$ via $\mathbf{R}_A$"

$$\mathbf{R_A}\left(true \vdash \left(\begin{array}{l} (wait' \land \#\, tr' - \#\, tr < t_1 + t_2) \lor \\ (\neg\, wait' \land \#\, tr' - \#\, tr = t_1 + t_2 \land state = state') \end{array}\right) \land trace(tr') - trace(tr) = \langle\rangle \right)$$

$\equiv$ "folding definition of *delay* and **wait** (Def. 6)"

wait $t_1 + t_2$ $\square$

With Law 26 being proved, we finally tackle the proof of the budget splitting law (Law 15) that our initial goal was to verify. Starting from the left-hand side $\mathbf{wait}\, 0 \mathrel{..} t$ of the law, we obtain

$\mathbf{wait}\, 0 \mathrel{..} t$

$\equiv$ "unfolding definition of time budget, nondeterminism, and **wait** (Def. 6 and Def. 7)"

$\exists\, d : 0 \mathrel{..} t \bullet \mathbf{R_A}(true \vdash delay(d) \land trace' = \langle\rangle)$

$\equiv$ "lemma: $(\exists\, d : 0 \mathrel{..} t \bullet P(d)) \Leftrightarrow (\exists\, d_1 : 0 \mathrel{..} t_1 \bullet \exists\, d_2 : 0 \mathrel{..} t_2 \bullet P(d_1 + d_2))$"

$\exists\, d_1 : 0 \mathrel{..} t_1 \bullet \exists\, d_2 : 0 \mathrel{..} t_2 \bullet \mathbf{R_A}(true \vdash delay(d_1 + d_2) \land trace' = \langle\rangle)$

$\equiv$ "folding definition of **wait** (Def. 6)"

$\exists\, d_1 : 0 \mathrel{..} t_1 \bullet \exists\, d_2 : 0 \mathrel{..} t_2 \bullet \mathbf{wait}\; t_1 + t_2$

$\equiv$ "application of the Law 26 for splitting a simple delay"

$\exists\, d_1 : 0 \mathrel{..} t_1 \bullet \exists\, d_2 : 0 \mathrel{..} t_2 \bullet \mathbf{wait}\; d_1 \;;\; \mathbf{wait}\; d_2$

$\equiv$ "distribution law: $(\exists\, x \bullet A_1 \;;\; A_2) \Leftrightarrow A_1 \;;\; (\exists\, x \bullet A_2)$ if A_1 does not reference x"

$\exists\, d_1 : 0 \mathrel{..} t_1 \bullet \mathbf{wait}\; d_1 \;;\; (\exists\, d_2 : 0 \mathrel{..} t_2 \bullet \mathbf{wait}\; d_2)$

$\equiv$ "distribution law: $(\exists\, x \bullet A_1 \;;\; A_2) \Leftrightarrow (\exists\, x \bullet A_1) \;;\; A_2$ if A_2 does not reference x"

$(\exists\, d_1 : 0 \mathrel{..} t_1 \bullet \mathbf{wait}\; d_1) \;;\; (\exists\, d_2 : 0 \mathrel{..} t_2 \bullet \mathbf{wait}\; d_2)$

$\equiv$ "folding definition of generalised nondeterminism and time budget (Def. 7)"

$\mathbf{wait}\, 0 \mathrel{..} t_1 \;;\; \mathbf{wait}\, 0 \mathrel{..} t_2 \quad \Box$

The two proofs we discussed in this section are primarily meant to illustrate the possibility and the general approach to proving the laws that we presented earlier on. Additional examples that illustrate the use of algebraic strategies are available in [ZCW+12].

8. 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. 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 and sharing laws 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 introduction of cycle timings and 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 and encapsulation of shared data; this ultimately creates a positive outlook.

Like in SCJ, our model and strategy supports data sharing between missions, and novel refinement laws have been presented that encapsulate shared data to refine communication patterns while accounting for sequential data flow, parallel computations, and control mechanisms by virtue of SCJ events. The soundness of the laws guarantees the absence of race conditions in the emerging SCJ program model.

Though we have focused on handler architectures, the mission design in fact emerges where sequential actions of an abstract centralised model are retained during refinement. In terms of sharing, sequential composition is not an issue. Accordingly, data shared between missions is kept as state components of the process in Figure 5 that defines the refinement target. Data shared between handlers must, however, be encapsulated and accessed through communications, as *Circus* parallel composition prohibits data sharing as to retain monotonicity of that operator with respect to refinement, which is crucial for compositionality.

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 ongoing work, however, their mechanisation may use a tool like [ZOC12] interacting with a prover to ensure soundness of refinements and laws alike.

In terms of the laws, it is still an open issue how the application of more specialised laws like Law 22, Law 25 and Law 35 (in Appendix A) can be automated. There may again be scope for using heuristics and tactics, but more experience needs to be gained to ascertain this. Furthermore, we observe that the decomposition (Section 4.2), parallelisation (Section 4.4) and sharing (Section 4.5) laws are defined so that they can be applied in succession: each law for a later stage targets the result of the application of another law from an earlier stage. An interesting opportunity is to consider the fusion of matching laws. While this offers the potential to increase automation by reducing the number of law applications, it has the downside of reducing modularity and thereby the design space of realisable SCJ program designs. We thus observe a trade-off and delicate balance to be struck between flexibility of the approach and ease of its use. A lesson learned here is that parallel control fragments can provide genericity in laws as they enable us to postpone certain aspects of the refinement and support the definition of laws whose application requires less context.

For validation, we have presented the proofs for two key *Circus Time* laws. In [ZCW$^+$12], we moreover sketch a proof of Law 22 which uses a few novel and interesting elementary laws. That proof, however, uses existing *Circus* laws rather than the UTP-based semantics of *Circus Time* we recaptured in Section 7.2. We note that standard *Circus* laws like those in [CSW03, CCO11] remain valid in *Circus Time*.

Related work includes action systems and their refinement [Bac89, BKS83]. Action systems combine state and behaviour by way of atomic guarded 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 [Bac89, BvW03]. The execution model is typically *a priori* fixed, nondeterministically choosing an action whose guard is enabled, and performing the respective state update. Expressivity is constrained by the fact that any form of synchronisation has to be achieved through guards and the only way of communicating data is via shared variables. While the semantics of the guarded command language is simpler than that of *Circus* and CSP, it is not obvious how the mission-based execution paradigm can be expressed in terms of (one of) the common execution models for action systems.

Event-B [Abr10] 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 [But09] and time [CMR06]. Fundamentally, however, the same restrictions as for action systems apply: that is the lack of synchronisation and communication primitives. Some effort has been made to combine B with CSP to reap the benefits of both worlds [ST05, STW10]. It would thus be interesting to examine whether Event-B and its combination with CSP are indeed expressive enough for SCJ handler models, and whether the refinement laws we propose can be formulated and validated in that setting.

To overcome the issue of complexity of our refinement strategy for SCJ, a first important future work is to develop a semi-automatic refinement tool to facilitate the application of that strategy. This is to make the strategy amenable to use by engineers without expert knowledge in Z or *Circus*. The tool will provide an extendible collection of laws and tactics for each anchor, and, in particular, adopt the ideas for automation discussed earlier on in Section 6. Rather than applying laws one by one, we envisage that the developer will be able to select high-level patterns to decompose an abstract operation or orchestrate handler execution. Such patterns will be verified independently outside the tool, and the integration with a theorem prover based on the LCF principle is envisaged to guarantee that all laws and patterns are sound.

An automatic translator from SCJ Level 1 into the P model (Figure 4) [ZLCW13] is already available to pave the way for the verification of existing programs, and we are currently addressing translation into the opposite direction to cater for program development by virtue of the laws. The translation between the P model and S anchor is pending work but technically not as challenging; we are also looking at this issue.

Further work is required to integrate the semantics of *Circus Time* 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 UTP being 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.

SCJ is still a very new 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 use of object references, SCJ libraries, 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 We first would like to thank the anonymous reviewers for their careful reading and many useful suggestions. We are also grateful to Andy Wellings for helping us to clarify numerous subtleties surrounding SCJ. This work was funded by the EPSRC grant EP/H017461/1.

References

[Abr10] J. R. Abrial. *Modeling in Event-B.* Cambridge University Press, Cambridge, CB2 8BS, UK, May 2010.

[AH07] J.-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, May 2007.

[Bac89] R. J. R. Back. Refinement Calculus, Part II: Parallel and Reactive Programs. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 67–93. Springer, May 1989.

[BKS83] R. J. R. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *Proceedings of PODC '83, Second ACM Symposium on Principles of Distributed Computing*, pages 131–142. ACM, August 1983.

[Bur99] A. Burns. The Ravenscar Profile. *ACM SIGAda Ada Letters*, XIX(4):49–52, 1999.

[But09] M. Butler. Decomposition Structures for Event-B. In *Proceedings of IFM 2009, Integrated Formal Methods*, volume 5423 of *LNCS*, pages 20–38. Springer, February 2009.

[BvW03] R. J. R. Back and J. von Wright. Compositional Action System Refinement. *Formal Aspects of Computing*, 15(2-3):103–117, November 2003.

[Cav97] A. Cavalcanti. *A Refinement Calculus for Z.* PhD thesis, University of Oxford, Oxford, OX1 3QD, UK, 1997.

[CCO11] A. Cavalcanti, P. Clayton, and C. O'Halloran. From control law diagrams to Ada via Circus. *Formal Aspects of Computing*, 23(4):465–512, July 2011.

[CMR06] D. Cansell, D. Méry, and J. Rehm. Time Constraint Patterns for Event B Development. In *Proceedings of B 2007: Formal Specification and Development in B*, volume 4355 of *LNCS*, pages 140–154. Springer, January 2006.

[CSW03] A. Cavalcanti, A. Sampaio, and J. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2-3):146–181, November 2003.

[CSW05] A. Cavalcanti, A. Sampaio, and J. Woodcock. Unifying classes and processes. *Software and Systems Modeling*, 4(3):277–296, July 2005.

[CW98] A. Cavalcanti and J. Woodcock. ZRC — A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267–289, March 1998.

[CWW11a] A. Cavalcanti, A. Wellings, and J. Woodcock. The Safety-Critical Java Memory Model: A Formal Account. In *Proceedings of FM 2011: Formal Methods*, volume 6664 of *LNCS*, pages 246–261. Springer, June 2011.

[CWW$^+$11b] A. Cavalcanti, A. Wellings, J. Woodcock, K. Wei, and F. Zeyda. Safety-Critical Java in *Circus*. In *Proceedings of JTRES 2011, 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 20–29. ACM, September 2011.

[CZW$^+$13] A. Cavalcanti, F. Zeyda, A. Wellings, J. Woodcock, and K. Wei. Safety-Critical Java programs from *Circus* models. *Real-time Systems*, 49:614–667, September 2013.

[DHS12] A. E. Dalsgaard, R. R. Hansen, and M. Schoeberl. Private Memory Allocation Analysis for Safety-Critical Java. In *Proceedings of JTRES 2012, 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 9–17. ACM, 2012.

[Gro02] L. Groves. Refinement and the Z Schema Calculus. In *Proceedings of REFINE 2002: The BCS FACS Refinement Workshop, ENTCS*, volume 70(3), pages 70–93, November 2002.

[HHL$^+$09] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for Safety-Critical Applications. In *Proceedings of SafeCert 2009*, pages 1–11, March 2009.

[HJ98] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming.* Prentice Hall Series in Computer Science. Prentice Hall, Upper Saddle River, NJ, USA, 1998.

[HL11] G. Haddad and G. T. Leavens. Specifying Subtypes in SCJ Programs. In *Proceedings of JTRES 2011, 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 40–46. ACM, 2011.

[HU01] I. J. Hayes and M. Utting. A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385–448, 2001.

[KHP$^+$09] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CD_x: A Family of Real-time Java Benchmarks. In *Proceedings of JTRES 2009, 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 41–50. ACM, September 2009.

[Mor94] C. C. Morgan. *Programming from Specifications.* Prentice Hall International Series in Computer Science. Prentice Hall, Upper Saddle River, NJ, USA, January 1994.

[Nor03] M. Norrish. Complete Integer Decision Procedures as Derived Rules in HOL. In *Proceedings of TPHOLs 2003, Theorem Proving in Higher Order Logics*, volume 2758 of *LNCS*, pages 71–86. Springer, September 2003.

[OCW09] M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for *Circus*. *Formal Aspects of Computing*, 21(1-2):3–32, February 2009.

[Oli05] M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus.* PhD thesis, University of York, York, YO10 5GH, UK, 2005.

[OZC11] M. Oliveira, F. Zeyda, and A. Cavalcanti. A tactic language for refinement of state-rich concurrent specifications. *Science of Computer Programming*, 76(9):792–833, September 2011.

[PFHV04] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time Java Scoped Memory: Design Patterns and Semantics. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 101–110. IEEE, May 2004.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science. Prentice Hall, Upper Saddle River, NJ, USA, November 1997.

[Ros11] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.

[RR88] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1-3):249–261, June 1988.

[RTC11] RTCA/EUROCAE joint committee. Software Considerations in Airborne Systems and Equipment Certification. Technical Report DO-178C, RTCA Inc., Washington, DC, USA, December 2011.

[SCJS10] A. Sherif, A. Cavalcanti, H. Jifeng, and A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, March 2010.

[ST05] S. Schneider and H. Treharne. CSP theorems for communicating B machines. *Formal Aspects of Computing*, 17(4):390–422, December 2005.

[STR06] H. Søndergaard, B. Thomsen, and A. P. Ravn. A Ravenscar-Java Profile Implementation. In *Proceedings of JTRES 2006, 4th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 38–47. ACM, 2006.

[STW10] S. Schneider, H. Treharne, and H. Wehrheim. A CSP Approach to Control in Event-B. In *Proceedings of IFM 2010, Integrated Formal Methods*, volume 6396 of *LNCS*, pages 260–274. Springer, October 2010.

[The11] The Open Group. Safety Critical Java Technology Specification — Version 0.94. Technical Report JSR-302, Java Community Process, January 2011. Available from http://jcp.org/en/jsr/detail?id=302.

[TPV10] D. Tang, A. Plsek, and J. Vitek. Static Checking of Safety Critical Java Annotations. In *Proceedings of JTRES 2010, 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 148–154. ACM, August 2010.

[W$^+$08] R. Wilhelm et al. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, April 2008.

[WD96] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, Upper Saddle River, NJ, USA, July 1996.

[Wel04] A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, West Sussex, PO19 8SQ, UK, 2004.

[Woo13] J. Woodcock. CML definition 4. Technical Report COMPASS Deliverable 23.5, Seventh Framework Programme: Comprehensive Modelling for Advanced Systems of Systems, Grant Agreement 287829, 2013. Available from http://www.compass-research.eu/deliverables.html.

[ZC13] F. Zeyda and A. Cavalcanti. Refining SCJ Mission Specifications into Parallel Handler Designs. In *Proceedings of REFINE 2013: 16th BCS FACS Refinement Workshop, EPTCS*, volume 115, pages 52–67, May 2013.

[ZCW11] F. Zeyda, A. Cavalcanti, and A. Wellings. The Safety-Critical Java Mission Model: A Formal Account. In *Proceedings of ICFEM 2011, 13th International Conference on Formal Engineering Methods*, volume 6991 of *LNCS*, pages 49–65. Springer, October 2011.

[ZCW$^+$12] F. Zeyda, A. Cavalcanti, A. Wellings, J. Woodcock, and K. Wei. Refinement of the Parallel CD_x. Technical report, University of York, York, YO10 5GH, UK, July 2012. Available from http://www.cs.york.ac.uk/circus/publications/techreports/.

[ZLCW13] F. Zeyda, L. Lalkhumsanga, A. Cavalcanti, and A. Wellings. *Circus* Models for Safety-Critical Java Programs. *The Computer Journal*, 57(7):1046–1091, July 2013.

[ZOC12] F. Zeyda, M. Oliveira, and A. Cavalcanti. Mechanised support for sound refinement tactics. *Formal Aspects of Computing*, 24(1):127–160, January 2012.

A. Supplementary Refinement Laws

Law 27. $(c \longrightarrow A_1 \,;\, A_2) \blacktriangleleft d \;\equiv\; (c \longrightarrow A_1) \blacktriangleleft d \,;\, A_2$

Law 28. $(Op \,;\, A_2) \blacktriangleleft d \;\equiv\; Op \,;\, (A_2 \blacktriangleleft d)$

Law 29. $(\mathbf{wait}\, t \,;\, A_2) \blacktriangleleft d \;\equiv\; \mathbf{wait}\, t \,;\, (A_2 \blacktriangleleft d - t)$ provided $t \leq d$

Law 30. $(A_1 \sqcap A_2) \blacktriangleleft d \;\equiv\; (A_1 \blacktriangleleft d) \sqcap (A_2 \blacktriangleleft d)$

Law 31. $(A_1 \Box A_2) \blacktriangleleft d \;\equiv\; (A_1 \blacktriangleleft d) \Box (A_2 \blacktriangleleft d)$

Law 32. $(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) \blacktriangleleft d \;\equiv\; (A_1 \blacktriangleleft d) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$ provided $\mathrm{usedC}(A_2) = \varnothing$

Law 33. $(A \blacktriangleleft d_1) \blacktriangleleft d_2 \;\equiv\; A \blacktriangleleft \min(d_1, d_2)$

Law 34. $(A \blacktriangleright d_1) \blacktriangleleft d_2 \;\equiv\; (A \blacktriangleleft d_2) \blacktriangleright d_1$

Fig. 24. Distribution laws for synchronisation deadlines.

Law 35. Specialised sharing law for unidirectional communication between sequential handlers.

$$\left(\begin{array}{l} (\mu X \bullet A_1 \,;\, c\,!\,x \longrightarrow \mathbf{skip} \,;\, lockstep \longrightarrow X) \\ \quad \llbracket ns_1 \mid cs \mid ns_2 \rrbracket \\ (\mu X \bullet c\,?\,x \longrightarrow A_2(x) \,;\, lockstep \longrightarrow X) \end{array}\right) \setminus \{\!|\, c \,|\!\}$$

$$\equiv$$

$$\left(\begin{array}{l} \left(\begin{array}{l} (\mu X \bullet A_1 \,;\, c_{write}\,!\,x \longrightarrow \mathbf{skip} \,;\, c_{pivot} \longrightarrow \mathbf{skip} \,;\, lockstep \longrightarrow X) \\ \quad \llbracket ns_1 \mid (cs - \{\!|\, c \,|\!\}) \cup \{\!|\, c_{pivot} \,|\!\} \mid ns_2 \rrbracket \\ (\mu X \bullet c_{pivot} \longrightarrow \mathbf{skip} \,;\, c_{read}\,?\,x \longrightarrow A_2(x) \,;\, lockstep \longrightarrow X) \end{array}\right) \setminus \{\!|\, c_{pivot} \,|\!\} \\ \llbracket ns_1 \cup ns_2 \mid \{\!|\, c_{read}, c_{write} \,|\!\} \mid \varnothing \rrbracket \\ \left(\begin{array}{l} \mathbf{var}\, v : T \bullet \\ \quad \mu X \bullet \left(\begin{array}{l} (c_{read}\,!\,v \longrightarrow \mathbf{skip}) \,\Box \\ (c_{write}\,?\,x \longrightarrow v := x) \end{array}\right) ; \; X \end{array}\right) \end{array}\right) \setminus \{\!|\, c_{read}, c_{write} \,|\!\}$$

provided $\{\!|\, c, lockstep \,|\!\} \subseteq cs$ and $\{\!|\, c, lockstep \,|\!\} \cap (\mathrm{usedC}(A_1) \cup \mathrm{usedC}(A_2)) = \varnothing$ and c_{read}, c_{write} and c_{pivot} are fresh channels.

Fig. 25. Specialised sharing law for unidirectional communication between sequential handlers.

B. Evaluation of *TakesAtMost* in the CD_x example

$$TakesAtMost \left(\begin{array}{l} (next\_frame\,?\,frame \longrightarrow RecordFrame) \blacktriangleleft INP\_DL; \\ \mathbf{wait}\, 0 \,..\, FRAME\_PERIOD - OUT\_DL - INP\_DL; \\ \mathbf{var}\, colls : \mathbb{N} \bullet CalcCollisions; \\ (output\_collisions\,!\,colls \longrightarrow \mathbf{skip}) \blacktriangleleft OUT\_DL \end{array}\right)$$

$=$ "definition of $TakesAtMost(A_1 \,;\, A_2)$ in Figure 6"

$$\left(\begin{array}{l} TakesAtMost((next\_frame\,?\,frame \longrightarrow RecordFrame) \blacktriangleleft INP\_DL) + \\ TakesAtMost(\mathbf{wait}\, 0 \,..\, FRAME\_PERIOD - OUT\_DL - INP\_DL) + \\ TakesAtMost(\mathbf{var}\, colls : \mathbb{N} \bullet CalcCollisions) + \\ TakesAtMost((output\_collisions\,!\,colls \longrightarrow \mathbf{skip}) \blacktriangleleft OUT\_DL) \end{array}\right)$$

$= $ “definition of $\mathit{TakesAtMost}((c \longrightarrow A) \blacktriangleleft d)$ in Figure 6”

$$\left(\begin{array}{l} \mathit{TakesAtMost}(\mathit{RecordFrame}) + \mathit{INP\_DL} + \\ \mathit{TakesAtMost}(\mathbf{wait}\ 0 \mathinner{..} \mathit{FRAME\_PERIOD} - \mathit{OUT\_DL} - \mathit{INP\_DL}) + \\ \mathit{TakesAtMost}(\mathbf{var}\ \mathit{colls} : \mathbb{N} \bullet \mathit{CalcCollisions}) + \\ \mathit{TakesAtMost}(\mathbf{skip}) + \mathit{OUT\_DL} \end{array}\right)$$

$= $ “definition of $\mathit{TakesAtMost}(\mathbf{wait}\ t_1 \mathinner{..} t_2)$ in Figure 6”

$$\left(\begin{array}{l} \mathit{TakesAtMost}(\mathit{RecordFrame}) + \mathit{INP\_DL} + \\ \mathit{FRAME\_PERIOD} - \mathit{OUT\_DL} - \mathit{INP\_DL} + \\ \mathit{TakesAtMost}(\mathbf{var}\ \mathit{colls} : \mathbb{N} \bullet \mathit{CalcCollisions}) + \\ \mathit{TakesAtMost}(\mathbf{skip}) + \mathit{OUT\_DL} \end{array}\right)$$

$= $ “definition of $\mathit{TakesAtMost}(\mathbf{var}\ v : T \bullet A)$ in Figure 6”

$$\left(\begin{array}{l} \mathit{TakesAtMost}(\mathit{RecordFrame}) + \mathit{INP\_DL} + \\ \mathit{FRAME\_PERIOD} - \mathit{OUT\_DL} - \mathit{INP\_DL} + \\ \mathit{TakesAtMost}(\mathit{CalcCollisions}) + \\ \mathit{TakesAtMost}(\mathbf{skip}) + \mathit{OUT\_DL} \end{array}\right)$$

$= $ “definition of $\mathit{TakesAtMost}(\mathbf{skip})$ and $\mathit{TakesAtMost}(\mathit{Op})$ in Figure 6”

$$0 + \mathit{INP\_DL} + \mathit{FRAME\_PERIOD} - \mathit{OUT\_DL} - \mathit{INP\_DL} + 0 + 0 + \mathit{OUT\_DL}$$

$= $ “arithmetic simplification”

$\mathit{FRAME\_PERIOD}$ $\square$

C. Class definition for the *Parition* class

The *Partition* class is used in the CD_x program to record partitions of the voxel space that define the computational work for the parallel detector handlers. The aircraft in one voxel are encoded by a *List* of *Motion* objects. The class constructor (**initial** paragraph) receives the number of work partitions.

$\mathbf{class}\ \mathit{Partition} \mathrel{\widehat=} \mathbf{begin}$

$\mathbf{state}\ \mathit{PartitionState}$

$\mathbf{private}\ \mathit{parts} : \mathit{Array}[\mathit{List}[\mathit{VoxelMotions}]]$ (where $\mathit{VoxelMotions}$ abbreviates $\mathit{List}[\mathit{Motion}]$)

$\mathbf{private}\ \mathit{counter} : \mathit{int}$

$\mathit{parts} \neq \mathbf{null} \land 0 \leq \mathit{counter} < \mathit{parts.length}$

$$\begin{array}{l}
\mathbf{initial}\ \mathit{Init} \mathrel{\widehat=} \mathbf{val}\ n : \mathit{int} \bullet \\
\quad \left(\begin{array}{l} \mathit{parts} := \mathbf{newM}\ \mathit{Array}[\mathit{List}[\mathit{VoxelMotions}]](n); \\ \left(\begin{array}{l} \mathbf{for}\ \mathit{index} = 0\ \mathbf{to}\ \mathit{parts.length} - 1 \bullet \\ \mathit{parts.setArray}(\mathit{index}, \mathbf{newM}\ \mathit{LinkedList}[\mathit{Motion}]()) \end{array}\right); \\ \mathit{counter} := 0 \end{array}\right) \\
\mathbf{public\ sync}\ \mathit{clear} \mathrel{\widehat=} \\
\quad \left(\begin{array}{l} \left(\begin{array}{l} \mathbf{for}\ \mathit{index} = 0\ \mathbf{to}\ \mathit{parts.length} - 1 \bullet \\ \mathit{parts.clear}()) \end{array}\right); \\ \mathit{counter} := 0 \end{array}\right) \\
\mathbf{public\ sync}\ \mathit{recordVoxelMotions}(\mathit{motions} : \mathit{VoxelMotions}]) \mathrel{\widehat=} \\
\quad \left(\begin{array}{l} \mathit{parts.getArray}(\mathit{counter}).\mathit{add}(\mathit{motions}); \\ \mathit{counter} := (\mathit{counter} + 1) \bmod \mathit{parts.length} \end{array}\right) \\
\mathbf{public\ sync}\ \mathit{getDetectorWork} \mathrel{\widehat=} \mathbf{val}\ \mathit{detector} : \mathit{int};\ \mathbf{res}\ \mathit{ret} : \mathit{List}[\mathit{VoxelMotions}] \bullet \\
\quad \mathit{ret} := \mathit{parts.getArray}(\mathit{detector} - 1) \\
\mathbf{end}
\end{array}$$

D. Decomposed data operations of the CD_x example

Z data operation	**Handle class in the SCJ program**	**Description**
StoreFrame	`InputFrameHandler`	read frame and calculate motions
PartitionWork	`ReducerHandler`	voxel-hashing and partitioning voxels
CalcPartCollisions	`DetectorHandler` (4 instances)	compute collisions for each voxel partition
SetCollisionsFromParts	`OutputCollisionsHandler`	obtain and output collisions result

Table 1. Mapping between Z operations and handlers in the verified CD_x program design.

StoreFrame

$$\begin{array}{l}
\Delta[\mathit{currentFrame} : \mathit{RawFrame};\ \mathit{state} : \mathit{StateTable};\ \mathit{work} : \mathit{Partition};\ \mathit{collisions} : \mathbb{Z}] \\
\mathit{frame}? : \mathit{Frame} \\
\hline
\exists\, \mathit{posns}, \mathit{posns}' : \mathit{Frame};\ \mathit{motions}, \mathit{motions}' : \mathit{Frame} \mid \\
\quad \mathrm{dom}\ \mathit{posns} = \mathrm{dom}\ \mathit{motions} \land \mathrm{dom}\ \mathit{posns}' = \mathrm{dom}\ \mathit{motions}' \bullet \\
\left(\begin{array}{l}
\mathit{posns}' = \mathit{frame}? \land \\
\mathit{motions}' = (\lambda\, a : \mathrm{dom}\ \mathit{posns}' \bullet \mathbf{if}\ a \in \mathrm{dom}\ \mathit{posns}\ \mathbf{then}\ (\mathit{posns}'\ a) -_V (\mathit{posns}\ a)\ \mathbf{else}\ \mathit{ZeroV}) \land \\
\mathit{posns} = F(\mathit{currentFrame}) \land \mathit{motions} = G(\mathit{currentFrame}, \mathit{state}) \land \\
\mathit{posns}' = F(\mathit{currentFrame}') \land \mathit{motions}' = G(\mathit{currentFrame}', \mathit{state}')
\end{array}\right)
\end{array}$$

PartitionWork

$$\begin{array}{l}
\Delta[\mathit{currentFrame} : \mathit{RawFrame};\ \mathit{state} : \mathit{StateTable};\ \mathit{work} : \mathit{Partition};\ \mathit{collisions} : \mathbb{Z}] \\
\hline
\mathit{currentFrame}' = \mathit{currentFrame} \land \mathit{state}' = \mathit{state} \\
\exists\, \mathit{posns} : \mathit{Frame};\ \mathit{motions} : \mathit{Frame} \mid \mathrm{dom}\, \mathit{posns} = \mathrm{dom}\, \mathit{motions} \bullet \\
\left(\begin{array}{l}
\mathit{posns} = F(\mathit{currentFrame}) \land \mathit{motions} = G(\mathit{currentFrame}, \mathit{state}) \land \\
\exists\, \mathit{voxel\_map} : \mathit{HashMap}[\mathit{Vector2d}, \mathit{List}[\mathit{Motion}]] \mid \mathit{voxel\_map} \neq \mathbf{null} \bullet \\
\left(\begin{array}{l}
\forall\, a_1, a_2 : \mathit{Aircraft} \mid \{a_1, a_2\} \subseteq \mathrm{dom}\, \mathit{posns} \bullet \\
(a_1, a_2) \in \mathit{CollSet}(\mathit{posns}, \mathit{motions}) \Rightarrow \\
\qquad \left(\begin{array}{l}
\exists\, l : \mathit{List}[\mathit{Motion}] \mid l \in \mathit{voxel\_map}'.\mathit{values}().\mathit{elems}() \bullet \\
\mathit{MkMotion}(a_1, \mathit{posns}\ a_1 -_V \mathit{motions}\ a_1, \mathit{posns}\ a_1) \in l.\mathit{elems}() \land \\
\mathit{MkMotion}(a_2, \mathit{posns}\ a_2 -_V \mathit{motions}\ a_2, \mathit{posns}\ a_2) \in l.\mathit{elems}()
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}$$

CalcPartCollisions

$$\begin{array}{l}
\Xi[\mathit{currentFrame} : \mathit{RawFrame};\ \mathit{state} : \mathit{StateTable};\ \mathit{work} : \mathit{Partition};\ \mathit{collisions} : \mathbb{Z}] \\
i? : 1 \mathrel{..} 4 \\
\mathit{pcolls}! : \mathbb{Z} \\
\hline
\mathit{pcolls}! = \# \left\{\begin{array}{l}
a_1 : \mathit{Aircraft};\ a_2 : \mathit{Aircraft} \mid \exists\, l : \mathit{work}.\mathit{getDetectorWork}(i?).\mathit{elems}() \bullet \\
\left(\begin{array}{l}
\exists\, v_1, v_2 : \mathit{Vector};\ w_1, w_2 : \mathit{Vector} \bullet \\
\qquad \mathit{MkMotion}(a_1, v_1, w_1) \in l.\mathit{elems}\ () \land \\
\qquad \mathit{MkMotion}(a_2, v_2, w_2) \in l.\mathit{elems}\ () \land \\
\qquad \mathit{collide}((v_1, w_1 -_V v_1), (v_2, w_2 -_V v_2))
\end{array}\right)
\end{array}\right\} \mathrm{div}\ 2
\end{array}$$

SetCollisionsFromParts

$$\begin{array}{l}
\Delta[\mathit{currentFrame} : \mathit{RawFrame};\ \mathit{state} : \mathit{StateTable};\ \mathit{work} : \mathit{Partition};\ \mathit{collisions} : \mathbb{Z}] \\
\mathit{collsbag}? : \mathrm{bag}\ \mathit{int} \\
\hline
\mathit{currentFrame}' = \mathit{currentFrame} \land \mathit{state}' = \mathit{state} \land \mathit{work}' = \mathit{work} \\
\exists\, s : \mathrm{seq}\ \mathit{int} \mid \mathit{collsbag}? = \mathit{items}\ s \bullet \mathit{collisions}' = \Sigma_{\mathrm{seq}}\ s \quad (\Sigma_{\mathrm{seq}} \text{ yields the sum of sequence elements})
\end{array}$$

E. *Circus Time* UTP model

Def 8. Definition of prefix and subtraction for time traces in *Circus Time*.

$$tr_1 \leq_A tr_2 \mathrel{\widehat=} front(tr_1) < tr_2 \wedge last(tr_1).1 \leq head(tr_2 - front(tr_1)).1$$

$$tr_1 -_A tr_2 \mathrel{\widehat=} \langle (head(t_2 - front(tr_1)).1 - last(tr_1).1, head(tr_2 - front(tr_1)).2) \rangle \frown tail(tr_2 - front(tr_1))$$

where $\leq$ and $<$ are the standard prefix operators on sequences, and $-$ is (standard) sequence subtraction. The notations $c.1$ and $c.2$ are used for pair selection, and *head*, *tail*, *front* and *last* have their usual meanings.

F. *Circus Time* UTP Laws

Law 36. Law for sequential composition of reactive designs in *Circus Time*.

$$\mathbf{R_A}(P_1 \vdash Q_1) \,;\, \mathbf{R_A}(P_2 \vdash Q_2) \equiv$$

$$\mathbf{R_A}\left(\begin{array}{l} \neg\,(\mathbf{R1_A}(\neg\, P_1) \,;\, \mathbf{R1_A}(true)) \wedge \neg\,(\mathbf{R1_A}(Q_1) \,;\, \mathbf{R1_A}(\neg\, wait \wedge \mathbf{R2_A}(\neg\, P_2))) \\ \vdash \\ \mathbf{R1_A}(Q_1) \,;\, \mathbf{R1_A}(\mathbf{I\!I} \lhd wait \rhd \mathbf{R2_A}(Q_2)) \end{array}\right)$$

Fig. 26. Sequential composition of reactive designs in *Circus Time*.

Law 37. Law for sequential composition of terminating reactive designs in *Circus Time*.

$$\mathbf{R_A}(true \vdash Q_1) \,;\, \mathbf{R_A}(true \vdash Q_2) \equiv \mathbf{R_A}(true \vdash \mathbf{R1_A}(Q_1) \,;\, \mathbf{R1_A}(\mathbf{I\!I} \lhd wait \rhd \mathbf{R2_A}(Q_2)))$$

Proof. $\mathbf{R_A}(true \vdash Q_1) \,;\, \mathbf{R_A}(true \vdash Q_2)$

$\equiv$ "application of Law 36"

$$\mathbf{R_A}\left(\begin{array}{l} \neg\,(\mathbf{R1_A}(\neg\, true) \,;\, \mathbf{R1_A}(true)) \wedge \neg\,(\mathbf{R1_A}(Q_1) \,;\, \mathbf{R1_A}(\neg\, wait \wedge \mathbf{R2_A}(\neg\, true))) \\ \vdash \\ \mathbf{R1_A}(Q_1) \,;\, \mathbf{R1_A}(\mathbf{I\!I} \lhd wait \rhd \mathbf{R2_A}(Q_2)) \end{array}\right)$$

$\equiv$ "logic simplification"

$$\mathbf{R_A}\left(\begin{array}{l} \neg\,(\mathbf{R1_A}(false) \,;\, \mathbf{R1_A}(true)) \wedge \neg\,(\mathbf{R1_A}(Q_1) \,;\, \mathbf{R1_A}(\neg\, wait \wedge \mathbf{R2_A}(false))) \\ \vdash \\ \mathbf{R1_A}(Q_1) \,;\, \mathbf{R1_A}(\mathbf{I\!I} \lhd wait \rhd \mathbf{R2_A}(Q_2)) \end{array}\right)$$

$\equiv$ "unfolding definitions of $\mathbf{R1_A}$ and $\mathbf{R2_A}$ in precondition"

$$\mathbf{R_A}\left(\begin{array}{l} \neg\,((false \wedge tr \leq_A tr') \,;\, (true \wedge tr \leq_A tr')) \wedge \neg\,((Q_1 \wedge tr \leq_A tr') \,;\, (\neg\, wait \wedge false)) \\ \vdash \\ \mathbf{R1_A}(Q_1) \,;\, \mathbf{R1_A}(\mathbf{I\!I} \lhd wait \rhd \mathbf{R2_A}(Q_2)) \end{array}\right)$$

$\equiv$ "logic simplification"

$$\mathbf{R_A}\left(\begin{array}{l} \neg\,(false \,;\, tr \leq_A tr') \wedge \neg\,((Q_1 \wedge tr \leq_A tr') \,;\, false) \\ \vdash \\ \mathbf{R1_A}(Q_1) \,;\, \mathbf{R1_A}(\mathbf{I\!I} \lhd wait \rhd \mathbf{R2_A}(Q_2)) \end{array}\right)$$

$\equiv$ "composition with empty relation: $(false \,;\, P) \equiv false \equiv (P \,;\, false)$"

$$\mathbf{R_A}(\neg\, false \wedge \neg\, false \vdash \mathbf{R1_A}(Q_1) \,;\, \mathbf{R1_A}(\mathbf{I\!I} \lhd wait \rhd \mathbf{R2_A}(Q_2)))$$

$\equiv$ "logic simplification"

$$\mathbf{R_A}(true \vdash \mathbf{R1_A}(Q_1) \,;\, \mathbf{R1_A}(\mathbf{I\!I} \lhd wait \rhd \mathbf{R2_A}(Q_2))) \quad \square$$

Fig. 27. Sequential composition of terminating reactive designs in *Circus Time*.

G. Mission class of the Level 1 $\textsf{CD}_x$

```
import javax.safetycritical.Mission;

public class CDxMission extends Mission {
  /* Records the current radar frame of aircraft positions. */
  public RawFrame currentFrame;

  /* Holds previous aircraft positions; it is used to predict their motions. */
  public StateTable state;

  /* Records the computational work for the detector handlers. */
  public Partition work;

  /* Accumulates the number of collisions calculated during detection. */
  public int collisions;

  /* Control object that is used to orchestrate handler execution. */
  public DetectorControl control;

  public CDxMission() {
    /* Here we create shared data objects in mission memory. */
    currentFrame = new RawFrame();
    state = new StateTable();
    work = new Partition(4);
    collisions = 0;
  }

  public @Override void initialize() {
    /* SCJ event that releases ReducerHandler. */
    AperiodicEvent reduce = new AperiodicEvent();

    /* SCJ event that releases all four DetectorHandlers. */
    AperiodicEvent detect = new AperiodicEvent();

    /* SCJ event that releases OutputCollisionsHandler. */
    AperiodicEvent output = new AperiodicEvent();

    /* Control object that fires the output event when detection is completed. */
    control = new DetectorControl(output, 4);

    /* InputFrameHandler reads radar frames; the only periodic handler. */
    InputFrameHandler h1 = new InputFrameHandler(this, reduce);

    /* ReducerHandler performs voxel-hashing and then subdivides the work. */
    ReducerHandler h2 = new ReducerHandler(this, detect, control, reduce);

    /* Four DetectorHandler instances perform the actual detection work. */
    DetectorHandler h3 = new DetectorHandler(this, control, 1, detect);
    DetectorHandler h4 = new DetectorHandler(this, control, 2, detect);
    DetectorHandler h5 = new DetectorHandler(this, control, 3, detect);
    DetectorHandler h6 = new DetectorHandler(this, control, 4, detect);

    /* OutputCollisionsHandler outputs the collisions results. */
    OutputCollisionsHandler h7 = new OutputCollisionsHandler(this, output);
    /* Below we register all handlers with the mission. */
    h1.register(); h2.register(); h3.register(); h4.register();
    h5.register(); h6.register(); h7.register();
  }

  /* SCJ method that specifies the amount of mission memory required. */
  public @Override long missionMemorySize() {
    return Constants.MISSION_MEMORY_SIZE;
  }

  /* Method to get the current frame of aircraft positions. */
  public synchronized RawFrame getFrame() {
    return currentFrame;
  }

  /* Method to set the current frame of aircraft positions. */
  public synchronized void setFrame(RawFrame frame) {
    currentFrame = frame;
  }

  /* Method to get previous aircraft positions. */
  public synchronized StateTable getState() {
    return state;
  }

  /* Method to set previous aircraft positions. */
  public synchronized void setState(StateTable state) {
    this.state = state;
  }

  /* Method to get the shared work variable. */
  public synchronized Partition getWork() {
    return work;
  }

  /* Method to set the shared work variable. */
  public synchronized void setWork(Partition work) {
    this.work = work;
  }

  /* Resets the number of detected collisions. Called by ReducerHandler. */
  public synchronized void initColls() {
    collisions = 0;
  }

  /* Records a partial collisions result. Called by the DetectorHandlers. */
  public synchronized void recColls(int n) {
    collisions += n;
  }

  /* Returns the cumulative collisions. Called by OutputCollisionsHandler. */
  public synchronized int getColls() {
    return collisions;
  }
}
```