Refinement in *Circus*

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

[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*. $\mathsf{CircusPar}^*$ 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 $|\!\}$, 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.

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

Inputs and outputs are taken from two different channels.

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

$$
\begin{array}{lcl}
\mathsf{Program} & ::= & \mathsf{CircusPar}^* \\
\mathsf{CircusPar} & ::= & \mathsf{Par} \mid \mathbf{channel}\ \mathsf{CDecl} \mid \mathbf{chanset}\ \mathsf{N} == \mathsf{CSExp} \mid \\
& \mid & \mathbf{process}\ \mathsf{N} \mathrel{\widehat=} \mathsf{Proc} \\
\mathsf{CDecl} & ::= & \mathsf{SimpleCDecl} \mid \mathsf{SimpleCDecl};\ \mathsf{CDecl} \\
\mathsf{SimpleCDecl} & ::= & \mathsf{N}^+ \mid \mathsf{N}^+ : \mathsf{Exp} \mid \mathsf{Schema\text{-}Exp} \\
\mathsf{Proc} & ::= & \mathbf{begin}\ \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} \lll \mathsf{Proc} \mid \mathsf{Proc} \setminus \mathsf{CSExp} \\
& \mid & \mathsf{Decl} \odot \mathsf{Proc} \mid \mathsf{Proc} \lfloor \mathsf{Exp}^+ \rfloor \mid \mathsf{Process}[\mathsf{N}^+ := \mathsf{N}^+] \\
& \mid & \mathsf{Decl} \bullet \mathsf{Proc} \mid \mathsf{Proc}(\mathsf{Exp}^+) \mid [\mathsf{N}^+]\mathsf{Proc} \mid \mathsf{Proc}[\mathsf{Exp}^+] \\
\mathsf{PPar} & ::= & \mathsf{Par} \mid \mathsf{N} \mathrel{\widehat=} \mathsf{Action} \\
\mathsf{Action} & ::= & \mathsf{Schema\text{-}Exp} \mid \mathsf{CSPAction} \mid \mathsf{Command} \\
\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{CSExp} \rrbracket \mathsf{Action} \mid \mathsf{Action} \lll \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}^+) \\
\mathsf{Comm} & ::= & \mathsf{N}\ \mathsf{CParameter}^* \\
\mathsf{CParameter} & ::= & ?\,\mathsf{N} \mid ?\,\mathsf{N} : \mathsf{Predicate} \mid !\,\mathsf{Expression} \mid .\,\mathsf{Expression} \\
\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} \mid \mathbf{con}\ \mathsf{Decl} \bullet \mathsf{Action} \\
\mathsf{GActions} & ::= & \mathsf{Pred} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \rightarrow \mathsf{Action} \Box \mathsf{GActions}
\end{array}
$$

Fig. 1. *Circus* syntax

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.

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

$$\mathit{BufferState} \mathrel{\widehat=} [\, \mathit{buff} : \operatorname{seq} \mathbb{N};\ \mathit{size} : 1 \mathinner{..} \mathit{maxbuff} \mid \mathit{size} = \# \mathit{buff} \leq \mathit{maxbuff} \,]$$

Initially, the buffer is empty.

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

Input is possible if there is space in the buffer; the input element is appended to the bounded sequence and the size incremented.

InputCmd

$\Delta BufferState$
$x? : \mathbb{N}$

$size < maxbuff \land buff' = buff \frown \langle x? \rangle \land size' = size + 1$

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

The *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.

OutputCmd

$\Delta BufferState$

$size > 0$
$buff' = tail\ buff \land size' = size - 1$

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

Finally, the main action initialises the *Buffer* and repeatedly offers the choice of input and output.

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

end

The guards guarantee that *Input* is available only if the buffer is not full, and *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 *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 $(i : T \odot P)\lfloor e \rfloor$ 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 $P[oldc := newc]$, the communications of P through channel *oldc* are done through the channel *newc* instead. An example of the use of the indexing and renaming operators is found in our case study (Section 5).

The semantics of *Circus* [19, **?**] is based on unifying theories of programming [8]: 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 *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 $P = a \rightarrow Skip$; 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 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: $tr' = tr \land a \notin ref'$.

Finally, suppose that $wait'$ is false and so P has terminated. P must have added the event a to the trace: $tr' = tr \frown \langle a \rangle$. 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 ($\sqsubseteq_{\mathcal{A}}$).

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:* $A \sqsubseteq_{\mathcal{A}} B$ **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 ($\sqsubseteq_{\mathcal{P}}$).

Let $P.st$, $P.init$, and $P.act$ denote the local state, initialisation, and main action of a process P, respectively.

Definition 2 (Process refinement). *We define $P \sqsubseteq_{\mathcal{P}} Q$ to mean that process P is refined by process Q if, and only if,*

$$(\exists\, P.st;\ P.st' \bullet P.init \wedge P.act\,) \sqsubseteq_{\mathcal{A}} (\exists\, Q.st;\ Q.st' \bullet Q.init \wedge Q.act\,) \qquad \Box$$

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 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) $\quad [\,\forall\, Q.st \bullet Q.init \Rightarrow (\,\exists\, P.st \bullet P.init \wedge R\,)\,]$
2. (correctness) $\quad [\,\forall\, P.st;\ Q.st;\ Q.st' \bullet R \wedge B \Rightarrow (\,\exists\, P.st' \bullet R' \wedge A\,)\,]$

A forwards simulation between P and Q is a forwards simulation between their main actions. $\Box$

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 $P \sqsubseteq_{\mathcal{P}} Q$.*

Proof

$$\begin{aligned}
&\exists\, Q.st;\ Q.st' \bullet Q.init \wedge Q.act \\
&\Rightarrow \exists\, P.st;\ Q.st;\ Q.st' \bullet P.init \wedge R \wedge Q.act && \textit{[initialisation]} \\
&\Rightarrow \exists\, P.st;\ P.st';\ Q.st;\ Q.st' \bullet P.init \wedge R \wedge R' \wedge P.act && \textit{[correctness]} \\
&\Rightarrow \exists\, P.st;\ P.st' \bullet P.init \wedge P.act && \textit{[schema calculus]}
\end{aligned}$$

$\Box$

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) $[\forall P.st;\ Q.st \bullet R \land \text{pre } PSExp \Rightarrow \text{pre } QSExp\,]$
2. (correctness) $\left[\begin{array}{l}\forall P.st;\ Q.st;\ Q.st' \bullet \\ \quad R \land \text{pre } PSExp \land QSExp \Rightarrow (\exists P.st' \bullet R' \land PSExp\,)\end{array}\right]$

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

$$
\begin{array}{l}
R(P.st, Q.st) \land (B_1(Q.st, Q.st');\ B_2(Q.st, Q.st')) \\
\Leftrightarrow \exists\, Q.st_0 \bullet R(P.st, Q.st) \land B_1(Q.st, Q.st_0) \land B_2(Q.st_0, Q.st') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{[sequential composition]} \\
\Rightarrow \exists\, P.st_0, Q.st_0 \bullet R(P.st_0, Q.st_0) \land A_1(P.st, P.st_0) \land B_2(Q.st_0, Q.st') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{[assumption]} \\
\Rightarrow \exists\, P.st_0, P.st' \bullet R(P.st', Q.st') \land A_1(P.st, P.st_0) \land A_2(P.st_0, P.st') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{[assumption]} \\
\Leftrightarrow \exists\, P.st' \bullet R(P.st', Q.st') \land (A_1(P.st, P.st');\ A_2(P.st, P.st')) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{[sequential composition]}
\end{array}
$$

□

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 $c.pxp \rightarrow Skip$, 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, $[\,\Xi P.st;\ o! : V \mid o! = pxp\,]$. 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 $c.pxp \rightarrow Skip$ and $c.qxp \rightarrow Skip$.

Proof

$$\begin{array}{ll}
& R \wedge (c.qxp \rightarrow Skip) \\
\Leftrightarrow & R \wedge (\,(x' = x \wedge okay' \wedge \\
& \quad (Id \lhd wait \rhd (tr' = tr \wedge c.qxp \notin ref' \lhd wait' \rhd tr' = tr \frown \langle c.qxp \rangle))) \\
& \quad \lhd okay \rhd tr \ \mathsf{prefix}\ tr'\,) & \textit{[definition]} \\
\Leftrightarrow & R \wedge (\,(x' = x \wedge okay' \wedge \\
& \quad (Id \lhd wait \rhd (tr' = tr \wedge c.pxp \notin ref' \lhd wait' \rhd tr' = tr \frown \langle c.pxp \rangle))) \\
& \quad \lhd okay \rhd tr \ \mathsf{prefix}\ tr'\,) & \textit{[assumption]} \\
\Leftrightarrow & \exists\, P.st' \bullet \\
& \quad R' \wedge (\,(nx' = x \wedge okay' \wedge \\
& \quad (Id \lhd wait \rhd (tr' = tr \wedge c.pxp \notin ref' \lhd wait' \rhd tr' = tr \frown \langle c.pxp \rangle))) \\
& \quad \lhd okay \rhd tr \ \mathsf{prefix}\ tr'\,) & \textit{[existential introduction]} \\
\Leftrightarrow & \exists\, P.st' \bullet R' \wedge (c.pxp \rightarrow Skip) & \textit{[definition]}
\end{array}$$

□

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.* □

The parallel composition $A_1 \,[\![\, C \,]\!]\, A_2$ 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 $A_1 \upharpoonright S' \Rightarrow A_2 \upharpoonright S'$. (Here, $A_1 \upharpoonright S'$ 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, $(A_1 \upharpoonright S') \fatsemi R \Rightarrow (A_2 \upharpoonright S') \fatsemi R$. 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:*

$$
\begin{array}{ll}
A_1 \upharpoonright S' \Rightarrow A_2 \upharpoonright S' & (A_1 \upharpoonright S') \fatsemi R = (A_1 \fatsemi R) \upharpoonright T' \\
A_2 \setminus S' \Rightarrow A_1 \setminus S' & (A_1 \setminus S') \fatsemi R = (A_1 \fatsemi R) \setminus T' \\
(A_1 \upharpoonright S') \fatsemi R \Rightarrow (A_2 \upharpoonright S') \fatsemi R & (A_2 \upharpoonright S') \fatsemi R = (A_2 \fatsemi R) \upharpoonright T' \\
(A_2 \setminus S') \fatsemi R \Rightarrow (A_1 \setminus S') \fatsemi R & (A_2 \setminus S') \fatsemi R = (A_2 \fatsemi R) \setminus T'
\end{array}
$$

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

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

$$
\begin{array}{ll}
& R \fatsemi (B_1 \wedge B_2) \\
\Rightarrow & (R \fatsemi B_1) \wedge (R \fatsemi B_2) \qquad \textit{[schema calculus]} \\
\Rightarrow & (A_1 \fatsemi R) \wedge (A_2 \fatsemi R) \qquad \textit{[hypothesis]} \\
\Leftrightarrow & (((A_1 \fatsemi R) \upharpoonright T') \vee ((A_1 \fatsemi R) \setminus T')) \wedge (((A_2 \fatsemi R) \upharpoonright T') \vee ((A_2 \fatsemi R) \setminus T')) \\
& \qquad \textit{[T' and its complement partition the state space]} \\
\Leftrightarrow & (((A_1 \fatsemi R) \upharpoonright T') \wedge ((A_2 \fatsemi R) \upharpoonright T')) \vee (((A_1 \fatsemi R) \setminus T') \wedge ((A_2 \fatsemi R) \setminus T')) \\
& \qquad \textit{[schema calculus]} \\
\Leftrightarrow & ((A_1 \upharpoonright S') \fatsemi R) \vee ((A_2 \setminus S') \fatsemi R) \qquad \textit{[assumption]} \\
\Leftrightarrow & ((A_1 \upharpoonright S') \vee (A_2 \setminus S')) \fatsemi R \qquad \textit{[schema calculus]} \\
\Leftrightarrow & (((A_1 \upharpoonright S') \wedge (A_2 \upharpoonright S')) \vee ((A_1 \setminus S') \wedge (A_2 \setminus S'))) \fatsemi R \qquad \textit{[assumption]} \\
\Leftrightarrow & (((A_1 \upharpoonright S') \vee (A_1 \setminus S')) \wedge ((A_2 \upharpoonright S') \vee (A_2 \setminus S'))) \fatsemi R \qquad \textit{[schema calculus]} \\
\Leftrightarrow & (A_1 \wedge A_2) \fatsemi R \qquad \textit{[S' and its complement partition the state space]}
\end{array}
$$

□

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 $A \sqsubseteq_{\mathcal{A}} B$, 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.

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

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)

$$pd = (\,\mathbf{process}\ P \mathrel{\widehat=} F(Q.act, R.act)\,)$$

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, $pps = pps \upharpoonright s'$ and $pps' = pps' \upharpoonright s$, 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 *local*, whereas the collection is called *global*. When the local type is completely encapsulated (as an abstract data type) in the global type, we say that the promotion is *free*; otherwise it is called *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 *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 *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.

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

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

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

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.

$$\mathbf{promote}(c.e \rightarrow A) \mathrel{\widehat=} pc.\mathbf{promote}(e) \rightarrow \mathbf{promote}(A)$$

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.

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

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

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

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

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

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.

$$\begin{array}{l}
\mathbf{process}\ P \mathrel{\widehat=} \mathbf{begin} \\
\quad State \mathrel{\widehat=} [\, f : Range \pfun L.st \mid pred \,] \\
\quad L.action_k \uparrow State \\
\quad L.act \mathrel{\widehat=} \mu\, X \bullet F(L.action_k);\ X \\
\quad Promotion \mathrel{\widehat=} \\
\qquad [\, \Delta L.st;\ \Delta State;\ i? : Range \mid \\
\qquad\quad i? \in \mathrm{dom}\, f \land \theta L.st = f(i?) \land f' = f \oplus \{i? \mapsto \theta L.st'\} \,] \\
\quad action_k \mathrel{\widehat=} \mathbf{promote}(L.action_k) \\
\quad \bullet\ (\, \mu\, X \bullet F(action_k);\ X \,) \\
\mathbf{end}
\end{array}$$

As discussed before, the global state component is 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.

Consider also the indexed process IL below.

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

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.

Law 2 (Process indexing)

$$pd = \mathbf{process}\ P \mathrel{\widehat=} |||\, i : Range \odot IL\lfloor i \rfloor$$

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

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 *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.

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.

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

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

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

$$\textit{BufferState} \mathrel{\widehat=} \textit{ControllerState} \land \textit{RingState}$$

The constant *maxring*, defined as *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 *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 *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 *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, as follows.

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

These channels are hidden in the *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.

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

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.

CacheInput

$\Delta ControllerState$
$\Xi RingState$
$x? : \mathbb{N}$

$size = 0$
$size' = 1 \land cache' = x?$
$bot' = bot \land top' = top$

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.

StoreInputController

$\Delta ControllerState$
$\Xi RingState$
$x? : \mathbb{N}$

$0 < size < maxbuff$
$size' = size + 1 \land cache' = cache$
$bot' = bot \land top' = (top \bmod maxring) + 1$

The action below gets the new input and, if necessary, sends it to the ring using channel *write*.

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

The extra value communicated through *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 *OutputController*. The behaviour of the controller is as follows.

$$
\begin{array}{l}
ControllerAction \mathrel{\widehat=} ControllerInit; \\
\qquad\qquad \mu\, X \bullet (InputController \;\Box\; OutputController);\; X
\end{array}
$$

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*.

StoreRingCmd

$\Delta RingState$
$\Xi ControllerState$
$i? : 1 \mathinner{..} maxring$
$x? : \mathbb{N}$

$ring' = ring \oplus \{i? \mapsto x?\}$

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

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

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

$$NewCacheRing \mathrel{\widehat=} read?i!ring[i] \rightarrow Skip$$

In its main action, the ring repeatedly offers the external choice between *StoreRing* and *NewCacheRing* actions.

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

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

$$\bullet\ (ControllerAction \,[\![\, \{\!|\, write, read, |\!\} \,]\!]\, RingAction) \setminus \{\!|\, write, read \,|\!\}$$

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, which was obtained by data refining the actions *BufferInit*, *Input*, and *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 *Buffer* has two disjoint sets of paragraphs with respect to *ControllerState* and *RingState*. Therefore, with an application of Law 1, *Buffer* can be split into two independent processes: a controller and a ring process .

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

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

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*.

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

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

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

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

$$RingState \mathrel{\widehat=} [\, ring : \operatorname{seq} CellState \mid \#ring = maxring \,]$$

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.

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

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 $i?$.

Promotion

$$\Delta CellState$$
$$\Delta RingState$$
$$i? : 1 \,..\, maxring$$

$$\theta CellState = ring[i?]$$
$$ring' = ring \oplus \{i? \mapsto \theta CellState'\}$$

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

$$StoreRingCmd \mathrel{\widehat=} \exists\, CellState \bullet CellWrite \wedge Promotion$$

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.

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

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

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

$$NewCacheRing \mathrel{\widehat=} read?i!ring[i] \rightarrow Skip$$

The promotion of *rd* is *read*. Promoting *val* we get $ring[i]$.

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

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

$\bullet\ RingAction$

end

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 *Circus* process as the result of an application of Law 2 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 2.

A process *RingCell* is defined to include the paragraphs *CellState*, *Read*, *CellWrite*, *Write*, and *RingCellController* as the main action. An indexed ring cell is defined as follows.

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

The indexed process operates on the channels rd_i and wrt_i, which have type $(1\,..\,maxring) \times \mathbb{N}$. We rename them to *read* and *write*, respectively. 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.

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

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 *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

1. R. J. R. Back. Refinement of parallel and reactive programs. In *Proceedings of the Summer School on Program Design Calculi*, Lecture Notes in Computer Science. Springer-Verlag, 1992.
2. A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC—A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267—289, 1999.
3. E. W. Dijkstra. Guarded commands, nondeterminacy and the formal derivation of programs. *Communication of the ACM*, 18:453 – 457, 1975.
4. 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.
5. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576 – 580, 1969.
6. C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271—281, 1972.
7. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
8. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
9. 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.
10. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
11. C. C. Morgan and P. H. B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, 27(6):481—503, 1990.
12. J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287 – 306, 1987.
13. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319 – 340, 1976.
14. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach . *Communications of the ACM*, 19(5):279 – 285, 1976.
15. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
16. J. M. Spivey. *The Z Notation: A Reference Manual*. 2nd. Prentice-Hall, 1992.
17. J. C. P. Woodcock and A. L. C. Cavalcanti. A Concurrent Language for Refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.
18. J. C. P. Woodcock and A. L. C. Cavalcanti. The steam boiler in a unified theory of Z and CSP. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*. IEEE Press, 2001.
19. 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.

20. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.