A Concurrent Language for Refinement

Jim Woodcock
Oxford University/OUCL
Wolfson Building, Parks Road
Oxford OX1 3QD, England
e-mail: jimw@comlab.ox.ac.uk

Ana Cavalcanti
Centro de Informática/UFPE
P.O. Box 7851
50740-540 Recife - PE Brazil
e-mail: alcc@cin.ufpe.br

Abstract

We present a combination of the well-established formal specification languages Z and CSP; our objective is to provide support for the specification of both data and behaviour aspects of concurrent systems, and a development technique. The resulting language, *Circus*, distinguishes itself in that it is aimed at the calculational refinement of specifications to programs written in a language similar to occam and *Handel-C*. In this paper, we present *Circus*, the rationale for its design, and a case study in its use.

1 Introduction

Of all formal methods, the Z specification language [4, 24, 31] has been most widely accepted in academia and industry. Recently, proposals have been made to integrate it with a process algebra [15, 18, 21]; Fischer gives a survey of some of this research [11]. Such a combination has obvious advantages: Z is good at describing rich information structures in a system's state, and process algebra is good at describing behavioural patterns of communication and synchronisation. Several interesting languages have been proposed, but very little has been accomplished in terms of understanding the formal development of programs starting from such specifications.

Of all the process algebras, CSP [15, 21] is perhaps the most successful for industrial application. Its key feature is that the language has been designed around the notion of refinement, making it suitable for the development of large-scale systems. Commercial tools for analysis and simulation of CSP specifications are available [13, 12].

With the aim of proposing and formalising a refinement calculus for concurrent programs, we have designed *Circus*. It combines Z and CSP, but it also includes specification constructs usually found in refinement calculi [19, 2, 20] and Dijkstra's language of guarded commands [9]. As a refinement language, *Circus* is a unified programming language, in which we can write specifications, designs, and programs.

The development technique we want to provide is in the style of [19]. In particular, we want to rely on ZRC, the refinement calculus for Z presented in [6, 7]. Therefore, specifications in *Circus* are based largely 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 that change the state. The result is a general programming language adequate for developing concurrent programs.

In the next section we present *Circus*, giving its syntax, its well-formedness restrictions, and the grounds for its design. Section 3 presents a case study in the use of *Circus*. Finally, in Section 4 we present our conclusions, along with related and future work.

2 *Circus*

Like in a Z specification, a *Circus* program is formed by a sequence of paragraphs; each of these can either be a Z paragraph, a channel definition, a channel set definition, or a process definition. Figure 1 presents part of the BNF description of the syntax of *Circus*, omitting for brevity the syntax of channel set expressions, communications, and guarded commands. We use $\mathsf{CircusParagraph}^*$ to denote a list of 0 or more elements of the syntactic category CircusParagraph; similarly for $\mathsf{PParagraph}^*$. The notation N^+ is used for a comma-separated list of identifiers and similarly for $\mathsf{Expression}^+$. The syntactic category N is that of the valid Z identifiers. The categories called Paragraph, Schema-Exp, Predicate, and Expression are, as expected, those of Z paragraphs, schema expressions, predicates, and expressions; their definitions are standard and can be found in [24].

To explain the main constructs of *Circus*, we use a small example taken from [15]: we define a process that outputs the Fibonacci sequence.

2.1 Channels

A channel definition declares the channels to which the processes can refer: it gives the name of each of the channels and the type of the values it can communicate. Our example process outputs through a channel *out* that communicates natural numbers.

channel $out : \mathbb{N}$

More than one channel can be declared in such a paragraph. When a channel is not used to communicate values, but just as a synchronisation event, its declaration consists of only its name: no type is defined. Finally, we can use a schema to declare channels. Such a schema groups channel declarations, but does not have a predicate part. The notion of type here is more general than that of the maximal type of Z; we need to make sure that the values output on a channel belong to its declared type.

Sets of previously defined channels may be introduced in a **chanset** paragraph. We give a name to the channel set and a channel-set expression that determines the members of this set. The syntactic category CSExpression of channel set expressions contains the empty set of channels $\{\!|\,|\!\}$, channel enumerations enclosed in $\{\!|$ and $|\!\}$, and set expressions formed by the usual set operators. These sets of channels are used in process expressions.

Program	::=	CircusParagraph*
CircusParagraph	::=	Paragraph
	|	ChannelDefinition | ChanSetDefinition | ProcessDefinition
ChannelDefinition	::=	**channel** CDeclaration
CDeclaration	::=	SimpleCDeclaration | SimpleCDeclaration; CDeclaration
SimpleCDeclaration	::=	N^+ | N^+ : Expression | Schema-Exp
ChanSetDefinition	::=	**chanset** N == CSExpression
ProcessDefinition	::=	**process** N $\widehat{=}$ Process
Process	::=	**begin** PParagraph* • Action **end** | N
	|	Process; Process | Process $\Box$ Process | Process $\sqcap$ Process
	|	Process $[\![$ CSExpression $]\!]$ Process | Process $\|\|\|$ Process
	|	Process $\setminus$ CSExpression
	|	Declaration $\odot$ Process | Process$[$Expression$^+]$ | Process$[N^+ := N^+]$
	|	Declaration • Process | Process(Expression$^+$)
	|	$[N^+]$Process | Process[Expression$^+$]
PParagraph	::=	Paragraph | N $\widehat{=}$ Action
Action	::=	Schema-Exp | CSPActionExp | Command
CSPActionExp	::=	*Skip* | *Stop* | *Chaos*
	|	Communication $\rightarrow$ Action | Predicate & Action
	|	Action; Action | Action $\Box$ Action | Action $\sqcap$ Action
	|	Action $[\![$ CSExpression $]\!]$ Action | Action $\|\|\|$ Action
	|	Action $\setminus$ CSExpression | μ N • Action
	|	Declaration • Action | Action(Expression$^+$)

Figure 1: *Circus* syntax

2.2 Processes

A process definition declares its name and gives a process specification. The most basic sort of process specification is formed by a sequence of process paragraphs and a distinguished nameless action at the end delimited by **begin** and **end**. A process paragraph can be a Z paragraph or an action definition; together, they define the state and the behaviour of the process. In Figure 2, we define a *Circus* process that generates the Fibonacci sequence. The internal state of the process is described in the schema *FibState* to contain two natural numbers, x and y. The latter records the last value output, and the former records the value output before the last.

The definitions that follow are action specifications. The behaviour of *Fib* is described by the last, unnamed action; *Fib* behaves first as described by the action *InitFib* and then as described by the action *OutFib*. We use the CSP sequential composition operator.

The action *InitFib* outputs the number 1 twice and then records this by initialising the state components. It uses the prefix operator of CSP twice to output 1 through *out*

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

$FibState \mathrel{\widehat=} [\, x, y : \mathbb{N} \,]$

$InitFibState \mathrel{\widehat=} [\, FibState' \mid x' = y' = 1 \,]$
$InitFib \mathrel{\widehat=} out!1 \rightarrow out!1 \rightarrow InitFibState$

$OutFibState \mathrel{\widehat=} [\, \Delta FibState;\ next! : \mathbb{N} \mid next! = y' = x + y \land x' = y \,]$
$OutFib \mathrel{\widehat=} \mu\, X \bullet \textbf{var}\ next : \mathbb{N} \bullet OutFibState;\ out!next \rightarrow X$

$\bullet\ InitFib;\ OutFib$

end

Figure 2: A Fibonacci generator

and the schema *InitFibState* to initialise the state. This is a schema that follows the standard style of Z of defining initialisation operations.

The action *OutFib* is defined recursively with the use of the CSP operator μ. It consists of a local variable definition, an operation on the state, an output on the *out* channel, and a recursive call. The declaration of *next* is required so that it is in scope for both the operation schema *OutFibState* and the outputting action $out!next \rightarrow X$.

The schema *OutFibState* actually defines the value of *next*!, which represents the value of *next* in the state after the execution of *OutFibState*. The outputting action refers to *next*, the value of this variable in the state before its execution. In pure Z, dash and shriek decorations are used to refer to after-state and output variables, respectively. In the above example, however, we can use either *next*! or *next*′ to refer to the after-state value of *next*. Our choice has the purpose of emphasising the fact that *next* is a local variable, and so not really part of the state of *Fib*, and its value is output in the next action. In *Circus*, dashes and shrieks can be used interchangeably.

In summary, the action *OutFib* first behaves like *OutFibState*. This changes the state: it records in y the next output value $x + y$ and records in x the value of the previously output value y. This action also initialises the value of *next* to be $x + y$. Afterwards, *OutFib* outputs the value of *next* and then proceeds recursively.

It is possible to give a simpler definition to *OutFib*. As already explained, we define the output using the schema component *next*!; the following action outputs this value. As a consequence, we have to bring *next* into scope, which we do using a local variable declaration. This construction is very useful for implicit specifications, but here the output is deterministic, so we can write the action as follows.

$OutFibState \mathrel{\widehat=} [\, \Delta FibState \mid y' = x + y \land x' = y \,]$
$OutFib \mathrel{\widehat=} \mu\, X \bullet out!(x + y) \rightarrow OutFibState;\ X$

In this case, the action *OutFibState* only changes the state. As for the action *OutFib*, it first outputs the value $x + y$, then it changes the state, and finally proceeds recursively.

2.3 Process Expressions

A process definition like that of *Fib* is explicit, in that its definition uses Z and CSP constructs to define the state and the behaviour of the process. We can also use CSP operators to define processes in terms of others previously defined.

We can use sequence, internal and external choice, parallel, and interleaving operators. For instance, we can define a process *FibTwice* as follows.

$$FibTwice \mathrel{\widehat=} Fib \mid\mid\mid Fib$$

In this case we are using the interleave operator: communications in either process occur independently, with no need for synchronisation.

The state of the resulting process includes all the components of the state of the operand processes. In the above example, since both operands are *Fib*, the state of *FibTwice* includes two copies of the components of *FibState*.

The behaviour of the resulting process is defined by composing the actions that determine the behaviour of the operand processes using the operator applied. The process *FibTwice* outputs the Fibonacci sequence through the channel *out* twice. The sequences are merged arbitrarily, as the communications occur independently. Further examples of the use of the above process operators are presented in the next section.

Our parallel operator is alphabetised: an extra argument determines the channels on which the operand processes are required to synchronise (following [21], rather than [15]). Communications through channels that are not listed occur independently.

We can also use hiding to define a process. In this case, the state and behaviour of the resulting process is exactly like that of the operand process, except only that communications through the specified channels are hidden from the environment.

A *Circus* operator not available in CSP is indexing. We can, for instance, define a process as $i : T \odot P$. It behaves like P, but uses different channels. For each channel c of P, we have a fresh channel c_i, which must be a fresh channel name. It communicates pairs: the first element, the index, is a value of type T, and the second is a value of the type of c. The declarations of the channels c_i are implicit. The index is a parameter; correspondingly, we have an instantiation operation. If P is an indexed process, $P\lfloor e \rfloor$ behaves like P, but communicates, through all the channels, pairs whose first element is the value of the expression e. The value of e is the value of the index.

The process $i : \{1, 2\} \odot Fib$ outputs the pairs $(i, 1)$, $(i, 1)$, $(i, 2)$, $(i, 3)$, $(i, 5)$, ... through out_i, where in each case i is 1 or 2. The instantiated process $(i : \{1, 2\} \odot Fib)\lfloor 1 \rfloor$ outputs only pairs with 1 as first element; similarly for $(i : \{1, 2\} \odot Fib)\lfloor 2 \rfloor$. The interleaving $(i : \{1, 2\} \odot Fib)\lfloor 1 \rfloor \mid\mid\mid (i : \{1, 2\} \odot Fib)\lfloor 2 \rfloor$ behaves in a similar way to *FibTwice*, but each element of the output sequence is a pair whose first element identifies the Fibonacci generator that produced the second element.

We can declare an arbitrary number of indexes of arbitrary types with the indexing operator. The communicated values are tuples whose last elements are the values originally communicated and the others are values for the indexes; the instantiation operation, therefore, may take a list of indexes as argument. Partial instantiation is possible.

In CSP, indexing is achieved by renaming. Channels do not have types: a communication of a value 2 through a channel c is regarded as an event $c.2$. To achieve the effect of a channel c that can communicate natural numbers, we need the infinite set of events containing $c.0$, $c.1$, $c.2$, and so on. Indexing amounts to defining a process $l.P$, where l is a label, or rather, a name. This renames all channels, resulting in a process that engages in $l.c$ when P engages in c. We can also apply an injective function f on event names to a process P to obtain a process that uses channel $f(c)$ in the same way that P uses c.

In *Circus*, channels have a name and a type, due to the need for strong typing in the spirit of Z. Therefore, we have two operations: the first, indexing, changes the type of the channels; the second is renaming. In the process $P[old := new]$, the communications of P through channel old are done through the channel new, which is implicitly declared by this operation, if it has not already been declared. Usually, indexing and renaming are used in conjunction. An example is presented in the next section.

Parametrisation of processes is also available: parameters can be used in the process specification as values of their declared type. For example, we could define *Fib* to have parameters $a, b : \mathbb{N}$, declared just before the process specification. They could be used, for instance, to initialise the state components x and y. In this case, the instantiated process $Fib(5, 8)$ outputs the Fibonacci sequence starting from its fourth element.

We can also define generic processes; the mechanism is similar to that of a generic schema in Z. In the process $[X]P$, the name X is a generic parameter that can be used as a type in the definition of P. Later references to P need to define a value for this parameter. It can be either inferred from the context or defined explicitly as in $P[\mathbb{N}]$.

2.4 Actions

As already explained, an explicit process definition contains both Z and CSP constructs. Mainly, we have a Z specification where some paragraphs are action definitions. There is also a main action, which defines the behaviour of the process. As exemplified in the definition of the *Fib* process, an action can be a schema, a CSP process, a guarded command, or a combination of these constructs.

A schema expression defining an operation over the process state is an action. It changes the state, but does not communicate any value.

The action *Skip* terminates immediately, without communicating any value or changing the state; the action *Stop* deadlocks, and *Chaos* diverges. The prefixing operator is standard, but can be associated with the guard construct. The action $p \mathbin{\&} c?x \rightarrow A$, for instance, inputs a value through channel c, assigns it to the variable x, and then behaves like the action A, if the condition p is true; otherwise, it blocks: the predicate is an enabling condition. A guard may be associated with any kind of action.

In an action, all free variables have to be in scope; the state components are always in scope, and input communications introduce further variables into scope. Input variables, however, may not be used as the target of assignment statements.

We can also use the CSP operators of sequence, internal and external choice, parallelism, interleaving, hiding, and recursion. We observe that CSP operators are used both at the level of processes and at the level of actions. The operations that apply, however,

are different in each case. At the level of processes, we do not handle communications and, for the sake of simplicity, we do not have recursive definitions. The usefulness and the modelling of recursive processes is a topic for further research.

Like parametrised processes, parametrised actions declare variables that can be used locally in their definition. Instantiations give fixed values to these parameters.

In the parallel composition $A \,[\![\, C \,]\!]\, B$, the user state is affected by both A and B. It is the responsibility of the programmer to guarantee that no conflict arises. As a general policy, operations that modify the state should not be run in parallel with other operations that also modify the state; this restriction is enforced by occam, for instance. We must observe, however, that a *Circus* process, as opposed to an action, encapsulates its state. Therefore, we can run processes in parallel without the need to worry about interference.

An action can also be defined using Dijkstra's guarded commands. An action can be an assignment, possibly multiple, or a guarded alternation. For example, using an assignment, the *InitFib* action of the process *Fib* can be defined as follows.

$$InitFib \mathrel{\widehat=} out!1 \rightarrow out!1 \rightarrow x, y := 1, 1$$

As in the definition of *OutFib*, we can also use variable blocks. To support a calculational approach to development, an action can also be a specification statement in the style of Morgan's refinement calculus [19]. Finally, we can also declare logical constants.

The model of *Circus* is the *Unifying Theory of Programming* of Hoare & He [16]. We do not present this model here, but it can be found in a companion paper [30].

3 Case study: a reactive buffer

In this section, we give a specification in *Circus* of a simple bounded reactive buffer that is used to store natural numbers. We go on to describe a possible implementation as a ring of cells with a central controller and a cached head.

3.1 Abstract behaviour

As already explained, *Circus* specifications are sequences of paragraphs containing Z paragraphs, channel definitions, channel set definitions, or process definitions. Typically, standard Z paragraphs are used to define given sets and global constants that are used in several process definitions throughout a specification.

The buffer is bounded in its length: it may hold no more than *maxbuff* elements.

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

It is sensible to require that it can hold at least one value.

Usually, the definition of a process is preceded by a declaration of channels.

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

The process *Buffer* has two channels: *input* and *output*.

3.1.1 Process state

The state of a process is defined as in Z. For the *Buffer*, there are two state components: the contents of the buffer and its size, a derived component.

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

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

The size of the buffer has to be less than or equal to $maxbuff$.

3.1.2 Process actions

In describing a *Circus* process, a series of actions is defined. For the *Buffer*, there are three actions: initialisation, input, and output. Initialisation sets the buffer to empty.

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

The input action is enabled if there is space in the buffer for the new input; the corresponding input operation on the state has this as its precondition. The new element is appended to the buffer's contents and the size is updated.

InputCmd

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

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

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

The action *Input* guarantees the precondition by being guarded.

The output action is enabled if there is something in the buffer. The first element is output; the others are retained in order; the size of the buffer is suitably updated.

OutputCmd

$\Delta BufferState$

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

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

The output is actually defined in the *Output* action.

In every *Circus* process an unnamed action defines the externally-visible behaviour.

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

end

First, the buffer is initialised, and then it loops offering to the environment the choice to input and output. The guards of these actions guarantee that if the buffer is full, the input is blocked, and if it is empty, the output is blocked.

3.2 A cached-head ring buffer

A well-known implementation of a FIFO-buffer uses a ring: a circular array, with two indexes showing where the first and last elements reside. Our refinement uses a ring of cells, each implemented as a *Circus* process, and a central controller that keeps track of the indexes of the first and last elements, offering the input and output services.

An interesting problem arises immediately: the buffer must not refuse to output if it is non-empty, so how should this be achieved? One way is to distribute control around the ring, so that the cell owning the head of the buffer is enabled for output, but the others are disabled. The cost of this solution is the overhead of the protocol for distributed control. Another solution is to cache the head of the ring in the controller, and distribute only the tail of the buffer around the ring. The resulting protocol is very simple.

3.2.1 Controller process

The process *Controller* specifies the behaviour of the buffer, but does not contain the buffer itself in its state, just the cache; the ring is specified by another process. First, there are two global declarations to define the indexes of the ring cells. The maximum size of the ring is one less than the size of the buffer, as the head is cached.

$$maxring : \mathbb{N}$$
$$\hline$$
$$maxring = maxbuff - 1$$

The indexes of the ring go from 0 to $maxring - 1$.

$$RingIndex == 0 \mathinner{..} maxring - 1$$

The channels *read* and *write* are used for communication with the ring cells.

$$\textbf{channel}\ read, write : RingIndex \times \mathbb{N}$$

The values communicated through *read* and *write* are pairs, where the first element identifies a cell, and the second element is the natural number actually communicated.

3.2.2 Controller state

The state of the controller contains the size of the buffer, the size of the ring, the cache, and two ring indexes, *top* and *bot*, keeping track of the index of the next available position

and the index of the first value stored.

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

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

The size of the ring may be computed from the positions of the *top* and *bot* indexes: the number of elements between the two indexes is $(top - bot) \bmod maxring$. Since this confuses two values of the state (when the ring is full and when the ring is empty), it is necessary to add the first equation that relates *ringsize* and *size*.

3.2.3 Controller actions

Initially, the buffer is empty; we choose some suitable values for *top* and *bot*.

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

The input action depends on whether the buffer is empty or not. If it is empty, then the input must be kept in the cache; if it is non-empty, then it must be passed on to the appropriate ring cell; if it is full, then no input action is possible. When the input is cached, the *top* and *bot* indexes do not change.

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

When the input is passed on to the ring, the *top* index advances.

$$
\begin{array}{|l}
\hline
StoreInput \\
\Delta ControllerState \\
\hline
size > 0 \\
size' = size + 1 \land cache' = cache \\
bot' = bot \land top' = (top + 1) \bmod maxring \\
\hline
\end{array}
$$

The input action is enabled when there is space in the buffer; the subsequent behaviour

depends on whether the buffer is empty or not. If it is non-empty, the controller transmits the input x to the cell at the *top* of the ring.

$$\begin{aligned} \mathit{InputController} \mathrel{\widehat=} \mathit{size} < \mathit{maxbuff} \ \&\ \mathit{input}?x \rightarrow \\ \quad \mathit{size} = 0 \ \&\ \mathit{CacheInput} \\ \quad \Box \\ \quad \mathit{size} > 0 \ \&\ \mathit{write.top}!x \rightarrow \mathit{StoreInput} \end{aligned}$$

We observe that there is not, for this action, a one-to-one correspondence between the communication $\mathit{input}?x$ and operations that change the state.

There is a similar case analysis for output: the output always comes from the cache, which must be replaced if the ring is non-empty. In the case that the ring is empty, we have $\mathit{size} = 1$; *size* is reset; nothing else changes.

$$\begin{aligned} &\mathit{NoNewCache} \mathrel{\widehat=} \\ &\quad [\, \Delta \mathit{ControllerState} \mid \mathit{size} = 1 \land \mathit{size}' = 0 \land \mathit{bot}' = \mathit{bot} \land \mathit{top}' = \mathit{top} \,] \end{aligned}$$

If the ring is non-empty, then a new element (obtained from the ring) is stored in the cache; *bot* must be advanced.

StoreNewCache

$\Delta \mathit{ControllerState}$
$x? : \mathbb{N}$

$\mathit{size} > 1$
$\mathit{size}' = \mathit{size} - 1 \land \mathit{cache}' = x?$
$\mathit{bot}' = (\mathit{bot} + 1) \bmod \mathit{maxring} \land \mathit{top}' = \mathit{top}$

So, the output action is enabled when the buffer is non-empty. If the ring is non-empty, the controller obtains the input x from the cell at the *bot* of the ring.

$$\begin{aligned} \mathit{OutputController} \mathrel{\widehat=} \mathit{size} > 0 \ \&\ \mathit{output}!\mathit{cache} \rightarrow \\ \quad \mathit{size} > 1 \ \&\ \mathit{read.bot}?x \rightarrow \mathit{StoreNewCache} \\ \quad \Box \\ \quad \mathit{size} = 1 \ \&\ \mathit{NoNewCache} \end{aligned}$$

Neither of the operation schemas describe the value communicated on *output*.

3.2.4 Controller behaviour

We conclude the description of the controller by describing its overall behaviour: after being initialised, it repeatedly offers inputs and outputs.

$$\bullet\ \mathit{InitController};\ \mu X \bullet (\mathit{InputController} \Box \mathit{OutputController});\ X$$

end

We have now to specify the ring.

3.2.5 Ring cell process

Each cell has a *rd* and a *wrt* channel.

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

A cell contains one value.

process $RingCell \mathrel{\widehat=} \textbf{begin}\ CellState \mathrel{\widehat=} [\, val : \mathbb{N}]$

There are just two actions on the ring cell state. The *Read* action is so simple that we do not bother to define a corresponding operation on the state: it merely outputs *val*.

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

The *Write* action updates *val*.

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

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

The ring cell starts with a *Write* action; subsequently, it allows *Read* or *Write* actions.

$$\bullet\ Write;\ \mu X \bullet (Read \Box Write);\ X$$

end

The ring itself is formed by composing the cells in parallel.

3.2.6 The cached-head ring buffer's behaviour

Our first step is to assemble the ring. From *RingCell*, we define an indexed process, with indexes taken from the set of ring indexes.

process $IRingCell \mathrel{\widehat=} (i : RingIndex \odot RingCell)[read, write / rd\_i, wrt\_i]$

The resulting indexed process operates on channels rd_i and wrt_i of type $RingIndex \times \mathbb{N}$. We rename them to *read* and *write*, respectively. The behaviour of the indexed ring cell is exactly the same as that of a ring cell, except that the communications $rd!val$ and $wrt?x$ are replaced by $read.i!val$ and $write.i?x$, respectively.

There is no interaction between the ring's cells, so the ring is constructed by interleaving the indexed ring cells.

process $Ring \mathrel{\widehat=} |||\, i : RingIndex \odot IRingCell\lfloor i \rfloor$

Here we are using an iterated combination of $IRingCell\lfloor i \rfloor$ using interleaving. *Ring* is defined as the interleaving of the processes $IRingCell\lfloor v \rfloor$, with v in *RingIndex*. Iterated operators like this are available in *Circus*, but are omitted from Figure 1 for brevity.

Finally, the cached-head ring is composed by putting the controller and the ring in parallel, interacting along the internal *read* and *write* channels.

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

This completes our specification.

4 Conclusions

We have presented a unified language of specification, design, and programming that combines Z and CSP, and is suitable for refinement. Many approaches to the integration of Z or one of its extensions with a process algebra have been presented elsewhere. Our main objective is not to propose yet another language, but provide support for the formal development of concurrent programs. Nevertheless, we believe the language presented here has some interesting characteristics of its own.

In [11], Fischer surveys several integrations of Z with a process algebra. This work considers the combinations of Z with CCS in [14, 26]; the combinations of Z with CSP in [10, 22]; and the combination of Z with Object-Z [5] in [10]. Several issues involved in the integration of Z with a process algebra are discussed.

Fischer differentiates two styles for combining Z and process algebra: syntactic and semantic. In the former, the combination has a single syntax, with semantic definitions lifted from the two languages. In the latter, the Z specification is identified with a process.

We adopt the first style: the model of *Circus* is the unifying theory of programming of Hoare & He. All the other combinations of Z and CSP cited above adopt the semantic approach. The syntactic approach provides a deeper integration of the notations; the disadvantage is that we have to define the semantics of both the Z and CSP constructs, but we are already using an existing semantic model. As we express this model using Z, we are able to use Z tools to analyse *Circus* specifications.

Another issue is the relation between CSP events and Z operations; Fischer identifies three possibilities. Firstly, there is the single event approach, in which there is a one-to-one correspondence between operations and events; this leads to abstractions where operations are atomic transactions. Refinement to code must preserve this abstract atomicity. Secondly, there is the double event approach, where operations are divided into two parts: a front-end involving input and a back-end involving output. This corresponds to a natural way of using CSP, but a less natural way of using Z. Finally, there is the multiple event approach, in which there is no fixed connection between operations and events.

In *Circus*, there is no identification of operations with events: we adopt the multiple event approach. Operations can be seen as occurring in the background between events. We observe, however, that we can both adopt the abstract single event style of specification, as in our abstract buffer example, and consider programs where events and state changes are not associated. For a refinement language this is certainly more adequate. Furthermore, we do not split the input and output of parameters of the Z operations.

In most state-based formalisms, an operation may be given a partial specification, and a meaning is given to what happens when the operation is activated outside its precondition. There are two obvious choices: the operation waits, which is Fischer's blocking semantics; or the operation aborts, Fischer's non-blocking semantics. In Z, operations are given a non-blocking semantics; in *Circus*, operations have the usual semantics. Blocking is specified by guards in actions. Therefore, we can capture the enabling and the divergence conditions of an operation, while maintaining the usual semantics of Z operations.

A further issue is related to the typing of inputs. The Z type system is very simple: types are maximal, and there are only four constructors (given set, powerset, cartesian

product, and schema product); however, it is common practice to constrain components to range over subsets of types. As inputs are non-blocking, then the activation of a Z operation with an input outside its constraint leads to abort; in CSP, such an input is blocked. In *Circus*, this is exactly what happens.

Finally, regarding the issue of refinement, we observe that the major objective of *Circus* is to provide a theory of refinement and an associated calculus. Refinement has been studied for combinations of Object-Z and CSP [23], but nothing in the style of a calculus has been proposed.

At the moment, we are working on a number of more substantial case studies on the use of *Circus*. We are considering an IP-packet filter firewall [32], the steam boiler control system [1, 3, 29, 28], a smart card system for electronic finance [25], and a railway signalling application [27].

The model of *Circus* is a Z specification that can be analysed with almost no changes using the Z/EVES theorem prover [17]. Analysing the model and its properties using Z/EVES is in our research agenda. We are also building a tool that calculates the Z specification corresponding to a *Circus* program, for analysis using Z/EVES.

We are already considering the extension of *Circus* to include the operators of Timed CSP [8]. The resulting language is expected to be adequate to the specification of data, behavioural, and timing aspects of real-time systems. We intend to define its model by extending the unifying theory of programming to cover aspects of time.

Our main goal, however, is the proposal and proof of refinement laws for *Circus*. We want data refinement rules that allow, for instance, the proof that the process *CRing* defined in section 3 refines the process *Buffer*. We also want rules that allows the stepwise refinement of *CRing* to code.

Acknowledgements

We would like to thank Augusto Sampaio and Adnan Sherif for discussions and suggestions. The work of Ana Cavalcanti is financially supported by CNPq, grant 520763/98-0. Jim Woodcock gratefully acknowledges the support of CNPq and the University of Oxford for his visit to the Federal University of Pernambuco during July and August 2000.

References

[1] J. R. Abrial, E. Borger, and J. Langmaack, editors. *Formal Methods for Industrial Application*, volume 1165 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[2] R. J. R. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[3] J. C. Bauer. Specification for a software program for a boiler water content monitor and control system. Technical report, Institute of Risk Research, University of Waterloo, 1993.

[4] S. M. Brien and J. E. Nicholls. Z Base Standard, Version 1.0. Technical Monograph TM-PRG-107, Oxford University Computing Laboratory, Oxford - UK, November 1992.

[5] D. Carrington, D. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An Object-oriented Extension to Z. *Formal Description Techniques, II (FORTE'89)*, pages 281 – 296, 1990.

[6] A. L. C. Cavalcanti. *A Refinement Calculus for Z.* PhD thesis, Oxford University Computing Laboratory, Oxford - UK, 1997. Technical Monograph TM-PRG-123, ISBN 00902928-97-X.

[7] A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC - A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267 – 289, 1999.

[8] J. Davies. *Specification and Proof in Real-time CSP.* Cambridge University Press, 1993.

[9] E. W. Dijkstra. Guarded commands, nondeterminacy and the formal derivation of programs. *Communication of the ACM*, 18:453 – 457, 1975.

[10] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, volume 2, pages 423 – 438. Chapman & Hall, 1997.

[11] 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.

[12] Formal Systems (Europe) Ltd. *Process Behaviour Explorer - ProBE User Manual*, 1998.

[13] Formal Systems (Europe) Ltd. *FDR: User Manual and Tutorial, version 2.28*, 1999.

[14] A. J. Galloway. *Integrated Formal Methods with Richer Methodological Profiles for the Development of Multi-perspective Systems.* PhD thesis, University of Teeside, School of Computing and Mathematics, 1996.

[15] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[16] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice-Hall, 1998.

[17] I. Meisels. *Software Manual for Windows Z/EVES Version 2.1.* ORA Canada, 2000. TR-97-5505-04g.

[18] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[19] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[20] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287 – 306, 1987.

[21] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[22] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through Determinism. In D. Gollmann, editor, *ESORICS 94*, volume 1214 of *Lecture Notes in Computer Science*, pages 33 – 54. Springer-Verlag, 1994.

[23] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18:249–284, May 2001.

[24] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.

[25] S. Stepney, D. Cooper, and J. C. P. Woodcock. An Electronic Purse: Specification, Refinement, and Proof. Technical Monograph PRG-126, Oxford University Computing Laboratory, 2000.

[26] K. Taguchi and K. Araki. The State-based CCS Semantics for Concurrent Z Specification. In M. Hinchey and Shaoying Liu, editors, *International Conference on Formal Engineering Methods*, pages 283 – 292. IEEE, 1997.

[27] J. C. P. Woodcock. Montigel's Dwarf, a treatment of the dwarf-signal problem using CSP/FDR. In *Proceedings of the 5th FMERail Workshop*, Toulouse, France, September 1999.

[28] J. C. P. Woodcock and A. L. C. Cavalcanti. A *Circus* steam boiler: using the unifying theory of Z and CSP. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK, July 2001.

[29] 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)*, 2001.

[30] J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of Circus. In *ZB 2002 International Conference*, 2002. To appear.

[31] J. C. P. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[32] J. C. P. Woodcock and Alistair McEwan. Specifying a Handel-C program in the Unifying Theory. In *Proceedings of the Workshop on Parallel Programming*, Las Vegas, November 1999.