Refinement in Circus

Augusto Sampaio¹, Jim Woodcock², and Ana Cavalcanti¹

¹ Centro de Informática/UFPE
Recife PE Brazil
² University of Kent
Canterbury England - UK

Abstract. We describe refinement in Circus, a concurrent specification language that integrates imperative CSP, Z, and the refinement calculus. Each Circus process has a state and accompanying actions that define both the internal state transitions and the changes in control flow that occur during execution. We define the meaning of refinement of processes and their actions, and propose a sound data refinement technique for process refinement. Refinement laws for CSP and Z are directly relevant and applicable to Circus, but our focus here is on new laws for processes that integrate state and control. We give some new results about the distribution of data refinement through the combinators of CSP. We illustrate our ideas with the development of a distributed system of co-operating processes from a centralised specification.

Keywords: Z, CSP, distribution, unifying theories of programming.

1 Introduction

A recent, interesting, and challenging trend in computing is the combination of theories and tools. One important topic in this research context is language integration, with the major objective of addressing the several facets (data, control, time) of realistic software engineering problems.

In particular, much work has been done in combining Z [20] and process algebras, including CSP [7]; Fischer gives a survey of some of this research [4]. Such a combination has obvious advantages: Z is useful for describing rich information structures in a state, and process algebra is useful for describing the behavioural patterns of communication and synchronisation. Some interesting work has been undertaken, but very little has been accomplished in terms of understanding the formal development of programs starting from such mixed specifications.

Circus [17] combines Z and CSP, and includes specification constructs usually found in refinement calculi (as, for instance, in [10]) and Dijkstra’s language of guarded commands [3]. As a result, Circus is a unified programming language for presenting specifications, designs, and programs. Specifications are based on the use of Z constructs and specification statements. These constructs can be combined with executable commands, like assignments, conditionals, and loops; reactive behaviour, including communication, parallelism, and choice, is defined with the use of CSP constructs. All existing combinations of Z with a process
algebra model concurrent programs as communicating abstract data types, but we do not insist on identifying events with operations on the state. The result is a general programming language adequate for developing concurrent programs.

There are several complex issues to be considered in the integration of languages: syntax, semantics, proof theory, development methods, structuring techniques, and reuse, among others. Circus already has a well-defined syntax and a formal semantics [17,19] based on unifying theories of programming [8], together with case studies that illustrate its expressive power [18].

The central aim of this paper is to describe a development method for Circus, based on refinement. A refinement calculus for Circus should clearly extend similar work for CSP [15] and Z [2], since Circus integrates these two languages. In addition, however, new laws are necessary to deal directly with Circus processes, which combine state and control behaviour. The focus of this paper is on describing some of these new laws for processes.

We propose a refinement strategy whose typical starting point is a centralised specification of an application. In the development process, we move towards a distributed solution. The strategy is supported by two families of laws that allow the incremental splitting of Circus processes using parallelism. The overall approach is illustrated by a case study that, although simple, is interesting enough to demonstrate the proposed strategy in all its relevant details.

In the next section, we present Circus: its syntax and semantics. Our main results are in sections 3 and 4, where we present the notions of refinement appropriate for Circus and refinement laws. In Section 5, we present a case study, and conclude in Section 6 with a discussion of related and future work.

2 Circus

A Circus program is a sequence of paragraphs: a Z paragraph, a channel definition, a channel set definition, or a process definition. In Figure 1 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*. 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 [16]. The syntactic category CSExp of channel set expressions contains the empty set of channels ({}), channel enumerations enclosed in {{}}, and expressions involving the usual set operators.

To illustrate the use of Circus, we give a specification of a simple bounded reactive buffer that is used to store natural numbers. The maximum size of the buffer is a positive constant.

\[
\text{maxbuff : } N_1
\]

Inputs and outputs are taken from two different channels.

channel input, output : N
The basic form of process definition describes the process's state and operations, as in a Z specification. In Circus, we use process paragraphs; the operations are called “actions” and can be specified using schemas, CSP operators, and guarded commands. The nameless action at the end of a process description defines its behaviour; we refer to this action as the “main action” of the process.

In our example, we have a process Buffer, whose state components are the contents of the buffer and its size.

```
process Buffer ≡ begin
  BufferState ≡ [ buff : seq N; size : 1..maxbuff | size = #buff ≤ maxbuff ]
```

Initially, the buffer is empty.

```
BufferInit ≡ [ BufferState' | buff' = ⟨⟩ ∧ size' = 0 ]
```

Input is possible if there is space in the buffer; the input element is appended to the bounded sequence and the size incremented.
The \textbf{Input} action is enabled providing the buffer is not full. It outputs the head of the buffer, giving the FIFO discipline, and updates the size accordingly.

\begin{align*}
\text{Input} & \triangleq size < \text{maxbuff} & \& \ input?x \rightarrow \text{InputCmd} \\
\text{InputCmd} & \triangleq BufferState \\
\begin{array}{ll}
\Delta BufferState \\
x? : \mathbb{N} \\
\end{array} \\
\begin{array}{ll}
\text{size} < \text{maxbuff} & \wedge \text{buff'} = \text{buff} \setminus \{x?\} & \wedge \text{size'} = \text{size} + 1 \\
\end{array}
\end{align*}

The \textbf{Output} action is enabled providing the buffer is not empty. It outputs the head of the buffer, giving the FIFO discipline, and updates the size accordingly.

\begin{align*}
\text{Output} & \triangleq size > 0 & \& \ output!(\text{head buff}) \rightarrow \text{OutputCmd} \\
\text{OutputCmd} & \triangleq BufferState \\
\begin{array}{ll}
\Delta BufferState \\
size > 0 \\
\text{buff'} = \text{tail buff} & \wedge \text{size'} = \text{size} - 1 \\
\end{array}
\end{align*}

Finally, the main action initialises the \textbf{Buffer} and repeatedly offers the choice of input and output.

\begin{itemize}
\item \textbf{BufferInit; }$\mu X \star (\text{Input }\Box \text{Output}) ; X$
\end{itemize}

\textbf{end}

The guards guarantee that \textbf{Input} is available only if the buffer is not full, and \textbf{Output}, only if the buffer is not empty.

CSP operators can also be applied to processes: their states are conjoined and their main actions are combined using the operator applied. An unusual operator available in \textit{Circus} is indexing: a process 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 $(i : T \odot P)[e]$ 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 \textit{Circus}. For example, in $P[\text{olde} := \text{newc}]$, the communications of $P$ through channel $\text{olde}$ are done through the channel $\text{newc}$ instead. An example of the use of the indexing and renaming operators is found in our case study (Section 5).

The semantics of \textit{Circus} \cite{Circus} is based on unifying theories of programming \cite{Goguen}: an alphabetised relational model for imperative programming, concurrency, and communication. In our work, Z is the concrete syntax for the relational model, so that a \textit{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.
In the unifying theory, distinguished variables are used to describe relevant observations. In the semantics of Circus, these variables comprise the state components of a process denotation. In addition to the state components in the process specification, there are 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.

To illustrate the unifying theory, we give a description of the semantics of the simple prefixing operator. Consider the process \( P = a \rightarrow \text{Skip} \); we explain \( P \)'s behaviour by case analysis on the observational variables okay and wait.

Suppose that okay is false; in this case, \( P \) has been activated in the final state of a process that is diverging. Divergence is a left-zero for sequential composition, so the only thing that \( P \) can guarantee is that it leaves the final value of \( tr \) as an extension of its initial value: \( tr \text{ prefix} \ tr' \).

Suppose instead that okay is true and so \( P \)'s predecessor is not diverging. There are two cases to consider: the predecessor may or may not have terminated; this is described by the observation wait. Suppose that wait is true and so the predecessor has not terminated, then \( P \) has no effect on the observations.

Suppose instead that wait is false and so the predecessor has terminated. There are two possible states: \( P \) itself may or may not have terminated. Suppose that wait' is true and so \( P \) has not terminated. \( P \) must leave the trace \( tr \) unchanged, but it must not be refusing the event \( a \): \( tr' = tr \land a \notin \text{ref}' \).

Finally, suppose that wait' is false and so \( P \) has terminated. \( P \) must have added the event \( a \) to the trace: \( tr' = tr \land \langle a \rangle \). The final value of the refusal set is irrelevant, since \( P \) has now terminated and can do nothing further. In all these okay cases, the state variables are left unchanged and \( P \) doesn’t diverge.

3 Refinement notions

In the unifying theory, refinement is expressed as implication; that is, an implementation \( P \) satisfies a specification \( S \), providing that \([P \Rightarrow S]\), where the square brackets denote universal quantification over the alphabet, which must be the same for both implementation and specification. In Circus, this notion is used to formalise the situation when one action \( B \) refines another \( A \) \((\sqsubseteq_A)\).

**Definition 1 (Action refinement).** Suppose that \( A \) and \( B \) are actions on the same state space. Action \( A \) is refined by action \( B \) if, and only if, every observation of \( B \) is permitted by \( A \) as well: \( A \sqsubseteq_A B \iff [B \Rightarrow A] \).  

The state of a process is encapsulated; therefore, when refining a process we may change the local state if we wish. In the standard theory of data refinement [11], this possibility is handled by regarding the states as existing in local blocks. In Circus, as a result of hiding the details of the states of two processes \( P \) and \( Q \), we are left with two main actions with a common alphabet; this allows us to define process refinement in terms of action refinement of local blocks \((\sqsubseteq_P)\).

Let \( P.\text{st}, P.\text{init}, \) and \( P.\text{act} \) denote the local state, initialisation, and main action of a process \( P \), respectively.
Definition 2 (Process refinement). We define $P \sqsubseteq Q$ to mean that process $P$ is refined by process $Q$ if, and only if,

$$(\exists P \cdot st; P \cdot st' \cdot P \cdot \text{init} \land P \cdot \text{act}) \sqsubseteq_A (\exists Q \cdot st; Q \cdot st' \cdot Q \cdot \text{init} \land Q \cdot \text{act})$$

The techniques of data refinement are well-known for proving the correctness of a development step involving local blocks. They require the formalisation of a link between abstract and concrete states, usually referred to as forwards and backwards simulations [6, 9, 20]. A well-established result is that the completeness of data refinement requires both techniques; however, in this paper, we restrict ourselves to the most widely-used technique of forwards simulation, and leave backwards simulation as a topic for further work.

Definition 3 (Forwards simulation). A forwards simulation between actions $A$ and $B$ of processes $P$ and $Q$ is a relation $R$ satisfying

1. (initialisation) $[(\forall Q \cdot st \cdot Q \cdot \text{init} \Rightarrow (\exists P \cdot st \cdot P \cdot \text{init} \land R))]$
2. (correctness) $[(\forall P \cdot st; Q \cdot st; Q \cdot \text{act} \land R \Rightarrow (\exists P \cdot st' \cdot R' \land A)]$

A forwards simulation between $P$ and $Q$ 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 the semantics of actions are total.

The next theorem ensures that, if we provide a forwards simulation between processes $P$ and $Q$, then we can substitute $Q$ for occurrences of $P$ in a program.

Theorem 1 (Forwards simulation is sound). Whenever a forwards simulation exists between two processes $P$ and $Q$, we also have that $P \sqsubseteq P Q$.

Proof

$\exists Q \cdot st; Q \cdot \text{init} \land Q \cdot \text{act}$

$\Rightarrow \exists P \cdot st; Q \cdot \text{init} \land R \land Q \cdot \text{act}$ [initialisation]

$\Rightarrow \exists P \cdot st; Q \cdot st' \cdot P \cdot \text{init} \land R \land R' \land P \cdot \text{act}$ [correctness]

$\Rightarrow \exists P \cdot st; P \cdot \text{init} \land P \cdot \text{act}$ [schema calculus]

We still need, however, support for the proof that a particular relation $R$ is a forwards simulation.

Definition 3 imposes proof obligations related to the main actions of the processes. To support a calculational approach and the reuse of well-established techniques, it is useful to be able to prove simulation for primitive actions and rely on distribution properties through the action combinators. More specifically, we want to be able to be assured of the existence of a simulation by discharging proof obligations for schema expressions, as in Z, but keeping the structure of the main action. This is the approach supported by the following theorems.

First of all, data refinement leaves Skip, Stop, and Chaos unchanged. If an action is described by a schema, then the familiar proof obligations of $Z$ apply.
Theorem 2 (Forwards simulation of schema expressions). The following are sufficient conditions for the forwards simulation of schema expressions.

1. (applicability) \[ \forall P, st ; Q, st \bullet R \land \text{pre } P\text{Exp} \Rightarrow \text{pre } Q\text{Exp} \]
2. (correctness) \[ \forall P, st ; Q, st ; Q, st' \bullet R \land \text{pre } P\text{Exp} \land Q\text{Exp} \Rightarrow (\exists P, st' \bullet R' \land P\text{Exp}) \]

Proof From the semantics of schema expressions.

Results exist about the distribution of data refinement through the combinators of sequential programming languages \[11,12\], and some of these have been re-expressed in the unifying theory \[8\]. More interesting is the distribution through the combinators of CSP; in this paper, we have space for demonstrating a few cases: sequential composition, prefixing, and concurrency.

Theorem 3 (Data refinement distributes through sequential composition). Suppose that \(R\) is a forwards simulation between \(A_1\) and \(B_1\) and between \(A_2\) and \(B_2\), then \(R\) is also a forwards simulation between \(A_1; A_2\) and \(B_1; B_2\).

Proof

\[
R(P, st, Q, st) \land (B_1(Q, st, Q, st'); B_2(Q, st, Q, st'))
\]

\[ \Leftrightarrow \exists Q, st_0 \bullet R(P, st, Q, st) \land B_1(Q, st, Q, st_0) \land B_2(Q, st_0, Q, st') \]

[sequential composition]

\[ \Rightarrow \exists P, st_0, Q, st_0 \bullet R(P, st_0, Q, st_0) \land A_1(P, st, P, st_0) \land B_2(Q, st_0, Q, st') \]

[assumption]

\[ \Rightarrow \exists P, st_0, P, st' \bullet R(P, st', Q, st') \land A_1(P, st, P, st_0) \land A_2(P, st_0, P, st') \]

[assumption]

\[ \Leftrightarrow \exists P, st' \bullet R(P, st', Q, st') \land (A_1(P, st, P, st'); A_2(P, st, P, st')) \]

[sequential composition]

\[ \square \]

The proof of the last theorem is given in the relational calculus; the proof in the schema calculus is rather longer.

We consider the simple prefixing action \(c.pxp \rightarrow \text{Skip}\), where \(c\) is a channel name and \(pxp\) is an expression in terms of the abstract state denoting a communicable value on that channel. The abstract description of the event \(c.pxp\) must be transformed into a concrete description \(c.qxp\) of the same event. Externally, the same value is communicated; it is the description in terms of the internal state that has to change in a data refinement.

The correctness of replacing \(pxp\) by \(qxp\) may be explained by considering an expression as an interrogation of the state; that is, \([\Xi P, st; o! : V | o! = pxp] \). The assumption in the following theorem is then a consequence of the simulation of these interrogations: \(pxp\) and \(qxp\) are equal, modulo \(R\).
Theorem 4 (Data refinement distributes through simple prefixing).
Suppose that \( pxp \) and \( qxp \) are expressions in terms of the states \( P.st \) and \( Q.st \) of processes \( P \) and \( Q \), respectively, and that \( R \) is a forwards simulation between these processes. If the expressions are equal, modulo \( R \),

\[
\forall P.st; \ Q.st \bullet R \Rightarrow pxp = qxp
\]

then we have that the relation \( R \) is also a forwards simulation between the simple prefixed actions \( c.pxp \rightarrow \text{Skip} \) and \( c.qxp \rightarrow \text{Skip} \).

**Proof**

\[
R \land (c.qxp \rightarrow \text{Skip}) \\
\Leftrightarrow R \land ((x' = x \land \text{okay}' \land \\
(Id < wait \triangleright (tr' = tr \land c.qxp \notin ref' < wait' \triangleright tr' = tr \land (c.qxp)))) \\
< okay \triangleright tr \text{ prefix } tr') \quad \text{[definition]} \\
\Leftrightarrow R \land ((x' = x \land \text{okay}' \land \\
(Id < wait \triangleright (tr' = tr \land c.pxp \notin ref' < wait' \triangleright tr' = tr \land (c.pxp)))) \\
< okay \triangleright tr \text{ prefix } tr') \quad \text{[assumption]} \\
\Leftrightarrow \exists P.st' \bullet R' \land (c.pxp \rightarrow \text{Skip}) \\
\quad \text{[existential introduction]} \\
\quad \text{[definition]}
\]

Prefixing is defined in terms of sequential composition and simple prefixing.

**Corollary 1 (Data Refinement distributes through prefixing).** Suppose that \( pxp \) and \( qxp \) are expressions in terms of the states \( P.st \) and \( Q.st \) of processes \( P \) and \( Q \), and that \( R \) is a forwards simulation between these processes and their actions \( A \) and \( B \). If the expressions are equal, modulo \( R \), then \( R \) is a forwards simulation between the actions \( c!pxp \rightarrow A \) and \( c!qxp \rightarrow B \).

**Proof** Directly from Theorems 3 and 4. \( \Box \)

The parallel composition \( A_1 \parallel [C] \parallel A_2 \) describes two actions synchronising on events in the set \( C \). The resulting action is formed by merging the observations and conjoining the state changes. The traces are merged to produce a trace where the events in \( C \) occur synchronously. An event is refused if either component refuses it, divergence arises if either component diverges, and termination occurs when both components terminate. To achieve distribution of data refinement through parallelism, we must show its effect on the conjunction of state changes.

The assumptions for distributing data refinement through concurrency require that we can partition the state space in a particular way to avoid interference, so that \( A_1 \) has precedence in one partition, and \( A_2 \) has precedence in
the other. More formally, there is a sub-space $S$ of the abstract state such that $A_1 \mid S' \Rightarrow A_2 \mid S'$. (Here, $A_1 \mid S'$ is the projection of $A_1$ onto the variables of $S'$; the complementary operation is $A_1 \setminus S'$, which is $A_1$ with the variables of $S'$ hidden.) In other words, every result that $A_1$ can produce in $S'$ is acceptable to $A_2$. Furthermore, every result that $A_2$ can produce in the complement of $S'$ is acceptable to $A_1$; that is, $A_2 \setminus S' \Rightarrow A_1 \setminus S'$. In general, this allows each action to make compatible changes in the other’s partition. Furthermore, the simulation $R$ must respect the same noninterference properties between $A_1$ and $A_2$, so that, for example, $(A_1 \mid S') \not\triangleleft R \Rightarrow (A_2 \mid S') \not\triangleleft R$. Moreover, $R$ must respect the partition by identifying a corresponding region $T$ in the concrete state space.

**Theorem 5 (Data Refinement distributes through concurrency).** Suppose that $R$ is a forwards simulation between $A_1$ and $B_1$ and between $A_2$ and $B_2$. Furthermore, suppose that the after-variables can be partitioned as follows:

\[
\begin{align*}
A_1 \mid S' &\Rightarrow A_2 \mid S' \\
A_2 \setminus S' &\Rightarrow A_1 \setminus S' \\
(A_1 \mid S') \not\triangleleft R &\Rightarrow (A_2 \mid S') \not\triangleleft R \\
(A_2 \setminus S') \not\triangleleft R &\Rightarrow (A_1 \setminus S') \not\triangleleft R
\end{align*}
\]

Then $R$ is also a forwards simulation between $A_1 \parallel C \parallel A_2$ and $B_1 \parallel C \parallel B_2$.

**Proof** Our result follows directly from the schema calculus and our assumptions.

\[
R \triangleleft (B_1 \land B_2)
\]

\[
\Rightarrow (R \not\triangleleft B_1) \land (R \not\triangleleft B_2) \tag{schema calculus}
\]

\[
\Rightarrow (A_1 \not\triangleleft R) \land (A_2 \not\triangleleft R) \tag{hypothesis}
\]

\[
\Leftrightarrow (((A_1 \not\triangleleft R) \mid T') \lor ((A_2 \not\triangleleft R) \setminus T')) \land (((A_2 \not\triangleleft R) \mid T') \lor ((A_2 \not\triangleleft R) \setminus T'))
\]

\[
[T' \text{ and its complement partition the state space}]
\]

\[
\Leftrightarrow (((A_1 \not\triangleleft R) \mid T') \land ((A_2 \not\triangleleft R) \mid T')) \lor (((A_1 \not\triangleleft R) \setminus T') \land ((A_2 \not\triangleleft R) \setminus T'))
\]

\[
[\text{schema calculus}]
\]

\[
\Leftrightarrow ((A_1 \mid S') \not\triangleleft R) \lor ((A_2 \setminus S') \not\triangleleft R) \tag{assumption}
\]

\[
\Leftrightarrow ((A_1 \mid S') \lor (A_2 \setminus S')) \not\triangleleft R \tag{schema calculus}
\]

\[
\Leftrightarrow (((A_1 \mid S') \lor (A_2 \setminus S')) \lor ((A_1 \setminus S') \land (A_2 \setminus S'))) \not\triangleleft R \tag{assumption}
\]

\[
\Leftrightarrow (((A_1 \mid S') \lor (A_2 \setminus S')) \land ((A_2 \setminus S') \lor (A_2 \setminus S'))) \not\triangleleft R \tag{schema calculus}
\]

\[
\Leftrightarrow (A_1 \land A_2) \not\triangleleft R \tag{[S' and its complement partition the state space]}
\]

\[
\square
\]

The next result ensures that algorithmically refining an action (Definition 1) is a proper way of refining the process as a whole, justifying the use of action refinements in developments. As usual, we must prove the initialisation and applicability theorems.

**Theorem 6 (Feasible refinement).** Suppose we have have a process $P$ with actions $A$ and $B$. If $A \sqsubseteq_A B$, then the identity is a forwards simulation between
A and B, provided \( P \) satisfies the Z initialisation theorem and its schema actions are feasible.

**Proof** Direct from definitions.

The results just presented are applied in the case study in Section 5.

4 Refinement laws

Both laws of CSP and laws of Z, for which we have a refinement calculus [2], are relevant to our work; nevertheless, our focus here are on the laws of processes. 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 splitting of processes. Here we present two families of refinement laws.

4.1 Process splitting

The first family of laws, called process splitting, applies 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 include a partition of the state and the corresponding paragraphs, and the third process has the same behaviour as the original one.

Let \( pd \) stand 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 arbitrary context (function on processes). This is the general form of processes to which the process split laws apply.

\[
\text{process } P \equiv \text{begin} \\
\quad \text{State} \equiv Q.st \land R.st \\
\quad Q.ppS \uparrow R.st \\
\quad R.ppS \uparrow Q.st \\
\quad \subseteq F(Q.act, R.act) \\
\text{end}
\]

The state of \( P \) is defined as the conjunction of two other state schemas: \( Q.st \) and \( R.st \). The actions of \( P \) are \( Q.ppS \uparrow R.st \) and \( R.ppS \uparrow Q.st \), which handle the partitions of the state separately. In \( Q.ppS \uparrow R.st \), 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 \( R.ppS \uparrow Q.st \).

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. We can formulate our family of laws as follows.
Law 1 (Process splitting)

\[ pd = (\text{process } P \triangleq F(Q.\text{act}, R.\text{act})) \]

provided \(Q.pps\) and \(R.pps\) are disjoint with respect to \(R.st\) and \(Q.st\).

We say that two sets of process paragraphs \(pps\) and \(pps'\) are disjoint with respect to states \(s\) and \(s'\) if, and only if, \(pps = pps \uparrow s'\) and \(pps' = pps' \uparrow s\), and no command nor CSP action expression in \(pps\) refers to components of \(s'\) or to paragraph names in \(pps'\); further, no command nor CSP action expression in \(pps'\) refers to components of \(s\) or to paragraph names in \(pps\).

4.2 Process indexing

The second family of laws applies to processes defined using the promotion technique of Z. Broadly, the technique 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 sets, sequences, maps, etc.).

By convention, the basic (element) type is referred to as \(\text{local}\), whereas the collection is called \(\text{global}\). When the local type is completely encapsulated (as an abstract data type) in the global type, we say that the promotion is \(\text{free}\); otherwise it is called \(\text{constrained}\) [20]. 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.

One of the contributions of this work is to extend the Z technique of promotion to \(\text{Circus}\) actions. Below we give an inductive definition of the relevant promotion patterns; where \(L\) stands for the local process, \(G\) for the global process, and \(Promotion\) for the promotion schema.

For simplicity, we assume that the global state is a function \(f\) from elements of an arbitrary type \(\text{Range}\) 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.

\[ promote(SExp) \triangleq \exists \Delta L.st \cdot SExp \land Promotion \]

The promotion of \(\text{Skip}\), \(\text{Stop}\), and \(\text{Chaos}\) leaves them unchanged.

\[ promote(A) \triangleq A, \quad \text{for } A \in \{\text{Skip, Stop, Chaos}\} \]

To promote a communication \(c.e\), we need to communicate an extra value: the identifier of the value \(e\) 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. The latter may also need to be promoted, as it may include references to elements of the local state.

\[ promote(c.e \to A) \triangleq pc.promote(e) \to promote(A) \]
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 local state component, it does not need to be changed.

\[
\text{promote}(x) \equiv x, \quad \text{provided } x \text{ is not a component of } L.st
\]

If it is, then we need to access it through the global state.

\[
\text{promote}(x) \equiv f(i).x, \quad \text{if } x \text{ is a component of } L.st
\]

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

\[
f := f \oplus \{ i : \text{Range}; l : L.st \mid l.x = e \land l.y = f(i).y \land l.z = f(i).z \}
\]

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. We assume that actions like these are not used.

Let \( pd \) stand for the following process declaration. The family of process indexing laws applies to processes of this form.

\[
\text{process } P \equiv \begin{align*}
\text{State} & \equiv [f : \text{Range} \rightarrow L.st \mid \text{pred}] \\
L.\text{action}_k & \equiv \text{State} \\
L.\text{act} & \equiv \mu X \bullet F(L.\text{action}_k); X \\
\text{Promotion} & \equiv \\
& \begin{cases}
| \Delta L.st; \Delta \text{State}; i? : \text{Range} | \\
i? \in \text{dom } f \land \theta L.st = f(i?) \land f' = f \oplus \{ i? \mapsto \theta L.st' \}
\end{cases}
\end{align*}
\]

\[
\text{action}_k \equiv \text{promote}(L.\text{action}_k)
\]

\[
\bullet (\mu X \bullet F(\text{action}_k); X)
\]

As discussed before, the global state component is a function from \( \text{Range} \) to a local state \( L.st. \) Actions \( L.\text{action}_k \) over the local state do not affect the global state. The main local action \( L.\text{act} \) is defined recursively, as is the main global action. Both have the same structure, but the former uses the actions \( L.\text{action}_k \) on the local states, and the latter, the corresponding promoted actions \( \text{action}_k \). There is a promoted action \( \text{action}_k \) for every local action \( L.\text{action}_k \). We note that for each channel \( c \) used by the \( \text{action}_k \), 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.
Consider also the indexed process $IL$ below.

\[
\text{process } IL \triangleq i : \text{Range} \odot L[c_i := pc]
\]

The process $i : \text{Range} \odot L$ 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.

**Law 2 (Process indexing)**

\[
pd = \text{process } P \triangleq \| i : \text{Range} \odot IL[i]
\]

provided $L.pps$ and $pps$ are disjoint with respect to $L.st$ and $State$. \hfill \Box

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

## 5 Refining the reactive buffer

In this section, we develop an implementation for the bounded reactive buffer abstract specification presented in Section 2. The structure of the final implementation is a ring of cells with a central controller and a cached head. Broadly, the refinement progresses as follows: after a standard data refinement, we decompose the original process into a controller and a centralised ring (Law 1); through a second data refinement step, the centralised ring is redesigned as a promotion of individual ring cells. Finally, we apply Law 2 to decompose the ring process into the interleaving of ring cell, each one storing a single value.

### 5.1 A centralised ring buffer

Our first development step is a data refinement, in which we introduce a cache and a ring to represent the internal state of the process $Buffer$. When the buffer is non-empty, the cache stores the head of the buffer. In a circular array, the two ends are considered to be joined. We maintain two indexes into this array: a bottom 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.

This step can be justified applying Theorems 2 and 3, Corollary 1, and other similar theorems. For conciseness we omit the details of this data refinement which is very much like a standard Z data refinement. The resulting state is as follows. Its definition is partitioned because we aim at applying Law 1.

\[
\text{process } CBuffer \triangleq \text{begin}
\]
ControllerState

\[
\begin{align*}
\text{size} & : 0 \ldots \text{maxbuff} \\
\text{cache} & : \mathbb{N} \\
\text{ringsize} & : 0 \ldots \text{maxring}; \text{top}, \text{bot} : 1 \ldots \text{maxring} \\
\text{ringsize} &= \max\{0, \text{size} - 1\} \\
\text{ringsize} \mod \text{maxring} &= (\text{top} - \text{bot}) \mod \text{maxring}
\end{align*}
\]

\[
\text{RingState} \triangleq [\text{ring} : \text{seq}\mathbb{N} | \#\text{ring} = \text{maxring}]
\]

\[
\text{BufferState} \triangleq \text{ControllerState} \land \text{RingState}
\]

The constant \textit{maxring}, defined as \textit{maxbuff} − 1, gives the bound for the ring. There is a subtle situation when the bottom and the top indexes 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 of values stored in the ring.

The structure of the main action is exactly that used in the abstract specification in Section 2. The primitive actions, however, are changed to act on the concrete state.

### 5.2 Isolate access to the ring component

According to Theorem 6, we can also refine the individual actions of the \textit{Buffer}. Indeed, in our second and third development steps we refine these actions with the aim of obtaining two independent sets of paragraphs. One set of paragraphs accesses exclusively the \textit{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 which remains unchanged up to the end of the development.

In some circumstances, this partitioning of the state space is not direct. For example, the \textit{StoreInput} operation updates both \textit{top} and \textit{ring}. Splitting it into two operations is not immediate, because the operation that is concerned with updating the \textit{ring} needs the input value (\(x?\)) and the current value of \textit{top}. The main design tool to solve such data dependencies is introduction of communication. We need two new channels, as follows.

\[
\text{channel write, read} : (1 \ldots \text{maxring}) \times \mathbb{N}
\]

These channels are hidden in the \textit{Buffer} design and implementation.

The first set of paragraphs has ControllerState as its state space, whilst preserving RingState. The initialisation is for an empty buffer.

\[
\text{ControllerInit} \triangleq [\text{ControllerState'} | \text{size}' = 0 \land \text{bot}' = 1 \land \text{top}' = 1]
\]

In the case the buffer is empty, an input is cached. The ring indexes do not change and the buffer now contains a single item.
If the buffer is not empty, the cache is not changed; the indexes and the size of the ring are updated, but the ring itself is not changed.

The action below gets the new input and, if necessary, sends it to the ring using channel \textit{write}.

The extra value communicated through \textit{write} identifies the position in which the input is to be stored in the ring.

The handling of outputs by the controller can be specified in a similar way. For conciseness, we omit the definition of the action \textit{OutputController}. The behaviour of the controller is as follows.

After initialisation, inputs and outputs are offered repeatedly, whenever possible. The second set of paragraphs has as its state space \textit{RingState}, whilst preserving \textit{ControllerState}. The next action stores a value in the ring.
Although all state components are in scope, we confine the direct access to \textit{RingState} and receive the current value of \textit{top} through the \textit{write} internal channel.

\[ \text{StoreRing} \triangleq \text{write}\!\,?\,i\,?\,x \rightarrow \text{StoreRingCmd} \]

To send the value stored at a given position of the \textit{ring} requires no state change.

\[ \text{NewCacheRing} \triangleq \text{read}\!\,?\,!\,\text{ring}\![i] \rightarrow \text{Skip} \]

In its main action, the ring repeatedly offers the external choice between \textit{StoreRing} and \textit{NewCacheRing} actions.

\[ \text{RingAction} \triangleq \mu \, X \cdot (\text{StoreRing} \parallel \text{NewCacheRing}) \cdot X \]

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

\[ \cdot (\text{ControllerAction} \parallel [\{ \text{write, read} \}] \parallel \text{RingAction}) \parallel \{ \text{write, read} \} \]

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, which was obtained by data refining the actions \textit{BufferInit}, \textit{Input}, and \textit{Output} presented in the previous section. We could appeal to Definition 1, but the purpose is not to prove such obligations directly from the semantics of actions. Rather, the relevant tools are the algebraic laws of CSP (adapted for actions); however, they are not our concern here, as we concentrate on laws which relate processes.

### 5.3 Split centralised buffer into a controller and a ring

As a result of the previous development step, the process \textit{Buffer} has two disjoint sets of paragraphs with respect to \textit{ControllerState} and \textit{RingState}. Therefore, with an application of Law 1, \textit{Buffer} can be split into two independent processes: a controller and a ring process.

We call the first process \textit{Controller}; its paragraphs include \textit{ControllerState}, \textit{ControllerInit}, \textit{CacheInput}, \textit{StoreInputController}, \textit{InputController}, those that define \textit{OutputController}, and \textit{ControllerAction}. The latter is the main action. The second process, \textit{Ring}, includes \textit{RingState}, \textit{StoreRingCmd}, \textit{StoreRing}, \textit{NewCacheRing}, and \textit{RingAction}, which is the main action. The main action of \textit{Buffer} is the basis for its new definition.

\[ \text{process} \, \text{Buffer} \triangleq (\text{Controller} \parallel [\{ \text{write, read} \}] \parallel \text{Ring}) \parallel \{ \text{write, read} \} \]

This step is a direct application of Law 1.
5.4 The ring process as a promotion of ring cells

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. The ring cells communicate over channels rd and wrt.

```
channel rd, wrt : N
process Ring ≡ begin
```

A ring cell is required to store only a natural number; the ring is simply a sequence of cells.

```
CellState ≡ [ val : N ]
RingState ≡ [ ring : seq CellState | #ring = maxring ]
```

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

```
Read ≡ rd!val → Skip
```

The Write action updates val.

```
CellWrite ≡ [ ΔCellState; x? : N | val' = x? ]
Write ≡ wrt?x → CellWrite
```

The ring cell allows either Read or Write actions.

```
RingCellController ≡ μ X • (Read □ Write); X
```

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

\[
\begin{align*}
\text{Promotion} \\
\Delta\text{CellState} \\
\Delta\text{RingState} \\
i? : 1 \ldots \text{maxring}
\end{align*}
\]

\[
\begin{align*}
\theta\text{CellState} &= \text{ring}[i?] \\
\text{ring'} &= \text{ring} \oplus \{ i? \mapsto \theta\text{CellState}' \}
\end{align*}
\]

\[\text{StoreRingCmd}\text{ is defined as a promotion of } \text{CellWrite}, \text{ in a standard way.}\]

\[\text{StoreRingCmd} ≡ \exists \text{CellState} • \text{CellWrite} \land \text{Promotion}\]

The StoreRing action is not touched. It is a prefixing involving the action StoreRingCmd, which has already been promoted. The values it communicates are not in the local state, and so are not affect by promotion.

```
StoreRing ≡ write?i?x → StoreRingCmd
```

17
If we consider that the promotion of the channel \(\texttt{wrt}\) is the channel \(\texttt{write}\), then \(\texttt{StoreRing}\) is the result of promoting \(\texttt{Write}\).

The \(\texttt{NewCacheRing}\) action is defined by promoting \(\texttt{Read}\) in a similar way.

\[
\texttt{NewCacheRing} \triangleq \texttt{read?}!\texttt{ring[i]} \to \texttt{Skip}
\]

The promotion of \(\texttt{rd}\) is \(\texttt{read}\). Promoting \(\texttt{val}\) we get \(\texttt{ring[i]}\).

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

\[
\texttt{RingAction} \triangleq \mu X \bullet (\texttt{StoreRing} \boxdot \texttt{NewCacheRing}); X
\]

\* \(\texttt{RingAction}\) \*

The actions involved, however, have been promoted. This step can be justified by a simulation relating the sequence of cells to the sequence of natural numbers.

### 5.5 A distributed cached-head ring buffer

This is the final step of the development process, where each ring cell is implemented as an independent \(\texttt{Circus}\) process as the result of an application of Law 2 to \(\texttt{Ring}\). 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 2.

A process \(\texttt{RingCell}\) is defined to include the paragraphs \(\texttt{CellState}, \texttt{CellWrite}, \texttt{Write}, \texttt{and RingCellController}\) as the main action. An indexed ring cell is defined as follows.

\[
\text{process IRCell} \triangleq (i : 1..\texttt{maxring} \odot \texttt{RingCell})[\texttt{rd\_i}, \texttt{wrt\_i} := \texttt{read}, \texttt{write}]
\]

The indexed process operates on the channels \(\texttt{rd\_i}\) and \(\texttt{wrt\_i}\), which have type \((1..\texttt{maxring}) \times \mathbb{N}\). We rename them to \(\texttt{read}\) and \(\texttt{write}\), respectively. The indexed ring cell behaves like a ring cell, except that the communications \(\texttt{rd!val}\) and \(\texttt{wrt?x}\) are replaced by \(\texttt{read.i!val}\) and \(\texttt{write.i?x}\).

The ring is constructed by interleaving the indexed ring cells.

\[
\text{process Ring} \triangleq \parallel i : 1..\texttt{maxring} \odot \texttt{IRCell}\{i\}
\]

There is no interaction between the ring’s cells, so the definition is appropriate as a refinement of a sequence. This results from a direct application of Law 2.

### 6 Related and future work

In this paper, we outlined a process for developing distributed implementations from centralised \(\texttt{Circus}\) specifications. Although the application domain is concurrent programming, the process is similar in spirit to the development techniques used for sequential programming.
We gave a semantic definition of refinement and a forwards simulation rule for proving refinements correct. We presented laws for distributing data refinement through some of the combinators of CSP and laws for splitting processes. These laws are new contributions. In particular, we single out Law 2, which establishes a connection (original to our knowledge) between the Z promotion technique (for sequential programming) and the indexed interleaving of the promoted elements, which are processes in *Circus*. Expressing this law has required a generalisation of promotion of schemas to promotion of actions.

Previous work in this area includes that of Back [1], who has applied the refinement calculus to the stepwise development of parallel and reactive programs. In his work, action systems are used as the basic program model: they may be regarded as sequential programs, but they can be implemented as parallel programs. Back’s parallel refinement uses techniques originally developed for the sequential refinement calculus. Our work differs in that it is based on concepts taken from CSP, rather than action systems.

The assumptions in Theorem 5 require freedom from interference between two parallel actions in a manner that is essentially the same as a free promotion in Z [20]. In the work of Owicki and Gries [13, 14], noninterference also plays an important rôle. Their theory extends Hoare’s deductive system for partial correctness of sequential programs [5] by adding parallelism in the form of co-blocks, synchronisation, mutual exclusion, and wait statements. In their method, processes are considered in isolation and a proof of sequential correctness is obtained. These proofs must then be shown to be free from interference: no wait statement or assignment outside a wait statement in one process interferes with the proof of any other. The specification of the parallel program is then the conjunction of the preconditions and the postconditions of the components.

In their later work, the use of critical regions reduces much of the burden of the proofs of interference freedom. An invariant is required for each shared variable, and proofs of invariance replace proofs of noninterference. The difference between our notion of interference and that in the Owicki-Gries work is that we are interested in a design pattern that guarantees noninterference; the design pattern (promotion) is introduced by data refinement.

Our current work includes a weakest precondition semantics for *Circus*, and the indications are that this leads to simpler proofs for the soundness of the laws of refinement. We have already shown that the notion of refinement in this predicate transformer model is equivalent to that in the unifying theory. We shall continue to explore this matter. We also intend to address the completeness of data refinement by considering backwards simulation. Finally, adaptation of algebraic laws of CSP for actions is required to allow us to justify the refinement steps in detail, leading to a refinement calculus for *Circus*.

Acknowledgements

This work is partially supported by the EPSRC grant GR/R43211/01 on “Refinement calculi for sequential and concurrent programs” and by QinetiQ. The
work of Ana Cavalcanti and Augusto Sampaio is partially supported by CNPq; grants 520763/98-0 and 521039/95-9. The work of Jim Woodcock is partially supported by QinetiQ. We are grateful to Arthur Hughes for his suggestions.

References