Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 187 (2007) 107–123
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Denotational Semantics for *Circus*

Marcel Oliveira [1,2]

*Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
Natal, Brazil*

Ana Cavalcanti [3]   Jim Woodcock [4]

*Department of Computer Science
University of York
York, England*

Abstract

Circus specifications define both data and behavioural aspects of systems using a combination of Z and CSP. Previously, a denotational semantics has been given to *Circus*; however, as a shallow embedding of *Circus* in Z, it was not possible to use it to prove properties like the refinement laws that justify the distinguishing development technique associated with *Circus*. This work presents a final reference for the *Circus* denotational semantics based on Hoare and He's *Unifying Theories of Programming* (UTP). Finally, it discusses the library of theorems on the UTP that was created and used in the proofs of the refinement laws.

Keywords: Concurrency, refinement calculus, UTP, theorem proving.

1 Introduction

Throughout the past decades two schools have been developing formal techniques for precise and correct software development. Model based languages like Z [18] focus on data aspects of the systems; constructs to model behavioural aspects are not explicitly provided by any of these languages. On the other hand, CSP [9,15], among other process algebras, focuses on the behavioural aspects of the systems; however, it does not support a concise and elegant way to describe complex data aspects of the systems.

[1] The work of Marcel Oliveira is supported by CNPq: grant 551210/2005-2.
[2] E-mail: marcel@dimap.ufrn.br
[3] E-mail: alcc@cs.york.ac.uk
[4] E-mail: jim@cs.york.ac.uk

doi:10.1016/j.entcs.2006.08.047

Many formalisms combine constructs to specify data and behavioural aspects of the systems. For example, combinations of Z with CSP [7] and Object-Z with CSP [6], and new notations like RAISE [8], are some attempts to combine both schools of formalisms. As far as we know, however, none of them has a related refinement calculus. This lack of support for refinement of state-rich reactive systems in a calculational style, like that presented in [11], has motivated the creation of *Circus* [17]. In this concurrent language, systems are characterised as processes, which group constructs that describe data and control; the Z notation [16] is used to define most of the data aspects, and CSP is used to define behaviour.

In [17], Cavalcanti and Woodcock present a semantic model for *Circus* based on the *Unifying Theories of Programming* (UTP) [10], a relational framework that unifies programming science across many different computational paradigms. Although usable for reasoning about systems specified in *Circus*, the semantics in [17] is not appropriate to prove properties of *Circus* itself. This happens because it is a shallow embedding, in which the *Circus* constructs are mapped to their semantic as a Z specification, with yet another language being used as a meta-language.

For this reason, we redefined the *Circus* semantics and mechanised it using ProofPower-Z [14], a theorem prover for Z. Based on the new definitions, we proved over ninety percent of the one-hundred and forty-six proposed refinement laws. These proofs range over all the structure of the language and include all the data simulation laws; their proofs can be found in [12].

In Section 2 we present *Circus*. Section 3 introduces the UTP and reactive designs. In Section 4 we have the main contribution of this paper: we present a definitive reference for the *Circus* denotational semantics based on the UTP. Section 5 discusses the structure of the library of lemmas and theorems created during this work, and illustrates the usefulness of the library by presenting the proof of one of our refinement laws. Finally, we draw some conclusions in Section 6.

2 *Circus*

Circus is based on imperative CSP, and adds specification facilities in the Z style; this enables both state and communication aspects to be captured in the same specification. *Circus* programs are formed by a sequence of paragraphs. In Figure 1, we present the BNF of the *Circus* syntax. Here, $\mathsf{CircusPar}^*$ denotes a possibly empty list of elements of the syntactic category $\mathsf{CircusPar}$ of *Circus* paragraphs; similarly for PPar^* (process paragraphs). We use N^+ to denote a non-empty list of elements of the Z identifiers N. The syntactic categories Par, $\mathsf{SchemaExp}$, Exp, Pred, and Decl include the Z paragraphs, schema expressions, expressions, predicates and declarations defined in [16].

The declarations of all the channels give their names and the types of the values that they can communicate; however, if a channel does not communicate any value its declaration contains only its name. Generic channel declarations introduce families of channels. For instance, **channel** $[T]\ c : T$ declares a family of channels c. For every actual type S, we have a channel $c[S]$ that communicates values of type

$$
\begin{array}{lll}
\mathsf{Program} & ::= & \mathsf{CircusPar}^* \\
\mathsf{CircusPar} & ::= & \mathsf{Par} \mid \mathbf{channel}\ \mathsf{CDecl} \mid \mathbf{chanset}\ \mathsf{N} == \mathsf{CSExp} \mid \mathsf{ProcDecl} \\
\mathsf{CDecl} & ::= & \mathsf{SimpleCDecl} \mid \mathsf{SimpleCDecl};\ \mathsf{CDecl} \\
\mathsf{SimpleCDecl} & ::= & \mathsf{N}^+ \mid \mathsf{N}^+ : \mathsf{Exp} \mid [\mathsf{N}^+]\mathsf{N}^+ : \mathsf{Exp} \mid \mathsf{SchemaExp} \\
\mathsf{ProcDecl} & ::= & \mathbf{process}\ \mathsf{N} \mathrel{\widehat{=}} \mathsf{ProcDef} \mid \mathbf{process}\ \mathsf{N}[\mathsf{N}^+] \mathrel{\widehat{=}} \mathsf{ProcDef} \\
\mathsf{ProcDef} & ::= & \mathsf{Decl} \bullet \mathsf{ProcDef} \mid \mathsf{Decl} \odot \mathsf{ProcDef} \mid \mathsf{Proc} \\
\mathsf{Proc} & ::= & \mathbf{begin}\ \mathsf{PPar}^*\ \mathbf{state}\ \mathsf{SchemaExp}\ \mathsf{PPar}^* \bullet \mathsf{Action}\ \mathbf{end} \\
 & \mid & \mathsf{Proc};\ \mathsf{Proc} \mid \mathsf{Proc} \Box \mathsf{Proc} \mid \mathsf{Proc} \sqcap \mathsf{Proc} \mid \mathsf{Proc} \llbracket \mathsf{CSExp} \rrbracket \mathsf{Proc} \\
 & \mid & \mathsf{Proc} \interleave \mathsf{Proc} \mid \mathsf{Proc} \setminus \mathsf{CSExp} \mid (\mathsf{Decl} \bullet \mathsf{ProcDef})(\mathsf{Exp}^+) \mid \mathsf{N}(\mathsf{Exp}^+) \mid \mathsf{N} \\
 & \mid & (\mathsf{Decl} \odot \mathsf{ProcDef})\lfloor \mathsf{Exp}^+ \rfloor \mid \mathsf{N}\lfloor \mathsf{Exp}^+ \rfloor \mid \mathsf{Proc}[\mathsf{N}^+ := \mathsf{N}^+] \mid \mathsf{N}[\mathsf{Exp}^+] \\
 & \mid & ;\ \mathsf{Decl} \bullet \mathsf{Proc} \mid\ \Box\ \mathsf{Decl} \bullet \mathsf{Proc} \mid\ \sqcap\ \mathsf{Decl} \bullet \mathsf{Proc} \\
 & \mid & \llbracket \mathsf{CSExp} \rrbracket\ \mathsf{Decl} \bullet \mathsf{Proc} \mid\ \interleave\ \mathsf{Decl} \bullet \mathsf{Proc} \\
\mathsf{PPar} & ::= & \mathsf{Par} \mid \mathsf{N} \mathrel{\widehat{=}} \mathsf{ParAction} \mid \mathbf{nameset}\ \mathsf{N} == \mathsf{NSExp} \\
\mathsf{ParAction} & ::= & \mathsf{Action} \mid \mathsf{Decl} \bullet \mathsf{ParAction} \\
\mathsf{Action} & ::= & \mathsf{SchemaExp} \mid \mathsf{Command} \mid \mathsf{N} \mid \mathsf{CSPAction} \mid \mathsf{Action}\ [\mathsf{N}^+ := \mathsf{Exp}^+] \\
\mathsf{CSPAction} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \mathsf{Comm} \rightarrow \mathsf{Action} \mid \mathsf{Pred}\ \&\ \mathsf{Action} \\
 & \mid & \mathsf{Action};\ \mathsf{Action} \mid \mathsf{Action} \Box \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
 & \mid & \mathsf{Action} \llbracket\ \mathsf{NSExp} \mid \mathsf{CSExp} \mid \mathsf{NSExp}\ \rrbracket\ \mathsf{Action} \\
 & \mid & \mathsf{Action}\ \llbracket \mathsf{NSExp} \mid \mathsf{NSExp} \rrbracket\ \mathsf{Action} \\
 & \mid & \mathsf{Action} \setminus \mathsf{CSExp} \mid \mathsf{ParAction}(\mathsf{Exp}^+) \mid \mu\,\mathsf{N} \bullet \mathsf{Action} \\
 & \mid & ;\ \mathsf{Decl} \bullet \mathsf{Action} \mid\ \Box\ \mathsf{Decl} \bullet \mathsf{Action} \mid\ \sqcap\ \mathsf{Decl} \bullet \mathsf{Action} \\
 & \mid & \llbracket \mathsf{CSExp} \rrbracket\ \mathsf{Decl} \bullet \llbracket \mathsf{NSExp} \rrbracket \bullet \mathsf{Action} \mid\ \interleave\ \mathsf{Decl} \bullet \llbracket \mathsf{NSExp} \rrbracket\ \mathsf{Action} \\
\mathsf{Comm} & ::= & \mathsf{N}\ \mathsf{CParameter}^* \mid \mathsf{N}\ [\mathsf{Exp}^+]\ \mathsf{CParameter}^* \\
\mathsf{CParameter} & ::= & ?\mathsf{N} \mid ?\mathsf{N} : \mathsf{Pred} \mid !\mathsf{Exp} \mid .\mathsf{Exp} \\
\mathsf{Command} & ::= & \mathsf{N}^+ := \mathsf{Exp}^+ \mid \mathbf{if}\ \mathsf{GActions}\ \mathbf{fi} \mid \mathbf{var}\ \mathsf{Decl} \bullet \mathsf{Action} \\
 & \mid & \mathsf{N}^+ : [\mathsf{Pred}, \mathsf{Pred}] \mid \{\mathsf{Pred}\} \mid [\mathsf{Pred}] \\
 & \mid & \mathbf{val}\ \mathsf{Decl} \bullet \mathsf{Action} \mid \mathbf{res}\ \mathsf{Decl} \bullet \mathsf{Action} \mid \mathbf{vres}\ \mathsf{Decl} \bullet \mathsf{Action} \\
\mathsf{GActions} & ::= & \mathsf{Pred} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \rightarrow \mathsf{Action} \Box \mathsf{GActions}
\end{array}
$$

Fig. 1. *Circus* syntax

S. Channels can also be declared using schemas that group channel declarations. Channel sets may be introduced in a **chanset** paragraph. The empty set of channels $\{\!|\,|\!\}$, channel enumerations enclosed in $\{\!|$ and $|\!\}$, and expressions formed by some of the Z set operators are the elements of the syntactic category CSExp.

A process may be explicitly defined or defined in terms of other processes using CSP operators, iterated CSP operators, or indexed operators, which are particular to *Circus* specifications. An explicit process definition is delimited by the keywords **begin** and **end**, and is formed by a sequence of process paragraphs; a nameless action at the end defines the process behaviour.

The parallel operator follows the alphabetised approach adopted by [15]; we must declare a synchronisation channel set. Processes can also be composed in interleaving. An indexed process $i : T \odot P$ behaves exactly like P, but for each channel c of P, we have a freshly named channel c_i. These channels are implicitly declared by the indexing operator, and communicate pairs of values: the first element, the index, is a value i of type T, and the second element is the value of the original type of the channel. An indexed process P can be instantiated using the operator $P\lfloor e \rfloor$; it behaves just like P, however, the value of the expression e is used as the first element of the pairs communicated through all the channels.

An action can be a schema expression, a guarded command [5], a specification statement [11], an invocation to a previously defined action, a call by value, result, or by value-result, a recursive definition, or a combination of these constructs using CSP operators and their iterated versions. Furthermore, state components and local variables may be renamed; however, no channel name can be changed.

A guard may be associated with any action: given a Z predicate p, if the condition p is $true$, the action $p \mathbin{\&} A$ behaves like A; otherwise, it deadlocks. The parallel and the interleaving operators are slightly different from those for processes. In order to avoid conflicts in the access to the variables in scope (state components, and input and local variables), parallel composition and interleaving of actions must also declare two disjoint sets of variables. In $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, A_1 and A_2 synchronise on the channels in the set cs. Both actions have access to the initial values of all variables, but A_1 and A_2 may modify only the values of the variables in ns_1 and ns_2, respectively.

For a more detailed account of *Circus* and examples, please refer to [12].

3 UTP and Reactive Designs

Every program, design, and specification is interpreted in the UTP as a relation between an initial observation and a single subsequent observation, which may be either an intermediate or a final observation of the behaviour of a program execution. The relations are defined as predicates over observational variables; these are names that are important to describe all relevant aspects of a program behaviour. The initial observations of each variable are undecorated, and subsequent observations are decorated with a dash.

In this paper, four UTP observational variables are important: the boolean variable $okay$ indicates if the system has been properly started in a stable state, in which case its value is $true$, or not; $okay'$ means subsequent stabilisation in an observable state; the variable tr, whose type is a sequence of events, records all the events in which a program has engaged; the boolean variable $wait$ distinguishes the intermediate observations of waiting states from final observations on termination. In a stable intermediate state, $wait'$ has $true$ as its value; a $false$ value for $wait'$ indicates that the program has reached a final state. Finally, the variable ref describes the responsiveness properties of the process; its type is a set of events. All the events that may be refused by a process before the program has started are elements of ref, and possibly refused events at a later moment are referred by ref'.

Healthiness conditions are used in the UTP to test a specification or design for feasibility, and reject it, if it makes implementation impossible in the target language. They are often expressed in terms of an idempotent function ϕ that makes a program healthy; every ϕ-healthy program P is a fixed point of ϕ.

In [4], Cavalcanti and Woodcock present an introduction to CSP in the UTP. Their definitions correspond to the ones presented in [10], but with a different style of specification: every CSP process is defined as a reactive design $\mathbf{R}(pre \vdash post)$. A design $pre \vdash post$ is a pre-post specification, and $\mathbf{R}$ is a healthiness condition that gives a characterisation of a relation as a reactive process. Using this style, we use a design to define the behaviour of a process when its predecessor has terminated and not diverged; the process behaviour in the other situations is defined by $\mathbf{R}$, which is a composition of three healthiness conditions that we explain in the sequel.

The first healthiness condition, $\mathbf{R1}(P) \mathrel{\widehat{=}} P \wedge tr \leq tr'$, states that the history of interactions of a process cannot be changed, therefore, the value of tr can only get longer. The condition $tr \leq tr'$ holds if, and only if, the sequence tr is a prefix of or equal to tr'. The second healthiness condition, $\mathbf{R2}(P(tr, tr')) \mathrel{\widehat=} P(\langle\rangle, tr' - tr)$, establishes that a reactive process should not rely on the interactions that happened before its activation. The expression $s - t$ stands for the result of removing an initial copy of t from s; this partial operator is only well-defined if t is a prefix of s. The sequence $tr' - tr$ represents the traces of events in which the process itself has engaged from the moment it starts to the moment of observation. The last healthiness condition, $\mathbf{R3}(P) \mathrel{\widehat=} \mathit{II}_{rea} \lhd wait \rhd P$, defines the behaviour of a process that is waiting for another process to finish: it should not start. If the condition b is $true$, $P \lhd b \rhd Q$ is equivalent to P; otherwise, it is equivalent to Q.

In [10] it is not clear whether CSP processes have state or not; however, it is clear that, if there are state variables, they are not changed. We consider the state variables as part of the following definition for the reactive skip.

$$\begin{aligned}\mathit{II}_{rea} \mathrel{\widehat=}\ & (\neg\, okay \wedge tr \leq tr') \\ & \vee (okay' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \wedge v' = v)\end{aligned}$$

If the previous process diverged, the reactive skip only guarantees that the history of communication is not forgotten; otherwise, it terminates and keeps the values of the variables unchanged. For conciseness, throughout this paper, given a process with state components and local variables $x_1, \ldots, x_n$, the predicate $v' = v$ denotes the conjunction $x_1' = x_1 \wedge \ldots \wedge x_n' = x_n$.

CSP processes are reactive designs that satisfy two other healthiness conditions: the only guarantee on divergence of a **CSP1** process is the extension of the trace ($\mathbf{CSP1}(P) \mathrel{\widehat=} P \vee (\neg\, okay \wedge tr \leq tr')$), and **CSP2** processes may not require non-termination ($\mathbf{CSP2}(P) \mathrel{\widehat=} P;\ J$). In the definition of **CSP2** we take the approach of [4] instead of that in [10]. We make use of an idempotent function **CSP2**, which is defined in terms of J defined as $(okay \Rightarrow okay') \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \wedge v' = v$.

Processes that can be defined using the notation of CSP satisfy other healthiness conditions. One of them, **CSP3**, requires that the behaviour of a process does not depend on the initial value of ref ($\mathbf{CSP3}(P) \mathrel{\widehat=} SKIP;\ P$). The value of ref' has no relevance after termination of **CSP4** processes and a deadlocked **CSP5** process that refuses some events offered by its environment will still be deadlocked in an environment that offers even fewer events. Both, **CSP4** and **CSP5**, are expressed in terms of CSP constructs that have a slightly different definition in *Circus*: **CSP4** processes satisfy the right unit law ($P;\ SKIP = P$) and **CSP5** processes satisfy the unit law of interleaving ($P \mathbin{|||} SKIP = P$) [10]. The healthiness conditions $\mathbf{C1}(A) \mathrel{\widehat=} A;\ Skip$ and $\mathbf{C2}(A) \mathrel{\widehat=} A \mathbin{[\![} ns_1 \mid ns_2 \mathbin{]\!]} Skip$ lift these two healthiness conditions to state-rich *Circus* processes. The last of the *Circus* healthiness conditions, **C3**, guarantees that every *Circus* action, when expressed as a reactive design, has no dashed variables in the precondition ($\mathbf{C3}(A) \mathrel{\widehat=} \mathbf{R}(\neg\, A^f_f;\ true \vdash A^t_f)$).

4 *Circus* Denotational Semantics

The denotational semantics of *Circus* that we present in the sequel is based on the work presented in [17] and [10], but provides a framework to prove properties of *Circus* as well as of *Circus* specifications. It follows the approach of [4]: the vast majority of the *Circus* actions are defined as reactive designs of the form $\mathbf{R}(pre \vdash post)$. Those which are not defined in this way, reuse the results of [10] and were proved to be indeed reactive. As a direct consequence of this, we have that every *Circus* action is **R** (**R1**, **R2**, and **R3**) healthy. The mechanisation of this semantics in ProofPower-Z is a prototype theorem prover for *Circus*; it is built on top of the UTP theories presented in [14].

The first action we present is the deadlock action *Stop*: it is incapable of engaging in any events and is always waiting.

$$Stop \mathrel{\widehat=} \mathbf{R}(true \vdash tr' = tr \wedge wait')$$

Stop has a *true* precondition because it never diverges. Furthermore, it never engages in any event and is indefinitely waiting; therefore, its trace is left unchanged and $wait'$ is true. Since it represents deadlock, *Stop* must refuse all events (the final value of the refusal set, ref', is left unconstrained because any refusal set is a valid observation). As state changes do not decide a choice, in order to be the unit for external choice, *Stop* must leave the values of the state components unconstrained. In [4], we have proven that this definition corresponds to that of the UTP.

Skip is the action that terminates immediately and makes no changes to the trace or to the state components: its reactive design has a *true* precondition and $tr' = tr \wedge \neg\, wait' \wedge v' = v$ as postcondition. The value of ref' is left unspecified because it is irrelevant after termination.

The worst *Circus* action is *Chaos*; it has an almost unpredictable behaviour and has $\mathbf{R}(false \vdash true)$ as its semantics. Since it is defined as a reactive design, *Chaos* cannot undo the events of a process history. For this reason, it is not the right zero for sequential composition. Next, the *Circus* sequential composition is defined as relational sequence.

The guarded action $g \,\&\, A$ deadlocks if g is false, and like A otherwise. For conciseness, in the definition that follows and throughout this chapter, we abbreviate $A[b/okay'][c/wait]$ as A^b_c. Basically, A^f_f are the conditions in which A diverges when it is not waiting for its predecessor to finish, and A^t_f are the conditions that are satisfied when A terminates without diverging.

$$g \,\&\, A \mathrel{\widehat=} \mathbf{R}((g \Rightarrow \neg\, A^f_f) \vdash ((g \wedge A^t_f) \vee (\neg\, g \wedge tr' = tr \wedge wait')))$$

If the guard g is *false*, this definition can be reduced to *Stop*. However, if the guard g is *true*, we are left with the reactive design $\mathbf{R}(\neg\, A^f_f \vdash A^t_f)$; this has already been proved to be A itself provided A is a CSP process [10].

An external choice $A_1 \Box A_2$ does not diverge if neither A_1 nor A_2 do. We capture this behaviour in the precondition of the following definition of external choice. The postcondition establishes that if the trace has not changed and the choice has not terminated, the behaviour of an external choice is given by the conjunction of the effects of both actions; otherwise, the choice has been made and the behaviour is either that of A_1 or A_2. It is an important consequence of this definition that a state change does not resolve a choice; this would be expressed by including $v' = v$ in the condition of the postcondition.

$$\begin{array}{l} A_1 \Box A_2 \mathrel{\widehat=} \\ \qquad \mathbf{R}(\neg\, {A_1}^f_f \wedge \neg\, {A_2}^f_f \vdash ({A_1}^t_f \wedge {A_2}^t_f) \lhd tr' = tr \wedge wait' \rhd ({A_1}^t_f \vee {A_2}^t_f)) \end{array}$$

The internal choice is not defined as a reactive design: it is the disjunction of both actions. This is a simple definition, and the use of reactive designs to define an internal choice gives rise to a slightly more complicated definition.

Our semantics for prefix uses the function $do_{\mathcal{C}}$ presented below, which gives the behaviour of the prefix regarding tr and ref. For us, an event is a pair (c, e), where the first element is the name of the channel and the second element is the value that is communicated. For synchronisation events, we have the special value $Sync$.

$$do_{\mathcal{C}}\,(c, e) \mathrel{\widehat=}\ tr' = tr \wedge (c, e) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (c, e) \rangle$$

While waiting, an action that is willing to synchronise on an event (c, e) has not changed its trace and cannot refuse this event. After the communication $(\neg\, wait')$, the event is included in the trace of the action. A synchronisation $c \longrightarrow Skip$ does not diverge; neither does it change the state.

$$c \longrightarrow Skip \mathrel{\widehat=}\ \mathbf{R}(true \vdash do_{\mathcal{C}}\,(c, Sync) \wedge v' = v)$$

In [12], we prove that this result corresponds to the one presented in the UTP, but considers state variables in the postcondition. In *Circus*, output communications are a syntactic sugaring for synchronisations on output values v, which are taken into account $(do_{\mathcal{C}}(c, v))$ in the corresponding semantics.

An input prefix considers every possible value that can be communicated through the channel. Besides, once the communication happens, the value of the input variable changes accordingly. The function $do_{\mathcal{I}}$ takes these aspects into account. We consider the availability of an environment δ, that stores the types of every channel in the system. Before the communication, an input prefix $c?x : P$ cannot refuse any communication on the set composed by the events on c that communicate values of the type of c that satisfy the predicate P. After the communication the trace is extended by one of these events. Besides, the final value of x is that which is communicated. The function snd returns the second element of a pair, and the function $last$ returns the last element of a non-empty list.

$$\begin{array}{rl} do_{\mathcal{I}}\,(c, x, P) \mathrel{\widehat=} & tr' = tr \wedge \{e : \delta(c) \mid P \bullet (c, e)\} \cap ref' = \emptyset \\ & \lhd wait' \rhd \\ & tr' - tr \in \{e : \delta(c) \mid P \bullet \langle (c, e) \rangle\} \wedge x' = snd(last(tr')) \end{array}$$

Similarly to non-input prefix, we define the input prefix in terms of $do_{\mathcal{I}}$; however, an input prefix $c?x : P \rightarrow A(x)$ implicitly declares a new variable x and, after the communication, uses the communicated value in A. In the definition below, we consider that v and v' do not contain x and x', respectively.

$$c?x : P \rightarrow A(x) \mathrel{\widehat=} \mathbf{var}\ x \bullet \mathbf{R}(true \vdash do_{\mathcal{I}}(c, x, P) \land v' = v);\ A(x)$$

In [12], we show that if the set $\{e : \delta(c) \mid P\}$ is finite, the input prefixing above corresponds to the external choice $\Box\, x : \{e : \delta(c) \mid P\} \bullet c.x \rightarrow A(x)$. In this paper, we do not consider all the possible combinations of inputs and outputs in a prefixing; their semantics is lengthy, but not illuminating.

The parallel composition $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$ models interaction between the two concurrent actions A_1 and A_2. Here, we assume that references to channels sets have already been expanded using their corresponding definitions. We present the semantics of parallel operator as a reactive design in two parts: first we discuss its precondition, and then, we discuss its postcondition.

Divergence can only happen if it is possible for either of the actions to reach divergence. This is characterised by a trace that leads one of the actions to divergence and on which both actions agree regarding cs. For instance, the predicate below characterises possibility of divergence for A_1.

$$\exists\, 1.tr', 2.tr' \bullet ({A_1}^f_f;\ 1.tr' = tr) \land ({A_2}_f;\ 2.tr' = tr) \land 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs$$

Basically, if there exist two traces $1.tr'$ and $2.tr'$, defined as a trace of A_1 after divergence and as a trace of A_2, and if these two traces are equal modulo cs, then it is possible for A_1 to reach divergence. First, we define the trace $1.tr'$ on which A_1 diverges as ${A_1}^f_f;\ 1.tr' = tr$. The first predicate of the sequence give us the conditions under which A_1 diverges; we record the final trace in $1.tr'$ in the second predicate of the sequence, which ignores the final values of the other variables. Similarly, we define $2.tr'$ for A_2 as ${A_2}_f;\ 2.tr' = tr$. Since we are not interested in divergence, we do not replace $okay'$ by any particular value. Finally, we compare these traces after removing all the events that are not communications on the channels in cs. These can occur independently, but for the communications that require synchronisation, $1.tr'$ and $2.tr'$ have to agree (using the sequence filtering function $\upharpoonright$).

In a very similar way as we presented above for A_1, we can also express the possibility of divergence for A_2. The parallel composition diverges if either of these two conditions is true; hence, the precondition of the reactive design for the parallel composition is the conjunction of the negation of both conditions.

The postcondition uses the parallel by merge from [10]. Conceptually, it runs both actions independently and merge their results afterwards:

$$(({A_1}^t_f;\ U1(out\alpha A_1)) \land ({A_2}^t_f;\ U2(out\alpha A_2)))_{+\{v,tr\}};\ M_{\parallel_{cs}}$$

In order to express their independent executions, we use a relabelling function Ul: the result of applying Ul to an output alphabet $\{v_1', \ldots, v_n'\}$ is the predicate $l.v_1' = v_1 \land \ldots \land l.v_n' = v_n$. Before the merge, however, we extend the alphabet of the predicate that expresses the independent execution of both actions with v' and tr'; in this way, we record the initial values of the trace tr and of the state components and local variables v in tr' and v', respectively. For a predicate P and name n, the alphabet extension $P_{+\{n\}}$ is equivalent to $P \wedge n' = n$. The initial values of tr and v are used by the merge function $M_{\|_{cs}}$, as we explain in the sequel.

The function $M_{\|_{cs}}$ is responsible for merging the traces of both actions, the state components, local variables, and the UTP observational variables.

$$\begin{array}{rl} M_{\|_{cs}} \mathrel{\widehat=} & tr' - tr \in (1.tr - tr \parallel_{cs} 2.tr - tr) \wedge 1.tr \upharpoonright cs = 2.tr \upharpoonright cs \\ \wedge & \left(\begin{array}{l} \left(\begin{array}{l} (1.wait \vee 2.wait) \wedge \\ ref' \subseteq ((1.ref \cup 2.ref) \cap cs) \cup ((1.ref \cap 2.ref) \setminus cs) \end{array} \right) \\ \lhd wait' \rhd \\ (\neg\, 1.wait \wedge \neg\, 2.wait \wedge MSt) \end{array} \right) \end{array}$$

The trace is extended with the merge of the new events that happened in both actions. The function $\parallel_{cs}$ takes the individual traces and gives a set containing all the possible combinations of these two traces taking cs into consideration. The expression before the merge gives us all the possible behaviours of running A_1 and A_2 independently; however, only those combinations that are feasible regarding the synchronisation on cs should be considered $(1.tr \upharpoonright cs = 2.tr \upharpoonright cs)$. The definition of $\parallel_{cs}$ is omitted here but can be found in [12]; it is similar to that presented in [15] for CSP. Finally, the parallel composition has not terminated if any of the actions have not terminated. In this case, the parallel composition refuses all events in cs that are being refused by any of the actions and all the events not in cs which are being refused by both actions. We merge the states when both actions terminate: for every local variable and state component v, if it is declared in ns_1, its final value is that of A_1; if, however, it is declared in ns_2, its final value is that of A_2; finally, if it is declared in neither ns_1 nor ns_2, its value is left unchanged.

The interleaving does not have to consider any synchronisation channel. An interesting aspect regarding the differences between the definitions of parallel composition and interleaving is the much simpler precondition for interleaving. Since both actions may execute independently, the interleaving of two actions diverges if either of the actions do so. Therefore, its precondition is the same as that for external choice $\neg\, {A_1}^f_f \wedge \neg\, {A_2}^f_f$. Its postcondition is very similar to that of the parallel operator, but uses a different merge function $M_{|||_{cs}}$. As a matter of fact, interleaving is equivalent to parallel composition on an empty synchronisation channel set.

The hiding operator is also not defined as a reactive design. The calculations to express hiding as a reactive design pointed out that the final definition would be quite complicated and extensive; hence, we preferred to base our definition on that presented in [10] for the CSP hiding.

Here, we consider only the explicitly definition $(\mu\, X \bullet F(X))$ of recursion; the implicit definition using action invocation can be simply syntactically transformed to it. The semantics of recursion is standard: for a monotonic function F from *Circus* actions to *Circus* actions, the weakest fixed-point is defined as the greatest lower bound (the *weakest*) of all the fixed-points of F ($\bigsqcap \{X \mid F(X) \sqsubseteq_{\mathcal{A}} X\}$); in a similar way, mutually recursive actions are also defined as weakest fixed-points, but the functions considered are vectorial and so is the refinement.

The iterated operators are used to generalise the binary operators of sequence, external and internal choice, parallel composition, and interleaving; only finite types can be used for the indexing variables. Basically, the semantics of all the iterated operators is given by the expansion of the operator.

The semantics of a reference to an action name is given by the copy rule: it is the body of the action. Invocation of unnamed parametrised actions is defined simply as the substitution of argument for the formal parameter. The renaming of the local variables and state components is simply the syntactic substitution of the new names for the old ones.

The semantics of assignment is rather simple: it never diverges, terminates successfully leaving the trace unchanged, and sets the final values of the variables in the left-hand side to their new corresponding values. The remaining variables, denoted in the definition below by u ($u = v \setminus \{x_1, \ldots, x_n\}$), are left unchanged.

$$\begin{array}{l} x_1, \ldots, x_n := e_1, \ldots, e_n \mathrel{\widehat=} \\ \qquad \mathbf{R}(\mathit{true} \vdash \mathit{tr}' = \mathit{tr} \land \neg\, \mathit{wait}' \land x_1' = e_1 \land \ldots \land x_n' = e_n \land u' = u) \end{array}$$

Specification statements only terminate successfully establishing the postcondition if its precondition holds; only the variables in the frame can be changed. Furthermore, on successful termination, the trace is left unchanged. Now, we use u to denote the variables that are not in the frame ($u = v \setminus w$).

$$w : [\, \mathit{pre}, \mathit{post}\,] \mathrel{\widehat=} \mathbf{R}(\mathit{pre} \vdash \mathit{post} \land \neg\, \mathit{wait}' \land \mathit{tr}' = \mathit{tr} \land u' = u)$$

Assumptions $\{g\}$ and coercions $[\, g\,]$ are simply syntactic sugaring for specification statements $: [g, \mathit{true}]$ and $: [\mathit{true}, g]$, respectively.

Alternation can only diverge if none of the guards is *true*, or if any action guarded by a valid guard diverges; any of the guarded actions whose guard is valid can be chosen for execution.

$$\mathbf{if} \, [\!] \; i \bullet g_i \rightarrow A_i \, \mathbf{fi} \mathrel{\widehat=} \mathbf{R}((\bigvee i \bullet g_i) \land (\bigwedge i \bullet g_i \Rightarrow \neg\, A_{i_f}^{f}) \vdash \bigvee i \bullet g_i \land A_{i_f}^{t})$$

Variable block is defined in terms of the UTP constructors **var** and **end**; the former begins the scope of a variable, and the latter ends it.

Parametrisation by value, result, or by value-result are defined in terms of variable blocks and assignments. For instance, in a parametrisation by value, the formal parameter receives the value of the actual argument, which is actually to be used by the action. Therefore, we may define $(\mathbf{val}\; x : T \bullet A)(e)$ as $\mathbf{var}\; x : T \bullet x := e;\, A$. If, however, the parametrisation is neither by value, result, nor by value-result, the parameter is considered as a local variable and its instantiation is the substitution of the argument for the formal parameter.

We use the basic conversion rule of [2] to characterise schema expressions as specification statements. We assume that the schema expressions have already been normalised using the normalisation techniques presented in [18]. Besides, in *Circus*, the Z notations for input (?) and output (!) variables are syntactic sugaring for undashed and dashed variables, respectively. This implies that we actually have schemas containing the declaration of dashed ($ddecl'$) and undashed ($udecl$) variables and the predicate that determines the effect of the action. As a small abuse of notation, $ddecl$ also stands for a comma-separated list of undashed variables introduced as dashed variables in $ddecl'$.

$$[udecl;\ ddecl' \mid pred] \mathrel{\widehat=} ddecl : [\exists\, ddecl' \bullet pred, pred]$$

An explicitly defined process has an encapsulated state, a sequence $PPars$ of *Circus* paragraphs, and a main action A. It declares the state components using a *Circus* variable block and behaves like A.

$$\mathbf{begin\ state}\ [decl \mid pred]\ PPars\ \bullet A\ \mathbf{end} \mathrel{\widehat=} \mathbf{var}\ decl \bullet A$$

All compound processes are defined in terms of an explicit process specification. For instance, sequence, external and internal choice is defined as follows.

$$\begin{array}{l}P\ op\ Q \mathrel{\widehat=} \mathbf{begin\ state}\ State \mathrel{\widehat=} P.State \land Q.State\\ \qquad\qquad\qquad (P.PPar \land_{\Xi} Q.State)\ (Q.PPar \land_{\Xi} P.State)\\ \qquad\qquad\qquad \bullet\ P.Act\ op\ Q.Act\\ \qquad\qquad\qquad \mathbf{end}\end{array}$$

The state of the process $P\ op\ Q$ is defined as the conjunction of the individual states of P and Q; for simplicity, we assume that name clashes are avoided through renaming. Furthermore, every schema in the paragraphs of P (Q), specify an operation on $P.State$ ($Q.State$); they are not by themselves operations on $P\ op\ Q$. For this reason, we need to lift them to operate on the global $State$. For a sequence of process paragraphs $P.PPar$, the operation $P.PPar \land_{\Xi} Q.State$ stands for the conjunction of each schema expression in the paragraphs $P.PPar$ with $\Xi Q.State$; this indicates that they do not change the components of the state of process Q ($Q.State$). The main actions are composed in the same way using op; all the references from $P.Act$ to the components of $P.State$ are through schemas, which have already been conjoined with $\Xi Q.State$; the same comment applies to $Q.Act$.

For parallel composition and interleaving the only difference is that we must determine the state partitions of the operators. These are trivially the state components of each individual process. The semantics of hiding includes all the process paragraphs as they are, but the main action includes the hiding.

Our semantics for an indexed process $x : T \odot P$ is that of a parametrised process $x : T \bullet P$. However, all the communications within the corresponding parametrised processes are changed. For every channel c used in P, we have a freshly named channel c_i, which communicates pairs of values: the first element is an index i of type T, and the second element is the value of the original type of the channel.

The semantics of the corresponding parametrised process is given using an extended channel environment δ that includes the new implicitly declared channels c_i.

$$x : T \odot P \mathrel{\widehat=} (x : T \bullet P)[c : usedC(P) \bullet c\_x.x]$$

The notation $P[c : usedC(P) \bullet c\_x.x]$ denotes the change, in P, of all the references to every used channel c by a reference to $c\_x.x$. Since our semantics for indexed processes are parametrised processes, the semantics for their instantiation is simply a parametrised process invocation.

Besides making it able to prove the refinement laws, the semantics presented here defines most of the operators as reactive designs. Throughout the proofs of the refinement laws we created a vast library of laws and lemmas on the UTP theories, and more specifically reactive designs, that is discussed in the next section.

5 The library - reusing results to prove laws

In this section, we discuss the strategy adopted in our proofs and the structure of this library, which fosters reuse of our results in the proof of other laws and properties of *Circus* and reactive designs in general. The full library and the respective proofs can be found in [12].

The strategy for proving that a program P is equal (or refined) to Q is:

(i) Flatten program P to a single reactive design $R(pre_P \vdash post_P)$.

(ii) Flatten program Q to a single reactive design $R(pre_Q \vdash post_Q)$.

(iii) Use lemmas and theorems from the library and predicate calculus to transform the first reactive design into the second one (in case of refinement an inverse implication is the required result).

The flattening stage involves definitions and theorems that transform program structures into a single reactive design. For instance, if P is the sequence $P_1 ; P_2$, the following lemma transforms it into a single reactive design.

Lemma 5.1

$$\begin{array}{l}\mathbf{R}(P_1 \vdash Q_1) ; \mathbf{R}(P_2 \vdash Q_2)\\ =\\ \mathbf{R}\left(\begin{array}{l} P_1 \wedge \neg\, ((okay' \wedge \neg\, wait' \wedge Q_1) ; \neg\, P_2)\\ \vdash ((wait' \wedge Q_1) \vee ((okay' \wedge \neg\, wait' \wedge Q_1) ; Q_2))\end{array}\right)\end{array}$$

*for P_1 not mentioning dashed variables, and P_1, Q_1, P_2 and Q_2 **R2**-healthy.*

It establishes that the sequence of two reactive designs diverges if either P_1 is already violated in the very beginning or if, on termination of the first reactive design $(okay' \wedge \neg\, wait')$, P_2 is violated. Otherwise, the whole sequence is either in an intermediate state that satisfies Q_1 (if the first program waits indefinitely) or in a final state that results from the execution of the second reactive design after the completion of the first one.

Two other lemmas give the conditions on which a reactive design diverges and the conditions that are satisfied on termination. They are specially useful in the transformation stage of proofs that involve operators and healthiness conditions like **C3**, which use these conditions in their representation as reactive designs. A reactive design diverges if it started in a divergent state ($\neg\, okay$) or in a state that does not satisfy its precondition. On termination, it establishes the postcondition, provided the precondition is satisfied.

Lemma 5.2 $(\boldsymbol{R}(P \vdash Q))^f_f = \boldsymbol{R1}(\neg\,(okay \wedge \boldsymbol{R2}(P)))$

Lemma 5.3 $(\boldsymbol{R}(P \vdash Q))^t_f = \boldsymbol{CSP1}(\boldsymbol{R1}(\boldsymbol{R2}(P \Rightarrow Q)))$

Both can be proved by applying the definitions of the healthiness conditions.

By way of illustration, we conclude this section by presenting one out of over a hundred proofs we presented in [12]: the external choice unit law ($Stop \;\Box\; A = A$). Its proof illustrates the use of our library and shows the reasons for choosing *Stop* to leave the state loose. Before presenting this proof, we present two lemmas that are used in the proof. These lemmas are also part of our library and are proved using Lemmas 5.2 and 5.3 discussed above. Lemmas 5.4 and 5.5 give the conditions on which *Stop* diverges and the effects of *Stop* when it does not diverge, respectively.

Lemma 5.4 $Stop^f_f = \neg\, okay \wedge tr \le tr'$

Lemma 5.5 $Stop^t_f = \boldsymbol{CSP1}(tr' = tr \wedge wait')$

Since the precondition of *Stop* is *true*, it only diverges if its predecessor has done so and, in this case, only guarantees that the trace history is not forgotten. Secondly, on termination, *Stop* does not change the trace and waits indefinitely; **CSP1** guarantees the expected behaviour on divergence of the predecessor.

We start our proof (i) by applying the definition of external choice.

(i)

$Stop \;\Box\; A$

$$= R\left(\begin{array}{l}(\neg\, Stop^f_f \wedge \neg\, A^f_f)\\ \vdash \\ ((Stop^t_f \wedge A^t_f) \lhd tr' = tr \wedge wait' \rhd (Stop^t_f \vee A^t_f))\end{array}\right) \qquad \text{[External choice]}$$

Next (iii), we start by using Lemmas 5.4 and 5.5 to transform $Stop^f_f$ and $Stop^t_f$, respectively. The application of predicate calculus gives us the following result.

(iii)

$$= R\left(\begin{array}{l}(\neg\,((\neg\,okay \wedge tr \leq tr') \vee A^f_f)) \\ \vdash \\ \left(\begin{array}{l}(\mathbf{CSP1}(tr' = tr \wedge wait') \wedge A^t_f) \\ \lhd tr' = tr \wedge wait' \rhd \\ (\mathbf{CSP1}(tr' = tr \wedge wait') \vee A^t_f)\end{array}\right)\end{array}\right) \qquad [\text{Lemmas 5.4 and 5.5}]$$

The predicate A^f_f corresponds to the substitution of $okay'$ and $wait$ in A; however, $\neg\,okay \wedge tr \leq tr'$ does not mention either of these variables. Therefore, we may expand the substitution; this leaves us with the definition of **CSP1**.

$$= R\left(\begin{array}{l}\neg\,(\mathbf{CSP1}(A))^f_f \\ \vdash \\ \left(\begin{array}{l}(\mathbf{CSP1}(tr' = tr \wedge wait') \wedge A^t_f) \\ \lhd tr' = tr \wedge wait' \rhd \\ (\mathbf{CSP1}(tr' = tr \wedge wait') \vee A^t_f)\end{array}\right)\end{array}\right) \qquad [\text{Substitution and } \mathbf{CSP1}]$$

In [12], we prove that every *Circus* action is a **CSP1-CSP3** process, and therefore, a **CSP** process [12]. For this reason, the application of **CSP1** to A can be removed.

$$= R\left(\neg\,A^f_f \vdash \left(\begin{array}{l}(\mathbf{CSP1}(tr' = tr \wedge wait') \wedge A^t_f) \\ \lhd tr' = tr \wedge wait' \rhd \\ (\mathbf{CSP1}(tr' = tr \wedge wait') \vee A^t_f)\end{array}\right)\right) \qquad [\text{From [12]}]$$

Next, by expanding the definition of **CSP1**, we get the following disjunction.

$$= R\left(\neg\,A^f_f \vdash \left(\begin{array}{l}(((tr' = tr \wedge wait') \vee (\neg\,okay \wedge tr \leq tr')) \wedge A^t_f) \\ \lhd tr' = tr \wedge wait' \rhd \\ (((tr' = tr \wedge wait') \vee (\neg\,okay \wedge tr \leq tr')) \vee A^t_f)\end{array}\right)\right) \qquad [\mathbf{CSP1}]$$

The simple expansion of designs shows us that *okay* cannot be *false* in the post-condition; hence, the predicate $\neg\,okay \wedge tr \leq tr'$ is *false*. This leaves us with the following reactive design.

$$= R\left(\neg\,A^f_f \vdash \left(\begin{array}{l}(tr' = tr \wedge wait' \wedge A^t_f) \\ \lhd tr' = tr \wedge wait' \rhd \\ ((tr' = tr \wedge wait') \vee A^t_f)\end{array}\right)\right) \qquad [\text{Design}]$$

At this point, we are able to contemplate our decision on the semantics of $Stop$. The next step in our proof is to remove the disjunction of the right-hand side of the condition and leave just the predicate A^t_f; this can be done because the expression $tr' = tr \wedge wait'$ is $false$. The condition comes direct from our definition of external choice, in which, as explained in Section 4, state changes have no direct consequence. If we had chosen state changes to decide the choice, this would be expressed by including the predicate $v' = v$ in the condition of the choice. If this were the case, then $Stop$ would also have to leave the state unchanged. However, this is not the case, and hence, in order to go ahead with our proof, it is clear that $Stop$ cannot restrict the state to be kept unchanged.

$$= R(\neg\, A^f_f \vdash ((tr' = tr \wedge wait' \wedge A^t_f) \vee A^t_f)) \qquad \text{[Conditional]}$$

Using predicate calculus we can remove the predicate $tr' = tr \wedge wait' \wedge A^t_f$. We are left with $R(\neg\, A^f_f \vdash A^t_f)$. A theorem proved in [4] guarantees that this reactive design corresponds to A itself, provided A is a **CSP** process; the application of this theorem establishes the stage (ii) of the proof strategy and concludes this proof. □

In total, our library contains one-hundred an twenty-two theorems and more than two-hundred lemmas, which are structured into three groups:

- **Lemmas on the healthiness conditions:** these are the lemmas that involve some particular structure resulting from each of the healthiness conditions discussed in Section 4.
- **Lemmas on theories:** these are related to particular theories and are subdivided into relations, designs, reactive designs and *Circus*. The lemmas presented in this paper belong to this group.
- **Lemmas on *Circus* operators:** these are the lemmas that involve some particular structure resulting from each of the *Circus* operators.

Besides the proofs of the refinement laws, this library was also used to prove the correspondence between the semantics of the CSP operators in *Circus* presented in this paper and the corresponding UTP semantics.

6 Conclusions

The *Circus* semantics presented in [17] did not allow us to prove meta-theorems in the *Circus* theory and, as a direct consequence, refinement laws. For this reason, in this paper, we provided *Circus* with a new and definitive denotational semantics. The approach taken by Cavalcanti and Woodcock [4] was an inspiration for this semantics: we express the semantics of the vast majority of the *Circus* constructs as reactive designs. This uniformity is reflected in the proofs of the refinement laws. Together, the work presented in this paper and the one presented in [4] provide us with a library of lemmas involving reactive designs and foster reuse of these results. Yet another contribution of this paper is the discussion on the strategy of proof used in [12] to prove the refinement laws proposed for *Circus*.

The semantic model for *Circus* processes presented in [17] was a Z specification. For this reason, the state invariant was implicitly maintained by all operators. In our semantics, this is no longer a fact: nothing is explicitly stated about the invariant in our semantics. We assume specifications that initially contain no commands, and therefore, change the state using only Z operations, which explicitly include the state invariant and guarantee that it is maintained. For this reason, our semantics ignores any existing state invariants, since they are considered in the refinement process, just as in Z.

As a direct consequence of our definition for external choice and the need for *Stop* to be its unit, our semantics of *Stop* does not keep the state unchanged, but loose. An alternative would be to allow state changes to resolve the choice, in which case, *Stop* would keep the state unchanged; however, the states of the processes are encapsulated and state changes should not be noticed by the external environment. Another major difference is the state partitions in the parallel composition and interleaving, which remove the problems intrinsic to shared variables and were suggested in [3]. These partitions also had a direct consequence in the semantics of the parallel composition and interleaving of processes. In [17], the parallel composition $P \llbracket cs \rrbracket Q$ conjoins each paragraph in P (Q) with $\Delta Q.State$ $(\Delta P.State)$; this lifts the paragraphs in P (Q) to a state containing also the elements of Q (P), but with no extra restrictions. For us, in the semantics of parallel composition and interleaving, each side of the composition has a copy of all the variables in scope. They may change the values of all these variables, but only the changes to those variables that are in their partition have an effect in the final state of the composition. For this reason, we do not need to leave $Q.State$ unconstrained. We use a definition that is very similar to the other binary process combinators; the only change is the consideration of state partitions.

Besides the healthiness conditions satisfied by reactive processes (**R1-R3**) and by CSP processes (**CSP1-CSP3**), *Circus* processes were also proved to satisfy three further healthiness conditions: the first two of them, **C1** and **C2**, have a direct correspondence with two of the extra CSP healthiness conditions, **CSP4** and **CSP5**. However, **C3** is novel; it guarantees that our reactive designs do not contain any dashed variables in the precondition.

The semantics presented in this paper has been mechanised in ProofPower-Z [1]; this work was based on a mechanisation of the UTP theories [14]. As far as we know, the mechanisation of its semantics in ProofPower-Z makes *Circus* the first specification language of concurrent systems that has a mechanised semantics. Based on this result, we intend to mechanise the proof of the theorems and lemmas of our library and, ultimately, the refinement laws. This will provide both academia and industry with a mechanised refinement calculus that can be used in the formal development of state-rich reactive programs as the one presented in [13].

References

[1] ProofPower. At http://www.lemma-one.com/ProofPower/index/index.html.

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

[3] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A refinement strategy for *Circus*. *Formal Aspects of Computing*, **15**(2–3):146–181, 2003.

[4] A. L. C. Cavalcanti and J. C. P. Woodcock. A tutorial introduction to CSP in Unifying Theories of Programming. In *Proceedings of the Pernambuco Summer School on Software Engineering: Refinement 2004*, 2004.

[5] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[6] 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.

[7] C. Fischer. How to combine Z with a process algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM '98: Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation*, pages 5–23. Springer-Verlag, 1998.

[8] The RAISE Language Group. *The RAISE Specification Language.* Prentice-Hall, 1992.

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

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

[11] C. Morgan. *Programming from Specifications.* Prentice-Hall, 1994.

[12] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus.* PhD thesis, Department of Computer Science, University of York, 2005. YCST-2006/02.

[13] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Formal development of industrial-scale systems. *Innovations in Systems and Software Engineering—A NASA Journal*, **1**(2):125–146, 2005.

[14] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Unifying theories in ProofPower-Z. In S. Dunne and B. Stoddart, editors, *UTP 2006: First International Symposium on Unifying Theories of Programming*, volume **4010** of *LNCS*, pages 123–140. Springer-Verlag, 2006.

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

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

[17] J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume **2272** of *LNCS*, pages 184–203. Springer-Verlag, 2002.

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