A Refinement Strategy for *Circus*

Ana Cavalcanti[1], Augusto Sampaio[2] and Jim Woodcock[3]

[1] Computing Laboratory, University of Kent at Canterbury, Canterbury, CT2 7NF, England
[2] Centro de Informática, Universidade Federal de Pernambuco, P.O. Box 7851, 50740-540, Recife PE, Brazil

Abstract. We present a refinement strategy for *Circus*, which is the combination of Z, CSP, and the refinement calculus in the setting of Hoare & He's unifying theories of programming. The strategy unifies the theories of refinement for processes and their constituent actions, and provides a coherent technique for the stepwise refinement of concurrent and distributed programs involving rich data structures. This kind of development is carried out using *Circus*'s refinement calculus, and we describe some of its laws for the simultaneous refinement of state and control behaviour, including the splitting of a process into parallel subcomponents. We illustrate the strategy and the laws using a case study that shows the complete development of a small distributed program.

1 Introduction

The last twenty years have seen the development and application of various formal methods for software engineering, typified by the state-based and the process algebraic schools. In the state-based approach, a model is constructed of the desired system, avoiding incidental details and concentrating only on essential properties; successive models are constructed, in ever-more detail, until an implementation is reached. For correctness, the models must be related by some well-understood refinement relation. The most prominent members of this school of development are VDM [15], Z [24, 28], and the refinement calculus [18]. In VDM and Z, emphasis is on specification and data refinement, whilst in the refinement calculus, emphasis is, naturally, on calculation. In the process algebraic approach, systems and components are represented by process behaviour patterns that evolve and interact through atomic events. The most prominent members of the process algebraic school are CCS [17] and CSP [11, 21]; the latter has a theory of behavioural refinement.

The two schools started out with complementary objectives: VDM, Z, and the refinement calculus have been used to understand how to specify and develop sequential programs with complex data structures; CCS and CSP have been used to understand complex patterns of concurrency and communication, but with simple data structures. Over the last decade, various attempts have been made to unite the two schools, and produce a method that is suitable for developing concurrent and distributed systems with sophisticated data structures. The first attempts involved understanding data refinement in the context of concurrency and communication [16, 13, 29]. Subsequent attempts focussed on combining particular notations, such as VDM, Z, CCS, and CSP [6, 7, 23, 25].

Most recently, the *Circus* language has been proposed; it is a thorough integration of Z, CSP, and the refinement calculus. The semantics of *Circus* are given in Hoare & He's unifying theories of programming [12], resulting in a notation with a single semantics and a single refinement ordering. The objective was to give a sound basis to the development of concurrent and distributed systems in the calculational style.

In *Circus*, systems are characterised by processes, which group constructs that describe data and control behaviour. Data is defined using mainly the Z notation, and behaviour is characterised by actions, which are defined using Z, CSP, and guarded command constructs.

In previous work [22, 3], we have dealt, separately, with action and process refinement in *Circus*. In that context, we proposed refinement notions and some useful laws. The purpose of this paper is to unify these results, and provide an accompanying iterative development strategy, involving the application of simulation, action and, most importantly, process refinement. In this integrated context, we propose a notion of backwards simulation, in addition to that of forwards simulation provided in [22]. We also provide some new refinement laws for simulation, actions, and processes. We illustrate the refinement strategy through a complete development of a case study which has been only partially addressed before.

In Section 2, we describe the specification of a bounded, reactive, buffer as a means of introducing the *Circus* notation. In Section 3, we explain the notions of refinement in *Circus* for processes and their constituent actions, and the simulation technique; we also describe our strategy for the development of centralised specifications into distributed implementations. In Section 4, we give laws for piece-wise data refinement. Sections 5 and 6 contain our algebraic laws for the simultaneous refinement of the state and control behaviour of processes and of actions. In Section 7, we illustrate our refinement strategy by showing the complete development of the reactive buffer into a distributed implementation. Finally, in Section 8, we draw our conclusions and discuss related work.

2 *Circus*

A *Circus* program is a list of paragraphs containing process definitions and their ancillary declarations of channels, types, and global constants. We provide a BNF description of the *Circus* syntax later on, but first we introduce *Circus* through an example: a bounded, reactive buffer. We present a specification for this buffer here, and in Section 7 we refine it to obtain a distributed design.

The maximum size of the buffer is a strictly positive constant.

$$\mid \; \mathit{maxbuff} : \mathbb{N}_1$$

It takes its inputs and supplies its outputs on two different typed channels.

$$\mathbf{channel}\ \mathit{input}, \mathit{output} : \mathbb{N}$$

In *Circus* we use the Z notation to define the state; the attendant operations are actions, which are specified using Z, CSP operators, and the guarded command language. In our example, the process *Buffer* encapsulates two state components: an ordered list of the contents and the size of this list.

$$\mathbf{process}\ \mathit{Buffer} \mathrel{\widehat=} \mathbf{begin}$$

$$\mathbf{state}\ \mathit{BufferState} \mathrel{\widehat=} [\, \mathit{buff} : \operatorname{seq} \mathbb{N};\ \mathit{size} : 0 \mathinner{..} \mathit{maxbuff} \mid \mathit{size} = \#\mathit{buff} \leq \mathit{maxbuff} \,]$$

Initially, the buffer is empty; this is specified as a state initialisation action.

$$\mathit{BufferInit} \mathrel{\widehat=} [\, \mathit{BufferState}' \mid \mathit{buff}' = \langle\rangle \land \mathit{size}' = 0 \,]$$

We need to describe the behaviour of the process on input and output. The buffer accepts an input whenever there is space to store the new value; in this case, the element input is appended to the bounded sequence and the size incremented. First, we specify *InputCmd*, which describes the effect of the input on the state; as usual in Z, a schema is used.

$$\mathit{InputCmd} \mathrel{\widehat=} [\, \Delta \mathit{BufferState};\ x? : \mathbb{N} \mid \mathit{size} < \mathit{maxbuff} \land \mathit{buff}' = \mathit{buff} \frown \langle x? \rangle \land \mathit{size}' = \mathit{size} + 1 \,]$$

The following action, *Input*, describes the constraint on the communication over the channel *input*, and uses the schema action *InputCmd* to specify the state changes.

$$\mathit{Input} \mathrel{\widehat=} \mathit{size} < \mathit{maxbuff} \mathbin{\&} \mathit{input}?x \rightarrow \mathit{InputCmd}$$

This action is guarded by $\mathit{size} < \mathit{maxbuff}$: if this condition does not hold, *Input* deadlocks. In contrast, if a precondition of a schema action is not satisfied, its execution diverges (aborts), as usual in Z.

The action $\mathit{input}?x \rightarrow \mathit{InputCmd}$ is a prefixing in the style of CSP. A new input variable x is introduced, and a value input through the channel *input* is assigned to it. Afterwards, the action *InputCmd* is executed.

The *Output* action is enabled when the buffer contains something; it outputs the head of the buffer, and updates the size accordingly.

$$\mathit{OutputCmd} \mathrel{\widehat=} [\, \Delta \mathit{BufferState} \mid \mathit{size} > 0 \land \mathit{buff}' = \mathit{tail}\ \mathit{buff} \land \mathit{size}' = \mathit{size} - 1 \,]$$

$$\mathit{Output} \mathrel{\widehat=} \mathit{size} > 0 \mathbin{\&} \mathit{output}!(\mathit{head}\ \mathit{buff}) \rightarrow \mathit{OutputCmd}$$

An unnamed main action at the end of a process description defines its extensional behaviour as a protocol in terms of the actions over the state. In our example, the main action initialises the buffer and repeatedly offers the choice of input and output.

$$\bullet\ \mathit{BufferInit};\ (\mu X \bullet (\mathit{Input} \mathbin{\Box} \mathit{Output});\ X)$$

$| \ maxbuff : \mathbb{N}_1$

channel $input, output : \mathbb{N}$

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

  state $BufferState \mathrel{\widehat=} [\, buff : \operatorname{seq} \mathbb{N};\ size : 0 \mathinner{..} maxbuff \mid size = \# buff \leq maxbuff \,]$

  $BufferInit \mathrel{\widehat=} [\, BufferState' \mid buff' = \langle\rangle \land size' = 0 \,]$

  $InputCmd \mathrel{\widehat=} [\, \Delta BufferState;\ x? : \mathbb{N} \mid size < maxbuff \land buff' = buff \frown \langle x? \rangle \land size' = size + 1 \,]$

  $Input \mathrel{\widehat=} size < maxbuff \ \&\ input?x \rightarrow InputCmd$

  $OutputCmd \mathrel{\widehat=} [\, \Delta BufferState \mid size > 0 \land buff' = tail\ buff \land size' = size - 1 \,]$

  $Output \mathrel{\widehat=} size > 0 \ \&\ output!(head\ buff) \rightarrow OutputCmd$

  $\bullet\ BufferInit;\ (\, \mu X \bullet (\, Input \ \Box\ Output \,);\ X \,)$

end

Fig. 1. *Buffer* process specification.

end

We summarise the complete specification of the buffer process in Figure 1.

In Figure 2 we present the BNF description of the syntax of *Circus*. CircusPar* is a possibly empty list of elements of the syntactic category CircusPar of *Circus* paragraphs; similarly for PPar* (process paragraphs). We use N^+ for a comma-separated list of Z identifiers (elements of N), and similarly for Exp^+. The syntactic categories Par, Schema-Exp, Exp, Pred, and Decl include the Z paragraphs, schema expressions, expressions, predicates, and declarations defined in [24]. The syntactic category CSExp of channel set expressions contains the empty set $\{\!|\,|\!\}$, channel enumerations enclosed in $\{\!|$ and $|\!\}$, and expressions involving the usual set operators. Similarly, NSExp contains sets of state components; in this case the delimiters are ordinary braces.

The *Buffer* example has shown how processes are constructed from actions, but processes may themselves be combined with CSP operators, such as (alphabetised) parallel composition. The meaning of a new process constructed in this way is obtained from the conjunction of the state of the constituent processes and the parallel combination of their main actions.

A perhaps unusual operator available in *Circus* is indexing: a process such as $i : T \odot P$ behaves like P, but uses different channels. For each channel c of P, we have a fresh channel c_i that communicates pairs of values: the first element is the index, a value of type T, and the second element is the value originally communicated through c. The instantiation $(i : T \odot P)\lfloor e \rfloor$ behaves like P, but the first element of the pairs communicated is the value of the index expression e. We also have a renaming operator in *Circus*. For example, in $P[c_1 := c_2]$, the communications of P through channel c_1 are done through the channel c_2 instead. An example of the use of the indexing and renaming operators is found in our case study (Section 7).

Parametrised processes are also available in *Circus*. In the process $x : T \bullet P$, for instance, x is a parameter that can be used in the specification of P as a value of type T. To use this process, we must instantiate it by providing a value for x, as in $(x : T \bullet P)(v)$, where v is a value of type T.

At the level of actions, the *Circus* parallel operator is slightly different from that of CSP. Instead of having simply a synchronisation channel set, we also have two sets that partition all the variables in scope: the state components, and the input and local variables. For instance, we write $A_1 [\![ns_1 \mid C \mid ns_2]\!] A_2$ for the parallel composition of A_1 and A_2 synchronising on the channels in the set C, so that A_1 can modify only the variables in ns_1 and, similarly, A_2 can modify just the variables in ns_2; both A_1 and A_2 have access to the initial value of the variables in ns_1 and ns_2. Examples are presented in our case study (Section 7). Sets containing names of state components can be defined to shorten the definition of parallel compositions of actions. The interleaving operator also requires state partitioning sets.

We also have iterated versions of the process and action operators. For example, if P is an indexed process, then $|||\, i : T \odot P\lfloor i \rfloor$ is the process defined by interleaving each of the processes $P\lfloor v \rfloor$ formed by instantiating P with a value v of T. Except for sequence, all the above operators are commutative

$$
\begin{array}{lll}
\mathsf{Program} & ::= & \mathsf{CircusPar}^{*} \\[4pt]
\mathsf{CircusPar} & ::= & \mathsf{Par} \mid \mathbf{channel}\ \mathsf{CDecl} \mid \mathbf{chanset}\ \mathsf{N} == \mathsf{CSExp} \mid \mathbf{process}\ \mathsf{N} \mathrel{\widehat=} \mathsf{Proc} \\[4pt]
\mathsf{CDecl} & ::= & \mathsf{SimpleCDecl} \mid \mathsf{SimpleCDecl};\ \mathsf{CDecl} \\
\mathsf{SimpleCDecl} & ::= & \mathsf{N}^{+} \mid \mathsf{N}^{+} : \mathsf{Exp} \mid \mathsf{Schema\text{-}Exp} \\[4pt]
\mathsf{CSExp} & ::= & \{\!|\ |\!\} \mid \{\!|\ \mathsf{N}^{+}\ |\!\} \mid \mathsf{N} \mid \mathsf{CSExp} \cup \mathsf{CSExp} \mid \mathsf{CSExp} \cap \mathsf{CSExp} \mid \mathsf{CSExp} \setminus \mathsf{CSExp} \\[4pt]
\mathsf{Proc} & ::= & \mathbf{begin}\ \mathsf{PPar}^{*}\ \mathbf{state}\ \mathsf{Schema\text{-}Exp}\ \mathsf{PPar}^{*} \bullet \mathsf{Action}\ \mathbf{end} \mid \mathsf{N} \\
 & \mid & \mathsf{Proc};\ \mathsf{Proc} \mid \mathsf{Proc} \Box \mathsf{Proc} \mid \mathsf{Proc} \sqcap \mathsf{Proc} \\
 & \mid & \mathsf{Proc} \llbracket \mathsf{CSExp} \rrbracket \mathsf{Proc} \mid \mathsf{Proc} \mathbin{|||} \mathsf{Proc} \mid \mathsf{Proc} \setminus \mathsf{CSExp} \\
 & \mid & \mathsf{Decl} \odot \mathsf{Proc} \mid \mathsf{Proc}\lfloor \mathsf{Exp}^{+} \rfloor \mid \mathsf{Proc}[\mathsf{N}^{+} := \mathsf{N}^{+}] \\
 & \mid & \fatsemi\ \mathsf{Decl} \odot \mathsf{Proc} \mid \ \Box\ \mathsf{Decl} \odot \mathsf{Proc} \mid \ \bigsqcap \mathsf{Decl} \odot \mathsf{Proc} \\
 & \mid & \big\| \mathsf{Decl} \llbracket \mathsf{CSExp} \rrbracket \odot \mathsf{Proc} \mid \ \big|\big|\big| \mathsf{Decl} \odot \mathsf{Proc} \\
 & \mid & \mathsf{Decl} \bullet \mathsf{Proc} \mid \mathsf{Proc}(\mathsf{Exp}^{+}) \\
 & \mid & \fatsemi\ \mathsf{Decl} \bullet \mathsf{Proc} \mid \ \Box\ \mathsf{Decl} \bullet \mathsf{Proc} \mid \ \bigsqcap \mathsf{Decl} \bullet \mathsf{Proc} \\
 & \mid & \big\| \mathsf{Decl} \llbracket \mathsf{CSExp} \rrbracket \bullet \mathsf{Proc} \mid \ \big|\big|\big| \mathsf{Decl} \bullet \mathsf{Proc} \\[4pt]
\mathsf{NSExp} & ::= & \{\,\} \mid \{\mathsf{N}^{+}\} \mid \mathsf{N} \mid \mathsf{NSExp} \cup \mathsf{NSExp} \mid \mathsf{NSExp} \cap \mathsf{NSExp} \mid \mathsf{NSExp} \setminus \mathsf{NSExp} \\[4pt]
\mathsf{PPar} & ::= & \mathsf{Par} \mid \mathsf{N} \mathrel{\widehat=} \mathsf{Action} \\[4pt]
\mathsf{Action} & ::= & \mathsf{Schema\text{-}Exp} \mid \mathsf{CSPAction} \mid \mathsf{Command} \mid \mathbf{nameset}\ \mathsf{N} == \mathsf{NSExp} \\[4pt]
\mathsf{CSPAction} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \mathsf{Comm} \rightarrow \mathsf{Action} \mid \mathsf{Pred}\ \&\ \mathsf{Action} \\
 & \mid & \mathsf{Action};\ \mathsf{Action} \mid \mathsf{Action} \Box \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
 & \mid & \mathsf{Action} \llbracket \mathsf{NSExp} \mid \mathsf{CSExp} \mid \mathsf{NSExp} \rrbracket \mathsf{Action} \mid \mathsf{Action} \mathbin{|||\lbrack} \mathsf{NSExp} \mid \mathsf{NSExp} \mathbin{\rbrack|||} \mathsf{Action} \\
 & \mid & \mathsf{Action} \setminus \mathsf{CSExp} \mid \ \mu\, \mathsf{N} \bullet \mathsf{Action} \mid \mathsf{Decl} \bullet \mathsf{Action} \mid \mathsf{Action}(\mathsf{Exp}^{+}) \\
 & \mid & \fatsemi\ \mathsf{Decl} \bullet \mathsf{Action} \mid \ \Box\ \mathsf{Decl} \bullet \mathsf{Action} \mid \bigsqcap \mathsf{Decl} \bullet \mathsf{Action} \\
 & \mid & \big\| \mathsf{Decl} \llbracket \mathsf{NSExp} \mid \mathsf{CSExp} \mid \mathsf{NSExp} \rrbracket \bullet \mathsf{Action} \mid \ \big|\big|\big| \mathsf{Decl} \mathbin{|||\lbrack} \mathsf{NSExp} \mid \mathsf{NSExp} \mathbin{\rbrack|||} \bullet \mathsf{Action} \\[4pt]
\mathsf{Comm} & ::= & \mathsf{N}\ \mathsf{CParameter}^{*} \\
\mathsf{CParameter} & ::= & ?\,\mathsf{N} \mid ?\,\mathsf{N} : \mathsf{Predicate} \mid !\,\mathsf{Expression} \mid .\,\mathsf{Expression} \\[4pt]
\mathsf{Command} & ::= & \mathsf{N}^{+} : [\,\mathsf{Pred}, \mathsf{Pred}\,] \mid \mathsf{N}^{+} := \mathsf{Exp}^{+} \mid \mathbf{if}\ \mathsf{GActions}\ \mathbf{fi} \mid \mathbf{var}\ \mathsf{Decl} \bullet \mathsf{Action} \\[4pt]
\mathsf{GActions} & ::= & \mathsf{Pred} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \rightarrow \mathsf{Action} \Box \mathsf{GActions}
\end{array}
$$

Fig. 2. *Circus* syntax

and associative, so there is no concern about the order of the elements of T or about the grouping of the processes. For the sequence operator, we require T to be a sequence and define $; \ i : T \odot P\lfloor i \rfloor$ to be the sequence of the processes $P\lfloor v \rfloor$, with v taken from T in the order that they appear. For the indexed parallel and interleave operators, we also need to provide the list of synchronising channels, and state partitioning sets.

The semantics of *Circus* [27, 26] is based on Hoare & He's unifying theories of programming [12]: an alphabetised relational model for imperative programming, concurrency, and communication. In this model, distinguished variables are used to describe relevant observations and the relations are defined by predicates over these variables. Decorations are used to differentiate references to the initial value of the variables from those in subsequent observations. As in most refinement theories, the same model is used for specifications and programming constructs, which can be mixed during development.

Refinement is a central notion in the unifying theories of programming, where it is defined as (reverse) implication: if an implementation is to behave satisfactorily, then every observation that we make of it must be permitted by the specification. Formally, a mechanism modelled as a predicate P satisfies a specification S, another predicate, providing that $[\, P \Rightarrow S \,]$, where the square brackets denote universal quantification over all observation variables. The set of observation variables must be the same for both P and S.

In our work, Z is used as the concrete syntax for the relational model, so that a *Circus* program denotes a Z specification. Each process corresponds to a part of that specification characterised by a state definition. Actions are modelled as operations over this state. The state components are the observation variables, which include the components of the process state and components to model behaviour: stability from divergence ($okay$), termination ($wait$), a history of interaction with the environment (tr), and a set of events that can be refused (ref). This is a state-based, failures-divergences model, with embedded imperative features.

3 Refinement notions and strategy

The basic notion of refinement in *Circus* is that of action refinement; refinement of processes is defined in terms of refinement of main actions. The definition of the relation $\sqsubseteq_{\mathcal{A}}$ of action refinement is shown below. For simplicity, we identify a *Circus* action with its semantics: a schema specifying a change on the state formed by the components of the state of the process where the action occurs, and the extra observation variables $okay$, $wait$, tr, and ref. We say that, together, these variables compose the state space of the action.

Definition 1 (Action refinement). *For actions A_1 and A_2 on the same state space, we define $A_1 \sqsubseteq_{\mathcal{A}} A_2$ if, and only if, $[\, A_2 \Rightarrow A_1 \,]$.* □

This relation is a partial order, and the action constructors are monotonic with respect to it. Therefore, we can adopt a piecewise and stepwise refinement technique. Typically, we use the action refinement relation to compare actions of the same process, which, of course, act on the same state space.

Refinement of processes must consider their state and behaviour. Since the main action of a process defines its behaviour, roughly, a process P_1 is refined by a process P_2 if the main action of P_1 is refined by that of P_2. These actions, however, may act on different state spaces, and so may not be comparable. Since the states are encapsulated, we are to compare the actions we obtain by hiding the components of the state of P_1 and P_2, as if they were declared locally in a variable block. Process refinement ($\sqsubseteq_{\mathcal{P}}$) is, therefore, defined in terms of action refinement of local blocks, whose semantics is given by existential quantification. In the following, we use $P.st$ and $P.act$ to denote the local state and main action of a process P.

Definition 2 (Process refinement). *We define $P_1 \sqsubseteq_{\mathcal{P}} P_2$ if, and only if,*

$$(\exists\, P_1.st;\ P_1.st' \bullet P_1.act\,) \sqsubseteq_{\mathcal{A}} (\exists\, P_2.st;\ P_2.st' \bullet P_2.act\,) \qquad \square$$

When we hide the local states of the processes, we are left with two actions on the same state space, which contains only $okay$, $wait$, tr, and ref as components.

Because the state of a process is private, we may change its components and their types during refinement, in much the same way as we can data refine variable blocks and modules in imperative programs [19]. In those contexts, forwards and backwards simulation are well-known techniques of data refinement [10, 14]. Here, we adapt the standard techniques used in Z [28] to handle processes and actions.

In *Circus*, a simulation is a relation between the states of two processes that satisfies a number of properties. For example, in our case study presented in Section 7, the first step is the refinement of the *Buffer* process to introduce a *cache* and a *ring* to represent the internal state. When the buffer is non-empty, the cache stores the head of the buffer. The *ring* is a circular array, modelled as a sequence whose two ends are considered to be joined. We maintain two indices into this array: a *bot*tom and a *top*, to delimit the relevant values. This part of the array is a concrete representation of the tail of the original bounded buffer. Modulo arithmetic is used to increment *bot* and *top*; in Z sequence indices start at 1. The constant *maxring*, defined as $maxbuff - 1$, gives the bound for the ring. In the new process, the state is defined as follows.

```
CBufferState
  size : 0 .. maxbuff; ringsize : 0 .. maxring
  cache : ℕ; ring : seq ℕ
  top, bot : 1 .. maxring
  ────────
  ringsize mod maxring = (top − bot) mod maxring
  ringsize = max{0, size − 1}
  #ring = maxring
```

There is a subtle situation when the *bot*tom and the *top* indices coincide; in this case it is not possible to distinguish whether the ring has reached its maximum storage capacity or whether it is empty. As a consequence, we need to keep a separate record of the number *ringsize* of values stored in the ring.

With this new state, all the actions of the original abstract *Buffer* process need to be changed accordingly. In order to justify the refinement, we provide a simulation between the state of the original buffer and the new state, much in the same way as we do when we data refine Z specifications. In our example, the simulation (or retrieve) relation is as follows. We use a shift operator: $n \ll a$ shifts the (circular) array a by n positions. For the sake of conciseness, we omit the simple inductive definition of this operator.

$$RetrBuffer \mathrel{\widehat{=}} [\, BufferState;\ CBufferState \mid buff = (1 \mathinner{..} size) \lhd (\langle cache \rangle \frown ((bot - 1) \ll ring)) \,]$$

If we shift the *ring* so that the oldest element at *bot* occurs at position 1, concatenate the *cache* at the front of the resulting array, and restrict everything to the first *size* elements, then we have the abstract buffer.

To prove that *RetrBuffer* is indeed a simulation, we have to show that it satisfies a few properties. We define below the restrictions for actions and processes. We actually consider a relation that involves, besides the states of the processes, a local state that can include input and local variables in scope, and any additional information inferred from guards and conditionals in context.

This context is easily defined by induction on the structure of the actions. For example, the context for actions like *Skip* and *Stop* is a schema with an empty declaration and predicate *true*. For a schema expression, the context contains the declaration of the input and output variables, if any. For a guarded action, the context is enriched by conjoining the guard to its predicate. We omit further discussion of contexts in this paper, for the sake of simplicity.

Definition 3 (Forwards simulation). *A forwards simulation between actions A_1 and A_2 of processes P_1 and P_2, with local state L, is a relation R between $P_1.st$, $P_2.st$, and L, satisfying*

1. (feasibility) $\forall P_2.st;\ L \bullet (\exists P_1.st \bullet R)$
2. (correctness) $\forall P_1.st;\ P_2.st;\ P_2.st';\ L \bullet R \land A_2 \Rightarrow (\exists P_1.st';\ L' \bullet R' \land A_1)$

In this case, we write $A_1 \preceq_{P_1,P_2,R,L} A_2$ and say that the action A_2 simulates the action A_1, according to the simulation R, and in a state extended by L. When clear from the context, we omit the subscripts. A forwards simulation between P_1 and P_2 is a forwards simulation between their main actions. □

In this definition, there is no applicability requirement concerning preconditions, as would usually be found in the definition of forwards simulation; this is because actions are total. An action that is executed outside its preconditions diverges, but this does not mean that its behaviour is arbitrary. Implicitly, it is guaranteed that the state invariant is maintained, and that arbitrary new synchronisations and communications can be observed, but no past observations are affected.

Another possibly surprising aspect of Definition 3 is the fact that we do not impose any specific conditions on the initialisation. It is not necessarily the case that there is a separate initialisation action and, even if there is, it has to be explicitly included in the main action, as illustrated in Figure 1.

A theorem reproduced below and proved in [22] ensures that, if we provide a forwards simulation between processes P_1 and P_2, then we can substitute P_2 for P_1 in a program.

Theorem 1 (Forwards simulation is sound). *When a forwards simulation exists between two processes P_1 and P_2, we also have that $P_1 \sqsubseteq_{\mathcal{P}} P_2$.* □

In the next section we present laws that support the refinement of a process using simulation.
For backwards simulation, we have similar definitions and results; first, we present the definition.

Definition 4 (Backwards simulation). *A backwards simulation between actions A_1 and A_2 of processes P_1 and P_2, with local state L, is a relation R between $P_1.st$, $P_2.st$, and L, satisfying*

1. (feasibility) $\forall P_2.st;\ L \bullet (\exists P_1.st \bullet R)$
2. (correctness) $\forall P_1.st';\ P_2.st;\ P_2.st';\ L' \bullet R' \land A_2 \Rightarrow (\exists P_1.st;\ L \bullet R \land A_1)$

A backwards simulation between P_1 and P_2 is a backwards simulation between their main actions. □

A soundness theorem can be proved in much the same way as in the case of forwards simulation; the challenge is to realise the need for the feasibility restriction, which is the same for both forms of simulation.

Although we have notions of forwards and backwards simulation, we do not have a completeness result due to the presence of unbounded nondeterminism. The technical difficulties involved are discussed in [9].

The refinement notions support the transformation of *Circus* processes for several purposes, such as to show the equivalence or refinement of processes, to carry out optimisations, or to justify change of data representation. Here, we are interested in a development strategy that we discuss in the sequel. The strategy is based on laws of simulation, and action and process refinement, which we present in Sections 4 to 6.

The strategy includes (possibly several iterations) of the following steps (see Figure 3):

- use of simulation to introduce the elements of the concrete system state;
- algorithmic refinement of actions for partitioning the process state space and actions; and
- process refinement (decomposition).

Although simple, the iterated application of these steps can effectively serve as a tool to guide and transform an abstract, and usually centralised, specification into a concrete distributed solution of the system.

Assuming a centralised process specification as the starting point of the development, the first two steps of the strategy are used to reorganise the internal structure of the process. In Figure 3, the first specification is a centralised process with state components $a1, \ldots, an$, actions $ActA1, \ldots, ActAk$, and main action $ActA$. Simulation lays the ground for the introduction of concrete state elements $c1, \ldots, cn$, which compose the state of the second process in Figure 3. The actions are also changed to operate on the modified state.

Action refinement allows the partitioning of the state space and the accompanying actions, in such a way that each partition groups some state elements and the actions which access (read or update) these elements. The resulting structure clearly reflects the fact that each partition will become an independent process. The third process in Figure 3 illustrates this partitioning. The state Sc is defined by the conjunction of two state schemas $Sc1$ and $Sc2$; sets of actions that operate independently on $Sc1$ and on $Sc2$ can be clearly identified; and the main action is defined as a combination of two of these independent actions: $Act1C$ and $Act2C$. In the next step, these become the main actions of the partitioned processes.

The upgrade of a partition into a process is precisely captured in the third step of the strategy. The resulting processes are combined in the same way as their main actions were in the previous process.

It is worth emphasising that several iterations might be necessary, even in a medium-sized development, since a process resulting from one decomposition can itself be further decomposed into other processes, and so on. An interesting issue is that the development of a distributed system does not necessarily start with a centralised specification. Nonetheless, even in such a more pragmatic scenario, some process decomposition is usually essential in order to progress towards the final implementation, which needs to satisfy non-functional requirements like efficiency. Therefore, the strategy can be applied to every

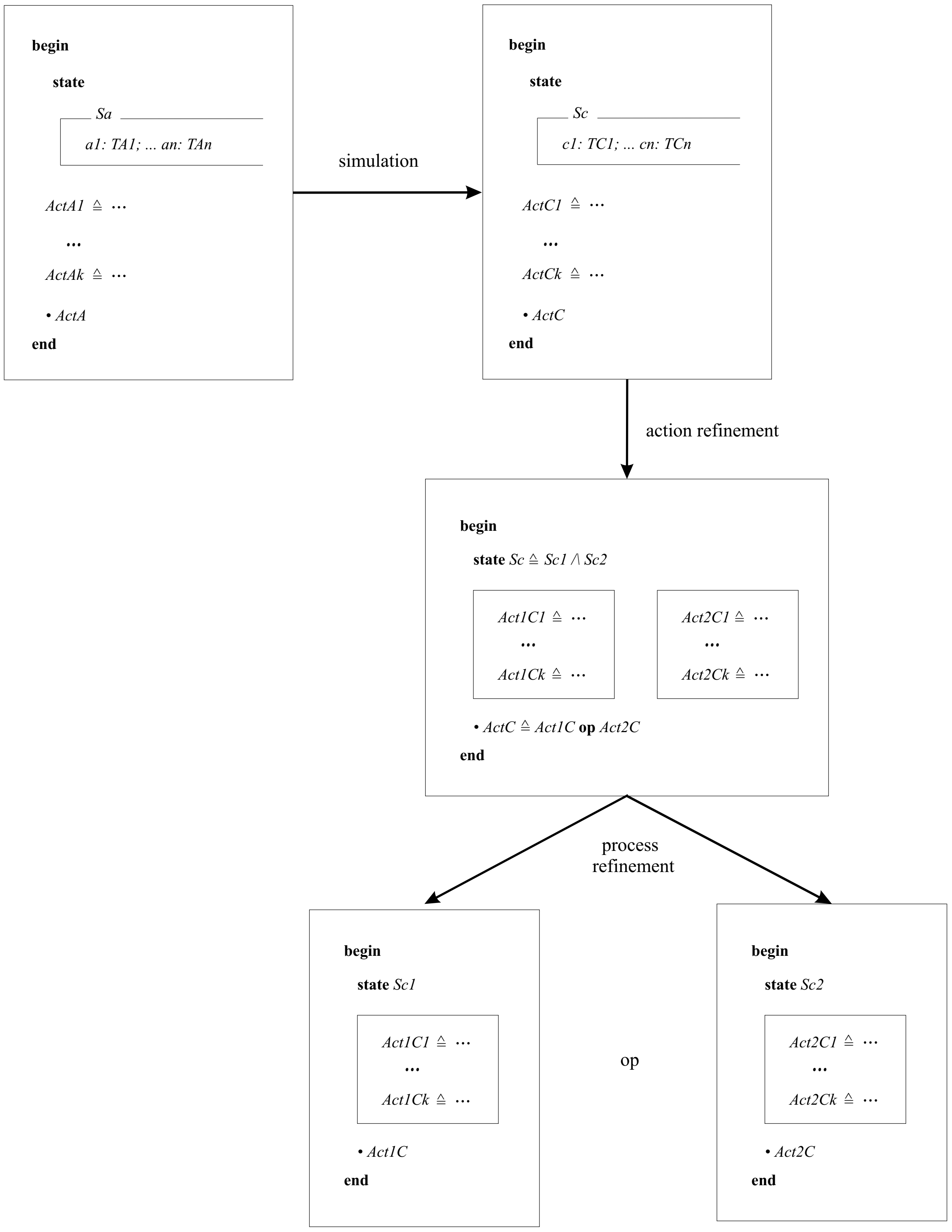

Fig. 3. Iteration of the refinement strategy

process that needs decomposing, just like it is applicable when the starting point is a centralised process specification.

An iteration of the strategy does not need to follow the above sequence of steps strictly. For example, a simulation step can subsume some action refinement. On the other hand, several applications of a given step (like data or action refinement) might be convenient for modularising the development. It might be even the case that a given process is ready for decomposition from start, when its internal structure is already partitioned. In this case, only the third step of the strategy would be necessary. An iteration of the strategy is characterised by one application of the process refinement step: whether several, one, or no application of simulation and action refinement is needed depends on the particular development at hand.

4 Laws of simulation

Although the definition of simulation and its soundness property give us the basis for a data refinement technique, in practice, we need laws to carry out data refinement in a stepwise way. The laws in the sequel provide support to prove that a relation R is indeed a forwards simulation. Using these laws, we can justify proving simulation for schema actions, in much the same way as we do in Z. Moreover, we are able to preserve the structure of the actions of the original process P_1 in the definition of the new process P_2.

The primitive actions *Skip*, *Stop*, and *Chaos* are not affected by forwards simulation. For instance $\mathit{Skip} \preceq \mathit{Skip}$, for any P_1, P_2, R, and L, which we omit. For schema actions, the provisos are those in the standard Z rule, which is a rather pleasing result in terms of using well-established techniques.

Law 41 (Schema Expressions)

$$\mathit{ASExp} \preceq \mathit{CSExp}$$

provided

- $\forall P_1.st;\ P_2.st;\ L \bullet R \land \mathrm{pre}\ \mathit{ASExp} \Rightarrow \mathrm{pre}\ \mathit{CSExp}$
- $\forall P_1.st;\ P_2.st;\ P_2.st';\ L \bullet R \land \mathrm{pre}\ \mathit{ASExp} \land \mathit{CSExp} \Rightarrow (\exists P_1.st';\ L' \bullet R' \land \mathit{ASExp})$ □

We refrain from presenting the particular case of initialisation operations and functional data refinement. Since the more generic rule presented above holds, the usual results follow.

This law includes an applicability condition, which does not appear in the definition of forwards simulation. This is because the definition is concerned with the semantics of actions, which are total operations on the state that includes the extra observation variables *okay*, *wait*, *tr*, and *ref*. An action described by a schema expression is an operation over the process state and it is not necessarily total. The proof of soundness for this and several other laws presented here can be found in [22, 3].

As already mentioned, forwards simulation distributes through the other constructs. Below, we present the rule for an input prefix.

Law 42 (Input prefix distribution)

$$c?x \to A_1 \preceq c?x \to A_2$$

provided $A_1 \preceq A_2$ □

For output prefixing, we need to relate the abstract and concrete expressions defining the output value.

Law 43 (Output prefix distribution)

$$c!ae \to A_1 \preceq c!ce \to A_2$$

provided

- $\forall P_1.st;\ P_2.st;\ L \bullet R \Rightarrow ae = ce$
- $A_1 \preceq A_2$ □

The concrete and abstract values have to be equal, with respect to the retrieve relation. As far as we know, this characterisation of equality for expressions modulo a retrieve relation is original. For guarded actions, we also need to relate the abstract and the concrete predicates that define the guard.

Law 44 (Guard distribution)

$$ag \mathbin{\&} A_1 \preceq cg \mathbin{\&} A_2$$

provided

- $\forall P_1.st;\ P_2.st;\ L \bullet R \Rightarrow (ag \Leftrightarrow cg)$
- $A_1 \preceq A_2$ □

The proviso is similar to that of Law 43.
For the other constructs, we have straightforward distribution laws.

Law 45 (Sequence distribution)

$$A_1;\ A_2 \preceq B_1;\ B_2$$

provided

- $A_1 \preceq B_1$
- $A_2 \preceq B_2$ □

Law 46 (External choice distribution)

$$A_1 \Box A_2 \preceq B_1 \Box B_2$$

provided

- $A_1 \preceq B_1$
- $A_2 \preceq B_2$ □

The law for parallelism uses the fact that the actions affect disjoint parts of the state; there is no interference.

Law 47 (Parallelism distribution)

$$A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 \preceq B_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B_2$$

provided

- $A_1 \preceq B_1$
- $A_2 \preceq B_2$ □

The fact that ns_1 and ns_2 partition the state is a well-formedness condition for the parallel operator. For recursion, we also have a simple result.

Law 48 (Recursion distribution)

$$\mu X \bullet F(A) \preceq \mu X \bullet F(B)$$

provided $A \preceq B$ □

We use F to stand for a context: a function on actions that determines the body of the recursion.

Based on this set of laws, we can conclude that forwards simulation distributes through the structure of arbitrary actions. Our case study (Section 7) illustrates the use of these laws; we give a new action on the state defined by *ControllerState* and deduce, from applications of Law 41 to the schema expressions, that the whole action simulates the original main action of *Buffer* according to *RetrBuffer*. Simulation is also used in a later stage of the development.

The laws for backwards simulation are very similar to those above, consequently, we present below only the law for schema expressions.

Law 49 (Schema Expressions)

$$ASExp \preceq CSExp$$

provided

- $\forall P_2.st;\ L \bullet (\forall P_1.st \bullet R \Rightarrow \text{pre } ASExp) \Rightarrow \text{pre } CSExp$
- $\forall P_2.st;\ L \bullet (\forall P_1.st \bullet R \Rightarrow \text{pre } ASExp) \Rightarrow$
  $(\forall P_1.st';\ P_2.st';\ L' \bullet CSExp \land R' \Rightarrow (\exists P_1.st \bullet R \land ASExp))$ □

Again, we have proof obligations similar to those needed for establishing refinement of Z specifications.

5 Process Refinement

In this section we propose refinement laws for processes. These laws deal simultaneously with state and control behaviour. As further discussed in Section 7, our approach to the refinement of *Circus* specifications is guided by the progressive and incremental distribution of a specification originally centralised. Surprisingly, perhaps, such a strategy can be supported by simple laws that allow the partitioning of processes.

We present two families of refinement laws, but beforehand we state a basic law that allows the introduction of a new process declaration *pd* as part of the sequence of paragraphs of a *Circus* program *cp*.

Law 51 (Process declaration introduction)

$$cp \ = \ pd \ cp$$

provided *the process declared in pd is not referenced in cp.* □

This law can be justified in much the same way as the introduction of fresh variables and procedure declarations in an imperative program, or new class declarations in an object-oriented program. As the declared process is assumed to be unused, its introduction has no effect whatsoever. The importance of this apparently innocuous law becomes evident in the sequel.

5.1 Process splitting

The first family of process partitioning laws, called *process splitting*, apply to processes whose state components can be partitioned in such a way that each partition has its own set of process paragraphs. The result is three processes: each of the first two includes a partition of the state and the corresponding paragraphs, and the third process, defined in terms of the first two, has the same behaviour as the original one.

In what follows, we assume that *pd* stands for the process declaration below, where we use $Q.pps$ and $R.pps$ to stand for the process paragraphs of the processes Q and R; and F for an action context which must also make sense as a function on processes, according to the *Circus* syntax.

process $P \mathrel{\widehat=}$ **begin**
  state $State \mathrel{\widehat=} Q.st \land R.st$
  $Q.pps \upharpoonright R.st$
  $R.pps \upharpoonright Q.st$
  $\bullet\ F(Q.act, R.act)$
end

The state of P is defined as the conjunction of two other state schemas: $Q.st$ and $R.st$. The actions of P are $Q.pps \upharpoonright R.st$ and $R.pps \upharpoonright Q.st$, which handle the partitions of the state separately. In $Q.pps \upharpoonright R.st$, each schema expression in $Q.pps$ is conjoined with $\Xi R.st$. This means that these process paragraphs do not change the state components of $R.st$; similarly for $R.pps \upharpoonright Q.st$.

We say that a process with the above internal structure is partitioned. We call a partition a group of paragraphs formed by a schema declaring a subset of the state components, together with the actions that use or constrain only these state components. For example, in P above, $Q.st$ and $Q.pps \upharpoonright R.st$ form a partition; and so does $R.st$ together with $R.pps \upharpoonright Q.st$. The law below applies to a partitioned process.

Law 52 (Process splitting)

Let qd and rd stand for the declarations of the processes Q and R, determined by $Q.st$, $Q.pps$, and $Q.act$, and $R.st$, $R.pps$, and $R.act$, respectively, and pd stand for the process declaration above. Then

$$pd = (qd\ rd\ \textbf{process}\ P \mathrel{\widehat=} F(Q, R))$$

provided *$Q.pps$ and $R.pps$ are disjoint with respect to $R.st$ and $Q.st$.* □

Two sets of process paragraphs pps_1 and pps_2 are disjoint with respect to states s_1 and s_2 if, and only if, no command nor CSP action expression in pps_1 refers to components of s_2 or to paragraph names in pps_2; similarly, for pps_2 and components of s_1.

This law can be informally justified by first adding the declarations qd and rd to the left-hand side (as stated by Law 51), and then promoting the context F from main actions to the corresponding processes.

5.2 Process indexing

The second family of laws applies to processes defined using Z's promotion technique. It is based on defining the specification of an abstract data type with its operations and then using this as the type of the elements of a more elaborate data structure like, for instance, a set, a sequence, or a map. For example, one can specify a bank account with its operations and then specify the bank itself as a collection of accounts. Promotion allows an elegant specification of such patterns. The name of the technique comes from the fact that the operations on the collection are defined in terms of those on the element type and a promotion schema.

By convention, the element type is referred to as local, whereas the collection is called global. When the local type is completely encapsulated in the global type, we say that the promotion is free; otherwise it is called constrained [28]. Here we are concerned solely with free promotions.

The proposed family of laws refines a specification structured using a free promotion to an indexed family of processes, each one representing an element of the local type. Considering the bank account example, the law would transform each account into an individual process, and the bank itself into an indexed family of such processes, combined using the interleaving operator.

One of the contributions of this work is to extend Z's promotion technique to *Circus* actions. Below we give an inductive definition of the relevant patterns; L stands for the local process, G for the global process, and *Promotion* for the promotion schema. We observe that we are promoting a process and not a simple abstract data type as in Z; however, we adopt the standard terminology and refer to local and global processes.

For simplicity, we assume that the global state is a function f from elements of an arbitrary type *Range* to elements of the local state; so, a local element is identified in the global state as $f\ i$. Promotion of schema expressions is as in Z.

$$\textbf{promote}(SExp) \mathrel{\widehat=} \exists \Delta L.st \bullet SExp \land Promotion$$

The promotion of *Skip*, *Stop*, and *Chaos* leaves them unchanged.

$$\textbf{promote}(A) \mathrel{\widehat=} A, \qquad \text{for } A \in \{ Skip, Stop, Chaos \}$$

To promote a communication $c.e$, where e is a reference to an element of the local state, we need to receive an extra value: the position i of e in the collection. Therefore, for each channel c, there is a corresponding promoted channel pc that communicates a pair formed by the identifier and the value.

$$\textbf{promote}(c.e \to A) \mathrel{\widehat=} pc?i.\textbf{promote}(e) \to \textbf{promote}(A) \qquad \text{provided } e \text{ is a component of } L.st$$

To promote any other communication, we only promote the communicated expression.

$$\textbf{promote}(c.e \to A) \mathrel{\widehat=} c.\textbf{promote}(e) \to \textbf{promote}(A) \qquad \text{provided } e \text{ is not a component of } L.st$$

Promotion for expressions is defined below; for the other forms of prefixing, the definition is similar. Promotion distributes through the other action operators. For a guarded action, we need to promote the guard. Promotion of predicates has an inductive definition based on promotion of expressions. For parallelism and hiding, the channels are replaced with corresponding promoted channels.

If a variable x is not a local state component, it does not need to be changed.

promote$(x) \mathrel{\widehat=} x$, provided x is not a component of $L.st$

If it is, then we need to access it through the global state. We assume that the position of x in the collection is given by i. This information is received as input, and can be regarded as an extra parameter for **promote**, which we omit for simplicity.

promote$(x) \mathrel{\widehat=} (f\ i).x$, if x is a component of $L.st$

Finally, promotion distributes through the expression operators; the simple but lengthy definition is omitted. If the local state includes components x, y, and z, for instance, a promoted assignment like $(f\ i).x := e$ is an abbreviation for $f := f \oplus \{l : L.st \mid l.x = e \wedge l.y = (f\ i).y \wedge l.z = (f\ i).z \bullet i \mapsto l\}$.

Promotion of multiple assignments may lead to aliasing if more than one component of the local state is being updated. For example, promotion of $x, y := 2, 3$ leads to $(f\ i).x, (f\ i).y := 2, 3$. A specification statement with a frame containing x and y is also problematic, as promotion leads to restrictions on $(f\ i).x$ and $(f\ i).y$ (and $(f'\ i).x$ and $(f'\ i).y$), and we need to promote these restrictions to f (and f'). We assume that actions like these are not used.

In the sequel, we assume that *gd* stands for the following process declaration. The family of process indexing laws applies to processes of this form.

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

  state $State \mathrel{\widehat=} [\, f : Range \pfun L.st \mid pred \,]$

  $L.action_k \uparrow State$

  $L.act \mathrel{\widehat=} \mu X \bullet F(L.action_k);\ X$

  $Promotion \mathrel{\widehat=} [\, \Delta L.st;\ \Delta State;\ i? : Range \mid i? \in \operatorname{dom} f \wedge \theta L.st = f\ i? \wedge f' = f \oplus \{i? \mapsto \theta L.st'\} \,]$

  $action_k \mathrel{\widehat=}$ **promote**$(L.action_k)$

  $\bullet\ (\, \mu X \bullet F(action_k);\ X \,)$

end

As discussed before, the global state component is assumed to be a function from *Range* to a local state *L.st*. Actions $L.action_k$ over the local state do not affect the global state. The main local action *L.act* is defined recursively, as is the main global action. Both have the same structure, but the former uses the actions $L.action_k$ on the local states, and the latter, the corresponding promoted actions $action_k$. There is a promoted action $action_k$ for every local action $L.action_k$. We note that for each channel c used by the $action_k$, the corresponding promoted action uses a corresponding promoted channel *pc*. A topic for further work is the generalisation of the process indexing family of laws in terms of the data structure used in the global state and the main action of both the local and the global states.

We also consider the declaration *ild* below of the indexed process *IL*.

process $IL \mathrel{\widehat=} i : Range \odot L[c\_i := pc]$

The process $i : Range \odot L$ acts on indexed channels c_i, where L acts on a channel c. Like the promoted channels *pc* used in P, they communicate pairs of values: the index and the original value. Above, we rename each channel c_i to *pc*. In this way, we can use *IL* in the refinement of P.

The family of laws for process indexing is as follows. We use the iterated interleaving operator; the iteration index i, whose value ranges in the set *Range*, is used to instantiate the *IL* process. In this way, we interleave several copies of the *IL* process, with each copy communicating a different index in *Range*.

Law 53 (Process indexing)
Let gd and ild be the process declarations above, and let ld be the declaration of the process L. Then,

$gd = ld\ ild$ **process** $G \mathrel{\widehat=} \interleave\, i : Range \odot IL\lfloor i \rfloor$

provided *L.pps and G.pps are disjoint with respect to L.st and G.st.* □

The local state is available through the indexed processes *IL*. Due to interleaving, there is no interference among the individual elements of the collection.

6 Action Refinement

Apart from data refining processes, we are also interested in algorithmically refining actions. The result below justifies the use of action refinement (Definition 1).

Theorem 2 (Soundness of action refinement). *Suppose we have a process P with actions A_1 and A_2. If $A_1 \sqsubseteq_{\mathcal{A}} A_2$, then the identity is a forwards simulation between A_1 and A_2.* $\Box$

This theorem is proved in [22]. It is a consequence of this theorem that we can refine a process by refining its actions. With this result, laws of CSP and Z, for which we have a calculus called ZRC [4], are relevant.

We concentrate here, however, on the laws that relate Z, guarded commands, and CSP constructs, which are novel. In the sequel, we present a few of them; others are listed in Appendix A. As the laws available in the literature for CSP are mostly aimed at characterising its algebraic semantics, we also need new laws of CSP; some are included in Appendix A.

The first set of laws that we present are related to guarded commands. The first law allows us to use a property assured by an assumption as a guard. An assumption $\{\, pre \,\}$ is a special specification statement $: [\, pre, true \,]$. This command does nothing if pre holds, and aborts, otherwise.

Law 61 (Guard Introduction—Assumption)

$\{\, g \,\};\ A = \{\, g \,\};\ g \mathbin{\&} A$ $\Box$

If g does not hold, both commands above abort; if g does hold, then its introduction as a guard of A has no effect. In other words, as g is already guaranteed to hold by the assumption, we can introduce it as a guard, as no extra possibility of deadlock is introduced.

Our second law allows us to eliminate a guard, in the presence of an assumption.

Law 62 (Assumption/Guard—Elimination 1)

$\{\, g_1 \,\};\ (g_2 \mathbin{\&} A) \sqsubseteq \{\, g_1 \,\};\ A$

provided $g_1 \Rightarrow g_2$ $\Box$

If g_1 is such that its validity guarantees that the guard holds, then we can eliminate the guard. Other laws presented in Appendix A (Laws A12 and A13) consider the case in which the assumption guarantees that the guard does not hold, and the possibility of modifying a guard in the presence of an assumption.

We also have new laws that handle guarded actions. The first of these laws introduces a choice of guarded actions from a schema expression $SExp$.

Law 63 (Guard Introduction—Schema Expression)

$SExp \ \sqsubseteq\ \Box\, i \bullet g_i \mathbin{\&} SExp \land [State \mid g_i]$

provided $\mathrm{pre}\ SExp \Rightarrow \bigvee i \bullet g_i$ $\Box$

The proviso guarantees that, if the precondition of $SExp$ holds, then at least one of the guards is enabled. In this case, an action associated with one of the enabled guards g_i is arbitrarily chosen for execution. The behaviour of this action is given by $SExp$ itself, and so we know that the original behaviour is attained; however, we conjoin $SExp$ with a schema that records that g_i holds. This may be useful in further refining $SExp$. If the precondition of $SExp$ does not hold, $SExp$ diverges and $\Box\, i \bullet g_i \mathbin{\&} SExp \land [State \mid g_i]$ may block. Therefore, we have a refinement as well.

If a schema action is expressed as a disjunction, and it is guarded by the precondition of one of the disjuncts, then the other disjunct can be eliminated.

Law 64 (Schema Disjunction Elimination)

$\mathrm{pre}\ SExp_1 \mathbin{\&} (SExp_1 \lor SExp_2) \ \sqsubseteq\ \mathrm{pre}\ SExp_1 \mathbin{\&} SExp_1$ $\Box$

This law is a refinement because, in general, $SExp_1 \lor SExp_2$ allows more nondeterminism than its disjunct $SExp_1$. In Appendix A there are a number of laws for manipulating guards: combining, distributing, and eliminating them.

We also have laws to handle assumptions. First, they distribute over external choice.

Law 65 (Assumption/External Choice—Distribution)

$\{p\};\ (A_1 \Box A_2) = (\{p\};\ A_1) \Box (\{p\};\ A_2)$ $\Box$

They also distribute over internal choice, parallelism, and interleaving. Other laws are already listed as part of ZRC. Two laws used in our case study to introduce and eliminate assumptions are in Appendix A.

A refinement calculus needs to provide a way of introducing programming constructs from specifications. ZRC includes laws to introduce guarded command constructs from schema expressions. Law 63 gives a way of introducing a guarded external choice. The following two laws are concerned with the introduction of sequence and parallelism. In the provisos, we use new notation. The function α gives the set of components of a given schema; it can also be applied to a declaration. The function FV defines the set of free variables of a predicate or expression; DFV determines the set of dashed free variables of a given predicate; finally, $UDFV$ gives the set of undashed free variables of a predicate.

Law 66 (Sequence Introduction—Schema Expression)

$$\begin{array}{l} \quad [\Delta S_1;\ \Delta S_2;\ i? : T \mid preS_1 \wedge preS_2 \wedge CS_1 \wedge CS_2] \\ = \\ \quad [\Delta S_1;\ \Xi S_2;\ i? : T \mid preS_1 \wedge CS_1];\ [\Xi S_1;\ \Delta S_2;\ i? : T \mid preS_2 \wedge CS_2] \end{array}$$

syntactic restrictions

- $\alpha(S_1) \cap \alpha(S_2) = \emptyset$;
- $FV(preS_1) \subseteq \alpha(S_1) \cup \{i?\}$;
- $FV(preS_2) \subseteq \alpha(S_2) \cup \{i?\}$;
- $DFV(CS_1) \subseteq \alpha(S_1')$;
- $DFV(CS_2) \subseteq \alpha(S_2')$;
- $UDFV(CS_2) \cap DFV(CS_1) = \emptyset$. □

This law applies to a schema action over a state composed of two disjoint sets of components specified in the schemas S_1 and S_2. The precondition of the action can be expressed as the conjunction of conditions $preS_1$ and $preS_2$ over the different parts of the state and the input variable(s). Also, the updates on the state are expressed as a conjunction of conditions CS_1 and CS_2 over the final values of the disjoint parts of the state.

The application of this law introduces a sequence of schema actions that update the disjoint parts of the state separately. An extra restriction is required: the final values of the state components of S_2 do not depend on the initial values of those of S_1, as these are potentially changed by the first action in the sequence.

There are no output variables. If we include them, we have to distinguish their specification and determine which action in the sequence is going to produce the output.

ZRC already includes laws to introduce sequences from specifications. This new law is needed in the context of the development of *Circus* programs because it supports the partitioning of the state space of actions. It introduces actions that update disjoint parts of the state.

The next law is concerned with the introduction of parallelism, again from a schema expression. It may seem slightly artificial to introduce communication between schema actions. We must have in mind, however, that these laws are used in stepwise developments, were the schemas are further developed and processes are split. So, the introduction of communication is an interesting step towards a more elaborate structure. This point is illustrated in examples, in the next section.

Law 67 (Parallelism Introduction—Schema Expression)

$$[\Delta S_1;\ \Delta S_2;\ i? : T \mid CS_1(i?, s_2) \land CS_2]$$

$$=$$

$$\begin{array}{l}(c?j?s \to [\Delta S_1;\ \Delta S_2;\ j? : T;\ s? : U \mid CS_1(j?, s?)] \\ \quad [\![\alpha(S_1) \mid \{\!| c |\!\} \mid \alpha(S2)]\!] \\ c!i!s_2 \to [\Delta S_1;\ \Delta S_2 \mid CS_2]) \setminus \{\!| c |\!\}\end{array}$$

syntactic restrictions

- $\alpha(S_1) \cap \alpha(S_2) = \emptyset$;
- *i is an input variable in scope;*
- *$s_2 \in \alpha(S_2)$ and s_2 has type U;*
- $FV(CS_1) \subseteq \alpha(\Delta S_1) \cup \{i?, s_2\}$;
- $FV(CS_2) \subseteq \alpha(\Delta S_2)$;
- *c is a valid channel of type $T \times U$.* □

Like Law 66, this law applies to a schema action over states composed by the conjunction of state schemas S_1 and S_2 with disjoint sets of components. This action takes an input $i?$ used to update the state S_1, but not S_2. The updates of the state are given by the conjunction of CS_1 and CS_2; the former defines the updates on S_1 and the latter, those on S_2. The updates on S_1 depend on the component s_2 of S_2, but the updates on S_2 do not depend on S_1. We use the notation $P(e_1)$ for a predicate P that potentially includes occurrences of an expression e_1; later references to $P(e_2)$ denote the result of replacing e_1 with e_2 in P.

With the application of the law above, we introduce a parallel composition, in which the disjoint parts of the state are updated separately by different schema actions, each restricted to the relevant part of the state. The hidden channel c is used to communicate the input value from one action to another, as well as the state component s_2 that CS_1 uses from S_2. The channel c, of course, needs to have been previously declared and have the appropriate type.

The value of the input variable i and of s_2 is sent by the second action to the first one using channel c. This communication introduces new input variables j and s, that are used by the first action, instead of $i?$ and s_2. Since the second action does not make use of $i?$, it is not in its declaration part.

An important concern in the development of *Circus* programs is the parallelisation of actions and, as a consequence, processes. We present next a law that transforms a sequence into a parallelism. The interesting point about this law is that we have to make sure the transformation does not affect either state transformations or communications.

In the following we use some new notation. The function *usedC* gives the set of channels referred in a given action. The function *usedV* gives the set of used variables: read, but not written. The function *wrtV* gives the set of variables written by a given action. In the case of a schema expression, *wrtV* actually gives the set of variables constrained by the schema. The definition is as follows.

$$wrtV(SExp) = \{v' : DFV(SExp) \mid SExp \neq (\exists\, v' : T \bullet SExp) \land [v, v' : T \mid v' = v] \bullet v\}$$

We use v' to denote the list $v'_1, \ldots, v'_n$ of dashed free variables in *SExp*, and v to denote the corresponding list of undecorated variables $v_1, \ldots, v_n$. The notation $v' : T$ stands for the declaration $v_1 : T_1;\ \ldots;\ v_n : T_n$ of each of the variables in v' with the corresponding types as defined in *SExp*. Informally, if we hide v' in *SExp* and include it back in the signature, we obtain a different schema. This means exactly that the final value of v is changed by *SExp*.

It is unfortunate that *wrtV* is not a purely syntactic function, as its calculation for schema expressions involves theorem proving. On the other hand, a tool can take the pragmatic approach of considering the whole of the state components as the set of written variables of a schema action, and request help from the user only if this worst-view approach fails.

Law 68 (Parallelism Introduction—Sequence 1)

$$A_1;\ A_2(e) \sqsubseteq ((A_1;\ c!e \rightarrow \mathit{Skip}) \llbracket \overline{\mathit{wrtV}(A_2)} \mid \{\!|c|\!\} \mid \mathit{wrtV}(A_2) \rrbracket\ c?y \rightarrow A_2(y)) \setminus \{\!|c|\!\}$$

syntactic restrictions

- *c is a valid channel of type T;*
- *$c \notin \mathit{usedC}(A_1) \cup \mathit{usedC}(A_2)$;*
- *$y \notin FV(A_2)$.*

provided

- *$\mathit{wrtV}(A_1) \cap \mathit{usedV}(A_2) = \emptyset$;*
- *$FV(e) \cap \mathit{wrtV}(A_2\ \mathit{before}\ e) = \emptyset$.* $\square$

To preserve the execution order, this law introduces a communication through a new hidden channel c with a new input variable y. This communication removes direct access of A_2 to the expression e. Even though the order of execution is preserved, in a parallelism both actions have access to the initial value of the variables. Therefore, a proviso is needed: the variables changed by the first action are not used by the second one.

This is strictly a refinement law, not an equality. To understand the reason, suppose A_1 is a schema action that leaves the value of a state component v unconstrained, and that A_2 uses this value. In the sequence, A_2 uses the arbitrary value of v, and in the parallelism, A_2 takes the initial value of v.

The sets of partition variables are defined as $\overline{\mathit{wrtV}(A_2)}$ and $\mathit{wrtV}(A_2)$, where the first is the set of all state components that are not written by A_2. We observe that $\mathit{wrtV}(A_1)$ and $\mathit{wrtV}(A_2)$ are not adequate, as we need to ensure coverage of the whole set of state components. Also, $\mathit{wrtV}(A_1)$ and $\overline{\mathit{wrtV}(A_1)}$ are not appropriate because A_2 has to be given priority to change the value of the variables it modifies. As an example, suppose A_1 changes a state component x to 1, and A_2 changes it to 2; the final value of x has to be 2. The proviso guarantees that A_2 does not use the variable x, but it may update it.

Finally, we need to guarantee that the value of e is not changed by A_2 before it is actually used. The function *before* gives an action that captures the behaviour of its action argument before it has to evaluate the given expression. A worst case view is taken in the definition of *before*, which is defined by recursion on the structure of actions. For example, using the notation $\mathit{SExp}[e]$ to represent the fact that the expression e occurs in the (predicate part of the) schema expression SExp, we have the definition below. We use similar notation for expressions, predicates, actions in general, and others.

$$\mathit{SExp}[e]\ \mathit{before}\ e = \mathit{Skip}$$

If the expression e occurs in the schema, the resulting action is *Skip*: the action that occurs before e and has to be evaluated is *Skip*.

In general, for any action A, if e does not occur in it, the result is A itself.

$$A\ \mathit{before}\ e = A, \text{ if } e \text{ does not occur in } A$$

This covers the actions *Skip*, *Stop*, and *Chaos*, for example.

For prefixing, the definition is as follows.

$$(c?x \rightarrow A)\ \mathit{before}\ e = c?x \rightarrow A, \text{ if } x \in FV(e)$$
$$(c?x \rightarrow A)\ \mathit{before}\ e = c?x \rightarrow (A\ \mathit{before}\ e), \text{ otherwise}$$

If a free variable x of e is reintroduced locally as an input variable, e cannot occur in its scope, since occurrences of x there are references to the input variable, and not to the same variable mentioned in e. If the input variable is not free in e, we proceed to the prefixed action A.

The definition for the other constructs is not very illuminating, except, perhaps, for sequence.

$$(A_1[e];\ A_2)\ \mathit{before}\ e = A_1[e]\ \mathit{before}\ e$$
$$(A_1;\ A_2)\ \mathit{before}\ e = A_1;\ (A_2\ \mathit{before}\ e), \text{ if } e \text{ does not occur in } A_1$$

If e occurs in the first action, the second one does not need to be considered.

Typically, with the application of Law 68 we want to avoid direct access of A_2 to a state component. So, we consider in Law A20 in Appendix A the particular case in which e is a variable. However, the generality above is necessary, because we work with structured variables, like sequences, and we do not want to communicate the entire component. If we have just a variable, we can have less restrictive provisos.

As we split and combine processes, it is often useful to manipulate channels accordingly. The following law allows us to combine channels: if all communications between two actions occur sequentially through channels c_1 and c_2, we can use only one channel c_3 to communicate the same values.

Law 69 (Channel Combination)

$$(A_1[c_1.com_1 \rightarrow c_2.com_2 \rightarrow B_1] \llbracket ns_1 \mid \{\!| c_1, c_2 |\!\} \mid ns_2 \rrbracket A_2[c_1.com_3 \rightarrow c_2.com_4 \rightarrow B_2]) \setminus \{\!| c_1, c_2 |\!\}$$
$$=$$
$$(A_1[c_3.com_1.com_2 \rightarrow B_1] \llbracket ns_1 \mid \{\!| c_3 |\!\} \mid ns_2 \rrbracket A_2[c_3.com_3.com_4 \rightarrow B_2]) \setminus \{\!| c_3 |\!\}$$

syntactic restrictions

- *all occurrences of* c_1 *and* c_2 *in* A_1 *and* A_2 *are as explicitly stated;*
- c_3 *is a valid channel of the appropriate type.* □

Obviously, the new channel has to have the appropriate type to communicate pairs of values: the first element is the value originally communicated by c_1, and the second, the value communicated by c_2.

The next laws allow the extension and reduction of channel synchronisation sets.

Law 610 (Channel Extension 1)

$$A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 = A_1 \llbracket ns_1 \mid cs \cup \{\!| c |\!\} \mid ns_2 \rrbracket A_2$$

provided $c \notin usedC(A_1) \cup usedC(A_2)$ □

The channel has to be new, in the sense that it is not used in the actions in parallel. Applied from right to left, this law can also be used to reduce channel sets.

The next law is more elaborate: it extends the synchronisation set and uses the new channel to communicate a value e from one of the parallel actions to the other.

Law 611 (Channel Extension 2)

$$A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2(e) = (c!e \rightarrow A_1 \llbracket ns_1 \mid cs \cup \{\!| c |\!\} \mid ns_2 \rrbracket c?x \rightarrow A_2(x)) \setminus \{\!| c |\!\}$$

syntactic restrictions

- c *is a valid channel of the appropriate type;*
- $c \notin usedC(A_1) \cup usedC(A_2)$*;*
- $x \notin FV(A_2)$*.*

provided $FV(e) \cap wrtV(A_2 \; before \; e) = \emptyset$ □

The idea is that the access of A_2 to e is removed. As in Law 68, we have to make sure that the value of e is not changed before it is actually used in A_2.

The following law eliminates unnecessary extra synchronisation between two parallel actions.

Law 612 (Synchronisation Elimination)

$$(\Box\, i \bullet g_i \,\&\, c_i.ccom_i \rightarrow d_i.acom_i \rightarrow A_i)$$
$$\llbracket ns_1 \mid cs \cup \{\!| i \bullet c_i |\!\} \mid ns_2 \rrbracket$$
$$(\Box\, i \bullet g_i \,\&\, c_i.ccom_i \rightarrow d_i.bcom_i \rightarrow B_i)$$
$$=$$
$$(\Box\, i \bullet d_i.acom_i \rightarrow A_i) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\Box\, i \bullet g_i \,\&\, c_i.ccom_i \rightarrow d_i.bcom_i \rightarrow B_i)$$

provided $\{i \bullet c_i\} \cap usedC(A_i) \cup usedC(B_i) = \emptyset$ □

It is not necessary for parallel actions to synchronise in sequence on c_i and d_i. Just one of the synchronisations is necessary to ensure the order of execution of the actions. An important point, of course, is that the communication parameter $ccom_i$ of c is common to both actions. This means that the communication between them is always possible; any restrictions are imposed by their environment. In other words, even though the channels c_i are used to communicate values, from the point of view of the parallel actions, they are just extra synchronisation points. At the same time we eliminate the communications, we can also eliminate the associated guards. If they are false, the guarded action blocks, the communication does not occur, and the matching actions $d_i.acom_i \rightarrow A_i$ cannot proceed either.

As already mentioned, laws of CSP are also relevant for *Circus*. In our case study, we use some, which are listed in Appendix A. We include laws for prefixing that relate this construct to sequence, choice, parallelism, and interleaving. In those laws, the function α is applied to a communication; it gives the input variables the communication introduces. These are the variables declared by the communication, which are in scope for the action that follows the communication in a prefixing.

We also have laws for external choice and parallelism. The following is a kind of step law for parallelism, where one of the actions put in parallel is a sequence. The law determines conditions in which the first step is the first action of the sequence.

Law 613 (Parallelism/Sequence—Step)

$$(A_1;\ A_2) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3 = A_1;\ (A_2 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3)$$

provided

- $initials(A_3) \subseteq cs$;
- $cs \cap usedC(A_1) = \emptyset$;
- $wrtV(A_1) \cap usedV(A_3) = \emptyset$ □

The functions *initials* gives the set of channels through which a given action may communicate first. The definition of this function is similar to that given in the CSP operational semantics presented in [21].

The above law requires that the set of channels through which A_3 is initially prepared to communicate is included in the synchronisation set. Therefore, A_3 cannot proceed independently. On the other hand, A_1 does not communicate through channels in the synchronisation set. Therefore, it can proceed independently and necessarily goes first. We need, however, to impose a further restriction related to the state changes. The set of variables written by A_1 and used by A_3 cannot overlap, since in the parallel composition A_3 acts on the initial value of the variables, and in the sequence, it acts on the values assigned by A_1.

An interesting law that relates parallelism and external choice is presented below. This is an exchange law that allows us to rearrange parallel actions; it is an important support for the task of splitting processes.

Law 614 (Parallelism/External Choice—Exchange)

$$(A_1 \llbracket cs \rrbracket A_2) \Box (B_1 \llbracket cs \rrbracket B_2) = (A_1 \Box B_1) \llbracket cs \rrbracket (A_2 \Box B_2)$$

provided $A_1 \llbracket cs \rrbracket B_1 = A_2 \llbracket cs \rrbracket B_1 = Stop$ □

The choice on the left-hand action is for the parallel execution of A_1 and A_2, or B_1 and B_2. On the right-hand action, there is also the possibility of parallel execution of A_1 and B_2, or B_1 and A_2. The proviso, however, ensures that these choices lead to deadlock, so they are not a possibility.

In our case study, we also use well-known laws of hiding and recursion. Finally, we make use of some very simple unit and zero laws. These are all listed in Appendix A.

7 Case study

To illustrate our refinement strategy, we develop an implementation for the bounded, reactive, buffer introduced in Section 2 using the laws presented previously. The structure of the final implementation is a ring of cells with a central controller and a cached head. The entire development exercises two iterations of the strategy described in Section 3.

First, by data refinement, the bounded sequence is replaced by a cache that stores the head of the buffer when it is not empty, and a ring (circular array) that stores the tail of the original bounded buffer. The simulation relation was presented in Section 3. Next, the state changes associated with the input and output actions are decomposed into small units to deal with the cases in which the cache is updated and those in which the ring is updated. This is a first step towards isolating access to the ring. Afterwards, action refinements are carried out to separate those actions that access the ring from those that access the other state components. Next, we decompose the original process into two: a controller and a centralised ring. This completes a first iteration of the strategy.

Through a second data refinement step, the centralised ring process is redesigned as a promotion of individual ring cells. In this case, no additional action refinement is necessary. Finally, we decompose the ring process into the interleaving of ring cell processes, each one storing a single value. This concludes the second iteration of the proposed strategy and the development of the entire case study. The following sections present the development in detail.

7.1 Data refinement: a centralised ring buffer

Our first step is a data refinement, in which we introduce a *cache* and a *ring* to represent the internal state of the process *Buffer*. In Section 3 we presented the new concrete state, which we reproduce below.

process *Buffer* $\widehat{=}$ **begin**
state

$$
\begin{array}{|l}
\textit{CBufferState} \\
\hline
\textit{size} : 0 \mathrel{..} \textit{maxbuff};\ \textit{ringsize} : 0 \mathrel{..} \textit{maxring} \\
\textit{cache} : \mathbb{N};\ \textit{ring} : \operatorname{seq} \mathbb{N} \\
\textit{top}, \textit{bot} : 1 \mathrel{..} \textit{maxring} \\
\hline
\textit{ringsize} \bmod \textit{maxring} = (\textit{top} - \textit{bot}) \bmod \textit{maxring} \\
\textit{ringsize} = \textit{max}\{0, \textit{size} - 1\} \\
\#\textit{ring} = \textit{maxring} \\
\hline
\end{array}
$$

This new state requires a new description for the main action. To establish that our new *Buffer* is a refinement of that presented in Section 2, we prove that the new main action is related to that of the original process by a forwards simulation (Theorem 1). The retrieve relation *RetrBuffer* from Section 3 is reproduced below.

$$\textit{RetrBuffer} \mathrel{\widehat{=}} [\, \textit{BufferState};\ \textit{CBufferState} \mid \textit{buff} = (1 \mathrel{..} \textit{size}) \lhd (\langle \textit{cache} \rangle \frown ((\textit{bot} - 1) \ll \textit{ring}))\,]$$

Instead of proposing a new action from scratch, we consider the schema actions and rely on the fact that forwards simulation distributes through the action constructors (laws of Section 4). The new actions have the same structure as the original ones, but use the new schema actions.

The first schema action is the initialisation *BufferInit*. Initially, the buffer is empty and so has size zero; for the concrete initialisation, we choose some suitable values for *top* and *bot*.

$$\textit{CBufferInit} \mathrel{\widehat{=}} [\, \textit{CBufferState}' \mid \textit{size}' = 0 \land \textit{bot}' = 1 \land \textit{top}' = 1\,]$$

To prove that this new initialisation is related to *BufferInit* by forwards simulation, we need to apply Law 41, which considers schema actions. In this case, however, the provisos are simplified because initialisation schemas do not include the state components that represent the before state and have true as precondition. We actually obtain the standard proviso for refinement of initialisations in Z. All we have to prove is that

$$
\begin{array}{l}
\forall\, \textit{BufferState};\ \textit{CBufferState};\ \textit{CBufferState}' \bullet \\
\qquad \textit{RetrBuffer} \land \textit{CBufferInit} \Rightarrow (\exists\, \textit{BufferState}' \bullet \textit{RetrBuffer}' \land \textit{BufferInit})
\end{array}
$$

This is a simple proof: the one-point rule, and the fact that $(\emptyset \lhd s) = \langle\rangle$ for any sequence s, can be used to reduce $\exists\, \textit{CBufferState}' \bullet \textit{RetrBuffer}' \land \textit{BufferInit}$ to *true*. As this is on the right-hand side of the implication above, the proof-obligation is discharged.

The concrete input action corresponding to *InputCmd* has to consider whether the buffer is empty or not. If it is empty, then the input must be kept in the cache; if it is non-empty, then it must be passed on to the appropriate ring cell. When the input is cached, the *top* and *bot* indices do not change.

$$\textit{CacheInput} \mathrel{\widehat{=}} [\, \Delta \textit{CBufferState};\ x? : \mathbb{N} \mid \textit{size} = 0 \land \textit{size}' = 1 \land \textit{cache}' = x? \land \textit{bot}' = \textit{bot} \land \textit{top}' = \textit{top}\,]$$

When the input is passed on to the ring, the corresponding value is stored and the *top* index advances.

$$\begin{array}{|l}
\textit{StoreInput} \\
\Delta \mathit{CBufferState} \\
x? : \mathbb{N} \\
\hline
0 < \mathit{size} < \mathit{maxbuff} \\
\mathit{size}' = \mathit{size} + 1 \land \mathit{cache}' = \mathit{cache} \\
\mathit{bot}' = \mathit{bot} \land \mathit{top}' = (\mathit{top} \bmod \mathit{maxring}) + 1 \\
\mathit{ring}' = \mathit{ring} \oplus \{\mathit{top} \mapsto x?\}
\end{array}$$

The overall state change caused by an input to the buffer can be captured by the disjunction below.

$\mathit{CInputCmd} \mathrel{\widehat=} \mathit{CacheInput} \lor \mathit{StoreInput}$

Again, we can justify this step with an application of Law 41. The proof-obligations are simple, if long; this is standard Z data refinement. We observe that the precondition of *CInputCmd* is the disjunction of the preconditions of *CacheInput* and *StoreInput*, and amounts to $\mathit{size} < \mathit{maxbuff}$.

Since *CInputCmd* simulates *InputCmd*, we can apply Laws 44 and 42 to obtain the following simulation of *Input*. The structure of *Input* is preserved and *InputCmd* is replaced with *CInputCmd*.

$\mathit{CInput}_0 \mathrel{\widehat=} \mathit{size} < \mathit{maxbuff}\ \&\ \mathit{input}?x \rightarrow \mathit{CInputCmd}$

In this case, the guard is not changed and the first proviso of Law 44 is trivial. In the next development step (Section 7.2) this action is refined to refer directly to the *CacheInput* and *StoreInput* operations.

The refinement of *Output* is similar; as for the input, there is a case analysis. The output always comes from the cache, which must be replaced if the ring is non-empty. If the ring is empty, we have $\mathit{size} = 1$; in this case *size* is reset and nothing else changes.

$$\begin{array}{|l}
\textit{NoNewCache} \\
\Delta \mathit{CBufferState} \\
\hline
\mathit{size} = 1 \\
\mathit{size}' = 0 \land \mathit{cache}' = \mathit{cache} \\
\mathit{bot}' = \mathit{bot} \land \mathit{top}' = \mathit{top} \land \mathit{ring}' = \mathit{ring}
\end{array}$$

If the ring is non-empty, then an element obtained from the ring is stored in the cache; the index *bot* is advanced, and the index *top* is unchanged.

$$\begin{array}{|l}
\textit{StoreNewCache} \\
\Delta \mathit{CBufferState} \\
\hline
\mathit{size} > 1 \\
\mathit{size}' = \mathit{size} - 1 \land \mathit{cache}' = \mathit{ring}\ \mathit{bot} \\
\mathit{bot}' = (\mathit{bot} \bmod \mathit{maxring}) + 1 \land \mathit{top}' = \mathit{top} \\
\mathit{ring}' = \mathit{ring}
\end{array}$$

The overall state change caused by an output from the buffer can be captured by the disjunction below.

$\mathit{COutputCmd} \mathrel{\widehat=} \mathit{NoNewCache} \lor \mathit{StoreNewCache}$

Law 41 can be used to justify that *COutputCmd* simulates *OutputCmd*. Laws 44 and 43 justify that $\mathit{COutput}_0$ below simulates *Output*.

$\mathit{COutput}_0 \mathrel{\widehat=} \mathit{size} > 0\ \&\ \mathit{output}!(\mathit{cache}) \rightarrow \mathit{COutputCmd}$

The output expression *head buff* is replaced with *cache*. This is justified by *RetrBuffer*, which amounts to $\mathit{buff} = (1 \mathinner{\ldotp\ldotp} \mathit{size}) \lhd (\langle \mathit{cache} \rangle \frown ((\mathit{bot} - 1) \ll \mathit{ring}))$, which implies that *head buff* is *cache*, since, in the context of the communication, $\mathit{size} > 0$.

Finally, the main action of the centralised ring buffer also has the same structure of that of the original *Buffer*. We simply replace the original input and output actions with those presented above.

$\bullet\ \mathit{CBufferInit};\ \mu X \bullet (\mathit{CInput}_0 \mathrel{\Box} \mathit{COutput}_0);\ X$
end

This step can be justified applying Laws 45 and 48.

7.2 Action refinement: decompose input and output actions

In the previous step, we have structured the state change resulting from an input to, or output from, the buffer in terms of separate operation schemas. This reflected a case analysis on whether the *ring* or just the *cache* needed to be accessed. Nevertheless, the $CInput_0$ as well as the $COutput_0$ actions still refer to the compound operation which combines the two cases. This was intentional: we kept an explicit correspondence between the abstract and concrete operations in order to allow a simpler justification of the data refinement.

We perform a simple design step to promote the case analysis from the operations on the state to the actions. We show only the refined actions and, afterwards, give Lemma 1 that justifies the refinement.

Input is enabled when the buffer is not full; in that case, the behaviour depends on whether the buffer is empty or not. If the buffer is empty, the corresponding state change is captured by *CacheInput*; if it is not, the behaviour is captured by *StoreInput*.

$$CInput \mathrel{\widehat=} size < maxbuff \mathbin{\&} input?x \rightarrow (size = 0 \mathbin{\&} CacheInput \mathbin{\Box} size > 0 \mathbin{\&} StoreInput)$$

The output action is enabled when there is something in the buffer; the subsequent behaviour depends on whether the ring is empty or not.

$$COutput \mathrel{\widehat=} size > 0 \mathbin{\&} output!cache \rightarrow (size > 1 \mathbin{\&} StoreNewCache \mathbin{\Box} size = 1 \mathbin{\&} NoNewCache)$$

The following lemma formalises the refinement of the input action.

Lemma 1 (Refinement of $CInput_0$). $CInput_0 \sqsubseteq_{\mathcal{A}} CInput$

Proof

$$\begin{array}{ll}
& LHS \\
\sqsubseteq_{\mathcal{A}} & size < maxbuff \mathbin{\&} input?x \rightarrow size = 0 \mathbin{\&} CInputCmd \wedge [\, CBufferState \mid size = 0 \,] \qquad \textit{[Law 63]} \\
& \qquad\qquad\qquad\qquad \Box\; size > 0 \mathbin{\&} CInputCmd \wedge [\, CBufferState \mid size > 0 \,] \\
\sqsubseteq_{\mathcal{A}} & RHS \qquad \textit{[schema and predicate calculus]}
\end{array}$$

□

The refinement of $COutput_0$ is analogous; its proof follows from the same laws above.

7.3 Action refinement: controller and ring partitions

The purpose of this refinement step is to reorganise the internal structure of the *Buffer* with the aim of obtaining two independent sets of paragraphs (partitions). One set of paragraphs accesses exclusively the *ring* and is, in the next step, promoted into an independent process. The other set of paragraphs accesses the remaining components, and is, also in the next step, turned into a controller process that remains unchanged up to the end of the development.

In some circumstances, this partitioning of the state space is not direct. For example, the *StoreInput* operation updates both *top* and *ring*. Splitting it into two operations is not immediate, because the operation that is concerned with updating the *ring* needs the input value ($x?$) and the current value of *top*. The main design tool to solve such data dependencies is introduction of communication. We need two new channels, which are used to exchange information between the ring and the controller processes.

$$\textbf{channel}\ write, read : (1 \mathinner{..} maxring) \times \mathbb{N}$$

These channels are hidden in the *Buffer* design and implementation. These declarations allow multi-part communication; we use *write* and *read* to communicate pairs of values, as in $write.i?x$.

We decompose the state of the buffer into two separate schemas, each one is a state space for a set of process paragraphs.

$$\begin{array}{|l}
\multicolumn{1}{l}{ControllerState} \\ \hline
size : 0 \mathinner{..} maxbuff;\ ringsize : 0 \mathinner{..} maxring \\
cache : \mathbb{N} \\
top, bot : 1 \mathinner{..} maxring \\ \hline
ringsize = max\{0, size - 1\} \\
ringsize \bmod maxring = (top - bot) \bmod maxring \\ \hline
\end{array}$$

$$RingState \mathrel{\widehat=} [\, ring : \mathrm{seq}\,\mathbb{N} \mid \#ring = maxring \,]$$

$$\textbf{state } CBufferState \mathrel{\widehat=} ControllerState \land RingState$$

The first set of paragraphs has *ControllerState* as its state space, whilst not constraining *RingState*. The initialisation is for an empty buffer.

$$ControllerInit \mathrel{\widehat=} [\, ControllerState';\ \ RingState' \mid size' = 0 \land bot' = 1 \land top' = 1 \,]$$

In the case the buffer is empty, an input is cached. The ring indices do not change and the buffer now contains a single item.

$$\begin{array}{|l}
\multicolumn{1}{l}{CacheInput} \\
\hline
\Delta ControllerState \\
\Xi RingState \\
x? : \mathbb{N} \\
\hline
size = 0 \\
cache' = x? \land size' = 1 \\
bot' = bot \land top' = top \\
\hline
\end{array}$$

If the buffer is not empty, the *cache* is not changed; the indices and the size of the ring are updated, but the ring itself is not changed.

$$\begin{array}{|l}
\multicolumn{1}{l}{StoreInputController} \\
\hline
\Delta ControllerState \\
\Xi RingState \\
\hline
0 < size < maxbuff \\
size' = size + 1 \land cache' = cache \land bot' = bot \land top' = (top \mathbin{\mathsf{mod}} maxring) + 1 \\
\hline
\end{array}$$

The action below gets the new input and, if necessary, sends it to the ring along with the position *top* in which the input is to be stored. This communication is through the channel *write*.

$$\begin{array}{l}
InputController \mathrel{\widehat=} \\
\qquad size < maxbuff \mathbin{\&} input?x \to \begin{array}[t]{l} size = 0 \mathbin{\&} CacheInput \\ \Box \\ size > 0 \mathbin{\&} write.top!x \to StoreInputController \end{array}
\end{array}$$

Concerning output, the value in the *cache* is always the one which is communicated. If the buffer has a single element, communicating this element is the only relevant action.

$$\begin{array}{|l}
\multicolumn{1}{l}{NoNewCache} \\
\hline
\Delta ControllerState \\
\Xi RingState \\
\hline
size = 1 \\
size' = 0 \land cache' = cache \land bot' = bot \land top' = top \\
\hline
\end{array}$$

Nevertheless, if there are elements stored in the *ring*, the value $x?$ at position *bot* must be recovered. In this case, the *cache* is updated with this value and *bot* is incremented.

$$\begin{array}{|l}
\multicolumn{1}{l}{StoreNewCacheController} \\
\hline
\Delta ControllerState \\
\Xi RingState \\
x? : \mathbb{N} \\
\hline
size > 1 \\
size' = size - 1 \land cache' = x? \land bot' = (bot \mathbin{\mathsf{mod}} maxring) + 1 \land top' = top \\
\hline
\end{array}$$

The following action captures the necessary case analysis for output. The channel *read* is used to recover the element $x?$ at position *bot* in the ring.

$$
\begin{array}{l}
\mathit{OutputController} \mathrel{\widehat=} \\
\quad \mathit{size} > 0 \mathrel{\&} \mathit{output}!\mathit{cache} \rightarrow \begin{array}[t]{l} \mathit{size} > 1 \mathrel{\&} \mathit{read}.\mathit{bot}?x \rightarrow \mathit{StoreNewCacheController} \\ \Box \\ \mathit{size} = 1 \mathrel{\&} \mathit{NoNewCache} \end{array}
\end{array}
$$

The behaviour of the controller is as follows.

$$\mathit{ControllerAction} \mathrel{\widehat=} \mathit{ControllerInit};\ \mu X \bullet (\mathit{InputController} \Box \mathit{OutputController});\ X$$

After initialisation, inputs and outputs are offered repeatedly, whenever possible.

The second set of paragraphs has as its state space *RingState*, whilst preserving *ControllerState*. The next action stores a value in the *ring*.

$$
\begin{array}{|l}
\hline
\mathit{StoreRingCmd} \\
\Delta \mathit{RingState} \\
\Xi \mathit{ControllerState} \\
i? : 1 \mathrel{..} \mathit{maxring};\ x? : \mathbb{N} \\
\hline
\mathit{ring}' = \mathit{ring} \oplus \{i? \mapsto x?\} \\
\hline
\end{array}
$$

Although all state components are in scope, we restrict direct access to *RingSate* and receive the current value of *top* through the *write* channel.

$$\mathit{StoreRing} \mathrel{\widehat=} \mathit{write}?i?x \rightarrow \mathit{StoreRingCmd}$$

To send the value stored at a given position of the *ring* requires no state change.

$$\mathit{NewCacheRing} \mathrel{\widehat=} \mathit{read}?i!(\mathit{ring}\ i) \rightarrow \mathit{Skip}$$

In a multi-part communication as $\mathit{read}?i!(\mathit{ring}\ i)$, it is possible to use the input value i to express the output value $\mathit{ring}\ i$, since i is immediately in scope after the input.
The ring repeatedly offers the external choice between *StoreRing* and *NewCacheRing* actions.

$$\mathit{RingAction} \mathrel{\widehat=} \mu X \bullet (\mathit{StoreRing} \Box \mathit{NewCacheRing});\ X$$

The control behaviour of the process *Buffer* is given by the parallel execution of the controller and the ring, hiding the internal channels.

$$\bullet\ (\mathit{ControllerAction} \llbracket \alpha(\mathit{ControllerState}) \mid \{\!| \mathit{write}, \mathit{read}, |\!\} \mid \alpha(\mathit{RingState}) \rrbracket \mathit{RingAction}) \setminus \{\!| \mathit{write}, \mathit{read} |\!\}$$

end

This is actually a significant refinement step, but it involves no change of data representation. To prove that it is valid, we need to compare the above main action to that of the data refined buffer. The relevant tools are the action refinement laws.

We start carrying out some refinement steps in the actions *CInput* and *COutput*. The purpose is to linearise guards and prefixes (through their distribution over external choice) and parallelise the state update to separate the controller from the ring components. These transformations are captured by the following lemmas, which are used in Theorem 3 to justify this refinement step. Their proofs illustrate the application of the laws for introducing parallelism. We also reference some simple laws of CSP which are presented in Appendix A. From the identification of each law that justifies a refinement step, it is clear whether it is presented in Section 6 or in Appendix A.

Lemma 2 (Refinement of CInput).

$$
\begin{array}{l}
\mathit{CInput} \sqsubseteq_{\mathcal{A}} \begin{array}[t]{l}
\mathit{size} = 0 \mathrel{\&} \mathit{input}?x \rightarrow \mathit{CacheInput} \\
\Box \\
0 < \mathit{size} < \mathit{maxbuff} \mathrel{\&} \mathit{input}?x \rightarrow \\
\quad (\mathit{write}?i?y \rightarrow \mathit{StoreRingCmd} \\
\qquad \llbracket \alpha(\mathit{RingState}) \mid \{\!| \mathit{write} |\!\} \mid \alpha(\mathit{ControllerState}) \rrbracket \\
\quad\ \mathit{write}.\mathit{top}!x \rightarrow \mathit{StoreInputController}) \setminus \{\!| \mathit{write} |\!\}
\end{array}
\end{array}
$$

Proof

$$\begin{array}{l}
\mathit{LHS} \\
\sqsubseteq_{\mathcal{A}} \mathit{size} < \mathit{maxbuff} \mathrel{\&} (\mathit{size} = 0 \mathrel{\&} \mathit{input}?x \rightarrow \mathit{CacheInput} \mathrel{\Box} \mathit{size} > 0 \mathrel{\&} \mathit{input}?x \rightarrow \mathit{StoreInput}) \\
\hfill [\text{Law A22}] \\
\sqsubseteq_{\mathcal{A}} \mathit{size} = 0 \mathrel{\&} \mathit{input}?x \rightarrow \mathit{CacheInput} \mathrel{\Box} 0 < \mathit{size} < \mathit{maxbuff} \mathrel{\&} \mathit{input}?x \rightarrow \mathit{StoreInput} \quad [\text{Laws A3, A1}] \\
\sqsubseteq_{\mathcal{A}} \mathit{RHS} \hfill [\text{Law 67 and schema refinement}]
\end{array}$$

□

We proceed in much the same way for the output action.

Lemma 3 (Refinement of COutput).

$$\begin{array}{l}
\mathit{COutput} \sqsubseteq_{\mathcal{A}} \mathit{size} > 1 \mathrel{\&} \mathit{output}!\mathit{cache} \rightarrow \\
\qquad (\mathit{read}?i!(\mathit{ring}\ i) \rightarrow \mathit{Skip} \\
\qquad\quad [\![\alpha(\mathit{RingState}) \mid \{\!|\ \mathit{read}\ |\!\} \mid \alpha(\mathit{ControllerState})]\!] \\
\qquad \mathit{read}!\mathit{bot}?x \rightarrow \mathit{StoreNewCacheController}) \setminus \{\!|\ \mathit{read}\ |\!\} \\
\quad \Box \\
\quad \mathit{size} = 1 \mathrel{\&} \mathit{output}!\mathit{cache} \rightarrow \mathit{NoNewCache}
\end{array}$$

Proof *In this proof we make use of intermediate channels that need to be declared.*

$$\textbf{channel}\ \mathit{read}_1 : 1 \mathrel{..} \mathit{maxring};\ \mathit{read}_2 : \mathbb{N};$$

The communications over these channels are later combined so that read is used instead of read_1 *and* read_2*. Unused channel declarations can be introduced and removed from a program, as much as processes can (see Law 51). When the uses of* read_1 *and* read_2 *are eliminated, their declaration can be eliminated. This use of channels for development purposes only is similar to that of logical constants in [18, 4].*

$$\begin{array}{l}
\mathit{LHS} \\
\sqsubseteq_{\mathcal{A}} \mathit{size} > 0 \mathrel{\&} (\mathit{size} > 1 \mathrel{\&} \mathit{output}!\mathit{cache} \rightarrow \mathit{StoreNewCache} \mathrel{\Box} \mathit{size} = 1 \mathrel{\&} \mathit{output}!\mathit{cache} \rightarrow \mathit{NoNewCache}) \\
\hfill [\text{Law A22}] \\
\sqsubseteq_{\mathcal{A}} \mathit{size} > 1 \mathrel{\&} \mathit{output}!\mathit{cache} \rightarrow \mathit{StoreNewCache} \mathrel{\Box} \mathit{size} = 1 \mathrel{\&} \mathit{output}!\mathit{cache} \rightarrow \mathit{NoNewCache} \\
\hfill [\text{Laws A3 and A1}] \\
\sqsubseteq_{\mathcal{A}} \mathit{size} > 1 \mathrel{\&} \mathit{output}!\mathit{cache} \rightarrow (\mathit{Skip};\ \mathit{read}_2!(\mathit{ring}\ \mathit{bot}) \rightarrow \mathit{Skip} \hfill [\text{Laws A32 and 68}] \\
\qquad\qquad [\![\alpha(\mathit{RingState}) \mid \{\!|\ \mathit{read}_2\ |\!\} \mid \alpha(\mathit{ControllerState})]\!] \\
\qquad\quad \mathit{read}_2?x \rightarrow \mathit{StoreNewCacheController}) \setminus \{\!|\ \mathit{read}_2\ |\!\} \\
\quad \Box \\
\quad \mathit{size} = 1 \mathrel{\&} \mathit{output}!\mathit{cache} \rightarrow \mathit{NoNewCache} \\
\sqsubseteq_{\mathcal{A}} \mathit{size} > 1 \mathrel{\&} \mathit{output}!\mathit{cache} \rightarrow \hfill [\text{Laws A32, 611}] \\
\qquad ((\mathit{read}_1?i \rightarrow \mathit{read}_2!(\mathit{ring}\ i) \rightarrow \mathit{Skip} \\
\qquad\quad [\![\alpha(\mathit{RingState}) \mid \{\!|\ \mathit{read}_1, \mathit{read}_2\ |\!\} \mid \alpha(\mathit{ControllerState})]\!] \\
\qquad \mathit{read}_1!\mathit{bot} \rightarrow \mathit{read}_2?x \rightarrow \mathit{StoreNewCacheController}) \setminus \{\!|\ \mathit{read}_1\ |\!\}) \setminus \{\!|\ \mathit{read}_2\ |\!\}) \\
\quad \Box \\
\quad \mathit{size} = 1 \mathrel{\&} \mathit{output}!\mathit{cache} \rightarrow \mathit{NoNewCache} \\
\sqsubseteq_{\mathcal{A}} \mathit{RHS} \hfill [\text{Laws A28, 69}]
\end{array}$$

□

The initialisation of the controller (*ControllerInit*) refines the initialisation of the buffer (*CBufferInit*), since all the state elements are initialised with the same values, except for the ring, which can take an arbitrary value upon initialisation; but this is the case in both schemas. The following lemma records this refinement.

Lemma 4 (Initialisation).

$$\mathit{CBufferInit} \sqsubseteq_{\mathcal{A}} \mathit{ControllerInit}$$

□

We proceed to justify the proof obligation for this development step. In the development below, all occurrences of the parallel operator are of the form $A_1 \llbracket \alpha(RingState) \mid \{\!| cs |\!\} \mid \alpha(ControllerState) \rrbracket A_2$, for actions A_1 and A_2, and synchronisation set cs. For conciseness, we omit the alphabets on both sides and write simply $A_1 \llbracket \{\!| cs |\!\} \rrbracket A_2$.

Theorem 3 (Refinement of the centralised buffer action).

$$\begin{array}{l} \quad CBufferInit;\ \mu X \bullet (CInput \mathbin{\Box} COutput);\ X \\ \sqsubseteq_{\mathcal{A}} \\ \quad (ControllerAction \llbracket \{\!| write, read |\!\} \rrbracket RingAction) \setminus \{\!| write, read |\!\} \end{array}$$

Proof *The first step of the proof applies the previous lemmas to introduce parallelism in the CInput and in the COutput actions, as well as to replace the initialisation of the buffer with that of the controller.*

$$\begin{array}{ll} LHS & \\ \sqsubseteq_{\mathcal{A}} ControllerInit; & \text{[Lemmas 2, 3, and 4.]} \\ \quad \mu X \bullet (size = 0 \mathbin{\&} input?x \rightarrow CacheInput & \\ \qquad\quad \Box & \\ \qquad\quad 0 < size < maxbuff \mathbin{\&} input?x \rightarrow (write?i?y \rightarrow StoreRingCmd & \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \llbracket \{\!| write |\!\} \rrbracket & \\ \qquad\qquad\qquad\qquad\qquad\qquad write.top!x \rightarrow StoreInputController) \setminus \{\!| write |\!\} & \\ \qquad\quad \Box & \\ \qquad\quad size > 1 \mathbin{\&} output!cache \rightarrow (read?i!(ring\ i) \rightarrow Skip & \\ \qquad\qquad\qquad\qquad\qquad\qquad \llbracket \{\!| read |\!\} \rrbracket & \\ \qquad\qquad\qquad\qquad\qquad read!bot?x \rightarrow StoreNewCacheController) \setminus \{\!| read |\!\} & \\ \qquad\quad \Box & \\ \qquad\quad size = 1 \mathbin{\&} output!cache \rightarrow NoNewCache);\ X & \end{array}$$

The second step distributes guards and prefixes over parallelism; for that, first the scopes of the hidings have to be extended to encompass the prefixes and the guards.

$$\begin{array}{ll} \sqsubseteq_{\mathcal{A}} ControllerInit; & \text{[Laws A29, A24, A5]} \\ \quad \mu X \bullet (size = 0 \mathbin{\&} input?x \rightarrow CacheInput & \\ \qquad\quad \Box & \\ \qquad\quad (0 < size < maxbuff \mathbin{\&} input?x \rightarrow write?i?y \rightarrow StoreRingCmd & \\ \qquad\qquad \llbracket \{\!| write, input |\!\} \rrbracket & \\ \qquad\quad\ 0 < size < maxbuff \mathbin{\&} input?x \rightarrow write.top!x \rightarrow StoreInputController) \setminus \{\!| write |\!\} & \\ \qquad\quad \Box & \\ \qquad\quad (size > 1 \mathbin{\&} output!cache \rightarrow read?i!(ring\ i) \rightarrow Skip & \\ \qquad\qquad \llbracket \{\!| read |\!\} \rrbracket & \\ \qquad\quad\ size > 1 \mathbin{\&} output!cache \rightarrow read!bot?x \rightarrow StoreNewCacheController) \setminus \{\!| read |\!\} & \\ \qquad\quad \Box & \\ \qquad\quad size = 1 \mathbin{\&} output!cache \rightarrow NoNewCache);\ X & \end{array}$$

The following two steps further extend the hiding and unify the synchronisation sets.

$$\begin{array}{ll} \sqsubseteq_{\mathcal{A}} ControllerInit; & \text{[Laws A29, A28]} \\ \quad \mu X \bullet ((size = 0 \mathbin{\&} input?x \rightarrow CacheInput & \\ \qquad\quad \Box & \\ \qquad\quad (0 < size < maxbuff \mathbin{\&} input?x \rightarrow write?i?y \rightarrow StoreRingCmd & \\ \qquad\qquad \llbracket \{\!| write, input |\!\} \rrbracket & \\ \qquad\quad\ 0 < size < maxbuff \mathbin{\&} input?x \rightarrow write.top!x \rightarrow StoreInputController) & \\ \qquad\quad \Box & \\ \qquad\quad (size > 1 \mathbin{\&} output!cache \rightarrow read?i!(ring\ i) \rightarrow Skip & \\ \qquad\qquad \llbracket \{\!| read, output |\!\} \rrbracket & \\ \qquad\quad\ size > 1 \mathbin{\&} output!cache \rightarrow read!bot?x \rightarrow StoreNewCacheController) & \\ \qquad\quad \Box & \\ \qquad\quad size = 1 \mathbin{\&} output!cache \rightarrow NoNewCache);\ X) \setminus \{\!| read, write |\!\} & \end{array}$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} \mathit{ControllerInit}; \hfill \text{[Law 610]} \\
\quad \mu X \bullet ((\mathit{size} = 0 \;\&\; \mathit{input}?x \rightarrow \mathit{CacheInput} \\
\qquad\qquad \Box \\
\qquad\qquad (0 < \mathit{size} < \mathit{maxbuff} \;\&\; \mathit{input}?x \rightarrow \mathit{write}?i?y \rightarrow \mathit{StoreRingCmd} \\
\qquad\qquad\qquad [\![\{\!| \mathit{write}, \mathit{input}, \mathit{read}, \mathit{output} |\!\}]\!] \\
\qquad\qquad\; 0 < \mathit{size} < \mathit{maxbuff} \;\&\; \mathit{input}?x \rightarrow \mathit{write}.\mathit{top}!x \rightarrow \mathit{StoreInputController}) \\
\qquad\qquad \Box \\
\qquad\qquad (\mathit{size} > 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{read}?i!(\mathit{ring}\ i) \rightarrow \mathit{Skip} \\
\qquad\qquad\qquad [\![\{\!| \mathit{write}, \mathit{input}, \mathit{read}, \mathit{output} |\!\}]\!] \\
\qquad\qquad\; \mathit{size} > 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{read}!\mathit{bot}?x \rightarrow \mathit{StoreNewCacheController}) \\
\qquad\qquad \Box \\
\qquad\qquad \mathit{size} = 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{NoNewCache});\; X) \setminus \{\!| \mathit{read}, \mathit{write} |\!\}
\end{array}
$$

Now we are ready to apply a law that allows exchanging parallelism and external choice.

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} \mathit{ControllerInit}; \hfill \text{[Law 614]} \\
\quad \mu X \bullet ((\mathit{size} = 0 \;\&\; \mathit{input}?x \rightarrow \mathit{CacheInput} \\
\qquad\qquad \Box \\
\qquad\qquad ((0 < \mathit{size} < \mathit{maxbuff} \;\&\; \mathit{input}?x \rightarrow \mathit{write}?i?y \rightarrow \mathit{StoreRingCmd} \\
\qquad\qquad\quad \Box \\
\qquad\qquad\quad \mathit{size} > 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{read}?i!(\mathit{ring}\ i) \rightarrow \mathit{Skip}) \\
\qquad\qquad\qquad [\![\{\!| \mathit{write}, \mathit{input}, \mathit{read}, \mathit{output} |\!\}]\!] \\
\qquad\qquad\; (0 < \mathit{size} < \mathit{maxbuff} \;\&\; \mathit{input}?x \rightarrow \mathit{write}.\mathit{top}!x \rightarrow \mathit{StoreInputController}) \\
\qquad\qquad\quad \Box \\
\qquad\qquad\quad \mathit{size} > 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{read}!\mathit{bot}?x \rightarrow \mathit{StoreNewCacheController})) \\
\qquad\qquad \Box \\
\qquad\qquad \mathit{size} = 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{NoNewCache});\; X) \setminus \{\!| \mathit{read}, \mathit{write} |\!\}
\end{array}
$$

The next transformation eliminates the unnecessary synchronisation on the channels input and output and the corresponding guards. The synchronisation set is reduced to $\{\!|\mathit{write}, \mathit{read}|\!\}$.

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} \mathit{ControllerInit}; \hfill \text{[Law 612]} \\
\quad \mu X \bullet ((\mathit{size} = 0 \;\&\; \mathit{input}?x \rightarrow \mathit{CacheInput} \\
\qquad\qquad \Box \\
\qquad\qquad ((\mathit{write}?i?y \rightarrow \mathit{StoreRingCmd} \;\Box\; \mathit{read}?i!(\mathit{ring}\ i) \rightarrow \mathit{Skip}) \\
\qquad\qquad\qquad [\![\{\!| \mathit{write}, \mathit{read} |\!\}]\!] \\
\qquad\qquad\; (0 < \mathit{size} < \mathit{maxbuff} \;\&\; \mathit{input}?x \rightarrow \mathit{write}.\mathit{top}!x \rightarrow \mathit{StoreInputController} \\
\qquad\qquad\quad \Box \\
\qquad\qquad\quad \mathit{size} > 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{read}!\mathit{bot}?x \rightarrow \mathit{StoreNewCacheController})) \\
\qquad\qquad \Box \\
\qquad\qquad \mathit{size} = 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{NoNewCache});\; X) \setminus \{\!| \mathit{read}, \mathit{write} |\!\}
\end{array}
$$

The following step is justified by a lemma which is itself derived by simple symbolic execution and fixed point calculation; this lemma and a sketch of its proof can be found in Appendix B. It guarantees that, in this particular context, it is possible to obtain some sort of distribution of recursion over parallelism.

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} \mathit{ControllerInit}; \hfill \text{[Lemma 5]} \\
\quad ((\mu X \bullet (\mathit{write}?i?y \rightarrow \mathit{StoreRingCmd} \;\Box\; \mathit{read}?i!(\mathit{ring}\ i) \rightarrow \mathit{Skip});\; X) \\
\qquad\quad [\![\{\!| \mathit{write}, \mathit{read} |\!\}]\!] \\
\quad\; (\mu X \bullet (\mathit{size} = 0 \;\&\; \mathit{input}?x \rightarrow \mathit{CacheInput} \\
\qquad\qquad \Box \\
\qquad\qquad 0 < \mathit{size} < \mathit{maxbuff} \;\&\; \mathit{input}?x \rightarrow \mathit{write}.\mathit{top}!x \rightarrow \mathit{StoreInputController} \\
\qquad\qquad \Box \\
\qquad\qquad \mathit{size} > 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{read}!\mathit{bot}?x \rightarrow \mathit{StoreNewCacheController} \\
\qquad\qquad \Box \\
\qquad\qquad \mathit{size} = 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{NoNewCache});\; X)) \setminus \{\!| \mathit{read}, \mathit{write} |\!\}
\end{array}
$$

The next step moves the initialisation close to the controller action; the final step restructures the guards.

$$
\begin{array}{ll}
\sqsubseteq_{\mathcal{A}} ((\mu X \bullet (write?i?y \rightarrow StoreRingCmd \mathrel{\Box} read?i!(ring\ i) \rightarrow Skip);\ X) & \textit{[Laws A29, 613]} \\
\qquad \lbrack\!\lbrack \{\!| write, read |\!\} \rbrack\!\rbrack & \\
\quad (ControllerInit; & \\
\quad\ \mu X \bullet (size = 0 \mathrel{\&} input?x \rightarrow CacheInput & \\
\qquad\qquad \Box & \\
\qquad\qquad 0 < size < maxbuff \mathrel{\&} input?x \rightarrow write.top!x \rightarrow StoreInputController & \\
\qquad\qquad \Box & \\
\qquad\qquad size > 1 \mathrel{\&} output!cache \rightarrow read!bot?x \rightarrow StoreNewCacheController & \\
\qquad\qquad \Box & \\
\qquad\qquad size = 1 \mathrel{\&} output!cache \rightarrow NoNewCache);\ X)) \setminus \{\!| read, write |\!\} & \\
 & \\
\sqsubseteq_{\mathcal{A}} RHS & \textit{[Laws A1, A3, A22]}
\end{array}
$$

□

This completes the current development step.

7.4 Process refinement: split centralised buffer into a controller and a ring

As a result of the previous development step, the process *Buffer* is partitioned: it has two disjoint sets of paragraphs with respect to *ControllerState* and *RingState*. Therefore, with an application of Law 52, *Buffer* can be split into two independent processes: a controller and a ring process.

Figure 4 gives the description of the *Controller* process, and Figure 5 specifies the *Ring*. The buffer process becomes the parallel composition of the *Ring* and the *Controller*.

$$\textbf{process}\ Buffer \mathrel{\widehat=} (Controller \lbrack\!\lbrack \{\!| write, read |\!\} \rbrack\!\rbrack Ring) \setminus \{\!| write, read |\!\}$$

This step is a direct application of Law 52, and concludes the first iteration of our development. As mentioned before, the *Controller* remains unchanged up to the end of the development, whereas the *Ring* is further refined into an indexed interleaving of ring cells.

7.5 Data refinement: the ring process as a promotion of ring cells

This is the first step of our second development iteration, which aims at partitioning the *Ring* process. In this step, we introduce the concept of a ring cell as an abstract data type and restructure the process *Ring* as a promotion of ring cells that communicate over channels *rd* and *wrt*.

$$\textbf{channel}\ rd, wrt : \mathbb{N}$$

Later, we use indexing to introduce channels that communicate the position of the cell in the ring, as well as the value of the cell, as *rd* and *wrt* do.
A ring cell is required to store only a natural number.

$$\textbf{process}\ Ring \mathrel{\widehat=} \textbf{begin}$$

$$CellState \mathrel{\widehat=} [\, val : \mathbb{N} \,]$$

There are two actions on the ring cell state. *Read* merely outputs *val*.

$$Read \mathrel{\widehat=} rd!val \rightarrow Skip$$

The *Write* action updates *val*.

$$CellWrite \mathrel{\widehat=} [\, \Delta CellState;\ x? : \mathbb{N} \mid val' = x? \,]$$

$$Write \mathrel{\widehat=} wrt?x \rightarrow CellWrite$$

The ring cell allows either *Read* or *Write* actions.

$$RingCellController \mathrel{\widehat=} \mu X \bullet (Read \mathrel{\Box} Write);\ X$$

process *Controller* $\widehat{=}$ **begin**

state

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ControllerState}} \\
\hline
size : 0 \mathrel{..} maxbuff;\ ringsize : 0 \mathrel{..} maxring \\
cache : \mathbb{N};\ top, bot : 1 \mathrel{..} maxring \\
\hline
ringsize = max\{0, size - 1\} \\
ringsize \bmod maxring = (top - bot) \bmod maxring \\
\hline
\end{array}
$$

$ControllerInit \mathrel{\widehat{=}} [\, ControllerState' \mid size' = 0 \land bot' = 1 \land top' = 1 \,]$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CacheInput}} \\
\hline
\Delta ControllerState;\ x? : \mathbb{N} \\
\hline
size = 0 \land size' = 1 \land cache' = x? \land bot' = bot \land top' = top \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{StoreInputController}} \\
\hline
\Delta ControllerState \\
\hline
0 < size < maxbuff \\
size' = size + 1 \land cache' = cache \\
bot' = bot \land top' = (top \bmod maxring) + 1 \\
\hline
\end{array}
$$

$$
\begin{array}{ll}
InputController \mathrel{\widehat{=}} size < maxbuff \mathbin{\&} input?x \rightarrow & size = 0 \mathbin{\&} CacheInput \\
 & \Box \\
 & size > 0 \mathbin{\&} write.top!x \rightarrow StoreInputController
\end{array}
$$

$NoNewCache \mathrel{\widehat{=}} [\, \Delta ControllerState \mid size = 1 \land size' = 0 \land cache' = cache \land bot' = bot \land top' = top \,]$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{StoreNewCacheController}} \\
\hline
\Delta ControllerState;\ x? : \mathbb{N} \\
\hline
size > 1 \land size' = size - 1 \land cache' = x? \land bot' = (bot \bmod maxring) + 1 \land top' = top \\
\hline
\end{array}
$$

$$
\begin{array}{ll}
OutputController \mathrel{\widehat{=}} size > 0 \mathbin{\&} output!cache \rightarrow & size > 1 \mathbin{\&} read.bot?x \rightarrow StoreNewCacheController \\
 & \Box \\
 & size = 1 \mathbin{\&} NoNewCache
\end{array}
$$

$\bullet\ ControllerInit;\ \mu X \bullet (InputController \mathbin{\Box} OutputController);\ X$

end

Fig. 4. *Controller* process specification.

process $\mathit{Ring} \mathrel{\widehat=}$ **begin**

state $\mathit{RingState} \mathrel{\widehat=} [\, \mathit{ring} : \mathrm{seq}\,\mathbb{N} \mid \#\mathit{ring} = \mathit{maxring} \,]$

$\mathit{StoreRingCmd} \mathrel{\widehat=} [\, \Delta \mathit{RingState};\ i? : 1 \mathinner{..} \mathit{maxring};\ x? : \mathbb{N} \mid \mathit{ring}' = \mathit{ring} \oplus \{i? \mapsto x?\} \,]$

$\mathit{StoreRing} \mathrel{\widehat=} \mathit{write}?i?x \rightarrow \mathit{StoreRingCmd}$

$\mathit{NewCacheRing} \mathrel{\widehat=} \mathit{read}?i!(\mathit{ring}\ i) \rightarrow \mathit{Skip}$

$\mathit{RingAction} \mathrel{\widehat=} \mu\, X \bullet (\mathit{StoreRing} \mathrel{\Box} \mathit{NewCacheRing});\ X$

$\bullet\ \mathit{RingAction}$

end

Fig. 5. *Ring* process specification.

In this context, *RingCellController* is like any other action, but it does characterise the behaviour of a ring cell, and, in the next step, becomes the main action of the process that describes a cell.

The *ring* is simply a sequence of cells.

state $\mathit{RingStateP} \mathrel{\widehat=} [\, \mathit{ringP} : \mathrm{seq}\, \mathit{CellState} \mid \#\mathit{ring} = \mathit{maxring} \,]$

The promotion schema relates the local state of ring cells to the sequence of cells. The relevant ring cell in the collection is that indexed by $i?$.

Promotion

$\Delta \mathit{CellState}$

$\Delta \mathit{RingStateP}$

$i? : 1 \mathinner{..} \mathit{maxring}$

$\theta \mathit{CellState} = \mathit{ringP}\ \ i?$

$\mathit{ringP}' = \mathit{ringP} \oplus \{\, i? \mapsto \theta \mathit{CellState}' \,\}$

StoreRingCmd is defined as a promotion of *CellWrite*, in a standard way.

$\mathit{StoreRingCmdP} \mathrel{\widehat=} \exists\, \Delta \mathit{CellState} \bullet \mathit{CellWrite} \land \mathit{Promotion}$

The *StoreRing* action is not touched, except that it now uses *StoreRingCmdP*, instead of *StoreRingCmd*.

$\mathit{StoreRingP} \mathrel{\widehat=} \mathit{write}?i?x \rightarrow \mathit{StoreRingCmdP}$

If we consider that the promotion of the channel *wrt* is the channel *write*, then *StoreRingP* is the result of promoting *Write*.

The *NewCacheRing* action is defined by promoting *Read* in a similar way.

$\mathit{NewCacheRingP} \mathrel{\widehat=} \mathit{read}?i!(\mathit{ringP}\ \ i).\mathit{val} \rightarrow \mathit{Skip}$

The promotion of *rd* is *read*. Promoting *val* we get $(\mathit{ringP}\ \ i).\mathit{val}$.

The main action of the promoted ring is defined by the same CSP expression as the original process.

$\mathit{RingAction} \mathrel{\widehat=} \mu\, X \bullet (\mathit{StoreRingP} \mathrel{\Box} \mathit{NewCacheRingP});\ X$

$\bullet\ \mathit{RingAction}$

The actions involved, however, have been promoted.

end

This step can be justified by a simulation relating the sequence of cells to the sequence of natural numbers. The retrieve relation is as follows.

$$RetrRing \mathrel{\widehat=} [\, RingState;\ RingStateP \mid ring = \{\, i : 1 \mathinner{.\,.} maxring \bullet i \mapsto (ringP\ \ i).val \,\} \,]$$

The sequence *ring* of integers is obtained from the sequence *ringP* of cells by extracting the *val* of each cell.

We have to prove that *RetrRing* is a simulation between the original and the new main actions. We proceed compositionally, based on Laws 46, 45, and 48. So, we focus on *StoreRingP* and *NewCacheRingP*.

For *NewCacheRingP*, we use Law 43. We have to prove the property below.

$$\forall\, RingState;\ RingStateP;\ i : \mathbb{N} \bullet RetrRing \Rightarrow ring\ \ i = (ringP\ \ i).val$$

This follows directly from the definition of *RetrRing*.

For *StoreRingP*, we can apply Law 42, if we can prove that *StoreRingCmdP* simulates *StoreRingCmd*, for which we apply Law 41. Since the preconditions of these schemas are both true, the first proof obligation is trivial. The second proof obligation is as follows.

$$\begin{array}{l} \forall\, RingState;\ RingStateP;\ RingStateP' \bullet RetrRing \land StoreRingCmdP \Rightarrow \\ \qquad (\exists\, RingState' \bullet RetrRing' \land StoreRingCmd) \end{array}$$

This can be discharged with a few applications of the one-point rules.

7.6 Process refinement: a distributed cached-head ring buffer

This is the final step of our second iteration and of the development process as a whole. Each ring cell is implemented as an independent *Circus* process as the result of an application of Law 53 to *Ring*. We observe that a sequence is a special case of a partial function, which is the kind of global component actually considered in the presentation of Law 53.

The process that represents a ring cell is defined as follows.

$$\begin{array}{l} \mathbf{process}\ RingCell \mathrel{\widehat=} \mathbf{begin} \\ \quad \mathbf{state}\ CellState \mathrel{\widehat=} [\, val : \mathbb{N} \,] \\ \quad Read \mathrel{\widehat=} rd!val \rightarrow Skip \\ \quad CellWrite \mathrel{\widehat=} [\, \Delta CellState;\ x? : \mathbb{N} \mid val' = x? \,] \\ \quad Write \mathrel{\widehat=} wrt?x \rightarrow CellWrite \\ \quad \bullet\ \mu X \bullet (Read \mathrel{\Box} Write);\ X \\ \mathbf{end} \end{array}$$

The indexed ring cell is defined as follows.

$$\mathbf{process}\ IRCell \mathrel{\widehat=} (i : 1 \mathinner{.\,.} maxring \odot RingCell)[rd\_i, wrt\_i := read, write]$$

The indexed process operates on the channels *rd_i* and *wrt_i*, which have type $(1 \mathinner{.\,.} maxring) \times \mathbb{N}$. We rename them to *read* and *write*, which are the promoted channels. The indexed ring cell behaves like a ring cell, except that the communications $rd!val$ and $wrt?x$ are replaced by $read.i!val$ and $write.i?x$. The ring is constructed by interleaving the indexed ring cells.

$$\mathbf{process}\ Ring \mathrel{\widehat=} \mathbin{|||}\ i : 1 \mathinner{.\,.} maxring \odot IRCell\lfloor i \rfloor$$

There is no interaction between the ring cells, so the definition is appropriate as a refinement of a sequence. This results from an application of Law 53. ZRC can now be used to refine the schema actions to code.

This design is just one of the possible implementations of the buffer. For example, we could have a fully distributed implementation, without a cached head; this involves a more complicated ring protocol.

8 Conclusions and related work

In the course of linking theories, methods, and techniques to support the development of concurrent systems, we have proposed a refinement strategy for *Circus*, based on existing approaches to refinement for Z, CSP, and the refinement calculus. As a consequence of the orthogonal design of *Circus*, the reuse of existing refinement laws is immediate. Notwithstanding, novel refinement concepts, laws, and techniques are needed and have been presented here.

A *Circus* process encapsulates a state in the style of Z, and has a behaviour given by an action described in a mixture of CSP, Z, and guarded commands. In this context, notions of process refinement, and forwards and backwards simulation have been introduced. Among the refinement laws, we single out those of processes, and the laws of actions which relate schema expressions with CSP operators like parallelism and choice; as far as we know, these are novel. The Z technique of promotion has also been generalised from ordinary data types to processes. All these have been linked together under a strategy which gives some guidance to conducting refinement in *Circus*, as illustrated by the complete development of our buffer case study.

The work that is most closely related to *Circus* is that of action systems. An action system consists of a state and a program described as a simple set of guarded commands. The behaviour of the action system is given by a simple interpreter for the program that repeatedly selects an enabled action and executes it. Action systems describe a general model for parallelism, where a process is reduced to the sequential interleaving of atomic steps. A model for concurrency with shared variables is obtained by partitioning the actions amongst different processes; a model for distributed systems is obtained by partitioning the variables amongst the processes. The emphasis is on the state of an action system, with interaction described through the interference of shared variables.

Back and Sere [1] describe the combination of the refinement calculus and action systems in the derivation of parallel and distributed algorithms. They start from a purely sequential algorithm and proceed by stepwise refinement until an efficient parallel program is derived. Most steps are accomplished as sequential refinements, with parallelism being introduced only through the decomposition of atomic actions.

The main difference between the action system approach and *Circus* is due to the very basic nature of the action system formalism in comparison with process algebra. Control flow in an action system is simple: select an enabled guard; execute it; repeat. This gives a very flat structure, where auxiliary variables simulating program counters are needed to guarantee the proper sequencing of actions. In *Circus*, control flow is described using the process algebraic operators of CSP and, as a result, a rich set of laws are available for process and action refinement that have no direct correspondence in action systems. The two approaches are formally linked: Woodcock and Morgan [29] show how to calculate the failures-divergences semantics of an action system, and they provide sound and complete techniques for data refinement without unbounded nondeterminism. Butler [2] extends this work to include internal actions and unbounded nondeterminism. With these links, we may be able to take inspiration from the rules related to decomposition in Back and Sere's work to propose further laws for *Circus*.

Olderog [20] introduces a design calculus for occam-like communicating programs that allows for the stepwise development of correct programs. The programs are given an imperative trace-readiness semantics, and specifications are given in terms of assertions. The program and specification semantics are uniformly presented in a predicative style similar in spirit to that of unifying theories of programming. In fact, both works have roots in the ESPRIT ProCoS project. The design rules of [20] can be another source of inspiration for further refinement laws for *Circus* actions.

We are conducting a series of formal developments of concurrent programs using *Circus*, both in academia, as case studies, and in industry, as part of a commercial project. The verification of the laws of *Circus* is a major task. We have recently completed a mechanisation of a major part of the semantic metalanguage in both Z/Eves and ProofPowerZ. This provides the possibility of machine-checking the proofs, although such an exercise is very labour-intensive.

We have also recently started work on tools for *Circus*. A parser is complete, and we are now working on a type-checker and a model-checker for *Circus* refinement. A tool to support the application of the laws presented here and the others that are to come is also in our plans.

In [5] we give an alternative semantics for actions, based on weakest preconditions. In the spirit of the unifying theories of programming, we calculate the new semantics from the relational definition. The new formulation of the action semantics is more adequate as a basis for verification techniques. Our refinement laws are, of course, laws of both models.

In some sense, we can relate our development strategy for *Circus* to refactoring in the object-oriented literature. Our process splitting laws allow partitioning of a process in a way similar to a refactoring known as extracting class [8]. Our law that deals with promotion (Law 53) captures a more elaborate transformation in that the result is an indexed interleaving of processes. The addition of object-oriented constructs to *Circus*, however, is left as future work.

Acknowledgements

This work is partially supported by the EPSRC grant GR/R43211/01 on "Refinement calculi for sequential and concurrent programs" and the QinetiQ grant CU009-019346 on "Advances in formal modelling and concurrency as represented by the *Circus* language". The work of Ana Cavalcanti and Augusto Sampaio is partially supported by CNPq: grants 520763/98-0 and 521039/95-9.

References

1. R. J. R. Back and K. Sere. Stepwise Refinement of Parallel Algorithms. *Science of Computer Programming*, 13:133—180, 1990.
2. M. Butler. Stepwise Refinement of Communicating Systems. *Science of Computer Programming*, 27:139 – 173, 1996.
3. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Refinement of Actions in *Circus*. In J. Derrick, E. Boiten, J. C. P. Woodcock, and J. Wright, editors, *Proceedings of REFINE'2002*, volume 70 of *Eletronic Notes in Theoretical Computer Science*. Elsevier, 2002.
4. A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC—A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267—289, 1999.
5. A. L. C. Cavalcanti and J. C. P. Woodcock. A Weakest Precondition Semantics for Circus. In *Proceedings of the Communicating Processing Architectures 2002*. IOS Press, 2002.
6. C. Fischer. How to Combine Z with a Process Algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*. Springer-Verlag, 1998.
7. C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Fachbereich Informatik Universität Oldenburg, 2000.
8. M. Fowler. *Refactoring*. Addison-Wesley, 1999.
9. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
10. C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271—281, 1972.
11. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
12. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
13. He Jifeng. Process Simulation and Refinement. *Formal Aspects of Computing*, 1(3):229—241, 1989.
14. He Jifeng, C. A. R. Hoare, and J. W. Sanders. Data Refinement Refined. In G. Goos and H. Hartmants, editors, *ESOP'86 European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187—196, 1986.
15. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.
16. M. B. Josephs. A State-based Approach to Communicating Processes. *Distributed Computing*, 3:9—18, 1988.
17. Robin Milner. *Communication and Concurrency*. Series in Computer Science. Prentice Hall International, 1989.
18. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
19. C. C. Morgan and P. H. B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, 27(6):481—503, 1990.
20. E. Olderog. Towards a Design Calculus for Communicating Programs. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR'91 – 2nd International Conference on Concurrency Theory*, pages 61—77. Springer-Verlag, 1991.
21. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
22. A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451 – 470. Springer-Verlag, 2002.
23. G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems—an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18:249—284, May 2001.
24. J. M. Spivey. *The Z Notation: A Reference Manual*. 2nd. Prentice-Hall, 1992.
25. W. J. Toetenel. VDM + CCS + Time = MOSCA. In *Proceedings of the IFIP/IFAC Workshop on Real-time Programming*, 1992.

26. J. C. P. Woodcock and A. L. C. Cavalcanti. *Circus*: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK, 2001.
27. J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184 – 203. Springer-Verlag, 2002.
28. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.
29. J. C. P. Woodcock and C. C. Morgan. Refinement of state-based concurrent systems. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z—Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 340—351. Springer, 1990.

A Action Refinement Laws

Here we introduce some additional laws of actions.

A.1 Guards

Law A1 (Guard combination)

$$g_1 \mathbin{\&} (g_2 \mathbin{\&} A) = (g_1 \land g_2) \mathbin{\&} A \qquad \Box$$

Law A2 (Guard/Sequence—Association)

$$(g \mathbin{\&} A_1);\ A_2 = g \mathbin{\&} (A_1;\ A_2) \qquad \Box$$

Law A3 (Guard/External choice—Distribution)

$$g \mathbin{\&} (A_1 \mathbin{\Box} A_2) = (g \mathbin{\&} A_1) \mathbin{\Box} (g \mathbin{\&} A_2) \qquad \Box$$

Law A4 (Guard/Internal choice—Distribution)

$$g \mathbin{\&} (A_1 \sqcap A_2) = (g \mathbin{\&} A_1) \sqcap (g \mathbin{\&} A_2) \qquad \Box$$

Law A5 (Guard/Parallelism—Distribution 1)

$$g \mathbin{\&} (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) = (g \mathbin{\&} A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (g \mathbin{\&} A_2) \qquad \Box$$

Law A6 (Guard/Parallelism—Distribution 2)

$$(g_1 \mathbin{\&} A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (g_2 \mathbin{\&} A_2) = (g_1 \lor g_2) \mathbin{\&} ((g_1 \mathbin{\&} A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (g_2 \mathbin{\&} A_2)) \qquad \Box$$

Law A7 (Guard/Interleaving—Distribution 1)

$$g \mathbin{\&} (A_1 \mathbin{|||}[ns_1 \mid ns_2]\mathbin{|||} A_2) = (g \mathbin{\&} A_1) \mathbin{|||}[ns_1 \mid ns_2]\mathbin{|||} (g \mathbin{\&} A_2) \qquad \Box$$

Law A8 (Guard/Interleaving—Distribution 2)

$$(g_1 \mathbin{\&} A_1) \mathbin{|||}[ns_1 \mid ns_2]\mathbin{|||} (g_2 \mathbin{\&} A_2) = (g_1 \lor g_2) \mathbin{\&} ((g_1 \mathbin{\&} A_1) \mathbin{|||}[ns_1 \mid ns_2]\mathbin{|||} (g_2 \mathbin{\&} A_2)) \qquad \Box$$

Law A9 (True Guard)

$$\mathit{true} \mathbin{\&} A = A \qquad \Box$$

Law A10 (False Guard)

$$\mathit{false} \mathbin{\&} A = \mathit{Stop} \qquad \Box$$

Law A11 (Guarded Stop)

$$g \mathbin{\&} \mathit{Stop} = \mathit{Stop} \qquad \Box$$

A.2 Assumptions

Law A12 (Assumption/Guard—Elimination 2)

$$\{ g_1 \};\ (g_2 \mathbin{\&} A) = \{ g_1 \};\ \mathit{Stop}$$

provided $g_1 \Rightarrow \neg\, g_2$ □

Law A13 (Assumption/Guard—Replacement)

$$\{ g_1 \};\ (g_2 \mathbin{\&} A) = \{ g_1 \};\ (g_3 \mathbin{\&} A)$$

provided $g_1 \Rightarrow (g_2 \Leftrightarrow g_3)$ □

Law A14 (Assumption/Internal Choice—Distribution)

$\{p\};\ (A_1 \sqcap A_2) = (\{p\};\ A_1) \sqcap (\{p\};\ A_2)$ □

Law A15 (Assumption/Parallelism—Distribution)

$\{p\};\ (A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2) = (\{p\};\ A_1) \llbracket\, cs \,\rrbracket\, (\{p\};\ A_2)$ □

Law A16 (Assumption/Interleaving—Distribution)

$\{p\};\ (A_1 \,|\!|\![ns_1 \mid ns_2]|\!|\!|\, A_2) = (\{p\};\ A_1) \,|\!|\![ns_1 \mid ns_2]|\!|\!|\, (\{p\};\ A_2)$ □

In the following law we refer to a predicate $assump'$. In general, for any predicate p, the predicate p' is formed by dashing all its free undecorated variables.

Law A17 (Assumption Introduction—Schema Expression)

$[\Delta State;\ i? : T_i;\ o! : T_o \mid p \land assump'] = [\Delta State;\ i? : T_i;\ o! : T_o \mid p \land assump'];\ \{assump\}$ □

The schema in this law is an arbitrary schema that specifies an action in *Circus*: it acts on a state schema *State* and, optionally, has input variables $i?$ of type T_i, and output variables $o!$ of type T_o.

Law A18 (Assumption Elimination)

$\{p\} \sqsubseteq \mathit{Skip}$ □

A.3 Parallelisation

Law A19 (Parallelism Introduction—Sequence 2)

$$A_1(x);\ A_2(x) = (c!x \rightarrow A_1(x) \llbracket\, \overline{wrtV(A_2)} \mid \{\!| c |\!\} \mid wrtV(A_2) \,\rrbracket\, c?y \rightarrow A_2(y)) \setminus \{\!| c |\!\}$$

syntactic restrictions

- *$wrtV(A_1) \cap usedV(A_2) = \emptyset$;*
- *c is a valid channel of type T;*
- *$c \notin usedC(A_1) \cup usedC(A_2)$;*
- *$y \notin FV(A_2)$.* □

Law A20 (Parallelism Introduction—Sequence 3)

$$A_1(x);\ A_2(x) \ \sqsubseteq\ ((A_1(x);\ c!x \rightarrow \mathit{Skip}) \llbracket\, \overline{wrtV(A_2)} \mid \{\!| c |\!\} \mid wrtV(A_2) \,\rrbracket\, (c?y \rightarrow A_2(y))) \setminus \{\!| c |\!\}$$

syntactic restrictions

- *c is a valid channel of type T;*
- *$c \notin usedC(A_1) \cup usedC(A_2)$;*
- *$y \notin FV(A_2)$.*

provided $wrtV(A_1) \cap usedV(A_2) = \{x\}$ □

A.4 Prefixing

Law A21 (Prefix/Sequential Composition—Association)

$$c \to (A_1;\ A_2) = (c \to A_1);\ A_2$$

syntactic restriction $FV(A_2) \cap \alpha(c) = \emptyset$ $\Box$

The following are laws for distribution.

Law A22 (Prefix/External choice—Distribution)

$$c \to \Box\, i \bullet g_i \ \&\ \ A_i = \Box\, i \bullet g_i \ \&\ c \to A_i$$

provided $\vee\, i \bullet g_i$

syntactic restriction $FV(g_i) \cap \alpha(c) = \emptyset$, *for all* i $\Box$

The proviso is needed to ensure that at least one guard is valid, so that in the right-hand side action the communication does take place.

Law A23 (Prefix/Internal choice—Distribution)

$$c \to (A_1 \sqcap A_2) = (c \to A_1) \sqcap (c \to A_2)$$ $\Box$

Law A24 (Prefix/Parallelism—Distribution)

$$c \to (A_1 \llbracket\, cs \,\rrbracket A_2) = (c \to A_1) \llbracket\, ns_1 \mid cs \cup \{\!|c|\!\} \mid ns_2 \,\rrbracket (c \to A_2)$$

syntactic restriction $c \notin usedC(A_1) \cup usedC(A_2)$ *or* $c \in cs$ $\Box$

A.5 External choice

Law A25 (External choice/Sequence—Distribution)

$$(\Box\, i \bullet g_i \ \&\ c_i \to A_i);\ B = \Box\, i \bullet g_i \ \&\ c_i \to A_i;\ B$$ $\Box$

A.6 Parallelism

Law A26 (Parallelism/External Choice - Distribution)

$$A_1 \llbracket\, cs \,\rrbracket (A_2 \Box A_3) = (A_1 \llbracket\, cs \,\rrbracket A_2) \Box (A_1 \llbracket\, cs \,\rrbracket A_3)$$

provided

- *$usedC(A_1) \subseteq cs$;*
- *A_1 is deterministic.*

$\Box$

Law A27 (Parallelism Deadlock)

$$g1 \ \&\ c_1 \to A_1 \llbracket\, ns_1 \mid cs \cup \{\!|c_1, c_2|\!\} \mid ns_2 \,\rrbracket\, g_2 \ \&\ c_2 \to A_2 = Stop$$

provided $c_1 \neq c_2$ $\Box$

A.7 Hiding

Law A28 (Hide combination)

$(A \setminus cs_1) \setminus cs_2 = A \setminus (cs_1 \cup cs_2)$ $\Box$

Law A29 (Hide expansion)

$F(A \setminus cs) = F(A) \setminus cs$

provided $cs \cap usedC(F(\_)) = \emptyset$ $\Box$

A.8 Recursion

Law A30 (Recursion Unfold)

$\mu X \bullet F(X) = F(\mu X \bullet F(X))$ $\Box$

Law A31 (Recursion—Least Fixed Point)

$F(Y) \sqsubseteq Y \Rightarrow \mu X \bullet F(X) \sqsubseteq Y$ $\Box$

A.9 Unit and Zero Laws

Law A32 (Sequence—Unit)

$Skip;\ A = A = A;\ Skip$ $\Box$

Law A33 (External Choice—Unit)

$Stop \Box A = A$ $\Box$

Law A34 (Sequence—Zero)

$Stop;\ A = Stop$ $\Box$

Law A35 (Parallelism—Zero)

$A \lbrack\!\lbrack cs \rbrack\!\rbrack Stop = Stop$ $\Box$

B Some fixed point calculation for the case study

In the case study (see Section 7.3), it is necessary to split a single recursion (whose body is the parallel composition of *ring* and *controller* actions) into the parallel combination of two recursive actions: one concerned with the behaviour of the ring and the other with the behaviour of the controller. This transformation is captured by the following lemma.

Lemma 5 (Fixed point calculation).

$$
\begin{array}{l}
\mu\, X \bullet \\
\quad ((\mathit{size} = 0 \;\&\; \mathit{input}?x \rightarrow \mathit{CacheInput} \\
\qquad \Box \\
\qquad ((\mathit{write}?i?y \rightarrow \mathit{StoreRingCmd} \;\Box\; \mathit{read}?i!\mathit{ring}[i] \rightarrow \mathit{Skip}) \\
\qquad\qquad [\![\{\!|\; \mathit{write}, \mathit{read} \;|\!\}]\!] \\
\qquad\; (0 < \mathit{size} < \mathit{maxbuff} \;\&\; \mathit{input}?x \rightarrow \mathit{write}.\mathit{top}!x \rightarrow \mathit{StoreInputController} \\
\qquad\quad \Box \\
\qquad\quad \mathit{size} > 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{read}!\mathit{bot}?x \rightarrow \mathit{StoreNewCacheController})) \\
\qquad \Box \\
\qquad \mathit{size} = 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{NoNewCache});\;\; X) \setminus \{\!|\; \mathit{read}, \mathit{write} \;|\!\}
\end{array}
$$

$\sqsubseteq_{\mathcal{A}}$

$$
\begin{array}{l}
((\mu\, X \bullet (\mathit{write}?i?y \rightarrow \mathit{StoreRingCmd} \;\Box\; \mathit{read}?i!\mathit{ring}[i] \rightarrow \mathit{Skip});\;\; X) \\
\qquad [\![\{\!|\; \mathit{write}, \mathit{read} \;|\!\}]\!] \\
\;\; (\mu\, X \bullet \\
\;\; (\mathit{size} = 0 \;\&\; \mathit{input}?x \rightarrow \mathit{CacheInput} \\
\;\;\; \Box \\
\;\;\; 0 < \mathit{size} < \mathit{maxbuff} \;\&\; \mathit{input}?x \rightarrow \mathit{write}.\mathit{top}!x \rightarrow \mathit{StoreInputController} \\
\;\;\; \Box \\
\;\;\; \mathit{size} > 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{read}!\mathit{bot}?x \rightarrow \mathit{StoreNewCacheController} \\
\;\;\; \Box \\
\;\;\; \mathit{size} = 1 \;\&\; \mathit{output}!\mathit{cache} \rightarrow \mathit{NoNewCache});\;\; X)) \setminus \{\!|\; \mathit{read}, \mathit{write} \;|\!\}
\end{array}
$$

□

It is well-known that recursion does not distribute through parallelism in general. The proof of the above lemma is derived by simple, although lengthy, fixed point calculation. We omit the detailed proof, but discuss some of the more relevant steps, pointing to the laws which are necessary to conduct the proof.

Starting with the right-hand side of the lemma, we unfold the second recursion (Law A30). Afterwards, we distribute the entire recursion through each of the choices inside of the recursion body (Law A25). We then combine, in parallel, the first recursive program with each of the branches of the second recursion (Law A26).

The strategy is then to show that each of these branches can be transformed into a branch of the recursion on the left-hand side of the lemma, followed by the entire right-hand side itself. For these we use a step law for parallelism (Law 613), recursion unfold (Law A30), and distribution of external choice through sequential composition (Law A25).

This results in a program which coincides with the body of the recursion on the left-hand side of the lemma, except that in place of the recursive call we have the right-hand side itself. Therefore we apply the least fixed point law (Law A31) to conclude the proof.