DOI 10.1007/s00165-007-0052-5
BCS © 2007
Formal Aspects of Computing (2009) 21: 3–32

Formal Aspects of Computing

A UTP semantics for *Circus*

Marcel Oliveira[1], Ana Cavalcanti[2], and Jim Woodcock[2]

[1] Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte, Natal, Brazil
[2] Department of Computer Science, University of York, York, UK

Abstract. *Circus* specifications define both data and behavioural aspects of systems using a combination of Z and CSP constructs. Previously, a denotational semantics has been given to *Circus*; however, a shallow embedding of *Circus* in Z, in which the mapping from *Circus* constructs to their semantic representation as a Z specification, with yet another language being used as a meta-language, was not useful for proving 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); as such, it allows the proof of meta-theorems about *Circus* including the refinement laws in which we are interested. Its correspondence with the CSP semantics is illustrated with some examples. We also discuss the library of lemmas and theorems used in the proofs of the refinement laws. Finally, we give an account of the mechanisation of the *Circus* semantics and of the mechanical proofs of the refinement laws.

Keywords: Relational model; Concurrency; Refinement calculus; 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 [Spi92, WD96] focus on data aspects of the systems; constructs to model behavioural aspects are not explicitly provided. On the other hand, CSP [Hoa85, Ros98], 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.

Combinations of Z with CCS [GS97, TA97], Z with CSP [Fis98, MS98, RWW94], Object-Z [CDD+90] with CSP [Fis97, MD98, Smi97], and Object-Z with timed CSP [MD98] are some attempts to combine both schools of formalisms. Furthermore, combinations of B and action systems [Abr03], B and CSP [TS99], and notations that describe both aspects, like RAISE [Gro92] have been used. As far as we know, however, none of them has a related refinement calculus to support code development. This has motivated the design of *Circus* [WC02], a language that characterises systems as processes, which group constructs that describe data and control; the Z notation is used to define most of the data aspects, and CSP is used to define behaviour.

Predicate transformers [Dij76] are commonly used as the basis of semantic models for imperative refinement calculi [Bac78, Mor87, Mor94]. However, a different model is used as the basis of theories of refinement for CSP, the failures-divergence model [Hoa85, Ros98]. Other works, such as those presented in [Fis97, Smi97], provide a failures-divergences model for Object-Z classes, in order to present the semantics for combinations of Object-Z and CSP. Although data refinement was investigated for these combinations, no refinement laws were proposed. In [WDB00], the failures model was used to give behavioural semantics to abstract data types. To give a semantics to *Circus*, we need a semantic model that is able to combine the notions of refinement for CSP and for imperative

Correspondence and offprint requests to: M. Oliveira, E-mail: marcel@dimap.ufrn.br

programs. The UTP [HJ98] is a framework that makes this combination possible by unifying the programming discipline across many different computational paradigms.

Every program, design, and specification is interpreted in the UTP as a relation between an initial observation and a 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; they represent concepts that are important to describe all relevant aspects of a program behaviour. The initial observations of each variable are represented using its undecorated name, and subsequent observations are represented using the name of the variable decorated with a dash, very much in the style of Z, for example.

Eight distinguished variables record important observations about a program: *okay* indicates whether the system has been properly started in a stable state or not; $okay'$ records the subsequent stabilisation in an observable state. The observational variable *wait* distinguishes the intermediate observations of waiting states from final observations on termination; $wait'$ distinguishes a stable intermediate state from a stable final state. But what is the role of $wait'$ when $okay'$ is *false*? First, note that any behaviour is better than divergence, so we cannot specify that a process *must* diverge, for such a specification would have only divergent refinements. This essential asymmetry is captured in a healthiness condition, **CSP2** (see Sect. 3.1 for details), that has the property that a CSP process P can be expressed in the form $P[false/okay'] \lor (okay' \land P[true/okay'])$. Because $okay'$ and $wait'$ are boolean variables, there are four possible combinations of their values, but another healthiness condition, **CSP1** (see Sect. 3.1 for details), conflates two of these possibilities. Suppose that a process P actually diverges (this is different from specifying that it must). Then $okay'$ will be *false*. This means that *okay* is *false* in the following process, Q. Now, **CSP1** says that, if a process is activated in a divergent state, then the only behaviour that can be relied upon is that the trace will be extended. In this way, Q continues P's divergence and behaves arbitrarily, but not even a divergent process can undo past events. A particular consequence of **CSP1** is that the value of *wait* in Q is irrelevant. So, there are only three situations that are important in P: $okay' \land wait'$, $okay' \land \neg\, wait'$, and $\neg\, okay'$.

The sequence of events *tr* records the events that occurred before the program was started; the sequence tr' records the events that occurred up to the intermediate or final state. The set of events ref' describe the events being refused in the intermediate or final state. Finally, the set of events *ref* is used only for a technicality: to make the program's relation homogeneous in its dashed and undashed components.

A denotational semantics for *Circus* was first published in [WC02]; it was also based on the UTP, but there our model for a *Circus* program is a Z specification. By using Z, that semantics allowed the use of tools like Z/EVES [Saa97] to analyse and validate the definitions, and to reason about systems specified in *Circus*. Unfortunately, that semantics is not appropriate to prove our refinement laws. The reason is that in [WC02] we provided a shallow embedding [BG95] of *Circus* in Z. Such an embedding does not allow us to express these laws; in order to prove properties about *Circus* itself, like our refinement laws, a new embedding of *Circus* in Z is needed. For this reason, in this paper, we provide *Circus* with a new and definitive denotational semantics. The approach taken in [CW06] described below was an inspiration for this semantics and fosters the reuse of the results presented there, simplifying and modularising the proofs of our refinement laws. Furthermore, based on this new semantics, we were able to mechanise the syntax and the semantics of *Circus* in Z. This allows us to mechanise the proofs of the *Circus* refinement laws that have already been done.

In [CW06], we present an introduction to CSP in the UTP. Our definitions correspond to the ones presented in [HJ98], but with a different style of specification: every CSP process is defined as a reactive design of the form $\mathbf{R}(pre \vdash post)$. A design $pre \vdash post$ is defined as $okay \land pre \Rightarrow okay' \land post$: if the program starts in a state satisfying its *pre*condition, the design will terminate, and, on termination, it will establish its *post*condition. 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 the healthiness condition **R**, which is a composition of three healthiness conditions that we explain in the sequel. As we prove in [CW06], they can be composed in any order.

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 Table 1, we summarise the three healthiness conditions that, together, characterise reactive processes. The first healthiness condition, **R1**, states that the history of interactions of a process cannot be changed, therefore, the value of *tr* can only get longer. The condition $tr \leqslant tr'$ holds if, and only if, the sequence *tr* is a prefix of or equal to the sequence tr'. The second healthiness condition, **R2**, 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

Table 1. Healthiness conditions: reactive processes

	Formal representation	Description
R1	$\mathbf{R1}(P) \mathrel{\widehat{=}} P \wedge tr \leqslant tr'$	The execution of a reactive process never undoes any event that has already been performed.
R2	$\mathbf{R2}(P(tr, tr')) \mathrel{\widehat{=}} P(\langle\rangle, tr' - tr)$	The behaviour of a reactive process is oblivious to what has gone before.
R3	$\mathbf{R3}(P) \mathrel{\widehat{=}} \mathit{II}_{rea} \lhd wait \rhd P$	Intermediate stable states do not progress.

of observation. The final healthiness condition, **R3**, defines the behaviour of a process that is still waiting for another process to finish: it should not start. If the condition b is true, the predicate $P \lhd b \rhd Q$ is equivalent to P; otherwise, it is equivalent to Q.

We consider the state variables v and v' as part of the following definition for the reactive skip.

$$\mathit{II}_{rea} \mathrel{\widehat{=}} (\neg\, okay \wedge tr \leqslant tr') \vee (okay' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \wedge v' = v)$$

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 $x'_1 = x_1 \wedge \cdots \wedge x'_n = x_n$.

The reactive skip could not be defined as a design, because the set of designs and the set of reactive processes are disjoint [CW06]. It could, however, be defined as a reactive design.

Theorem 1.1 $\mathit{II}_{rea} = \mathbf{R}(true \vdash \mathit{II})$

This theorem can be proved by expanding the definitions of the healthiness conditions and designs. The II is the relational skip; it is defined as $okay' = okay \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \wedge v' = v$.

The most recent denotational semantics of *Circus* was presented in [OCW06a]; it is based on the work presented in [HJ98, WC02], but follows the style of [CW06]. In [CW06], we write the UTP definitions of some CSP operators as reactive designs; we illustrate the approach in detail for the prefixing operator. In this paper, we use these results to define a definitive semantics for *Circus*. Each construct is formally described in terms of a predicate in the UTP theory which extends the CSP theory with new operators and extra healthiness conditions.

Based on the new definitions, we proved over 92% of the 146 refinement laws of *Circus* that were proposed in [Oli05b] inspired by industrial case studies. These proofs, which can be found in [Oli05a], are of soundness with respect to the semantics in this paper; they range over all the constructs of the language and include all the data simulation laws. Besides supporting a practical refinement technique, the proof of the refinement laws helped us to validate the semantics presented in this paper. Furthermore, this validation was also done by studying its relationship to the UTP CSP theory, as discussed in Sect. 3, and by proving the soundness of the *Circus* operational semantics [WCF05].

In [OCW06a], we also discussed the structure of the library of lemmas and theorems created during that work and illustrated its usefulness by presenting the proof of one of our refinement laws. In this paper, we extend this discussion by presenting further examples of lemmas and proofs that use the proof strategy.

In [OCW06b], we discuss a mechanisation of the UTP theory and our new *Circus* semantics using ProofPower-Z [PPW]; it uses Z as a meta-notation to provide a model for our theory. More specifically, we present a set-based model to UTP relations, and use it as a basis for the mechanisation of five theories: relations, designs, reactive processes, CSP, and *Circus*. It is a conservative extension of the existing theories of ProofPower-Z; this guarantees soundness. The automation of the proofs of the *Circus* refinement laws, however, was left as future work. In this paper, we describe the issues that were raised during this automation, which provide *Circus* with a mechanised refinement calculus; it is the basis for a refinement editor and a theorem prover for *Circus*.

In the next section we present *Circus*. Section 3 describes the *Circus* denotational semantics based on the UTP. A discussion on the *Circus* healthiness conditions is presented in Sect. 4 and the mechanisation of the *Circus* semantics in a theorem prover, ProofPower-Z is presented in Sect. 5. In sect. 6 we discuss the structure of the library of lemmas and theorems created during this work; its usefulness is illustrated with the proofs of some of our refinement laws. A discussion of the automation of these proofs is also presented in this section. Finally, we draw some conclusions in Sect. 7.

$$
\begin{array}{l}
\textbf{channel}\ init, liftNozzle, putNozzle, pressTrigger, releaseTrigger \\
\textbf{channel}\ enterAmount, reload : \mathbb{N} \\
\\
\textbf{process}\ Pump \mathrel{\widehat=} \\
\textbf{begin state}\ PState \mathrel{\widehat=} [\, fuelQ : \mathbb{N} \,] \\
\quad PInit \mathrel{\widehat=} [\, PState' \mid fuelQ' = 5000 \,] \\
\quad Reload \mathrel{\widehat=} [\, \Delta PState;\ q? : \mathbb{N} \mid fuelQ' = fuelQ + q? \,] \\
\quad Supply \mathrel{\widehat=} [\, \Delta PState;\ q? : \mathbb{N} \mid fuelQ' = fuelQ \ominus q? \,] \\
\\
\quad PumpIdle \mathrel{\widehat=} liftNozzle \rightarrow PumpActive \\
\qquad\qquad \Box\ reload?q \rightarrow Reload \\
\qquad\qquad \Box\ init \rightarrow PInit \\
\quad PumpActive \mathrel{\widehat=} putNozzle \rightarrow Skip \\
\qquad\qquad \Box\ enterAmount?q \rightarrow pressTrigger \rightarrow Supply;\ releaseTrigger \rightarrow Skip \\
\\
\bullet\ init \rightarrow PInit;\ \mu\, X \bullet PumpIdle;\ X \\
\textbf{end} \\
\\
\textbf{chanset}\ SyncCustomer \mathrel{\widehat=} \{\!|\ liftNozzle, pressTrigger, releaseTrigger, enterAmount\ |\!\} \\
\textbf{chanset}\ SyncEmployee \mathrel{\widehat=} \{\!|\ init, reload\ |\!\} \\
\\
\textbf{process}\ SinglePumpEmployee \mathrel{\widehat=} init \rightarrow reload!1000 \rightarrow Skip \\
\textbf{process}\ SinglePumpStation \mathrel{\widehat=} (SinglePumpEmployee \,[\![\, SyncEmployee \,]\!]\, Pump) \setminus SyncEmployee
\end{array}
$$

Fig. 1. Fuel pump in *Circus*

2. *Circus*

Circus is based on imperative CSP, but includes 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: channel declarations, channel set definitions, Z paragraphs, or process definitions. A process encapsulates its state and communicates through channels.

An example is given in Fig. 1: it specifies a process that controls a fuel pump. Seven channels are used in the system: *init* is used to initialise the pump, *liftNozzle* and *putNozzle* indicate to the pump that the nozzle has been lifted or put back, *pressTrigger* and *releaseTrigger* indicate to the pump that the trigger has been pressed or released, *enterAmount* is used by customers to enter the amount of fuel they want, and finally, *reload* is used by the gas station employee to reload the pump.

In Fig. 2, we present the BNF of the *Circus* syntax. We use CircusPar* to denote a possibly empty list of elements of the syntactic category CircusPar of *Circus* paragraphs; similarly for PPar* (process paragraphs). We use N$^+$ to denote a non-empty list of Z identifiers N. The categories Par, SchemaExp, Exp, Pred, and Decl include the Z paragraphs, schema expressions, expressions, predicates and declarations defined in [Spi92].

The declarations of all the channels give their names and the types of the values that they can communicate. In Fig. 1, channels *enterAmount* and *reload* communicate the amount of fuel that the client wants and the amount of fuel that was loaded into the pump. If, however, a channel does not communicate any value its declaration contains only its name. This is illustrated in Fig. 1 by the declaration of channels *init*, *liftNozzle*, *putNozzle*, *pressTrigger*, and *releaseTrigger*.

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 S. Channels can also be declared using schemas that group channel declarations. Channel sets may be introduced in a **chanset** paragraph. The empty set $\{\!|\,|\!\}$, channel enumerations enclosed in $\{\!|$ and $|\!\}$, and expressions formed by some of the Z set operators are the elements of the syntactic category CSExp. In a similar way, the empty set $\{\}$, name enumerations enclosed in { and }, and expressions formed by some of the Z set operators are the elements of the syntactic category NSExp. In our example, the set *SyncCustomer* groups the channels through which a customer may interact with the pump and the set *SyncEmployee* groups the channels through which an employee may interact with the pump.

A process may be explicitly defined or defined in terms of other processes (compound processes). 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 process *Pump* in Fig. 1 is defined in this way. The schema *PState* describes the internal state of the *Pump*: it contains a natural number *fuelQ* that stores the current quantity of fuel in the pump. The behaviour of *Pump* is described by the unnamed action after the $\bullet$. It is recursive: after a request to initialise, it performs the initialisation of the state, and behaves recursively, executing *PumpIdle*. The state component *fuelQ* is initialised with 5,000 using the Z operation *PInit*. The Z operation

```
Program     ::= CircusPar*
CircusPar   ::= Par | channel CDecl | chanset N == CSExp | ProcDecl
CDecl       ::= SimpleCDecl | SimpleCDecl; CDecl
SimpleCDecl ::= N+ | N+ : Exp | [N+]N+ : Exp | SchemaExp
ProcDecl    ::= process N ≙ ProcDef | process N[N+] ≙ ProcDef
ProcDef     ::= Decl • ProcDef | Decl ⊙ ProcDef | Proc
Proc        ::= begin PPar* state N ≙ SchemaExp PPar* • Action end
              | Proc; Proc | Proc □ Proc | Proc ⊓ Proc | Proc [| CSExp |] Proc
              | Proc ||| Proc | Proc \ CSExp | (Decl • ProcDef)(Exp+) | N(Exp+) | N
              | (Decl ⊙ ProcDef)⌊Exp+⌋ | N⌊Exp+⌋ | Proc[N+ := N+] | N[Exp+]
              | ; Decl • Proc | □ Decl • Proc | ⊓ Decl • Proc
              | [|CSExp|] Decl • Proc | ||| Decl • Proc
PPar        ::= Par | N ≙ ParAction | nameset N == NSExp
ParAction   ::= Action | Decl • ParAction
Action      ::= SchemaExp | Command | N | CSPAction | Action [N+ := Exp+]
CSPAction   ::= Skip | Stop | Chaos | Comm → Action | Pred & Action
              | Action; Action | Action □ Action | Action ⊓ Action
              | Action [| NSExp | CSExp | NSExp |] Action
              | Action [|[NSExp | NSExp]|] Action
              | Action \ CSExp | ParAction(Exp+) | μ N • Action
              | ; Decl • Action | □ Decl • Action | ⊓ Decl • Action
              | [|CSExp|] Decl • [|[NSExp]|] • Action | ||| Decl •[|[NSExp]|] Action
Comm        ::= N CParameter* | N [Exp+] CParameter*
CParameter  ::= ?N | ?N : Pred | !Exp | .Exp
Command     ::= N+ := Exp+ | if GActions fi | var Decl • Action
              | N+ : [Pred, Pred] | {Pred} | [Pred]
              | val Decl • Action | res Decl • Action | vres Decl • Action
GActions    ::= Pred → Action | Pred → Action □ GActions
```

Fig. 2. *Circus* syntax

Reload adds an input value $q?$ to the current quantity of fuel *fuelQ*; *Supply* subtracts (not below zero) the input $q?$ from the current quantity of fuel.

Compound processes are defined using the CSP operators of sequence, external and internal choice, parallel composition and interleaving, or their corresponding iterated operators, event hiding, or indexed operators, which are particular to *Circus* specifications, and are described later in this section. We can also instantiate a parametrised process by providing values for each of its parameters.

The parallel operator follows the alphabetised approach adopted by [Ros98]; we must declare a synchronisation channel set. By way of illustration, the process *SinglePumpStation* presented below is a parallel composition of an employee and a pump. The employee initialises the pump and reloads it with an additional 1,000 l. The employee and the pump synchronise on the set of events *SyncEmployee*.

Processes can also be composed in interleaving. For instance, a process that represents two pumps running independently can be defined as follows:

process $TwoPumps \mathrel{\widehat=} Pump \interleave Pump$

In this case the occurrence of any event in *SyncCustomer* or *SyncEmployee* in the environment leads to a non-deterministic choice of which *Pump* will synchronise with that event. For instance, if an event *init* occurs, one of the processes initialises, and the other one does not.

The event hiding operator $P \setminus cs$ is used to encapsulate the events that are in the channel set cs. This removes these events from the interface of P, which become no longer visible to the environment. For instance, the process *SinglePumpStation* encapsulates the interaction between the processes *SinglePumpEmployee* and *Pump* (*SyncEmployee*); the only way to interact with *SinglePumpStation* is via the channels in *SyncCustomer*.

Circus introduces a new operator that can be used to define processes. The 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, where i is the name of the variable. These channels are implicitly declared by the indexed 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 instantiation 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.

For instance, we may define a process similar to *TwoPumps*, in order to have the same process that represents two pumps running independently, but with an identification of which pump is initialised. In order to interact with the indexed process below we must use the channels *init_i*, *liftNozzle_i*, and so on.

process $IndexPump \mathrel{\widehat=} i : \{1, 2\} \odot Pump$

We may instantiate the process *IndexPump*: the process $IndexPump\lfloor 1 \rfloor$, for instance, inputs pairs through channel *enterAmount_i* whose first elements are 1 and the second elements are the amount of fuel requested by the customer. It may be initialised by sending the value 1 through the channel *init_i*. Similarly, we have the process

IndexPump⌊2⌋. Finally, we have the process presented below that represents a pair of pumps: the first element of the pairs that are communicated through channels like *enterAmount_i* identifies the pump.

process $TwoPumpsId \mathrel{\widehat=} IndexPump\lfloor 1\rfloor \mid\mid\mid IndexPump\lfloor 2\rfloor$

The renaming operator $P[oldc := newc]$ replaces all the communications that are done through channels *oldc* by communications through channels *newc*, which are implicitly declared, if needed.

An action can be a schema expression, a guarded command, an invocation of any action, or a combination of these constructs using CSP operators. Also, three primitive actions are available: *Skip*, *Stop*, and *Chaos*. The action *Skip* does not communicate any value nor changes the state: it terminates immediately. The action *Stop* deadlocks, and the action *Chaos* diverges.

The prefixing operator is standard, but a guard construction may be associated with it. For instance, if the condition *p* is *true*, the action $p \mathbin{\&} c?x \rightarrow A$ inputs a value through channel *c* and assigns it to the variable *x*, and then behaves like *A*, which has the variable *x* in scope. If, however, the condition *p* is *false*, the same action blocks. Such enabling conditions like *p* may be associated with any action.

The CSP operators of sequence, external and internal choice, parallel composition, interleaving, their corresponding iterated operators, and hiding may also be used to compose actions. Communications and recursive definitions are also available. In *Pump* we use an external choice to define *PumpIdle*: the nozzle can be lifted, in which case the *Pump* behaves like *PumpActive*, or the pump can be reloaded, using channel *reload*, or the pump can be initialised using channel *init*. In *PumpActive*, we have another external choice: the nozzle can be put back in the pump, in which case the pump recurses, or an amount of fuel may be requested via channel *enterAmount*, in which case the pump waits for the trigger to be pressed, and then it supplies the amount of fuel requested, releases the trigger, and recurses.

To avoid conflicts in the access to the variables in scope, parallel composition and interleaving of actions must declare a synchronisation channel set and two disjoint sets of variables. In the parallel composition $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, the actions A_1 and A_2 synchronise on the channels in the set *cs*. Both A_1 and A_2 have access to the initial values of all variables in both ns_1 and ns_2, but A_1 and A_2 may modify only the values of the variables in ns_1 and ns_2, respectively. The changes made by A_1 in the variables in ns_1 are not seen by A_2, and vice-versa.

Finally, an action may also be a variable block, a specification statement, an assumption, a coercion, an alternation, or an assignment. The semantics of *Circus* is an enriched failures-divergences model expressed in the UTP. It caters for communication and reaction, like the model of CSP, but also for data and data operations. The details of this semantics is the subject of the next section.

3. *Circus* denotational semantics

The original semantics given as a translation from *Circus* to Z [WC02] only allowed the proof of properties of particular *Circus* specifications, rather than general properties of *Circus* constructs, like the refinement laws. The denotational semantics of *Circus* that we present in the sequel provides a framework to prove properties of *Circus* as well as of *Circus* specifications; its summary can be found in Appendix B.

3.1. *Circus* actions

A *Circus* action can be a CSP process, a guarded command, an invocation of any action, a schema expression, or a combination of these constructs using CSP operators. The following sections present the semantics of each one of them.

3.1.1. Basic actions

The first action we define is the deadlock *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 [CW06], 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.

3.1.2. Sequence

The *Circus* sequential composition is not defined as a reactive design but as the relational sequence, which is defined in the UTP as an existential quantification on the intermediary state.

$$A_1; A_2 \mathrel{\widehat{=}} \exists\, x_0 \bullet A_1[x_0/x'] \wedge A_2[x_0/x]$$

In this definition, besides the state components and local variables v, the list of variables x also contains the four UTP observational variables.

3.1.3. Guarded action

The guarded action $g \mathbin{\&} A$ deadlocks if g is false, and behaves like A otherwise. For conciseness, in the definition that follows and throughout this paper, we abbreviate $A[b/okay'][c/wait]$ as A^b_c. Basically, $A^f_f \mathrel{\widehat{=}} A[false/okay'][false/wait]$ are the conditions in which A diverges when it is not waiting for its predecessor to finish, and $A^t_f \mathrel{\widehat{=}} A[true/okay'][false/wait]$ are the conditions in which A does not diverge when it is not waiting for its predecessor to finish.

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

If the guard g is *false*, this definition can be reduced to *Stop*. However, if g is *true*, we are left with the reactive design $\mathbf{R}(\neg\ A^f_f \vdash A^t_f)$; the following theorem shows us that this reactive design is exactly A itself.

Theorem 3.1 (from [HJ98]) For every CSP process A, $A = \mathbf{R}(\neg\ A^f_f \vdash A^t_f)$.

This theorem is proved in [HJ98] and applies to CSP processes. These processes are defined in the UTP as reactive processes that satisfy two other healthiness conditions presented in Table 2: the only guarantee on divergence of a **CSP1** process is the extension of the trace, and **CSP2** processes may not require non-termination. In the definition of **CSP2** we take the approach of [CW06] instead of that in [HJ98]. We make use of an idempotent function **CSP2**, which is defined in terms of a predicate J defined as follows:

$$J \mathrel{\widehat{=}} (okay \Rightarrow okay') \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \wedge v' = v$$

Besides **CSP1** and **CSP2**, processes that can be defined using the notation of CSP satisfy other healthiness conditions. The first one of them, **CSP3**, requires that the behaviour of a process does not depend on the initial value of ref; only the value of ref' is relevant to determine which events can be refused. Intuitively, the set of events that were previously refused (ref) should not be of any concern to the current process. Next, a terminating **CSP4** process does not restrict ref'. Finally, a deadlocked **CSP5** process that is refusing some events offered by the environment is still deadlocked in an environment that offers even fewer events.

The definition of **CSP3** refers to the CSP $SKIP$[1], which is defined as $\mathbf{R}(\exists ref \bullet \mathit{II})$. The UTP model of CSP processes, that is, those that satisfy **CSP1** and **CSP2**, is not isomorphic to the failures-divergences model [CW06]; it includes extra processes, and for those $SKIP$ is not necessarily an identity. These extra processes may exhibit, for example, miraculous behaviour. They are useful for a theory that combines data operations, CSP constructs, and a refinement notion.

[1] It is important to notice the difference between the CSP *SKIP* (capital letters) and the *Circus Skip* (capital *S*).

Table 2. Healthiness conditions: CSP processes

	Formal representation	Description
CSP1	$\mathbf{CSP1}(P) \mathrel{\widehat{=}} P \vee (\neg\, okay \wedge tr \leqslant tr')$	Extension of the trace is the only guarantee on divergence.
CSP2	$\mathbf{CSP2}(P) \mathrel{\widehat{=}} P ; J$ $J \mathrel{\widehat{=}} (okay \Rightarrow okay') \wedge tr' = tr$ $\wedge\, wait' = wait \wedge ref' = ref$ $\wedge\, v' = v$	A process may not require non-termination.
CSP3	$\mathbf{CSP3}(P) \mathrel{\widehat{=}} SKIP ; P$	A process does not depend on ref.
CSP4	$\mathbf{CSP4}(P) \mathrel{\widehat{=}} P ; SKIP$	A terminating process does not restrict ref'.
CSP5	$\mathbf{CSP5}(P) \mathrel{\widehat{=}} P \mid\mid\mid SKIP$	A deadlocked process that is refusing some events offered by the environment is still deadlocked in an environment that offers even fewer events.

3.1.4. External and internal choice

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 a 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.

$$A_1 \,\Box\, A_2 \mathrel{\widehat{=}} \mathbf{R}(\neg\, {A_1}_f^f \wedge \neg\, {A_2}_f^f \vdash ({A_1}_f^t \wedge {A_2}_f^t) \lhd tr' = tr \wedge wait' \rhd ({A_1}_f^t \vee {A_2}_f^t))$$

It is a direct and 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.

For example, let us consider the choice $(x := 0;\ c_1 \rightarrow Skip) \,\Box\, (x := 1;\ c_2 \rightarrow Skip)$. It does not happen instantly, but only when either c_1 or c_2 happens. The final value of x depends on which communication actually happens. We have chosen state changes not to resolve an external choice because states are encapsulated within a *Circus* process, and so their changes should not be noticed by the external environment.

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.

3.1.5. Prefixed action

Our semantics for prefixed actions uses the function *do* presented below, which gives the behaviour of the prefixed action 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 events that do not model the communication of any specific value, we have the special value *Sync*. 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.

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

This function is much simpler than the function $do_{\mathcal{A}}$ used in the UTP [HJ98] to define the CSP prefixing. Basically, the function $do_{\mathcal{A}}$ is a result of applying **R** to *do*. Lemma 3.1 presented below states this property; its proof and the proof of other lemmas used in this paper can be found in Appendix A.

Lemma 3.1 $do_{\mathcal{A}} = \mathbf{R}(do)$

In what follows, we use this lemma to calculate the definition of the CSP prefixing as a reactive design. Our calculation is presented in Fig. 3. In the first step, we use Theorem 3.1 to write the CSP prefixing as a reactive design.

$$
\begin{array}{ll}
c \rightarrow SKIP & \\
= \mathbf{R}(\neg\, (c \rightarrow SKIP)^f_f \vdash (c \rightarrow SKIP)^t_f) & \text{[Theorem 3.1]} \\
= \mathbf{R}(\neg\, (\neg\, okay \wedge tr \leq tr') \vdash \mathbf{CSP1}(do\,(c, Sync))) & \text{[Lemmas 3.2 and 3.3]} \\
= \mathbf{R}(true \vdash okay \wedge \mathbf{CSP1}(do\,(c, Sync))) & \text{[Designs and predicate calculus]} \\
= \mathbf{R}(true \vdash do\,(c, Sync)) & \text{[Lemma 3.4]}
\end{array}
$$

Fig. 3. Calculation of prefixing as a reactive design

Now, our concern is to transform the pre and the postcondition into more intuitive predicates. We use the two following lemmas.

Lemma 3.2 $(c \rightarrow SKIP)^f_f = \neg\, okay \wedge tr \leqslant tr'$

Lemma 3.3 $(c \rightarrow SKIP)^t_f = \mathbf{CSP1}(do\,(c, Sync))$

First, prefixing only diverges if it has already started in a divergent state, in which case, it only guarantees that the trace is not forgotten. At the end, the prefixing establishes the expected result given by the expression $do\,(c, Sync)$; the properties on divergence are guaranteed by **CSP1**. We continue our transformation by using both lemmas to transform the pre and the postcondition. Next, the definition of designs and simple predicate calculus can be used to simplify the precondition. Furthermore, the simple expansion of designs shows us that we may include (or remove) *okay* in (or from) the postcondition of any design. We use this fact, to introduce *okay* in the postcondition. The last lemma that we use in this calculation states that the application of **CSP1** is innocuous if we have that *okay* is true.

Lemma 3.4 $okay \wedge \mathbf{CSP1}(P) = okay \wedge P$

The expansion of **CSP1** and simple predicate calculus is enough to prove this lemma. We use this lemma and the freedom to remove *okay* from the postcondition to conclude the calculation in Fig. 3. □

The CSP prefixing never diverges and establishes the result of *do* on termination. This definition corresponds directly to the *Circus* one presented below; the only difference is that in *Circus* we must consider state variables: the *Circus* prefixing does not diverge nor does it change the state.

$$c \rightarrow Skip \mathrel{\widehat{=}} \mathbf{R}(true \vdash do\,(c, Sync) \wedge v' = v)$$

An input prefixing 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 use an environment δ that records for each channel c its type $\delta(c)$. Before the communication, $c?x : P$ cannot refuse any communication in 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.

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

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

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

In [Oli05b], 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.

3.1.6. Parallel composition and interleaving

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 channel sets have already been expanded using their corresponding

definitions. We present the semantics of the 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, $\exists\, 1.tr', 2.tr' \bullet (A_{1f}^{f}; (1.tr' = tr)) \wedge (A_{2f}; (2.tr' = tr)) \wedge 1.tr' \restriction cs = 2.tr' \restriction cs$ characterises possibility of divergence for A_1. 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 $A_{1f}^{f}; (1.tr' = tr)$. 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. In this case, we are not interested in the divergence of A_2 because A_1 is already divergent; hence, we do not replace $okay'$ by any particular value. Similarly, we define $2.tr'$ for A_2 as $A_{2f}; (2.tr' = tr)$. Finally, we compare these traces after removing all the events that are not in cs (using the sequence filtering function $\restriction$). These can occur independently, but for the communications that require synchronisation, $1.tr'$ and $2.tr'$ have to agree.

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 conditions are 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 [HJ98]. Conceptually, it runs both actions independently and merges their results afterwards:

$$((A_{1f}^{t};\ U1(out\alpha\ A_1)) \wedge (A_{2f}^{t};\ U2(out\alpha\ A_2)))_{+\{v,tr\}};\ M_{\parallel}(cs)$$

To express their independent executions, we use a relabelling function Ul: the result of applying Ul to a set $\{v'_1, \ldots, v'_n\}$ of dashed names is $l.v'_1 = v_1 \wedge \cdots \wedge l.v'_n = v_n$. For instance, the application of $U1$ to $\{wait'\}$ yields the predicate $1.wait' = wait$. Before the merge, however, we extend the alphabet of the predicate that expresses the independent execution of both actions with v and tr and their dashed counterparts; 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 $M_{\parallel}$, as we explain in the sequel.

The function $M_{\parallel}$ receives a channel set and merges the traces of both actions, the state components, local variables, and the UTP observational variables.

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

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 ($1.tr \restriction cs = 2.tr \restriction cs$). The definition of $\parallel_{cs}$ is omitted here but can be found in [Oli05b]; it is similar to that presented in [Ros98] 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. In order to terminate, both actions in the parallel composition must terminate; we merge the state as follows.

$$MSt \mathrel{\widehat{=}}\ \forall\, v \bullet (v \in ns_1 \Rightarrow v' = 1.v) \wedge (v \in ns_2 \Rightarrow v' = 2.v) \wedge (v \notin ns_1 \cup ns_2 \Rightarrow v' = v)$$

For every variable 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.

We present below the whole of the semantics of parallel composition.

$$A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 \mathrel{\widehat{=}} \\ \mathbf{R} \left(\begin{array}{l} \neg\ \exists\, 1.tr', 2.tr' \bullet (A_{1f}^{f}; (1.tr' = tr)) \wedge (A_{2f}; (2.tr' = tr)) \wedge 1.tr' \restriction cs = 2.tr' \restriction cs \\ \wedge \neg\ \exists\, 1.tr', 2.tr' \bullet (A_{1f}; (1.tr' = tr)) \wedge (A_{2f}^{f}; (2.tr' = tr)) \wedge 1.tr' \restriction cs = 2.tr' \restriction cs \\ \vdash \\ ((A_{1f}^{t};\ U1(out\alpha\ A_1)) \wedge (A_{2f}^{t};\ U2(out\alpha\ A_2)))_{+\{v,tr\}};\ M_{\parallel}(cs) \end{array} \right)$$

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 does. 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 $M_{|||}$. Interleaving is equivalent to parallel composition on an empty synchronisation channel set.

3.1.7. Hiding

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 [HJ98] for the CSP hiding. In the definition presented below, $EVENT$ denotes the universal set of events.

$$A \setminus cs \mathrel{\widehat{=}} \mathbf{R}(\exists\, s \bullet A[s, cs \cup ref'/tr', ref'] \wedge (tr' - tr) = (s - tr) \upharpoonright (EVENT - cs)); Skip$$

If A reaches a stable state in which it cannot perform any further events in cs, than the action $A \setminus cs$ has also reached such state. The new events $(tr' - tr)$ performed by $A \setminus cs$ are those new events performed by A (in this definition, we rename the final trace of A to s; so $s - tr$ gives us the new events of A), but filtered by the set of all events but those in cs. We also include the events in cs in the final refusal set of A by replacing ref' by $cs \cup ref'$. *Skip* guarantees that possible divergences introduced by hiding events in a recursive action are actually captured.

3.1.8. Recursion

We consider only the explicit definition $(\mu\, X \bullet F(X))$ of recursion; the implicit definition using action invocation can be 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 defined as weakest fixed-points, but the functions are vectorial and so is the refinement order. In our work, $\sqsubseteq_{\mathcal{A}}$ denotes action refinement; its definition can be found in [CSW03].

3.1.9. Iterated operators

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.

3.1.10. Action invocation, parametrised action and renaming

The semantics of a reference to an action name is given by the copy rule: it is the body of the action. Invocation of an unnamed parametrised action $(d \bullet A)(e)$ is defined simply as the substitution of the argument e for the formal parameter declared in d. The renaming of the local variables and state components is simply the syntactic substitution of the new names for the old ones.

3.1.11. Commands

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.

$$x_1, \ldots, x_n := e_1, \ldots, e_n \mathrel{\widehat{=}} \mathbf{R}(true \vdash tr' = tr \wedge \neg\, wait' \wedge x'_1 = e_1 \wedge \cdots \wedge x'_n = e_n \wedge u' = u)$$

A specification statement only terminates 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 $(u = v \setminus w)$.

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

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

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

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 used by the action; that is, we define $(\textbf{val } x : T \bullet A)(e)$ as $\textbf{var } x : T \bullet x := e; A$. 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. This is the parametrisation mechanism of CSP.

3.1.12. Schema expression

We use the basic conversion rule of [CW99] to characterise schema expressions as specification statements. We assume that the schema expressions have already been normalised using the techniques presented in [WD96]. Besides, in *Circus*, the Z notations for input (?) and output (!) variables are syntactic sugaring for undashed and dashed variables, respectively. This means that we actually have schemas containing the declaration of dashed (*ddecl′*) and undashed (*udecl*) variables, and, of course, a 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′*.

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

In this case, we are concerned with the Z constructs in *Circus*; the semantics is given by Z normalisation and conversion to a specification statement, both justified by the Z theory and refinement calculus [CW99], and by the use of **R** as a link to embed a data operation into the theory of reactive processes.

This definition concludes the semantics of the *Circus* actions. We are now left with the semantics of *Circus* process, which we present in the sequel.

3.2. *Circus* processes

An explicitly defined process has an encapsulated state, a sequence *PPars* of paragraphs, and a main action A. Its meaning is that of a variable block that declares the state components and whose body is A.

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

The invariant only plays a role when it is explicitly included in an operation schema. In other words, just like in Z, types are maximal and invariants, therefore, only play a role in arguments of correctness. Constructs like *Skip*, *Stop*, and *Chaos*, for example, do not enforce the maintenance of the invariant.

All compound processes are defined in terms of an explicit process specification. For instance, sequence, external and internal choice are defined as follows; we use *op* to stand for any of ; , $\Box$ or $\sqcap$.

$$P\ op\ Q \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{begin state } State \mathrel{\widehat{=}} P.State \wedge Q.State \\ \qquad P.PPar \wedge_{\Xi} Q.State \\ \qquad Q.PPar \wedge_{\Xi} P.State \\ \qquad \bullet\ P.Act\ op\ Q.Act \\ \textbf{end} \end{array}\right)$$

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 $P.PPar \wedge_{\Xi} Q.State$ stands for the conjunction of each schema expression in the paragraphs $P.PPar$

Table 3. Healthiness conditions: *Circus* processes

	Formal representation	Description
C1	$\mathbf{C1}(A) \mathrel{\widehat{=}} A ; Skip$	The value of the variable ref' has no relevance after termination
C2	$\mathbf{C2}(A) \mathrel{\widehat{=}} A \llbracket v \mid \emptyset \rrbracket Skip$	A deadlocked process that refuses some events offered by its environment will still be deadlocked in an environment which offers even fewer events
C3	$\mathbf{C3}(A) \mathrel{\widehat{=}} \mathbf{R}(\neg A_f^f ; true \vdash A_f^t)$	The precondition of a *Circus* process expressed as a reactive design contains no dashed variables

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.

For parallel composition and interleaving the only difference is that we must determine the state partitions of the operators. These are 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 $i : T \odot P$ is that of a parametrised process $i : T \bullet P$. 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.

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

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

All the predicates used to give semantics to *Circus* constructs satisfy a number of healthiness conditions that are important in the proof of laws. They are discussed in the next section.

4. Healthiness conditions

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 [HJ98] and were proved to be reactive. As a direct consequence of this, we have that the following theorem holds; its proof is by induction on the structure of the *Circus* actions.

Theorem 4.1 Every *Circus* action is **R** (**R1**, **R2**, and **R3**) healthy.

In Sect. 3, we used Theorem 3.1 to reason about the behaviour of guarded actions. The following theorem guarantees that *Circus* actions are indeed **CSP1**, **CSP2** and **CSP3** healthy, and therefore, Theorem 3.1 is applicable to them.

Theorem 4.2 Every *Circus* action is **CSP1**, **CSP2**, and **CSP3** healthy.

Part of the proof of this theorem is a direct result from the fact that reactive designs are indeed **CSP1** and **CSP2** [CW06]. The rest of the proof is done by induction on the syntax of the language; for the sake of conciseness, it is omitted here. This proof and the proof of all the new theorems presented in this paper can be found in [Oli05a].

From Theorems 4.1 and 4.2, we already know that every *Circus* action is **R** and **CSP1**-**CSP3** healthy. However, processes that can be defined using the notation of CSP also satisfy other healthiness conditions: 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 definitions in *Circus*: **CSP4** processes satisfy the right unit law ($P ; SKIP = P$) and **CSP5** processes satisfy the unit law of interleaving ($P \mid\mid\mid SKIP = P$) [HJ98]. The healthiness conditions **C1** and **C2** presented in Table 3 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. The sequential composition of the precondition with *true* guarantees that only those actions with no dashed variables in the precondition will be a fixed-point of the function **C3**.

The last theorem regarding healthiness conditions guarantees that every *Circus* operator is **C1**–**C3** healthy.

Theorem 4.3 Every *Circus* action is **C1**, **C2**, and **C3** healthy.

As for the similar theorems for **R** and **CSP**, the proof of this theorem is done by induction on the language.

Based on this new semantics, we proved over 90% of the 146 proposed refinement laws. The mechanical proof of these laws requires the mechanisation of the *Circus* semantics, which is the subject of the next section.

5. Mechanising the *Circus* semantics

Our aim is to provide a mechanisation that can support the development of *Circus* programs using the *Circus* refinement laws. In order to achieve such result, we mechanised the *Circus* semantics in a theorem prover, ProofPower-Z. In [OCW06b], we presented the first step towards our objective, which is summarised in Sect. 5.1: the mechanisation of the UTP in ProofPower-Z.

5.1. The UTP theories

A very important part of that work is the theory of relations. It provides a set-based model for alphabetised relations. In this theory, we define alphabets, relations, and basic programming constructs as we briefly describe in the sequel.

A name is an element of the given set $NAME$. Each relation has an $ALPHABET$, which is defined as $\mathbb{P}\, NAME$. Every alphabet a contains an input alphabet of *undashed* names, and an output alphabet of *dashed* names. Instead of using free-types, which would lead to more complicated proofs in ProofPower-Z, we use the injective ($\rightarrowtail$) function $dash : NAME \rightarrowtail NAME$ to model name decoration. The set of *dashed* names is defined as the range of *dash*. The complement of this set are the *undashed* names; hence, names are either *dashed* or *undashed*, but multiple dashes is allowed. For the sake of conciseness, we omit the definitions of the functions *in_a* and *out_a*, which return the input and the output alphabets of a given alphabet.

An alphabet a in which $n \in a \Leftrightarrow n' \in a$, for all *undashed* names n, is called *homogeneous*. For us, n' is mechanised as $dash(n)$. Similarly, a pair of alphabets $(a1, a2)$ is *composable* if $n \in a2 \Leftrightarrow n' \in a1$, for every *undashed* name n; this notion is used in the definition of relational sequence.

A value is an element of the free-type $VALUE$, which can be an integer, a boolean, a channel, a sequence of values, a set of values, a pair of values, or a special synchronisation value.

$$\begin{aligned} VALUE ::= {} & Int(\mathbb{Z}) \mid Bool(\mathbb{B}) \mid Channel(NAME) \mid Seq(\text{seq } VALUE) \\ \mid {} & Set(\mathbb{F}\, VALUE) \mid Pair(VALUE \times VALUE) \mid Sync \end{aligned}$$

In ProofPower-Z, $Bool(\mathbb{B})$ stands for the Z constructor $Bool\langle\langle\mathbb{B}\rangle\rangle$, which introduces a collection of constants, one for each element of the set $\mathbb{B}$. The ProofPower-Z type $\mathbb{B}$ is the booleans. The type $VALUE$ can be extended without any impact on the proofs because they do not depend on its structure.

Although we are defining an untyped theory, the observational variables have types; for instance, *okay* is a boolean. For this reason, we have defined some restricted sets; for instance, boolean values are in the set $BOOL\_VAL \mathrel{\widehat=} \{Bool(true), Bool(false)\}$.

Three definitions allow us to abstract from the syntax of expressions. The set of relations between values is $RELATION \mathrel{\widehat=} VALUE \leftrightarrow VALUE$. The set of unary functions is $UNARY\_F \mathrel{\widehat=} VALUE \nrightarrow VALUE$; similarly, for binary functions we have the set $BINARY\_F \mathrel{\widehat=} (VALUE \times VALUE) \nrightarrow VALUE$, which defines the set of partial functions from pairs of values to values. An expression can be a value, a name, a relation, or a unary or binary function application.

$$\begin{aligned} EXPRESSION ::= {} & Val(VALUE) \mid Var(NAME) \\ \mid {} & Rel(RELATION \times EXPRESSION \times EXPRESSION) \\ \mid {} & Fun_1(UNARY\_F \times EXPRESSION) \\ \mid {} & Fun_2(BINARY\_F \times EXPRESSION \times EXPRESSION) \end{aligned}$$

The definitions for unary functions, binary functions, and relations only deal with values. For instance, for a given unary function f, the expression $Fun_1(f, e)$ can only be evaluated once e is evaluated to some $VALUE$.

A binding is a partial function from $NAME$ to VAL, and therefore we define the set of all bindings, $BINDING$, as $NAME \pfun VAL$. The type $BINDINGS$ represents sets of bindings, $\mathbb{P}\, BINDING$. Given a binding b and an expression e with free variables in the domain (dom) of b, $Eval(b, e)$ gives the value of e in b (beta-reduction).

A relation is modelled in our work by the type $REL\_PREDICATE$ defined below. A relation is a pair: the first element is its alphabet, and the second is a set of bindings, which gives us all bindings that satisfy the UTP predicate modelled by the relation. The domain of the bindings must be equal to the alphabet.

$$REL\_PREDICATE \mathrel{\widehat=} \{a : ALPHABET;\ bs : BINDINGS \mid (\forall b : bs \bullet \mathrm{dom}(b) = a) \bullet (a, bs)\}$$

This corresponds directly to the definition of alphabetised predicates of the UTP.

In [OCW06b], each predicate construct is defined as an alphabetised relation. One of them is *true*: for a given alphabet a, $True_R\, a$ is as the pair with alphabet a, and with all the bindings with domain a.

$$\begin{array}{|l} True_R : ALPHABET \to REL\_PREDICATE \\ \hline \forall a : ALPHABET \bullet True_R(a) = (a, \{b : BINDING \mid \mathrm{dom}(b) = a\}) \end{array}$$

In our work, we subscript the names of the constructs in order to make it easier to identify to which theory they belong; we use R for the theory of relations.

Nothing satisfies *false*: the second element of $False_R(a)$ is the empty set. This operator is the main motivation for representing relations as pairs. If we had defined relations just as a set of bindings with the same domain a, which would be considered as the alphabet, we would not be able to tell the difference between $False_R(a_1)$ and $False_R(a_2)$, since both sets would be empty. Besides, it is important to notice the difference between $True_R(\emptyset)$ and $False_R(\emptyset)$: the former has a set that contains one empty set of bindings as its second element, and the latter has the empty set as its second element.

All usual predicate combinators are defined. Conjunctions and disjunctions extend the alphabet of each relation to the alphabet of the other. The function $\oplus_R$ is alphabet extension; the values of the new variables are left unconstrained. In the following definition we make use of the Z domain restriction $A \lhd B$: it restricts a relation $B : X \rel Y$ to a set A, which must be a subset of X, ignoring any member of B whose first element is not a member of A.

$$\begin{array}{|l} \_ \oplus_R \_ : REL\_PREDICATE \times ALPHABET \to REL\_PREDICATE \\ \hline \forall u : REL\_PREDICATE;\ a : ALPHABET \\ \quad \bullet\ u \oplus_R a = (u.1 \cup a, \{b : BINDING \mid (u.1 \lhd b) \in u.2 \land \mathrm{dom}(b) = u.1 \cup a\}) \end{array}$$

The conjunction is defined as the union of the alphabets and the intersection of the extended set of bindings of each relation.

$$\begin{array}{|l} \_ \land_R \_ : REL\_PREDICATE \times REL\_PREDICATE \to REL\_PREDICATE \\ \hline \forall u1, u2 : REL\_PREDICATE \bullet u1 \land_R u2 = (u1.1 \cup u2.1, (u1 \oplus_R u2.1).2 \cap (u2 \oplus_R u1.1).2) \end{array}$$

The definition of disjunction is similar, but the union of the extend set of bindings is the result.

An alphabetised equality compares the values of variables and expressions. It is modelled as a function from WF_Equals_R to UTP relations. The type WF_Equals_R is the set of triples (a, n, e) where the name n is a member of the alphabet a and the free variables of the expression e are a subset of a.

$$\begin{array}{|l} =_R : WF\_Equals_R \to REL\_PRED \\ \hline \forall a\_n\_e : WF\_Equals_R \bullet \\ \quad =_R (a\_n\_e) = (a\_n\_e.1, \{b : BINDING \mid \mathrm{dom}(b) = a\_n\_e.1 \land b(a\_n\_e.2) = Eval(b, a\_n\_e.3)\}) \end{array}$$

The alphabet of an equality is simply the alphabet given as argument. In Z, $t.n$ refers to the n-th element of a tuple t; for instance, $a\_n\_e.1$ represents the first element of a_n_e, which corresponds to the alphabet. For a given alphabet a, name n, and expression e, such that $n \in a$ and the free variables of e are in a, the function $=_R (a, n, e)$ returns a relational predicate (a, bs), in which for every binding b in bs, $b(n) = Eval(b, e)$.

The negation $\neg_R$ of a relation r does not change its alphabet. Only those bindings b that do not satisfy r ($b \notin r.2$) are included in the resulting bindings. For the sake of conciseness, we omit the definitions of the remaining relational operations like conditional ($\_ \lhd_R \_ \rhd_R \_$), which can be trivially defined in terms of the previously defined operators. All the definitions and proof scripts can be found in [Oli05a].

In [OCW06b], we describe theories that contains definitions and properties of the observational variables *okay*, *wait*, *tr*, and *ref*. For instance, in the theory *utp-okay*, we define *okay* as an *undashed* name.

$$okay : NAME \vdash okay \in undashed$$

We restrict the type of values which *okay* and *okay*′ can be associated with: they can only be boolean values.

$$\forall b : BINDING \mid okay \in \mathrm{dom}(b) \bullet b(okay) \in BOOL\_VAL$$
$$\wedge \forall b : BINDING \mid dash(okay) \in \mathrm{dom}(b) \bullet b(dash(okay)) \in BOOL\_VAL$$

The same restriction is valid for *wait* and *wait*′. Besides, only sequences of events can be associated with *tr* and *tr*′, and *ref* and *ref*′ can only be associated with sets of events.

Designs are defined in the theory *utp-des*. The set $ALPHABET\_DES$ is the set of all alphabets that contain *okay* and *okay*′. First we define $DES\_PREDICATE$, the set of relations u, such that $u.1$ is a $ALPHABET\_DES$. Designs with precondition p and postcondition q are written $p \vdash q$ and defined as $okay \wedge p \Rightarrow okay' \wedge q$. The expression *okay* is the equality $okay =_a true$, which is mechanised in our work as $=_R (a, okay, Val(Bool(true)))$. A design is defined as follows.

$$\_ \vdash_D \_ : WF\_DES\_PREDICATE\_PAIR \to REL\_PREDICATE$$
$$\forall d : WF\_DES\_PREDICATE\_PAIR \bullet$$
$$d.1 \vdash_D d.2 = (=_R (d.1.1, okay, Val(Bool(true))) \wedge_R d.1) \Rightarrow_R$$
$$(=_R (d.1.1, dash(okay), Val(Bool(true))) \wedge_R d.2)$$

The members of $WF\_DES\_PREDICATE\_PAIR$ are pairs of relations (r_1, r_2) of the type $DES\_PREDICATE$ with the same alphabet. The turnstile is used by both ProofPower-Z and the UTP. The former uses it to give names to theorems, and the later uses it to define designs. In our work, we have kept both of them, but we subscript the UTP design turnstile with a *D*.

In the theory *utp-rea*, we define $REA\_PREDICATE$, the set of relations whose alphabet is a member of $ALPHABET\_OWTR$; this is the set of alphabets that contain *okay*, *tr*, *wait*, *ref*, and their *dashed* counterparts. Finally, a $CSP\_PROCESS$ is defined in the theory *utp-csp* as a $CSP1\_healthy$ and $CSP2\_healthy$ reactive process: the sets containing all the **CSP1** healthy and **CSP2** healthy processes, respectively.

$$CSP\_PROCESS \mathrel{\widehat=} \{p : REA\_PROCESS \mid p \in CSP1\_healthy \wedge p \in CSP2\_healthy\}$$

The definitions of the theories of relations, designs, reactive processes, and CSP, and around five-hundred theorems, is the result of the work in [OCW06b]. This work was the basis for mechanising the *Circus* semantics in [Oli05b]. In what follows, we present the automation of some of the *Circus* constructors.

5.2. The *Circus* theory

Although the constructors of CSP do not contain state variables, the set of processes described by the theory of CSP contains processes that might have state components. The only restriction on the alphabet is that it must contain the observational variables *okay*, *wait*, *tr*, and *ref* and their *dashed* counterparts in the alphabet, but there may be more variables. Therefore, for us, *Circus* actions are modelled by predicates that are in the set $CSP\_PROCESS$; we do not define a new set of predicates.

The definitions of the ProofPower-Z theory of *Circus*, *utp-circus*, follow directly from the semantics presented in this paper. The *Circus* operators that are inherited from CSP have very similar definitions to their CSP counterparts; however, the state components of the *Circus* processes must be taken into account in these new definitions.

The first ProofPower-Z definition that we present is the one for *Stop*. For a given *homogeneous* alphabet *a* that contains the four observational variables and their dashed counterparts ($WF\_hom\_alpha_C$), *Stop* is the reactive design with a *true* precondition, which we mechanise using the relational $True_R$, and with the conjunction $tr' =_a tr \wedge_R wait'$ as its postcondition; the predicate $tr' =_a tr$ is mechanised as $=_R (a, dash(tr), Var(tr))$.

$$Stop : WF\_hom\_alpha_C \to CSP\_PROCESS$$
$$\forall a : WF\_hom\_alpha_C \bullet$$
$$Stop(a) = R(True_R(a) \vdash_D ((=_R (a, dash(tr), Var(tr))) \wedge_R (=_R (a, dash(wait), Val(Bool(true))))))$$

Four auxiliary functions mechanise the substitutions P^b_c; in order to make it more alike the textual notation, we use a prefix notation for them. For instance, $P\,\sigma_f\,\omega_f$ mechanises the predicate P^f_f.

$$\omega_f, \omega_t, \sigma_f, \sigma_t : CSP\_PROCESS \to CSP\_PROCESS$$

$$\begin{aligned}\forall\, c : CSP\_PROCESS \bullet\ & c\,\sigma_f = /_R(c, Val(Bool(false)), dash(okay)) \\ & \land\ c\,\sigma_t = /_R(c, Val(Bool(true)), dash(okay)) \\ & \land\ c\,\omega_f = /_R(c, Val(Bool(false)), wait) \land c\,\omega_t = /_R(c, Val(Bool(true)), wait)\end{aligned}$$

The expression $/_R(p, e, n)$ automates the substitution of a variable n for an expression e in a predicate p.

Using these auxiliary functions, we can define the external choice as follows. The pairs of CSP processes with the same alphabet compose the set $WF\_CSP\_PROCESS\_PAIR$.

$$\_ \,\Box_C\, \_ : WF\_CSP\_PROCESS\_PAIR \to CSP\_PROCESS$$

$$\begin{aligned}\forall\, PQ : WF\_CSP\_PROCESS\_PAIR \bullet\ & PQ.1 \,\Box_C\, PQ.2 = R(((\neg\, {}_R PQ.1\,\sigma_f\,\omega_f) \land_R (\neg\, {}_R PQ.2\,\sigma_f\,\omega_f)) \\ & \qquad \vdash_D \\ & \qquad (((\neg\, {}_R PQ.1\,\sigma_t\,\omega_f) \land_R (\neg\, {}_R PQ.2\,\sigma_t\,\omega_f)) \\ & \qquad\ \lhd_R ((=_R (PQ.1, dash(tr), Var(tr))) \\ & \qquad\qquad \land_R (=_R (PQ.1, dash(wait), Val(Bool(true))))) \rhd_R \\ & \qquad\ ((\neg\, {}_R PQ.1\,\sigma_t\,\omega_f) \lor_R (\neg\, {}_R PQ.2\,\sigma_t\,\omega_f)))\,)\end{aligned}$$

Although long, this definition is a direct mechanisation of the previously presented reactive design representation of external choice.

Besides making it possible to prove the refinement laws, the semantics presented here defines most of the operators as reactive designs. For 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.

6. The library: proof of refinement laws

In this section, we discuss the strategy adopted in our proofs and the structure of our library of laws and lemmas, 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 [Oli05b]. At the end of this section, we discuss some issues that were raised during the application of this strategy in the automated proof of *Circus* refinement laws.

6.1. The proof strategy

The strategy for proving that a program P is equal (or refined) to Q is illustrated graphically in Fig. 4. It involves three stages:

(1) Flatten program P to a single reactive design $\mathbf{R}(pre_P \vdash post_P)$.
(2) Flatten program Q to a single reactive design $\mathbf{R}(pre_Q \vdash post_Q)$.
(3) Use lemmas and theorems from the library and predicate calculus to transform the first reactive design into the second one; in the case of refinement, an inverse implication is the required result.

The stage (3) does not involve transformations on the healthiness condition **R**; rather, it is simply a proof of equivalence (or refinement) between de designs $pre_P \vdash post_P$ and $pre_Q \vdash post_Q$ to which **R** is being applied to. This simplification is supported by the following lemma.

Lemma 6.1 $\mathbf{R}(P) \sqsubseteq \mathbf{R}(Q)$ provided $P \sqsubseteq Q$

The proofs of this lemma and others that follow in this section can be found in Appendix A.

The flattening stage involves definitions and theorems that transform program structures into a single reactive design. For example, if P is a sequence, the lemma below transforms it into a single reactive design.

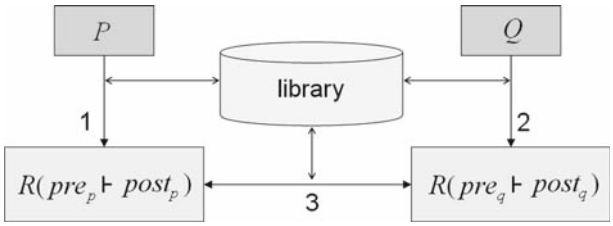

Fig. 4. Using the library to prove refinement laws

Lemma 6.2

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

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 $\mathbf{R}(P_1 \vdash Q_1); \mathbf{R}(P_2 \vdash Q_2)$ 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 or in a state that results from the execution of the second reactive design after the completion of the first one. This lemma applies to reactive designs. This is useful in the context of our strategy because it is based on reactive designs, which are used as the semantics of the vast majority of the *Circus* operators. In the proviso, we require that P_1 does not have free occurrences of dashed variables. This is not too restrictive because, from Theorem 4.3, our reactive designs are **C3**-healthy. For conciseness, we omit here the proof of this lemma; it can be found in [Oli05b].

6.1.1. Sequence zero

By way of illustration, we present one out of over a hundred proofs we presented in [Oli05b]: the sequence zero law ($Stop; A = Stop$). We start the proof of this refinement law (stage (1)) by applying the definition of external choice to the left hand-side of the law.

(1)

$$\begin{array}{ll} Stop;A & \\ = \mathbf{R}(true \vdash tr' = tr \wedge wait'); A & [Stop]\end{array}$$

Theorem 3.1 guarantees that every **CSP** process can be written as a reactive design; the application of this theorem transforms A into another reactive design.

$$= \mathbf{R}(true \vdash tr' = tr \wedge wait'); \mathbf{R}(\neg\, A_f^f \vdash A_f^t) \qquad [\text{Theorem 3.1}]$$

Now we have two reactive designs and, in order to complete stage (1), we must flatten these into a single reactive design. The Lemma 6.2 can be used to give us the desired result.

$$= \mathbf{R}\left(\begin{array}{l} true \wedge (\neg\, ((okay' \wedge \neg\, wait' \wedge tr' = tr \wedge wait'); \neg\, (\neg\, A_f^f))) \\ \vdash \\ (wait' \wedge tr' = tr \wedge wait') \vee ((okay' \wedge \neg\, wait' \wedge tr' = tr \wedge wait'); A_f^t)\end{array}\right) \qquad [\text{Lemma 6.2}]$$

[LHS]

The definition of *Stop* establishes the stage (2) of the proof strategy.

(2)

$$\begin{array}{ll} Stop & \\ = \mathbf{R}(true \vdash tr' = tr \wedge wait') & [Stop]\end{array}$$

[RHS]

We start the stage (3) of our proof strategy by applying trivial predicate calculus to the pre and the postcondition of the design.

(3)
LHS
$= \mathbf{R}(\neg\ (false;\ A_f^f) \vdash (tr' = tr \wedge wait') \vee (false;\ A_f^t))$ [Predicate Calculus]

In the UTP, sequence is defined as an existential quantification on the intermediary state. For this reason, it is trivial that the following lemma is valid.

Lemma 6.3 $false;\ P = P;\ false = false$

We use this lemma to simplify both the pre and the postcondition.

$= \mathbf{R}(\neg\ false \vdash (tr' = tr \wedge wait') \vee false)$ [Lemma 6.3]

Finally, by using simple predicate calculus we can conclude this proof.

=RHS [Predicate calculus]

□

Another example of a lemma on reactive designs that can be used to flatten programs into a single reactive design is presented below; it flattens the non-deterministic choice between two reactive designs.

Lemma 6.4 $\mathbf{R}(P_1 \vdash Q_1) \sqcap \mathbf{R}(P_2 \vdash Q_2) = \mathbf{R}((P_1 \wedge P_2) \vdash (Q_1 \vee Q_2))$

The internal choice between two reactive designs diverges if either P_1 or P_2 is violated; otherwise, the final state of the internal choice satisfies either Q_1 or Q_2.

6.1.2. Internal choice zero

Lemma 6.4 can be used in the stage (1) of the simple proof of the internal choice zero law ($A \sqcap Chaos$) presented below. This stage of this proof is very similar to the one of the previous proof, but uses Lemma 6.4.

(1)
$A \sqcap Chaos$
$= A \sqcap \mathbf{R}(false \vdash true)$ [*Chaos*]
$= \mathbf{R}(\neg\ A_f^f \vdash A_f^t) \sqcap \mathbf{R}(false \vdash true)$ [Theorem 3.1]
$= \mathbf{R}((\neg\ A_f^f \wedge false) \vdash (A_f^t \vee true))$ [Lemma 6.4]
[LHS]

The stage (2) is also very similar to the one of the previous proof, but uses the definition of *Chaos* instead.

(2)
Chaos
$= \mathbf{R}(false \vdash true)$ [*Chaos*]
[RHS]

The last stage of this proof is extremely simple and can be achieved with the application of predicate calculus.

LHS
=RHS [Predicate calculus]

□

Two other lemmas give the conditions under which a reactive design diverges and the conditions under which a reactive design does not diverge when it is not waiting for its predecessor to finish. They are useful in the transformation stage (3) of proofs that involve operators (after using Theorem 3.1) and healthiness conditions like **C3**, which refer explicitly to these conditions. A reactive design diverges if started in a divergent state or in a state that does not satisfy its precondition. Provided its precondition is satisfied and hence, it does not diverge, a reactive design establishes its postcondition.

Lemma 6.5 $(\mathbf{R}(P \vdash Q))^f_f = \mathbf{R1}(\neg\ (okay \wedge \mathbf{R2}(P)))$, provided *wait* and $okay'$ are not free in P and Q.

Lemma 6.6 $(\mathbf{R}(P \vdash Q))^t_f = \mathbf{CSP1}(\mathbf{R1}(\mathbf{R2}(P \Rightarrow Q)))$, provided *wait* and $okay'$ are not free in P and Q.

These lemmas can be proved by applying the definitions of the healthiness conditions.

6.1.3. External choice unit

In what follows, we discuss the proof of the external choice unit law ($Stop \,\Box\, A = A$), which besides illustrating the use of our library, shows the reasons for defining that *Stop* leaves the state unconstrained. 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 6.5 and 6.6 discussed above. Lemmas 6.7 and 6.8 give the conditions on which *Stop* diverges and the effects of *Stop* when it does not diverge, respectively.

Lemma 6.7 $Stop^f_f = \neg\ okay \wedge tr \leqslant tr'$

Lemma 6.8 $Stop^t_f = \mathbf{CSP1}(tr' = tr \wedge wait')$

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, *Stop* does not change the trace and waits indefinitely; **CSP1** guarantees the expected behaviour on divergence of the predecessor.

We start the proof of the external choice unit law (stage (1)) by applying the definition of external choice to the left hand-side of the law.

(1)

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

The application of Theorem 3.1 establishes the stage (2) of the proof strategy.

(2)

$$
\begin{array}{ll}
& A \\
= & \mathbf{R}(\neg\ A^f_f \vdash A^t_f) \qquad \text{[Theorem 3.1]} \\
& \text{[RHS]}
\end{array}
$$

We start the stage (3) of our proof strategy by transforming the precondition of the reactive design originated by step (1). Using Lemmas 6.7 and 6.8, we transform $Stop^f_f$ and $Stop^t_f$, respectively.

(3)

LHS

$$
= \mathbf{R}\left(\begin{array}{l}(\neg\ (\neg\ okay \wedge tr \leqslant tr') \wedge \neg\ A^f_f) \vdash \left(\begin{array}{l}(\mathbf{CSP1}(tr' = tr \wedge wait') \wedge A^t_f) \\ \lhd tr' = tr \wedge wait' \rhd \\ (\mathbf{CSP1}(tr' = tr \wedge wait') \vee A^t_f)\end{array}\right)\end{array}\right) \qquad \text{[Lemmas 6.7 and 6.8]}
$$

The application of predicate calculus gives us the following result.

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

At this point we have exactly the definition of **CSP1** applied to A^f_f in the precondition.

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

Another lemma from our library can be used at this point. It states that the substitutions of $okay'$ and $wait$ for any value commute with **CSP1**.

Lemma 6.9 $\mathbf{CSP1}(A_c^b) = (\mathbf{CSP1}(A))_c^b$

The predicate A_c^b corresponds to the substitution of $okay'$ and $wait$ in A; however, the disjunct $\neg\ okay \land tr \leqslant tr'$ that is used in the definition of **CSP1** does not mention either of these variables. Therefore, we may commute the substitution with **CSP1**; the use of this lemma leaves us with the following reactive design.

$$= \mathbf{R}\left(\neg\ (\mathbf{CSP1}(A))_f^f \vdash \left(\begin{array}{l}(\mathbf{CSP1}(tr' = tr \land wait') \land A_f^t)\\ \lhd tr' = tr \land wait' \rhd\\ (\mathbf{CSP1}(tr' = tr \land wait') \lor A_f^t)\end{array}\right)\right) \qquad \text{[Lemma 6.9]}$$

Since every *Circus* action is **CSP1**-healthy (Theorem 4.2), the application of **CSP1** to A can be removed.

$$= \mathbf{R}\left(\neg\ A_f^f \vdash \left(\begin{array}{l}(\mathbf{CSP1}(tr' = tr \land wait') \land A_f^t)\\ \lhd tr' = tr \land wait' \rhd\\ (\mathbf{CSP1}(tr' = tr \land wait') \lor A_f^t)\end{array}\right)\right) \qquad \text{[Theorem 4.2]}$$

At this point, we have already shown that the preconditions of the left-hand side and the right-hand side are the same. We now turn our attention to the postcondition.

As previously explained, we may include (or remove) *okay* in (or from) the postcondition of any design. Furthermore, using the definition of conditional and simple predicate calculus, we may distribute *okay* to both branches of the conditional as follows.

$$= \mathbf{R}\left(\neg\ A_f^f \vdash \left(\begin{array}{l}okay \land \mathbf{CSP1}(tr' = tr \land wait') \land A_f^t\\ \lhd tr' = tr \land wait' \rhd\\ okay \land \mathbf{CSP1}(tr' = tr \land wait') \lor A_f^t\end{array}\right)\right) \qquad \text{[Designs and predicate calculus]}$$

The use of Lemma 3.4 and the previously explained freedom to remove *okay* from the postcondition leaves us with the following reactive design.

$$= \mathbf{R}(\neg\ A_f^f \vdash (tr' = tr \land wait' \land A_f^t \lhd tr' = tr \land wait' \rhd (tr' = tr \land wait') \lor A_f^t)) \qquad \text{[Lemma 3.4]}$$

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_f^t; this can be done because the predicate $tr' = tr \land wait'$ is *false*: it is in the else-part of a conditional on the same predicate. The conditional comes directly from our definition of external choice, in which, as explained in Sect. 3, 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.

$$= \mathbf{R}(\neg\ A_f^f \vdash ((tr' = tr \land wait' \land A_f^t) \lor A_f^t)) \qquad \text{[Conditional]}$$

Using absorption we can remove the predicate $tr' = tr \land wait' \land A_f^t$. We are left with the reactive design originated by step (2); this concludes our proof.

$$= \text{RHS} \qquad \text{[Predicate calculus]}$$

□

6.1.4. Library summary

In total, our library contains 122 theorems and more than 200 lemmas, which are structured into three groups. With the exception of the lemmas used in stages (1) and (2), in our proof strategy, they are used in the following order.

- **Lemmas on *Circus* operators:** these are the lemmas that involve some particular structure resulting from each of the *Circus* operators. They are first used in the stage (3) of the proof strategy to remove references to *Circus* operators in the pre and the postcondition. In this paper, we have presented (and used) two lemmas (6.7 and 6.8) on *Stop*; both of them transform the substitution of observational variables on *Stop* into much simpler predicates.

Table 4. Theories mechanisation

	Number of theorems	Lines of proof script
utp-z-library	23	1,293
utp-rel	360	22,794
utp-okay	30	1,197
utp-des	57	4,263
utp-wtr	41	1,829
utp-rea	129	7,046
utp-csp	46	5,885
utp-circus	15	3,195
Total	701	47,502

- **Lemmas on the healthiness conditions:** they are directly related to the healthiness conditions discussed in Sect. 3. The vast majority of them, like Lemmas 3.4 and 6.9, are used during the stage (3) of our proof strategy to remove references to healthiness conditions in the pre and the postcondition.
- **Lemmas on UTP theories:** these are related to particular UTP theories and are subdivided into relations, designs, reactive designs, and CSP. Most lemmas of this category, like Lemmas 6.3, 6.5, and 6.6 are also used during the stage (3) of the proof strategy; however, some lemmas on reactive designs, like Lemmas 6.2 and 6.4, are used to flatten the programs into a single reactive design in the stages (1) and (2).

As expected, there is a dependence between the lemmas of these categories. Most lemmas on *Circus* operators use lemmas on healthiness conditions in their proofs, and most of these lemmas use lemmas on the UTP theories in their proofs. The mechanical proofs of these lemmas is currently under development.

6.2. Proof automation

As expected, the proof of the *Circus* refinement laws requires further work on the UTP theories discussed in Sect. 5. In Table 4 we summarise the amount of effort required so far. The new theorems exploit properties of the UTP theories that are the theoretical basis of *Circus*, which were not a major concern in [Oli05b, Oli05a]. Basically, they involve properties on the healthiness conditions (specially **R**, **R1**, **R2**, **R3**, and **CSP1**). This includes properties of expressions like $tr \leqslant tr'$, that are used in the definitions of the healthiness conditions and the *Circus* operators, properties of substitution, conditional, and equalities, and properties of the substitution functions ω_t, ω_f, σ_t, σ_f like distribution over predicate constructors and commutativity with healthiness conditions (that is Lemma 6.9).

An important aspect that has a direct impact on the effort spent in these proofs and on the size of the proof scripts is the typing requirements to use previously proved theorems. Although we have used strategies to reduce these requirements (see [OCW06b] for details), there are some requirements that are still needed and are very time consuming. For instance, suppose we have the goal presented below. In our work, the type *REL_PREDICATE* represents the alphabetised predicates from the UTP. In what follows, we use $P_1, \ldots, P_n ? \vdash G$, to represent a list of premises $P_1, \ldots, P_n$ and the goal G of the proof.

$$
\begin{array}{l}
P \in REL\_PREDICATE \land Q \in REL\_PREDICATE \\
R \in REL\_PREDICATE \land S \in REL\_PREDICATE \\
(P \lor Q) \land (R \lor S) \\
? \vdash \\
((P \lor Q) \land R) \lor ((P \lor Q) \land S)
\end{array}
$$

It is clear that we may apply the following distribution theorem to conclude the proof.

$$\forall X, Y, Z : REL\_PREDICATE \bullet X \land (Y \land Z) = (X \land Y) \lor (X \land Z)$$

However, in order to use this lemma in the proof, we need to instantiate it with $X = P \lor Q$, $Y = R$, and $Z = S$; hence, we need to assert that $P \lor Q$ is a member of *REL_PREDICATE*. In this case, using forward chaining and a theorem that is already defined in our theory of relations we may include this information. This task of including inductive typing information in the premisses, however, becomes very time consuming when we have more complex predicates. Furthermore, this task cascades to all the theories. A solution for this problem is the creation of a ProofPower-Z tactic that makes an inductive analysis on the structure of the goal and uses previously defined

theorems to include the typing information needed in the assumption of the goal. This tactic can be used right at the beginning of any proof, reducing considerably the effort needed; we intend to implement it in the near future.

The mechanical proofs pointed out that some information was missing in the UTP theories presented in [OCW06b]. First, in that work, we declared that the observational variables were members of the type $NAME$. We did not, however, declare that they were different names. This was left implicit in [OCW06a], but needed to be explicitly stated in order to guarantee, for instance, that the substitution of *wait* for *true* on *okay* is innocuous ($okay_f = okay$). A proviso of these lemmas was also left implicit in [OCW06a]: the observational variables *wait* and $okay'$ are not free in neither P nor Q. This proviso guarantees that, for any boolean values b and c, $P^b_c = P$ and $Q^b_c = Q$.

Another point is that in our theory of relations, the substitution $P[e/n]$ is defined only if the free-variables of the expression e and the name n are in the alphabet of the predicate P, which are the names that characterise a range of external observations of P. Intuitively, it makes no sense to replace a name by anything in a predicate, if that name is not of any concern for the predicate. Because of this restriction on substitutions, extra provisos relating the observational variables and the alphabets of the predicates are often needed. Another consequence of this restriction was the need to change the alphabet of $tr \leqslant tr'$ in the definition of **R1**, which was previously defined as $\{tr, tr'\}$. For example, we use the commutativity theorem $(\mathbf{R1}(P))^t = \mathbf{R1}(P^t)$ and, in order to prove it, we expand the definition of **R1** and distribute the substitution over the conjunction. However, as previously explained, the predicate $(tr \leqslant tr')[true/okay']$ is only well defined if $okay'$ is in the alphabet of $tr \leqslant tr'$. For this reason, in the definition of **R1**, we changed the alphabet of $tr \leqslant tr'$ to be an alphabet that contains all the four observational variables (and their dashed counterparts) needed to describe reactive processes.

So far, the proof strategy presented in Sect. 6 has proved to be extremely useful to modularise proofs and, as a direct consequence, to reduce effort in the mechanical proofs.

7. Conclusions

This paper presents an example of the application of the UTP to a practical language of some complexity. We have taken into consideration issues related to the design of a refinement, to program development, and to practical use through the provision of tools. Furthermore, it presents the CSP part of *Circus* as a collection of explicit reactive designs that clarify the link between the theories of designs and reactive processes, a library of lemmas, and a theorem prover.

The *Circus* semantics presented in [WC02] did not allow us to prove meta-theorems in the *Circus* theory and, as a direct consequence, refinement laws. For this reason, a new denotational semantics was presented in this paper; it is a final reference for the *Circus* denotational semantics.

We now have proved over 92% of the 146 refinement laws, and we are working to complete all the proofs. These proofs are of soundness with respect to the semantics presented in this paper. If we were to find any invalid law, then either we would propose an alternative law or, if the invalid law is particularly attractive, we would consider changing the semantics to accommodate. We understand that the latter would involve considerable regression in our work. Nevertheless, we have improved our confidence on this semantics in three ways: by studying its relationship to the UTP CSP theory as in Sect. 3, by proving the refinement laws, and by proving the soundness of the *Circus* operational semantics published in [WCF05].

Although based on the definitions from [HJ98, WC02] this new semantics follows the style of [CW06], where we express the semantics of the vast majority of the *Circus* constructs as reactive designs. The structural uniformity of the semantics given to the *Circus* operators is reflected in the proofs of the refinement laws, which use the strategy discussed in this paper. For instance, Lemma 6.1 that is analogous to a factorisation result, can only be used because our definitions are reactive designs. Our work shows that we can use a pre-postcondition style to specify processes and gives a foundation to reason about them in a familiar assertional style. The uniformity also facilitates mechanisation. Together, the library of lemmas presented in this paper and in [CW06], and the proof strategy fosters the reuse of our results in the mechanisation of other languages that have a UTP semantics like TCOZ [MD98, QDC03].

The semantic model for *Circus* processes presented in [WC02] was a Z specification, and every *Circus* operator was defined by a Z schema that included the state definition, and therefore, the invariant as well. For this reason, the state invariant was maintained by all constructs, including *Skip*, *Stop*, and *Chaos*, for example. As already mentioned, this is no longer the case. This ensures, for instance, that there are really no guarantees about the behaviour of *Chaos*.

The definitions of most control constructs are also affected by the presence of data. For example, we have addressed how data operations affect choice: 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 unconstrained. An alternative would be to allow state changes to resolve external choices, 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.

We have addressed sharing concerns in the definitions of the several forms of parallelism: state partitions in the parallel composition and interleaving remove the problems intrinsic to shared variables and were suggested in [CSW03]. These partitions also have a direct impact on the semantics of the parallel composition and interleaving of processes. In [WC02], 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 also satisfy three further healthiness conditions: the first two, **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 designs do not contain any dashed variables in the precondition.

The semantics presented in this paper has been mechanised in ProofPower-Z [Oli05b]; this work was based on a mechanisation of the UTP theories [OCW06b]. Based on this result, we are currently developing the mechanical proof of *Circus* refinement laws. This work has already shown us the need of tactics to reduce considerably the amount of effort needed; furthermore, it has shown subtleties on alphabets and free-variables that were left implicit during the hand-proofs, but needed to be made explicit.

Circus, however, is not the first specification language of concurrent systems that has its semantics mechanised in a theorem-prover. In [Cam90b], the CSP trace model is mechanised in HOL. This work was later extended in [Cam90a], where the failures-divergence model is considered. In both cases, besides mechanising the semantics, the author also proved some standard CSP laws based on his mechanisation. A semantic embedding of CSP's trace semantics in PVS was presented in [DS97] and used to verify the correctness of a verification protocol. This embedding is a little more general than Camilleri's one because they use parametric types for events instead of considering events as atomic symbols represented by strings. Their CSP variant also differs: [DS97] mechanises Roscoe's CSP [Ros98] whereas [Cam90b] mechanises Hoare's CSP [Hoa85]. In order to be able to mechanise the semantics of *Circus*, we need to use the mechanisation of a semantic model that is able to combine the notions of refinement for CSP and for imperative programs. For this reason, differently from [Cam90a, DS97], we mechanise *Circus* semantics based on our previous mechanisation of the UTP [OCW06b]. In [CW06], we relate our model to Roscoe's standard model.

Nuka and Woodcock formalised the alphabetised relational calculus in Z/EVES [NW04]. They did not restrict the set of bindings in the same way as we do, but the restriction on the domain of the bindings is satisfied by all the constructors. By including the restriction on the set of bindings, we make this information available in all the proofs, and not only in those including some particular operators. Moreover, we extend [NW04] by including many other operations, such as sequencing, assignment, refinement, and recursion. The hierarchical mechanisation of the theories of designs, reactive processes, CSP, and *Circus* is also a contribution of our work that provides a powerful tool for further investigation.

In [NW06], the authors present the same mechanisation that was presented in [NW04] but, this time, in ProofPower-Z. They also extend [NW04] by mechanising a specification language that includes, among other operators, skip, abort, miracle, Hoare triples, assertions, coercions, weakest preconditions, and iterations. However, their syntax is defined using Z free types; any extension to the language would require proving most of the laws again making it harder to extend their specification language.

Ultimately, we intend to mechanise the proofs of all 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 [OCW05].

Acknowledgments

We thank QinetiQ and the Royal Society for their financial support. The work of Marcel Oliveira is supported by CNPq: grant 551210/2005-2. Philip Clayton, Rob Arthan, Roger Bishop Jones, Mark Adams, and Will Harwood

provided valuable advice for our work. The reviews of the anonymous referees have also contributed to the final version of this paper.

A. Proof of lemmas

Lemma 3.1 $do_{\mathcal{A}} = \mathbf{R}(do)$.

Proof.

$$
\begin{array}{ll}
do_{\mathcal{A}}(a) & \text{[Definition of } do_{\mathcal{A}}\text{]} \\
= \Phi(tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle) & \text{[Definition of } \Phi\text{]} \\
= (\mathbf{R} \circ and_B)\,(tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle) & \text{[Definition of } and_B\text{]} \\
= \mathbf{R}(B \wedge (tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle)) & \text{[Definition of } B\text{]} \\
= \mathbf{R}(((tr' = tr \wedge wait') \vee tr < tr') \wedge (tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle)) & \text{[Distribution]} \\
= \mathbf{R}\left(\begin{array}{l}(tr' = tr \wedge wait' \wedge (tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle)) \\ \vee\, (tr < tr' \wedge (tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle))\end{array}\right) & \text{[Conditional]} \\
= \mathbf{R}\left(\begin{array}{l}(tr' = tr \wedge wait' \wedge a \notin ref') \\ \vee\, (tr < tr' \wedge (tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle))\end{array}\right) & \text{[Distribution]} \\
= \mathbf{R}\left(\begin{array}{l}(tr' = tr \wedge wait' \wedge a \notin ref') \\ \vee\, (tr < tr' \wedge tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr < tr' \wedge tr' = tr \frown \langle a \rangle)\end{array}\right) & \text{[Predicate Calculus]} \\
= \mathbf{R}((tr' = tr \wedge wait' \wedge a \notin ref') \vee (false \lhd wait' \rhd tr < tr' \wedge tr' = tr \frown \langle a \rangle)) & \text{[Predicate Calculus]} \\
= \mathbf{R}((tr' = tr \wedge wait' \wedge a \notin ref') \vee (\neg\, wait' \wedge tr' = tr \frown \langle a \rangle)) & \text{[Conditional]} \\
= \mathbf{R}(tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle) & \text{[Definition of } do\text{]} \\
= \mathbf{R}(do(a)) &
\end{array}
$$

Lemma 3.2 $(c \rightarrow SKIP)^{f}_{f} = \neg\, okay \wedge tr \leqslant tr'$.

Proof.

$$
\begin{array}{ll}
(c \rightarrow SKIP)^{f}_{f} & \\
= (\mathbf{CSP1}(okay' \wedge do_{\mathcal{A}}(a)))^{f}_{f} & \text{[Definition of Prefix]} \\
= \mathbf{CSP1}((okay' \wedge do_{\mathcal{A}}(a))^{f}_{f}) & \text{[Lemma 6.9]} \\
= \mathbf{CSP1}((false \wedge do_{\mathcal{A}}(a))_{f}) & \text{[Substitution]} \\
= \mathbf{CSP1}(false) & \text{[Predicate calculus]} \\
= (\neg\, okay \wedge tr \leqslant tr') \vee false & \textbf{[CSP1]} \\
= \neg\, okay \wedge tr \leqslant tr' & \text{[Predicate calculus]}
\end{array}
$$

Lemma 3.3 $(c \rightarrow SKIP)^{t}_{f} = \mathbf{CSP1}(do\,(c, Sync))$.

Proof.

$$
\begin{array}{ll}
(c \rightarrow SKIP)^{f}_{f} & \\
= (\mathbf{CSP1}(okay' \wedge do_{\mathcal{A}}(a)))^{f}_{f} & \text{[Definition of Prefix]} \\
= \mathbf{CSP1}((okay' \wedge do_{\mathcal{A}}(a))^{f}_{f}) & \text{[Lemma 6.9]} \\
= \mathbf{CSP1}((true \wedge do_{\mathcal{A}}(a))_{f}) & \text{[Substitution]} \\
= \mathbf{CSP1}((do_{\mathcal{A}}(a))_{f}) & \text{[Predicate calculus]} \\
= \mathbf{CSP1}((\mathbf{R}(do(a)))_{f}) & \text{[Lemma 3.1]} \\
= \mathbf{CSP1}((\mathbf{R3}(\mathbf{R2}(\mathbf{R1}(do(a)))))_{f}) & \textbf{[R]} \\
= \mathbf{CSP1}((\Pi_{rea} \lhd wait \rhd \mathbf{R2}(\mathbf{R1}(do(a))))_{f}) & \textbf{[R3]}
\end{array}
$$

$= \mathbf{CSP1}((\mathbf{R2}(\mathbf{R1}(do(a))))_f)$ [Substitution and conditional]
$= \mathbf{CSP1}(\mathbf{R2}(\mathbf{R1}(do(a))))$ [**R1**, **R2**, and *do* do not mention *wait*]
$= \mathbf{CSP1}(\mathbf{R2}(do(a)))$ [Lemma A.1]
$= \mathbf{CSP1}(do(a))$ [Lemma A.2]

Lemma A.1 $\mathbf{R1}(do(a)) = do(a)$.

Proof.

$\mathbf{R1}(do(a))$
$= (tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle) \wedge tr \leqslant tr'$ [**R1** and *do*]
$= tr' = tr \wedge a \notin ref' \wedge tr \leqslant tr' \lhd wait' \rhd tr' = tr \frown \langle a \rangle \wedge tr \leqslant tr'$ [Conditional and predicate calculus]
$= tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle$ [Sequences and predicate calculus]
$= do(a)$ [*do*]

Lemma A.2 $\mathbf{R2}(do(a)) = do(a)$.

Proof.

$\mathbf{R2}(do(a))$
$= (tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle)[\langle\rangle, tr' - tr/tr, tr']$ [**R2** and *do*]
$= tr' - tr = \langle\rangle \wedge a \notin ref' \lhd wait' \rhd tr' - tr = \langle\rangle \frown \langle a \rangle$ [Substitution]
$= tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle$ [Sequences]
$= do(a)$ [*do*]

Lemma 3.4 $okay \wedge \mathbf{CSP1}(P) = okay \wedge P$.

Proof.

$okay \wedge \mathbf{CSP1}(P)$
$= okay \wedge ((\neg\, okay \wedge tr \leqslant tr') \vee P)$ [**CSP1**]
$= (okay \wedge (\neg\, okay \wedge tr \leqslant tr')) \vee (okay \wedge P)$ [Predicate calculus]
$= false \vee (okay \wedge P)$ [Predicate calculus]
$= okay \wedge P$ [Predicate calculus]

Lemma 6.1 $\mathbf{R}(P) \sqsubseteq_{\mathcal{A}} \mathbf{R}(Q)$, provided $P \sqsubseteq_{\mathcal{A}} Q$.

Proof.

$\mathbf{R}(Q)$
$= \mathbf{R3}(\mathbf{R2}(\mathbf{R1}(Q)))$ [**R**]
$= \mathit{II}_{rea} \lhd wait \rhd Q[\langle\rangle, tr' - tr/tr, tr'] \wedge tr \leqslant tr'$ [**R3**, **R2**, and **R1**]
$\Rightarrow \mathit{II}_{rea} \lhd wait \rhd P[\langle\rangle, tr' - tr/tr, tr'] \wedge tr \leqslant tr'$ [Assumption and Predicate calculus]
$= \mathbf{R3}(\mathbf{R2}(\mathbf{R1}(P)))$ [**R3**, **R2**, and **R1**]
$= \mathbf{R}(P)$ [**R**]

Lemma 6.4 $\mathbf{R}(P_1 \vdash Q_1) \sqcap \mathbf{R}(P_2 \vdash Q_2) = \mathbf{R}((P_1 \wedge P_2) \vdash (Q_1 \vee Q_2))$.

Proof.

$\mathbf{R}(P_1 \vdash Q_1) \sqcap \mathbf{R}(P_2 \vdash Q_2)$
$= \mathbf{R}(P_1 \vdash Q_1) \vee \mathbf{R}(P_2 \vdash Q_2)$ [$\sqcap$]
$= (\mathbf{R2}(\mathbf{R3}(P_1 \vdash Q_1)) \wedge tr \leqslant tr') \vee (\mathbf{R2}(\mathbf{R3}(P_2 \vdash Q_2)) \wedge tr \leqslant tr')$ [**R** and **R1**]
$= \mathbf{R1}(\mathbf{R2}(\mathbf{R3}((P_1 \vdash Q_1))) \vee \mathbf{R2}(\mathbf{R3}((P_2 \vdash Q_2))))$ [Predicate calculus and **R1**]
$= \mathbf{R1}(\mathbf{R3}((P_1 \vdash Q_1))[\langle\rangle, tr' - tr/tr, tr'] \vee \mathbf{R3}((P_2 \vdash Q_2))[\langle\rangle, tr' - tr/tr, tr'])$ [**R2**]
$= \mathbf{R1}(\mathbf{R2}(\mathbf{R3}((P_1 \vdash Q_1)) \vee \mathbf{R3}((P_2 \vdash Q_2))))$ [Predicate calculus and **R2**]

$= \mathbf{R1}(\mathbf{R2}((\mathit{II}_{rea} \lhd \mathit{wait} \rhd (P_1 \vdash Q_1)) \lor (\mathit{II}_{rea} \lhd \mathit{wait} \rhd (P_2 \vdash Q_2))))$ [**R3**]
$= \mathbf{R1}(\mathbf{R2}((\mathit{II}_{rea} \lhd \mathit{wait} \rhd ((P_1 \vdash Q_1) \lor (P_2 \vdash Q_2)))))$ [Predicate calculus]
$= \mathbf{R}((P_1 \vdash Q_1) \lor (P_2 \vdash Q_2))$ [**R3** and **R**]
$= \mathbf{R}((P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2))$ [$\sqcap$]
$= \mathbf{R}((P_1 \land P_2) \vdash (Q_1 \lor Q_2))$ [UTP - Theorem 3.1.4 (1)]

Lemma 6.5 $(\mathbf{R}(P \vdash Q))^f_f = \mathbf{R1}(\neg\, \mathit{okay} \land \mathbf{R2}(P))$.

Proof.

$(\mathbf{R}(P \vdash Q))^f_f$
$= (\mathbf{R3}(\mathbf{R2}(\mathbf{R1}(P \vdash Q))))^f_f$ [**R**]
$= (\mathit{II}_{rea} \lhd \mathit{wait} \rhd (P \vdash Q)[\langle\rangle, tr' - tr/tr, tr'] \land tr \leqslant tr')^f_f$ [**R3**, **R2**, and **R1**]
$= ((P \vdash Q)[\langle\rangle, tr' - tr/tr, tr'] \land tr \leqslant tr')^f$ [Substitution and conditional]
$= (P[\langle\rangle, tr' - tr/tr, tr'] \vdash Q[\langle\rangle, tr' - tr/tr, tr'])^f \land tr \leqslant tr'$ [Substitution]
$= (\neg\, (\mathit{okay} \land P[\langle\rangle, tr' - tr/tr, tr'])) \land tr \leqslant tr'$ [Design, Substitution and predicate calculus]
$= \mathbf{R1}(\neg\, (\mathit{okay} \land \mathbf{R2}(P)))$ [**R2** and **R1**]

Lemma 6.6 $(\mathbf{R}(P \vdash Q))^t_f = \mathbf{CSP1}(\mathbf{R1}(\mathbf{R2}(P \Rightarrow Q)))$.

Proof.

$(\mathbf{R}(P \vdash Q))^t_f$
$= (\mathbf{R3}(\mathbf{R2}(\mathbf{R1}(P \vdash Q))))^t_f$ [**R**]
$= (\mathit{II}_{rea} \lhd \mathit{wait} \rhd (P \vdash Q)[\langle\rangle, tr' - tr/tr, tr'] \land tr \leqslant tr')^t_f$ [**R3**, **R2**, and **R1**]
$= ((P \vdash Q)[\langle\rangle, tr' - tr/tr, tr'] \land tr \leqslant tr')^t$ [Substitution and conditional]
$= (P \vdash Q)^t[\langle\rangle, tr' - tr/tr, tr'] \land tr \leqslant tr'$ [Substitution]
$= (\mathit{okay} \land P \Rightarrow Q)[\langle\rangle, tr' - tr/tr, tr'] \land tr \leqslant tr'$ [Designs and predicate calculus]
$= (\neg\, \mathit{okay} \lor \neg\, P \lor Q)[\langle\rangle, tr' - tr/tr, tr'] \land tr \leqslant tr'$ [Predicate calculus]
$= (\neg\, \mathit{okay} \land tr \leqslant tr') \lor ((P \Rightarrow Q)[\langle\rangle, tr' - tr/tr, tr'] \land tr \leqslant tr')$ [Predicate calculus]
$= \mathbf{CSP1}(\mathbf{R1}(\mathbf{R2}(P \Rightarrow Q)))$ [**CSP1**,**R1**,**R2**]

Lemma 6.7 $\mathit{Stop}^f_f = \neg\, \mathit{okay} \land tr \leqslant tr'$.

Proof.

Stop^f_f
$= (\mathbf{R}(\mathit{true} \vdash tr' = tr \land \mathit{wait}'))^f_f$ [*Stop*]
$= \mathbf{R1}(\neg\, (\mathit{okay} \land \mathbf{R2}(\mathit{true})))$ [Lemma 6.5]
$= \neg\, \mathit{okay} \land tr \leqslant tr'$ [**R2**, **R1**, Predicate calculus]

Lemma 6.8 $Stop^t_f = \mathbf{CSP1}(tr' = tr \wedge wait')$.

Proof.

$$
\begin{array}{ll}
Stop^t_f = (\mathbf{R}(true \vdash tr' = tr \wedge wait'))^t_f & [Stop] \\
= \mathbf{CSP1}(\mathbf{R1}(\mathbf{R2}(true \Rightarrow tr' = tr \wedge wait'))) & [\text{Lemma 6.6}] \\
= \mathbf{CSP1}(tr' = tr \wedge wait') & [\mathbf{R1}, \mathbf{R2}, \text{Predicate calculus}]
\end{array}
$$

B. Summary of the *Circus* semantics

Circus **Actions**	**Semantics**
$Stop$	$\mathbf{R}(true \vdash tr' = tr \wedge wait')$
$Skip$	$\mathbf{R}(true \vdash tr' = tr \wedge \neg\, wait' \wedge v' = v)$
$Chaos$	$\mathbf{R}(false \vdash true)$
$A_1 ; A_2$	$\exists x_0 \bullet A_1[x_0/x'] \wedge A_2[x_0/x]$
$g \;\&\; A$	$\mathbf{R}((g \Rightarrow \neg\, A^f_f) \vdash ((g \wedge A^t_f) \vee (\neg\, g \wedge tr' = tr \wedge wait')))$
$A_1 \,\Box\, A_2$	$\mathbf{R}\left((\neg\, {A_1}^f_f \wedge \neg\, {A_2}^f_f) \vdash \left(\begin{array}{l} ({A_1}^t_f \wedge {A_2}^t_f) \\ \lhd tr' = tr \wedge wait' \rhd \\ ({A_1}^t_f \vee {A_2}^t_f) \end{array} \right) \right)$
$A_1 \sqcap A_2$	$A_1 \vee A_2$
$c \rightarrow Skip$	$\mathbf{R}(true \vdash do\,(c, Sync) \wedge v' = v)$
$c.e \rightarrow Skip$	$\mathbf{R}(true \vdash do\,(c, e) \wedge v' = v)$
$c!e \rightarrow Skip$	$c.e \rightarrow Skip$
$c?x : P \rightarrow A(x)$	$\mathbf{var}\ x \bullet \mathbf{R}(true \vdash do_{\mathcal{I}}\,(c, x, P) \wedge v' = v) ; A(x)$
$c?x \rightarrow A(x)$	$c?x : true \rightarrow A(x)$
$A_1 \,[\![ns_1 \mid cs \mid ns_2]\!]\, A_2$	$\mathbf{R}\left(\begin{array}{l} \neg\ \exists 1.tr', 2.tr' \bullet ({A_1}^f_f ; 1.tr' = tr) \\ \qquad\qquad \wedge ({A_2}_f ; 2.tr' = tr) \\ \qquad\qquad \wedge 1.tr' \restriction cs = 2.tr' \restriction cs \\ \wedge \neg\ \exists 1.tr', 2.tr' \bullet ({A_1}_f ; 1.tr' = tr) \\ \qquad\qquad \wedge ({A_2}^f_f ; 2.tr' = tr) \\ \qquad\qquad \wedge 1.tr' \restriction cs = 2.tr' \restriction cs \\ \vdash \\ \left(\begin{array}{l} ({A_1}^t_f ; U1(out\alpha\, A_1)) \\ \wedge ({A_2}^t_f ; U2(out\alpha\, A_2)) \end{array} \right)_{+\{v,tr\}} ; M_{[\![}(cs) \end{array} \right)$
$A_1 \,[\![ns_2 \mid ns_2]\!]\, A_2$	$\mathbf{R}\left(\begin{array}{l} (\neg\, {A_1}^f_f \wedge \neg\, {A_2}^f_f) \\ \vdash \\ \left(\begin{array}{l} ({A_1}^t_f ; U1(out\alpha\, A_1)) \\ \wedge ({A_2}^t_f ; U2(out\alpha\, A_2)) \end{array} \right)_{+\{v,tr\}} ; M_{[\![}(cs) \end{array} \right)$
$A \setminus cs$	$\mathbf{R}\left(\begin{array}{l} \exists s \bullet \\ \quad A[s, cs \cup ref'/tr', ref'] \\ \quad \wedge (tr' - tr) = (s - tr) \restriction (EVENT - cs) \end{array} \right) ; Skip$
$\mu X \bullet F(X)$	$\bigsqcap \{X \mid F(X) \sqsubseteq_{\mathcal{A}} X\}$
Iterated operators	By expansion of the operators
Action invocation	By the copy-rule
$(d \bullet A)\,(e)$	$A[e/x]$
$A[old := new]$	$A[new/old]$
$x_1, \ldots, x_n := e_1, \ldots, e_n$	$\mathbf{R}\left(true \vdash \left(\begin{array}{l} tr' = tr \wedge \neg\, wait' \\ \wedge\, x'_1 = e_1 \wedge \cdots \wedge x'_n = e_n \wedge u' = u \end{array} \right) \right)$
$w : [pre, post]$	$\mathbf{R}(pre \vdash post \wedge \neg\, wait' \wedge tr' = tr \wedge u' = u)$
$\{g\}$	$: [g, true]$
$[g]$	$: [true, g]$
$\mathbf{if}\ [\!]\ i \bullet g_i \rightarrow A_i\ \mathbf{fi}$	$\mathbf{R}((\bigvee i \bullet g_i) \wedge (\bigwedge i \bullet g_i \Rightarrow \neg\, {A_i}^f_f) \vdash \bigvee i \bullet (g_i \wedge {A_i}^t_f))$

Circus **Actions**	**Semantics**
var $x : T \bullet A$	**var** $x : T ; A ;$ **end** $x : T$
(**val** $x : T \bullet A)(e)$	(**var** $x : T \bullet x := e ; A$)
(**res** $x : T \bullet A)(y)$	(**var** $x : T \bullet A ; y := x$)
(**vres** $x : T \bullet A)(y)$	(**var** $x : T \bullet x := y ; A ; y := x$)
$[udecl;\ ddecl' \mid pred]$	$ddecl : [\exists\, ddecl' \bullet pred, pred]$

Circus **Processes**	**Semantics**
begin state $[decl \mid pred]$ $PPars \bullet A$ **end**	**var** $decl \bullet A$
$P\ op\ Q$ $op \in \{ ;, \Box, \sqcap \}$	**begin** **state** $State \mathrel{\widehat=} P.State \land Q.State$ $P.PPar \land_{\Xi} Q.State$ $Q.PPar \land_{\Xi} P.State$ $\bullet\ P.Act\ op\ Q.Act$ **end**
$P \llbracket cs \rrbracket Q$	**begin** **state** $State \mathrel{\widehat=} P.State \land Q.State$ $P.PPar \land_{\Xi} Q.State$ $Q.PPar \land_{\Xi} P.State$ $\bullet\ P.Act$ $\llbracket \alpha(P.State) \mid cs \mid \alpha(Q.State) \rrbracket$ $Q.Act$ **end**
$P \mid\mid\mid Q$	**begin** **state** $State \mathrel{\widehat=} P.State \land Q.State$ $P.PPar \land_{\Xi} Q.State$ $Q.PPar \land_{\Xi} P.State$ $\bullet\ P.Act$ $\llbracket \alpha(P.State) \mid \alpha(Q.State) \rrbracket$ $Q.Act$ **end**
$P \setminus cs$	**begin** **state** $State \mathrel{\widehat=} P.State$ $P.PPar$ $\bullet\ P.Act \setminus cs$ **end**
$i : T \odot P$	$(i : T \bullet P)[c : usedC(P) \bullet c\_i.i]$
$(i : T \odot P) \lfloor v \rfloor$	$(i : T \odot P)(v)$
Iterated operators	By expansion of the operators
Process invocation	By the copy-rule
$(x : T \bullet P)(e)$	$P[e/x]$
$P[oldc := newc]$	$P[newc/oldc]$
$P[te_0, \ldots, te_n]$	By instantiating the type variables with the corresponding te_i in the body of P

References

[Abr03] Abrial J-R (2003) B$^{\#}$: toward a synthesis between Z and B. In: Bert D, Bowen JP, King S, Waldén M (eds) ZB, vol 3582 of LNCS. Springer, Heidelberg, pp 168–177

[Bac78] Back RJR (1978) On the correctness of refinement steps in program development. PhD Thesis, Department of Computer Science, University of Helsinki. Report A-1978-4

[BG95] Bowen JP, Gordon MJC (1995) A shallow embedding of Z in HOL. Inf Softw Technol 37(5–6):269–276

[Cam90a] Camilleri AJ (1990) A higher order logic mechanization of the CSP failure-divergence semantics. Technical Report HPL-90-194, HP Laboratories, Bristol

[Cam90b] Camilleri AJ (1990) Mechanizing CSP trace theory in higher order logic. IEEE Trans Softw Eng 16(9):993–1004

[CDD+90] Carrington D, Duke D, Duke R, King P, Rose GA, Smith G (1990) Object-Z: an object-oriented extension to Z. In: Vuong ST (ed) Formal description techniques, II (FORTE'89), pp 281–296

[CSW03] Cavalcanti ALC, Sampaio ACA (2003) Woodcock JCP A refinement strategy for *Circus*. Formal Aspects Comput 15(2–3):146–181

[CW99] Cavalcanti ALC, Woodcock JCP (1999) ZRC—a refinement calculus for Z. Formal Aspects Comput 10(3):267–289

[CW06] Cavalcanti ALC, Woodcock JCP (2006) A tutorial introduction to CSP in unifying theories of programming. In: Cavalcanti ALC, Sampaio ACA, Woodcock JCP (eds) Refinement techniques in software engineering, vol 3167 of LNCS. Springer, Heidelberg, pp 220–268

[Dij76] Dijkstra EW (1976) A discipline of programming. Prentice-Hall, Englewood Cliffs

[DS97] Dutertre B, Schneider S (1997) Using a PVS Embedding of CSP to Verify Authentication Protocols. In: Gunter EL, Felty A (eds) Theorem proving in higher order logics: 10th international conference. TPHOLs'97, vol 1275 of LNCS. Springer, Heidelberg, pp 121–136

[Fis97] Fischer C (1997) CSP-OZ: A combination of object-Z and CSP. In: Bowman H, Derrick J (eds) Formal methods for open object-based distributed systems (FMOODS'97), vol 2. Chapman & Hall, London, pp 423–438

[Fis98] Fischer C (1998) How to combine Z with a process algebra. In: Bowen J, Fett A, Hinchey M (eds) ZUM'98: proceedings of the 11th international conference of Z users on the Z formal specification notation. Springer, Heidelberg, pp 5–23

[Gro92] The RAISE Language Group (1992) The RAISE specification language. Prentice-Hall, Englewood Cliffs

[GS97] Galloway A, Stoddart B (1997) An operational semantics for ZCCS. In: Hinchey MG (ed) ICFEM'97: proceedings of the 1st international conference on formal engineering methods. IEEE Computer Society, Washington, p 272

[HJ98] Hoare CAR, Jifeng H (1998) Unifying theories of programming. Prentice-Hall, Englewood Cliffs

[Hoa85] Hoare CAR (1985) Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs

[MD98] Mahony BP, Dong JS (1998) Blending Object-Z and timed CSP: an introduction to TCOZ. In: Torii K, Futatsugi K, Kemmerer RA (eds) The 20th international conference on software engineering (ICSE'98). IEEE Computer Society Press, Washington, pp 95–104

[Mor87] Morris JM (1987) A theoretical basis for stepwise refinement and the programming calculus. Sci Comput Programm 9(3): 287–306

[Mor94] Morgan C (1994) Programming from Specifications. Prentice-Hall, Englewood Cliffs

[MS98] Mota AC, Sampaio ACA (1998) Model-checking CSP-Z. In: Astesiano E (ed) Proceedings of FASE'98, held as part of the ETAPS'98: European joint conference on theory and practice of software, vol 1382 of LNCS. Springer, Heidelberg, pp 205–220

[NW04] Nuka G, Woodcock JCP (2004) Mechanising the alphabetised relational calculus. In: WMF2003: 6th Braziliam Workshop on Formal Methods, vol 95. Campina Grande, Brazil, pp 209–225

[NW06] Nuka G, Woodcock JCP (2006) Mechanising a unifying theory. In: Dunne S, Stoddart B (eds) UTP 2006: first international symposium on unifying theories of programming, vol 4010 of LNCS. Springer, Heidelberg, pp 217–235

[OCW05] Oliveira MVM, Cavalcanti ALC, Woodcock JCP (2005) Formal development of industrial-scale systems. Innovat Syst Softw Eng NASA J 1(2):125–146

[OCW06a] Oliveira MVM, Cavalcanti ALC, Woodcock JCP (2006) A Denotational Semantics for *Circus*. In: Aichernig B, Boiten E, Derrick J, Groves L (eds) International refinement workshop, vol 187 of electronic notes in theoretical computer science. Elsevier, Amsterdam, pp 107–123

[OCW06b] Oliveira MVM, Cavalcanti ALC, Woodcock JCP (2006) Unifying theories in ProofPower-Z. In: Dunne S, Stoddart B (eds) UTP 2006: first international symposium on unifying theories of programming, vol 4010 of LNCS. Springer, Heiddelberg, pp 123–140

[Oli05a] Oliveira MVM Formal Derivation of state-rich reactive programs using *Circus*—additional material, 2005. At http://www.cs.york.ac.uk/circus/refinement-calculus/oliveira-phd/

[Oli05b] Oliveira MVM (2005) Formal derivation of state-rich reactive programs using *Circus*. PhD Thesis, Department of Computer Science, University of York, YCST-2006/02

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

[QDC03] Qin SC, Dong JS, Chin WN (2003) A semantic foundation of TCOZ in unifying theories of programming. In: Araki K, Gnesi S, Mandrioli D (eds) FME 2003: formal methods, vol 2805 of LNCS. Springer, Heidelberg, pp 321–340

[Ros98] Roscoe AW (1998) The theory and practice of concurrency. Prentice-Hall Series in Computer Science. Prentice-Hall, Englewood Cliffs

[RWW94] Roscoe AW, Woodcock JCP, Wulf L (1994) Non-interference through Determinism. In: Gollmann D (ed) ESORICS 94, vol 875 of LNCS. Springer, Heidelberg, pp 33–54

[Saa97] Saaltink M (1997) The Z/EVES System. In: Bowen JP, Hinchey MG, Till D (eds) ZUM'97: The Z formal specification notation, vol 1212 of LNCS. Springer, Heidelberg, pp 72–85

[Smi97] Smith G (1997) A semantic integration of Object-Z and CSP for the specification of concurrent systems specified in Object-Z and CSP. In: Fitzgerald J, Jones CB, Lucas P (eds) Proceedings of FME'97, vol 1313 of LNCS. Springer, Heidelberg, pp 62–81

[Spi92] Spivey JM (1992) The Z notation: a reference manual, 2nd edn. Prentice-Hall, Englewood Cliffs

[TA97] Taguchi K, Araki K (1997) The state-based CCS semantics for concurrent Z specification. In: Hinchey M, Liu S (eds) International conference on formal engineering methods. IEEE, Washington, pp 283–292

[TS99] Treharne H, Schneider S (1999) Using a process algebra to control B operations. In: Araki K, Galloway A, Taguchi K (eds) Proceedings of the 1st international conference on integrated formal methods. Springer, Heidelberg, pp 437–456

[WC02] Woodcock JCP, Cavalcanti ALC (2002) The semantics of *Circus*. In: Bert D, Bowen JP, Henson MC, Robinson K (eds) ZB 2002: formal specification and development in Z and B, vol 2272 of LNCS. Springer, Heidelberg, pp 184–203

[WCF05] Woodcock JCP, Cavalcanti ALC, Freitas L (2005) Operational semantics for model-checking *Circus*. In: Fitzgerald J, Hayes IJ, Tarlecki A (eds) FM 2005: formal methods, vol 3582 of LNCS. Springer, Heidelberg, pp 237–252

[WD96] Woodcock JCP, Davies J (1996) Using Z—specification, refinement, and proof. Prentice-Hall, Englewood Cliffs

[WDB00] Woodcock JCP, Davies J, Bolton C (2000) Abstract data types and processes. In: Roscoe AW, Davies J, Woodcock JCP (eds) Millennial perspectives in computer science, proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare. Palgrave, pp 391–405

Received 11 January 2007

Accepted in revised form 8 October 2007 by B. K. Aichernig, E. A. Boiten, M. J. Butler, J. Derrick, L. Groves and C. B. Jones

Published online 4 December 2007