The Semantics of *Circus*

Jim Woodcock[1] and Ana Cavalcanti[2]

[1] Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, UK
Jim.Woodcock@comlab.ox.ac.uk
[2] Universidade Federal de Pernambuco/Centro de Informática
P.O. Box 7851, 50740-540 Recife PE, Brazil
alcc@cin.ufpe.br

Abstract. *Circus* is a concurrent language for refinement; it is a unification of imperative CSP, Z, and the refinement calculus. We describe the language of *Circus* and the formalisation of its model in Hoare & He's unifying theories of programming.

1 Introduction

An important research agenda in software engineering is the integration of languages and techniques that have proved themselves successful in specific aspects of software development. In particular, there has been interest for many years in combining state-based and behavioural formalisms [36]. There are several proposals for combining Z [4, 26] with process algebras [15, 19, 23]. With *Circus* [32], we also want a calculational method of program development as those of [2, 21, 22]; a similar calculus has been proposed for Z [7].

Circus is a unified programming language containing Z and CSP constructs, specification statements, and guarded commands [9]. It includes assignments, conditionals, and loops, and the reactive behaviour of communication, parallelism, and choice. 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 suitable for developing concurrent programs.

Imperative refinement calculi are normally given predicate transformers semantics; however, theories of refinement for CSP are based on the failures-divergences model [15, 23]. A connection between weakest preconditions and CSP does exist [20], and a sound and complete theory of refinement has been developed based on it [36]. We use the unifying theory of [16], where both state and communication aspects of concurrent systems are integrated in a state-based failures-divergences model. This leads to a simple and elegant definition of refinement and makes a good underpinning for a refinement calculus. Our goals are: ease of use for those familiar with Z and CSP; encapsulation of the model; and the possibility of reusing existing theories, techniques, and tools.

In the next section we give an overview of the structure of *Circus* specifications and define its syntax. Section 3 provides a brief introduction to unifying theories

Program	::=	CircusParagraph*
CircusParagraph	::=	Paragraph
	|	ChannelDefinition | ChanSetDefinition | ProcDefinition
ChannelDefinition	::=	**channel** CDeclaration
CDeclaration	::=	SCDeclaration | SCDeclaration; CDeclaration
SCDeclaration	::=	N^+ | N^+ : Expression | Schema-Exp
ChanSetDefinition	::=	**chanset** N == CSExpression
CSExpression	::=	$\{\!\|\ \|\!\}$ | $\{\!\|\ N^+\ \|\!\}$ | N
	|	CSExpression $\cup$ CSExpression | CSExpression $\cap$ CSExpression
	|	CSExpression $\setminus$ CSExpression
ProcDefinition	::=	**process** N $\widehat{=}$ Process
Process	::=	**begin** PParagraph* $\bullet$ Action **end** | N
	|	Process; Process | Process $\Box$ Process | Process $\sqcap$ Process
	|	Process $[\![$ CSExpression $]\!]$ Process | Process $\|\|\|$ Process
	|	Process $\setminus$ CSExpression
	|	Declaration $\odot$ Process | Process$\lfloor$Expression$^+\rfloor$
	|	Process$[N^+ := N^+]$
	|	Declaration $\bullet$ Process | Process(Expression$^+$)
	|	$[N^+]$Process | Process[Expression$^+$]

Fig. 1. *Circus* syntax

of programming before, in Section 4, we give the formal semantics of *Circus*. Finally, in Section 5 we discuss related and future work.

2 *Circus*

As with Z specifications, *Circus* programs are sequences of paragraphs: besides the Z paragraphs, we have channel and channel set definitions, and process definitions. In Figures 1 and 2 we present the specification of the *Circus* syntax. We use an extended BNF notation, where SC^* and SC^+ represent a possibly empty list and a comma-separated list of elements of the syntactic category SC. The syntactic category N is that of the valid Z identifiers. The definition for the Z constructs in the syntactic categories called Paragraph, Schema-Exp, Predicate, and Expression can be found in [26]. In Figure 3, as a simple example, we present a process that generates the Fibonacci sequence; more substantial examples of the use of *Circus* can be found in [32, 33].

Processes communicate with each other and the environment by means of channels, whose definitions declare the types of the values that are communicated through them. For the Fibonacci generator, we need to declare the channel *out*

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 $\bullet$ Action
| Declaration $\bullet$ Action | Action(Expression$^+$)

Communication ::= N CParameter*

CParameter ::= ? N | ? N : Predicate | ! Expression | . Expression

Command ::= N$^+$: [Predicate, Predicate] | N$^+$:= Expression$^+$
| **if** GuardedActions **fi**
| **var** Declaration $\bullet$ Action | **con** Declaration $\bullet$ Action

GuardedActions ::= Predicate $\rightarrow$ Action | Predicate $\rightarrow$ Action $\Box$ GuardedActions

Fig. 2. *Circus* syntax

through which the Fibonacci sequence is output.

channel $out : \mathbb{N}$

Channel declarations may be grouped in schemas. For conciseness, we can define channel sets to be used as arguments to the parallelism and hiding operators.

The definition of a process gives its name and its specification: state and behaviour. An explicit process specification like that of *Fib* is a Z specification interspersed with action definitions: the state is defined as in Z and the behaviour is defined by a distinguished nameless action at the end. Typically, this action is defined in terms of other actions previously defined in the process specification. In our example, the state contains two components: x and y of type $\mathbb{N}$. A proof-obligation requires us to prove that the invariant is not *false*; in this case and in most cases it is simple. The main action is the CSP sequence *InitFib*; *OutFib*.

An action can be a schema that specifies an operation over the process state, a command of Dijkstra's language of guarded commands, or formed using CSP operators. We also have specification statements as in Morgan's calculus [21], and logical constants. The action *InitFib* is defined using the prefixing operator: it outputs 1 twice through *out* and then behaves like the initialisation *InitFibState*.

In *OutFib*, firstly, a local variable is declared; secondly, the *next* Fibonacci number is calculated and the state is appropriately updated by *OutFibState*; thirdly, the number is output; and finally, *OutFib* proceeds recursively. The variable *next* is referred to in *OutFibState* as *next*!. Unlike in Z, we interchangeably use dashes and shrieks to decorate after-state variables. Our choice in the example emphasises that *next* is a local variable. The following simpler definition of

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

Fig. 3. Fibonacci generator

OubFib is possible, though.

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

If, however, an implicit specification of the output is needed, the previous style should be adopted.

As actions, processes can also be combined using CSP operators. The state of the combination encompasses those of the operands, and the behaviour is given by combining the main actions of the operands with the used operator. As an example, consider $Fib \mid\mid\mid Fib$. Its state comprises two copies of the state components of *Fib*; renaming is used to avoid clashes. As the interleaving operator is used, two Fibonacci sequences are merged arbitrarily through *out*.

The *Circus* indexing operator is novel: the process $i : T \odot P$ behaves like P, but operates on different channels. Each communication over a channel c is turned into a communication of a pair over a channel c_i. The first element of the pair is i, and the second is the value originally communicated. The declaration of c_i is implicit. The index i in $i : T \odot P$ is a parameter; the process $(i : T \odot P)\lfloor e \rfloor$ communicates pairs as explained above, but the first element of the pairs is e: the index.

For example, suppose we want to generate two Fibonacci sequences, but we want to identify the generator of each element. We consider the process $i : \{1, 2\} \odot Fib$. It outputs through channel out_i the pairs $(i, 1)$, $(i, 1)$, $(i, 2)$, $(i, 3)$, $(i, 5)$, and so on, where in each case i is either 1 or 2. The process $(i : \{1, 2\} \odot Fib)\lfloor 1 \rfloor$ produces pairs through out_i whose first elements are 1; similarly for $(i : \{1, 2\} \odot Fib)\lfloor 2 \rfloor$. Finally, $(i : \{1, 2\} \odot Fib)\lfloor 1 \rfloor \mid\mid\mid (i : \{1, 2\} \odot Fib)\lfloor 2 \rfloor$ produces an arbitrary merge of the two sequences of pairs: the first element of the pair identifies the generator and the second is a Fibonacci number.

In CSP, indexing is achieved by renaming since a communication of the value 2 over c is just an event name $c.2$. In *Circus*, this is not the case and we need to handle channel names and types separately. The reason for this distinction is the need for strong typing of communications in the spirit of Z.

The process $P[oldc := newc]$ is similar to P, but communicates through the channel *newc* where P would communicate through *oldc*. The declaration of *newc*, if necessary, is implicit.

Parametrisation of processes is similar to indexing, but is not related to channels. It declares extra variables, the parameters, which are available in the specification of the process. Instantiation fixes the value of these variables. The Z facility for defining generic paragraphs is extended to processes in *Circus*. In $[X]P$, the generic parameter X is introduced to be used in P as a type. Instantiation can be explicit as in $P[\mathbb{N}]$ or inferred from the context.

The CSP recursion and prefixing operators are not available for processes. Since there is no notion of a global state, communication is handled by actions. It is also not clear that recursive definitions of processes are useful.

3 Unifying Theories of Programming

In Hoare & He's unifying theory of programming [16], the theory of relations is used as a unifying basis for the science of programming across many different computational paradigms. Programs, designs, and specifications are all interpreted as relations between an initial observation and a subsequent (intermediate or final) observation of the behaviour of a device executing a program.

In their unification, different theories share common ideas: sequence is relational composition; the conditional is a simple Boolean connective; nondeterminism is disjunction; and parallelism is a restricted form of conjunction. The miracle of the refinement calculus is the empty relation and abortion is the universal relation. Making assertions conditional-aborts brings all of assertional reasoning within the scope of the theory. Both correctness and refinement are interpreted as inclusion of relations, and all the laws of a relational calculus are valid for reasoning about correctness in all theories and in all languages.

Particular design calculi and programming languages are differentiated by their alphabet, signature, and healthiness conditions. The *alphabet* of a theory gives names for a range of external observations of program behaviour. By convention, the name of an initial observation is undecorated, but the name of a similar observation taken subsequently is decorated with a dash. This allows a relation to be expressed as in Z by its characteristic predicate. The *signature* provides syntax for denoting the objects of the theory. It names the relations corresponding to primitive operations directly, and provides operators for combining them. The *healthiness conditions* select the objects of a sub-theory from those of a more expressive theory in which it is embedded. Thus programs form a subset of designs, and designs form a subset of specifications.

The alphabet of each theory contains variables to describe all aspects of program behaviour that are considered relevant. In a purely procedural paradigm, these stand for the initial and final values of the global variables accessed and updated by a program block. Some variables are *external*, because they are globally shared with the real world in which the program operates, and so they cannot be declared locally. The first example is the Boolean variable *okay*: it means that

the system has been properly started in a stable state; $okay'$ means subsequent stabilisation in an observable state. This permits a description of programs that fail due to nonterminating loops or recursion.

In a theory of reactive processes, the variable tr records past interactions between a process and its environment; and the Boolean variable *wait* distinguishes the intermediate observations of waiting states from final observations of termination. During a wait, the process can refuse to participate in certain events offered by the environment; these are specified by the variable *ref*.

The signature of a theory varies in accordance with its intended use, whether in specification, in design, or in programming. A specification language has the least restricted signature. Design calculi successively remove unimplementable operators, starting with negation; all operators are then monotonic, and recursion can safely be introduced as a fixed-point operator. In the programming language, only implementable operations are left.

A healthiness condition distinguishes feasible descriptions of reality from infeasible ones. By a suitable restriction of signature, design languages satisfy many healthiness conditions, and programming languages satisfy even more. Hoare & He have shown [16] that all healthiness conditions of interest may be expressed in the form $P = \phi(P)$, where ϕ is an idempotent function mapping all relations to the healthy ones of a particular theory. These idempotents link higher-level designs to lower-level implementations.

The laws of CSP are not true for all predicates over the observational variables; there are eight healthiness conditions for CSP processes: three characterise *reactive* processes in general; two constrain reactive processes to be CSP ones; and a further three are more specific still. The first healthiness condition for a reactive process (**R1**) is that its execution can never undo any event that has already been performed. The second healthiness condition (**R2**) requires that a reactive process's behaviour is oblivious of what has gone before. The third healthiness condition (**R3**) is designed to make sequential composition work as intended. Suppose that we have the sequential composition of two processes P_1 and P_2. When P_1 terminates, the value of $wait'$ is false; therefore, P_2's value of *wait* is also false, and control passes from P_1 to P_2. On the other hand, if P_1 is still waiting for interaction with its environment, then its value of $wait'$ and P_2's value of *wait* are both true; in this case, P_2 must leave the state unchanged. Of course, this is sensitive to the previous process's stability: if activated when *okay* is false, then its only guarantee is to extend the trace.

R is the set of reactive processes, which satisfy these first three healthiness conditions. An interesting subset of **R** satisfies two additional conditions. The first (**CSP1**) states that, if a process has not started, then we can make no prediction about its behaviour. The second (**CSP2**) states that we cannot require a process to abort; this is characterised by the monotonicity of the $okay'$ variable.

CSP is the set of reactive processes that satisfy these two healthiness conditions. Further healthiness conditions are required to capture the standard theory of CSP; each of them is given as a simple unit law. The first (**CSP3**) requires that a CSP process does not depend on the initial value of the *ref* variable

when *wait* is false. Of course, when *wait* is true, then it must behave as required by **R3**. The second (**CSP4**) requires that the ref' variable is irrelevant after termination. The third (**CSP5**) requires that refusal sets must be subset-closed.

4 The Model of *Circus*

Our model for a *Circus* program is a Z specification that describes processes and actions as relations. The model of a process is itself a Z specification, and the model of an action is a schema.

4.1 Channel environment

We use the Z mathematical notation as a meta-language. The semantics of a process depends on the channels in scope. These are recorded in an environment:

$$ChanEnv == ChanName \pfun \mathsf{Expression}$$

The given set *ChanName* contains the channel names. The channel environment associates a channel name to its type.

The semantic function $[\![\_]\!]^{\mathcal{CD}} : \mathsf{ChannelDefinition} \fun ChanEnv$ gives the meaning of channel definitions as channel environments recording their declarations. Untyped channels, used as synchronisation events, and are given the type *Sync*, a given set. For conciseness, we omit the definition of $[\![\_]\!]^{\mathcal{CD}}$, which can be found in [31] along with a few other definitions also omitted from this paper.

A channel is not a value in our model and so we cannot define a set of channels. In a *Circus* program, channel sets are used to abbreviate process expressions like parallelism and hiding. We assume that these process expressions are expanded by replacing references to channel sets with the set of channels it defines. The channel set definitions can then be eliminated.

4.2 Process environment

A process definition may refer to other processes previously defined and so we also need a process environment that associates process names to their models:

$$ProcEnv == \mathrm{seq}(ProcName \times \mathsf{ZSpecification})$$

The given set *ProcName* contains the valid process names, and ZSpecification is the syntactic category of Z specifications. We use sequences because the order in which processes are declared is relevant: we cannot refer to a process before its definition. This is a restriction inherited from Z that can be lifted by tools, as long as they guarantee that there is an appropriate order to present the specification. The Z specification corresponding to a whole program includes those corresponding to the individual processes in the order that they appear.

A process definition enriches the environment by associating a process name to the Z specification corresponding to the declared process. Moreover, if the

process specification involves indexing or channel renaming, new channel names are implicitly declared. The semantic function $[\![\_]\!]^{\mathcal{PD}}$ gives the meaning of a process definition as a process environment that records just the single process it declares, and a channel environment recording the new channels it introduces.

$$[\![\_]\!]^{\mathcal{PD}} : \mathsf{ProcDefinition} \pfun \mathit{ChanEnv} \pfun \mathit{ProcEnv} \pfun (\mathit{ChanEnv} \times \mathit{ProcEnv})$$

$$[\![\mathbf{process}\ N \mathrel{\widehat=} P]\!]^{\mathcal{PD}} \gamma\ \rho = \mathbf{let}\ Ps == [\![P]\!]^{\mathcal{P}} \gamma\ \rho \bullet (Ps.1, \langle (N, Ps.2) \rangle)$$

The semantics Ps of P is taken in the current channel and process environments; it is a pair containing a channel environment and a Z specification. The semantics of the process definition includes the channel environment and a process environment that associates N to the Z specification. The function $[\![\_]\!]^{\mathcal{P}}$ gives the meaning of processes and is defined later on.

4.3 Programs

A program's meaning is given by $[\![\_]\!]^{\mathcal{PROG}} : \mathsf{Program} \pfun \mathsf{ZSpecification}$. The model of a *Circus* program is its Z paragraphs and the model of its processes. More precisely, the specification starts with the following four paragraphs. First, we define a boolean type *Bool* as a free type with two constants: False and True; we use variables of this type as predicates, for simplicity. The second paragraph declares the given sets *Sync* and *Event*; the former is the type of the synchronisation events, and the latter includes the possible communications of the program.

The third paragraph specifies the components that comprise the state of a process, in addition to the user state components. These are the variables of the unifying theory model and the additional *trace* variable, which records the events that occurred since the last observation. Our processes do not have an alphabet as in [16]; instead we consider the alphabetised parallel operator of [23].

$$\mathit{ProcessState} \mathrel{\widehat=} [\, \mathit{trace}, tr : \mathrm{seq}\, \mathit{Event};\ \mathit{ref} : \mathbb{P}\, \mathit{Event};\ \mathit{okay}, \mathit{wait} : \mathit{Bool} \,]$$

Changes to the process state are constrained as specified in the fourth paragraph: valid process observations increase the trace.

$$\mathit{ProcessStateObs} \mathrel{\widehat=} [\, \Delta \mathit{ProcessState} \mid tr \ \mathsf{prefix}\ tr' \land \mathit{trace}' = tr' - tr \,]$$

The remaining paragraphs are determined by $[\![\mathit{prog}]\!]^{\mathcal{CPARL}}\ \emptyset\ \emptyset$. This is the semantics of the list of paragraphs that compose the program itself.

4.4 Paragraphs

A *Circus* paragraph can contribute to the semantics of the whole program by extending the Z specification, and the channel and process environments.

$$[\![\_]\!]^{\mathcal{CPAR}} : \mathsf{CircusParagraph} \pfun \mathit{ChanEnv} \pfun \mathit{ProcEnv} \pfun (\mathsf{ZSpecification} \times \mathit{ChanEnv} \times \mathit{ProcEnv})$$

For a Z paragraph Zp, the definition of $[\![\_]\!]^{\mathcal{CPAR}}$ adds Zp to the Z specification

as it is, and does not affect the channel or the process environment.

$$[\![Zp]\!]^{\mathcal{CPAR}}\ \gamma\ \rho = (tc\ Zp, \gamma, \rho)$$

Slight changes to the Z paragraph may be needed because of schemas with untyped components, which are assumed to be synchronisation event declarations. The function tc, when applied to a schema that declares such components, yields the schema obtained by declaring the types of these components to be *Sync*.

A channel definition cd gives rise to a few paragraphs in the Z specification and enriches the channel environment.

$$[\![cd]\!]^{\mathcal{CPAR}}\ \gamma\ \rho = \mathbf{let}\ \gamma' == [\![cd]\!]^{\mathcal{CD}} \bullet (events\ \gamma', \gamma \oplus \gamma', \rho)$$

The environment γ' records the channels declared in cd. For each channel c recorded to have type T different from *Sync* in γ', we have that $events\ \gamma'$ yields an axiomatic description that declares c to be an injective function from T to *Event*. If T is *Sync*, then c is declared to be itself an event. These constants and injective functions are *Event* constructors.

A process definition pd determines a Z specification of its model, and enriches the process environment and possibly the channel environment as well.

$$[\![pd]\!]^{\mathcal{CPAR}}\ \gamma\ \rho = \mathbf{let}\ pds == [\![pd]\!]^{\mathcal{PD}}\ \gamma\ \rho \bullet ((pds.2\ 1).2, \gamma \oplus pds.1, \rho \oplus pds.2)$$

The semantics of pd is a pair pds containing the channel environment that records the channels pd (implicitly) declares, and a process environment that records the process defined by pd. The Z specification corresponding to the process is the second element of the pair in the first and unique position of the process environment, which is itself the second element of pds.

The function $[\![\_]\!]^{\mathcal{CPARL}}$ maps lists of paragraphs to Z specifications whose paragraphs are obtained from the *Circus* paragraphs as specified by $[\![\_]\!]^{\mathcal{CPAR}}$.

We eliminate repeated names used across different process definitions by prefixing each name with the name of the process in which it is declared. Also, in the model of a process, where several schemas are defined, their names should be fresh. Below, we leave this assumption implicit.

4.5 Processes

The semantic function $[\![\_]\!]^{\mathcal{P}}$ gives the meaning of a process declaration as a pair containing a channel environment and a Z specification.

$$[\![\_]\!]^{\mathcal{P}} : \mathsf{Process} \pfun ChanEnv \pfun ProcEnv \pfun (ChanEnv \times \mathsf{ZSpecification})$$

For a process name N, the semantics is the current channel environment and $modelOf\ N\ in\ \rho$, which is the Z specification associated to N in ρ.

For an explicit process specification **begin** $ppl \bullet A$ **end**, the semantics is a Z specification containing the following paragraphs: $ProcObs$, a schema describing the observations that may be made of the process; the Z paragraphs as they are,

except for those that are schemas that define operations as these are actions; and, for each action, a schema constraining the process observations. In order to define *ProcObs* we specify a schema *State* that defines the process state. It includes the components of the schema *ProcessState* previously defined and those of the state defined in *ppl*, which we assume to be named *UserState*.

$$State \mathrel{\widehat=} UserState \land ProcessState$$

The state of a *Circus* process in our model includes the components of the state in its specification, which we refer to as user state, and the observation variables of the unifying theory. A process observation corresponds to a state change.

$$ProcObs \mathrel{\widehat=} \Delta UserState \land ProcStateObs$$

As we explain later on, the state can be extended by the declaration of extra variables. Therefore, we actually consider a family of schemas $ProcObs(USt)$; for a schema reference *USt* that defines the user state, $ProcObs(USt)$ is the schema defined as *ProcObs* above, except that it includes ΔUSt, instead of $\Delta UserState$.

4.6 Actions

To each action $N \mathrel{\widehat=} A$ corresponds a schema named N. It is determined by the function $[\![\_]\!]^{\mathcal{A}}$ which takes as arguments the current channel environment and the name of the schema that defines the user state.

$$[\![\_]\!]^{\mathcal{A}} : \mathsf{Action} \pfun ChanEnv \pfun \mathsf{N} \pfun \mathsf{Schema\text{-}Exp}$$

The main action is nameless; the schema corresponding to it is given a fresh name. This schema is also determined by the function $[\![\_]\!]^{\mathcal{A}}$.

We distinguish three cases in the definition of the behaviour of an action: the normal case, the case in which the previous operation diverged, and the case in which the previous operation has not terminated.

$$[\![A]\!]^{\mathcal{A}}\gamma\ USt = [\![A]\!]^{\mathcal{A}_{\mathcal{N}}}\gamma\ USt \lor Diverge(USt) \lor Wait(USt)$$

The function $[\![\_]\!]^{\mathcal{A}_{\mathcal{N}}}$ characterises the normal behaviour of an action.

$$[\![\_]\!]^{\mathcal{A}_{\mathcal{N}}} : \mathsf{Action} \pfun ChanEnv \pfun \mathsf{N} \pfun \mathsf{Schema\text{-}Exp}$$

It is defined by induction below. The family of schemas $Diverge(USt)$ characterise the behaviour of an action in the presence of divergence.

$$Diverge(USt) \mathrel{\widehat=} [\, ProcObs \mid \neg\ okay \,]$$

Divergence is characterised by the fact that *okay* is false; the only guarantee is that *trace* can only be extended: a restriction enforced by *ProcessStateObs*. For $Wait(USt)$ we have the following definition.

$$Wait(USt) \mathrel{\widehat=} [\, \Xi ProcObs(USt) \mid okay \land wait \,]$$

The waiting state occurs when both *okay* and *wait* are true: there is no diver-

gence, but the previous action has not terminated; the state does not change. The normal case of the actions behaviour is characterised by the schema below.

$$Normal(USt) \mathrel{\widehat=} [\, ProcObs(USt) \mid okay \land \neg\, wait \,]$$

In this case, there is no divergence and the previous action has terminated.

Schema Expressions For a schema expression $SExp$, we have the following.

$$[\![SExp]\!]^{\mathcal{A}_{\mathcal{N}}}\gamma\ USt = SExp \land OpNormal \lor OpDiverge$$

$OpNormal$ describes when $SExp$ is activated in a state that satisfies its precondition; the $trace$ is not modified and $okay$ and $wait$ do not change: the operation terminates successfully.

$$OpNormal \mathrel{\widehat=} [\, Normal(USt) \mid trace' = \langle\rangle \land okay' \land \neg\, wait' \,]$$

If the precondition of $SExp$ is not satisfied, the action diverges.

$$OpDiverge \mathrel{\widehat=} [\, Normal(USt);\ SExp \lor \neg\, SExp \mid \neg \text{ pre } SExp \land \neg\, okay' \,]$$

We include $SExp \lor \neg\, SExp$ to put input and output variables in scope. In the sequel, we define the normal behaviour of the actions.

CSP Expressions The definition of the normal behaviour of $Skip$ is as follows.

$$[\![Skip]\!]^{\mathcal{A}_{\mathcal{N}}}\gamma\ USt = [Normal(USt) \land \Xi USt \mid trace' = \langle\rangle \land okay' \land \neg\, wait']$$

The user state is not changed, the trace is also not changed, and it terminates. For $Stop$, we have a similar definition, but deadlock is characterised by the fact that $wait'$ is true. For $Chaos$, we require $\neg\, okay$, which characterises divergence.

Sequencing is defined in terms of a function $sequence$ on $ProcObs(USt)$.

$$\frac{[\![A;\ B]\!]^{\mathcal{A}_{\mathcal{N}}}\gamma\ USt \qquad Normal(USt)}{\theta ProcObs(USt) = \theta([\![A]\!]^{\mathcal{A}}\gamma\ USt)\ sequence\ \theta([\![B]\!]^{\mathcal{A}}\gamma\ USt)}$$

The function $sequence$ takes two process observations and returns the process observation that characterises their sequential composition.

$$\_sequence\_ : ProcObs(USt) \times ProcObs(USt) \nrightarrow ProcObs(USt)$$

$$\forall a, b, c : ProcObs(USt) \mid c = a\ sequence\ b \Leftrightarrow$$
$$\quad before\ c = before\ a \land after\ a = before\ b \land after\ b = after\ c$$

The sequential composition is well-defined only if the final state of the first process is equal to the initial state of the second. The functions $before$ and $after$ project the initial and the final state out of a process observation. If a diverges,

then we have that $\neg\ a.okay'$ and consequently $\neg\ b.okay$. So, if b satisfies the healthiness condition **CSP1**, then the composite $a\ sequence\ b$ diverges. Similarly, if a is waiting, then we have $a.wait'$ and so $b.wait$. So, if b satisfies the healthiness condition **R3**, then $a\ sequence\ b$ waits.

We specify a communication as a process observation $Comm(USt)$. Its occurrence is an event characterised by a channel name and a communicated value; in our model, this is an element of $Event$. The process observation $Comm(USt)$ takes as input the set of events $accEvents?$ that can take place, and outputs the event $e!$ that actually takes place. We can make observations at two stages of the communication. The first is when the communication has not yet taken place.

$$
\begin{array}{|l}
\hline
CommWaiting(USt) \\
Normal(USt) \\
accEvents? : \mathbb{P}\ Event \\
\Xi USt \\
\hline
trace' = \langle\rangle \land accEvents? \cap ref' = \emptyset \land okay' \land wait' \\
\hline
\end{array}
$$

The trace is not extended, the acceptable events are not refused, and the user state is not changed. We can also observe a communication after it has occurred.

$$
\begin{array}{|l}
\hline
CommDone(USt) \\
Normal(USt) \\
accEvents? : \mathbb{P}\ Event \\
e! : Event \\
\Xi USt \\
\hline
e! \in accEvents? \land trace' = \langle e!\rangle \land okay' \land \neg\ wait' \\
\hline
\end{array}
$$

The trace is extended with a possible event; the user state is not changed.

$$
\begin{aligned}
Comm(USt) \mathrel{\widehat=}\ & CommWaiting(USt) \lor CommDone(USt) \lor \\
& Diverge(USt) \lor Wait(USt)
\end{aligned}
$$

A communication is actually a more primitive concept than a prefixing, which is the sequential composition of a communication and an action. For a prefixing $c!v \to A$, we have the following semantics.

$$
\begin{array}{|l}
\hline
[\![c!v \to A]\!]^{\mathcal{A}_{\mathcal{N}}}\gamma\ USt \\
Normal(USt) \\
\hline
\exists\, oc : Comm(USt) \mid oc.accEvents? = \{c(v)\} \bullet \\
\quad \theta ProcObs(USt) = (procObsC\ \ oc)\ sequence\ \theta([\![A]\!]^{\mathcal{A}}\gamma\ USt) \\
\hline
\end{array}
$$

The only possible communication is $c(v)$. The function $procObsC$ projects out the components of a communication that form a process observation. The semantics of $c.v \to A$ and $c \to A$ can be defined in a similar way. For $c?x \to A$ the definition is different as $c?x$ introduces the variable x in scope for A.

The action $p \mathbin{\&} A$ is enabled only if the p condition holds, in which case, the semantics is that of A; otherwise it behaves as $Stop$.

The behaviour of an external choice $A \Box B$ can be observed in two points. Before the choice is made, the trace is empty, the process is waiting, and the user state has not been changed. The refusal set is characterised by the restrictions of both A and B: an event is refused only if it is refused by both A and B.

$$
\begin{array}{|l}
\hline
ExtChoiceWait(USt) \\
Normal(USt) \\
\Xi USt \\
\hline
trace' = \langle\rangle \land okay' \land wait' \land [\![A]\!]^{\mathcal{A}}\gamma\ USt \land [\![B]\!]^{\mathcal{A}}\gamma\ USt \\
\hline
\end{array}
$$

If a choice has been made, then either the trace is not empty or the process has diverged or it is not waiting. The behaviour is either that of A or that of B.

$$
\begin{array}{|l}
\hline
ExtChoiceNotWait(USt) \\
Normal(USt) \\
\hline
(trace' \neq \langle\rangle \lor \neg\ okay' \lor \neg\ wait') \land ([\![A]\!]^{\mathcal{A}}\gamma\ USt \lor [\![B]\!]^{\mathcal{A}}\gamma\ USt) \\
\hline
\end{array}
$$

$$
[\![A \Box B]\!]^{\mathcal{A}_{\mathcal{N}}}\gamma\ USt \mathrel{\widehat=} ExtChoiceWait(USt) \lor ExtChoiceNotWait(USt)
$$

The internal choice is given simply by disjunction.

We define the semantics of parallelism as shown below.

$$
\begin{array}{|l}
\hline
[\![A \,[\![\, C \,]\!]\, B]\!]^{\mathcal{A}_{\mathcal{N}}}\gamma\ USt \\
Normal(USt) \\
\hline
\exists\, tracea, traceb : \mathrm{seq}\ Event;\ refa, refb : \mathbb{P}\ Event; \\
\quad okaya, okayb, waita, waitb : Bool \bullet \\
\qquad ([\![A]\!]^{\mathcal{A}}\gamma\ USt)[tracea, refa, okaya, waita / trace', ref', okay', wait'] \land \\
\qquad ([\![B]\!]^{\mathcal{A}}\gamma\ USt)[traceb, refb, okayb, waitb / trace', ref', okay', wait'] \land \\
\qquad trace' \in tracea \mid\mid traceb\ sync\ ([\![C]\!]\gamma) \land \\
\qquad ref' = (refa \cup refb) \cap [\![C]\!]\gamma \cup (refa \cap refb) \setminus [\![C]\!]\gamma \land \\
\qquad okay' = okaya \land okayb \\
\qquad wait' = waita \lor waitb \\
\hline
\end{array}
$$

We include the schemas that define the semantics of A and B. These actions start in the same state, so their restrictions on the initial state are conjoined. We rename the individual final state components to use in the definition of the parallel composition.

An event can be refused by the parallel composition if either A or B can refuse it and they have to synchronise on it, or rather, it is in the synchronisation set C. This set is denoted above by $[\![C]\!]\gamma$ and includes all events that represent

communications over a channel in C according to its type definition in γ. If an event is not in this synchronisation set, then it can only be refused if both A and B can refuse it. The parallel composition diverges if either A or B does, and it terminates when both A and B do. The trace is the combination of the traces of A and B where events in the channel set determined by C are synchronised. The definition of the $\_ \parallel \_sync\_$ operator can be found in [23]. The definition of interleaving is similar to that of parallelism.

The semantics of hiding is as follows.

$$
\begin{array}{|l}
[\![A \setminus C]\!]^{\mathcal{A}_{\mathcal{N}}} \gamma\ USt \\
Normal(USt) \\
\hline
\exists\, tracep : \mathrm{seq}\ Event;\ refp : \mathbb{P}\ Event \bullet \\
\quad [\![A]\!]^{\mathcal{A}} \gamma\ USt[tracep, refp / trace', ref'] \land \\
\quad trace' = tracep \upharpoonright (Event \setminus [\![C]\!]\gamma) \\
\quad refp = ref' \cup ([\![C]\!]\gamma) \\
\hline
\end{array}
$$

The traces and refusals of $A \setminus C$ are determined in terms of those of A: we have to eliminate all events that represented communications through the channels in C from the trace and from the refusals. In order to deal adequately with the semantics of hiding, we need to have infinite sequences in our model, as suggested in [16]. We leave the complications of this as a future work for now.

The definition of the semantics of a recursive action $\mu X \bullet A$ is standard. We must observe, however, that the action A may use X as an action and, as such, is regarded as a function from actions to actions. For clarity, we refer to this action as $F(X)$.

$$
\begin{array}{|l}
[\![\mu X \bullet F(X)]\!]^{\mathcal{A}_{\mathcal{N}}} \gamma\ USt \\
Normal(USt) \\
\hline
\theta ProcObs(USt) \in \bigcup \{ a : SProcess(USt) \mid a \subseteq [\![F(\_)]\!]^{\mathcal{F}} \gamma\ USt(\!|\ a\ |\!) \} \\
\hline
\end{array}
$$

In this context, a process is represented as a set of process observations that satisfies the healthiness conditions of CSP. The definition of the set $SProcess(USt)$ of such processes can be found in [31].

We use the semantic function $[\![\_]\!]^{\mathcal{F}}$, which gives the semantics of a function on actions as a function on process observations.

$$
\begin{array}{l}
[\![\_]\!]^{\mathcal{F}} : (Action \to Action) \nrightarrow ChanEnv \nrightarrow SchemaName \\
\qquad \nrightarrow (ProcObs(USt) \to ProcObs(USt))
\end{array}
$$

The function $[\![F(\_)]\!]^{\mathcal{F}} \gamma\ USt$ can be defined in terms of $[\![\_]\!]^{\mathcal{A}}$ as follows.

$$
\begin{array}{|l}
[\![F(\_)]\!]^{\mathcal{F}} \gamma\ USt : ProcObs(USt) \to ProcObs(USt) \\
\hline
\forall\, po : ProcObs(USt) \bullet \mathbf{let}\ \theta([\![X]\!]^{\mathcal{A}} \gamma\ USt) == po\ \bullet \\
\quad \exists\, [\![F(X)]\!]^{\mathcal{A}} \gamma\ USt \bullet f\ po = \theta ProcObs(USt) \\
\end{array}
$$

According to the definition of $[\![\_]\!]^{\mathcal{A}}$, to determine $[\![F(X)]\!]^{\mathcal{A}} \gamma\ USt$, we need to

know $\theta([\![X]\!]^{\mathcal{A}}\gamma\ USt)$. This is specified above as the argument of $[\![F(\_)]\!]^{\mathcal{F}}\gamma\ USt$.

For example, consider $\mu X \bullet out!2 \rightarrow X$, in an environment γ in which out is a channel of type integer. The function $[\![out!2 \rightarrow \_]\!]^{\mathcal{F}}\ \gamma\ USt$ is as follows.

$$[\![out!2 \rightarrow \_]\!]^{\mathcal{F}}\ \gamma\ USt : ProcObs(USt) \rightarrow ProcObs(USt)$$

$$\forall\, po : ProcObs(USt) \bullet \exists\, Output \bullet f\ po = \theta ProcObs(USt)$$

The schema $Output$ is defined according to the semantics of $out!2 \rightarrow X$.

Output

$$Normal(USt)$$

$$\exists\, oc : Comm(USt) \mid oc.accEvents? = \{out(2)\} \bullet$$
$$\theta ProcObs(USt) = (procObsC\ oc)\ sequence\ po$$

The semantics of X, which is required in the specification of $Output$ is taken to be po. The calculation of the semantics of other recursive actions, which apply to X operators whose semantics are given in terms of $[\![X]\!]^{\mathcal{A}}\gamma\ USt$ instead of $\theta([\![X]\!]^{\mathcal{A}}\gamma\ USt)$, requires more effort. We need to manipulate the definition of $[\![F(X)]\!]^{\mathcal{A}}\gamma\ USt$ to express it in terms of $\theta([\![X]\!]^{\mathcal{A}}\gamma\ USt)$.

A parametrised action $D \bullet A$ declares extra variables that can be used in A. We regard them as immutable state components whose values are fixed by initialisation. The semantics of $D \bullet A$ is that of A taken in the extended state. For an instantiation $A(e)$, we define the initial value of the extra state components in the semantics of A to be e and hide them.

Commands The behaviour of a specification statement $x : [pre, post]$ is highly dependent on whether its precondition holds or not. If it does, then the postcondition must be established and the operation must terminate successfully, but the trace is not affected and only the variables x in the frame can be changed.

$[\![x : [pre, post]]\!]^{\mathcal{A}_{\mathcal{N}}}\gamma\ USt$

$$ProcObs(USt)$$

$$Normal(USt) \wedge pre \Rightarrow$$
$$post \wedge trace' = \langle\rangle \wedge okay' \wedge \neg\ wait' \wedge \alpha USt' \setminus x' = \alpha USt \setminus x$$

We decorate the list of variables x to obtain the list of corresponding dashed variables. In an abuse of notation we use x and x' as sets to define the set of user state components that cannot be changed: all (αUSt) but those in x. The conjunction of equalities $\alpha USt' \setminus x' = \alpha USt \setminus x$ enforces this restriction.

If the precondition holds, but the postcondition cannot be established (under the given circumstances), then we have an infeasible (miraculous) operation. In such a circumstance, the predicate of the above schema is *false*. Therefore, the semantics of $x : [pre, post]$ is a partial relation.

The semantics of the assignment $x := e$ is rather simple. This action does not change the trace and, since we assume the expressions are always well-defined, it does not diverge and terminates. Of course, it sets the final value of x to e. The semantics of the conditional is also standard.

The semantics of variable declaration is given by existential quantification.

$$[\![\mathbf{var}\ x : T \bullet A]\!]^{\mathcal{A}_{\mathcal{N}}}\gamma\ USt = \exists\, x, x' : T \bullet [\![A]\!]^{\mathcal{A}}\gamma\ xUSt(USt)$$

The semantics of the action in the scope of the variable block is taken in the extended user state $xUSt(USt)$ that includes x. The semantics of constant declaration is similar, but given by universal quantification.

4.7 Process expressions

For the binary operators op, the semantics of $P\ \mathsf{op}\ Q$ can be given in terms of an explicit process specification.

$P\ \mathsf{op}\ Q = \mathbf{begin}$

$\quad State \mathrel{\widehat=} P.State \land Q.State$

$\quad P.PPar \uparrow Q.State$

$\quad Q.PPar \uparrow P.State$

$\quad \bullet\ P.Act\ \mathsf{op}\ Q.Act$

$\mathbf{end}$

The schemas $P.State$ and $Q.State$ are the schemas that define the user state of P and Q. The state of $P\ \mathsf{op}\ Q$ conjoins the states of P and Q; we assume that name clashes are avoided through renaming.

We also refer to $P.PPar$ and $Q.PPar$, which are the process paragraphs that compose the definitions of P and Q, except $P.State$ and $Q.State$ and the main actions. These are all included in $P\ \mathsf{op}\ Q$. We must notice, however, that the schemas in $P.PPar$ that specify an operation on $P.State$ are not by themselves actions in $P\ \mathsf{op}\ Q$; we need to lift them to act on the extended state defined by $State$. This is the aim of the operator $\uparrow$, which simply conjoins each such schema with $\Xi Q.State$: actions of P are not supposed to affect the state components that are inherited from Q. Similar comments apply to the actions in $Q.PPar$.

The specification of the action that defines the behaviour of $P\ \mathsf{op}\ Q$ combines those that specify the behaviour of P and Q, $P.Act$ and $Q.Act$, using op. We observe that $P.Act$ ($Q.Act$) operates on the part of the user state that is due to P (Q) and cannot change the components that are originally part of the Q (P) state. It does not refer to them directly or indirectly, except through schema actions which have been conjoined with $\Xi Q.State$ ($\Xi P.State$).

The semantics of a hiding expression $P \setminus C$ is even simpler. The process paragraphs of P are included as they are; only the main action is modified to include the hiding.

The indexed process $i : T \odot P$ implicitly declares channels c_i, for each channel c used in P. Its semantics, therefore, affects the channel environment.

$$[\![i : T \odot P]\!]^{\mathcal{P}}\ \gamma\ \rho = \mathbf{let}\ \gamma' == \{c : used\ P \bullet c\_i \mapsto T \times \gamma\ c\} \bullet [\![i : T \bullet (P[c : (used\ P) \bullet c\_i])]\!]^{\mathcal{P}}\ (\gamma \oplus \gamma')\ \rho$$

The environment γ' records the channels implicitly declared. The set $used\ P$ includes the channels used in P. For each such channel c, γ' records a channel c_i which communicate pairs of values: the index and whatever value was communicated originally. The semantics of $i : T \odot P$ is that of a parametrised process taken in the extended channel environment that includes γ'. The parameter is the index, and the process $P[c : (used\ P) \bullet c\_i])$ is that obtained from P by changing all the references to a used channel c by a reference to the channel c_i. Communications through these channels are also changed so that the index i is also communicated. The generalisation of this and the previous definition for an arbitrary indexed process $D \odot P$ whose indexes are declared by D is lengthy, but straightforward. The semantics of instantiation is given by substitution.

The semantics of renaming is given mainly by substitution on the process definition. In a parametrisation, the parameters are regarded as loose constants. The channel environment in the pair defined by $[\![D \bullet P]\!]^{\mathcal{P}}\ \gamma\ \rho$ is that in $[\![P]\!]^{\mathcal{P}}\ \gamma\ \rho$. The Z specification is also that in $[\![P]\!]^{\mathcal{P}}\ \gamma\ \rho$, preceded by an axiomatic description that introduces D. The instantiation of parametrised processes has the same definition of instantiation of indexed processes: substitution.

The semantics of a generic process $[X]P$ is given by a rewriting of the Z specification denoted by P: each of its paragraphs is turned into a similar generic paragraph with parameter X; the channel environment is the same. Instantiation $P[E]$ of a generic process P is defined by the instantiation of all definitions in the Z specification corresponding to P.

5 Conclusions

This paper presents a unified language for specifying, designing, and programming concurrent systems; it combines Z and CSP in a way suitable for refinement. Fischer reports [11] a survey of related work in combining Z and process algebras: Z and CCS in [14, 28]; Z with CSP in [10, 24]; and CSP with Object-Z [5] in [10]. The major objective of *Circus* is to provide a theory of refinement and an associated calculus. Nothing in the style of a calculus has been proposed.

Combinations of Object-Z and CSP have been given a failures-divergences model for Object-Z classes [25, 12]; data refinement has been briefly explored for such a combination, but no refinement laws have been proposed. Abstract data types have been given behavioural semantics in the failures model [34]. In [6], refinement rules have been proposed to support the development of Java programs, but no semantic model has been provided.

An early paper [17] presents a state-based semantics for a subset of occam in a style similar to ours, using predicates over failures and a stability/termination.

The subset includes *Stop*, *Skip*, assignment, communication, conditional, loop, sequence, alternation, and variable declarations. The semantics of parallelism is left as an exercise.

Our semantic definitions are based on those in [16]. We believe, however, that we have provided an accessible presentation of the theory of imperative communicating programs. In doing so, the use of Z as an elegant notation to define relations was very appropriate. It also means that we can make use of tools like Z/Eves [18] to analyse and validate our definitions. Currently, we are encoding our semantics in Z/Eves. It is our plan to prove that all the healthiness conditions hold for our semantic definitions.

We are linking tools for Z and CSP through the unifying theory: we are using FDR [13] and Z/Eves for analysing different aspects of *Circus* specifications. We are also building a tool that calculates the Z specification corresponding to a *Circus* program, producing a specification that is suitable for analysis using Z/Eves.

A new language needs demonstrations of its usefulness; an implementation in the form of tools for analysis and development; and additional theory to underpin and extend. We are considering case studies and examples including the steam boiler control system [1, 3, 33, 30], an IP-packet firewall [35], a smart-card system for electronic finance [27], and a railway signalling application [29].

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. In order to solve the difficulty with the semantics of the hiding operator, we also plan to extend our model to allow infinite traces [23].

Our main goal, however, is the proposal and proof of refinement laws for *Circus*. We want data refinement rules, and rules that allows the stepwise refinement to code in a calculational way.

Acknowledgments

We would like to thank Augusto Sampaio for his many suggestions on our work. Ana Cavalcanti is partly 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.

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 and A. C. A. Sampaio. From CSP-OZ to Java with Processes (Extended Version). Technical report, Centro de Informática/UFPE, 2001. Available at `http://www.cin.ufpe.br/~lmf`.
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. *A Discipline of Programming*. Prentice-Hall, 1976.
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. C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Fachbereich Informatik Universitat Oldenburg, 2000.
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. C. A. R. Hoare and A. W. Roscoe. Programs as executable predicates. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984 (FGCS'84)*, pages 220–228, Tokyo, Japan, November 1984. Institute for New Generation Computer Technology.
18. I. Meisels. *Software Manual for Windows Z/EVES Version 2.1*. ORA Canada, 2000. TR-97-5505-04g.
19. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
20. C. C. Morgan. Of wp and csp. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer, 1990.
21. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
22. J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287 – 306, 1987.
23. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

24. 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.
25. G. Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems Specified in Object-Z and CSP. In C. B. Jones J. Fitzgerald and P. Lucas, editors, *Proceedings of FME'97*, volume 1313 of *Lecture Notes in Computer Science*, pages 62 – 81. Springer-Verlag, 1997.
26. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.
27. 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.
28. 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.
29. 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.
30. 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.
31. J. C. P. Woodcock and A. L. C. Cavalcanti. *Circus*: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK, July 2001.
32. J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In Andrew Butterfield and Claus Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*. Computer Science Department, Trinity College Dublin, July 2001.
33. 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.
34. J. C. P. Woodcock, J. Davies, and C. Bolton. Abstract Data Types and Processes. In J. Davies, A. W. Roscoe, and J. C. P. Woodcock, editors, *Millenial Perspectives in Computer Science, Proceedings of the 1999 Oxford-Microsoft Symposium in honour of Sir Tony Hoare*, pages 391 – 405. Palgrave, 2000.
35. 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.
36. J. C. P. Woodcock and C. C. Morgan. Refinement of state-based concurrent systems. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z—Formal Methods in Software Development*, number 428 in LNCS, pages 340–351. Springer, 1990.