DOI 10.1007/s00165-007-0044-5
BCS © 2007
Formal Aspects of Computing (2013) 25: 133–158

Formal Aspects of Computing

Unifying theories in ProofPower-Z

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

[1]Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte, Campus Universitário, Natal, 59072-970, Brazil. E-mail: marcel@dimap.ufrn.br
[2]Department of Computer Science, University of York, York, UK

Abstract. The increasing interest in the combination of different computational paradigms is well represented by Hoare and He in the *Unifying Theories of Programming* (UTP). In this paper, we present a mechanisation of part of that work in a theorem prover, ProofPower-Z; the theories of alphabetised relations, designs, reactive and CSP processes are in the scope of this paper. Furthermore, the mechanisation of *Circus*, a language that combines Z, CSP, specification statements and Dijkstra's guarded command language, is also presented here. We also present an account of how this mechanisation is achieved, and more interestingly, of what issues were raised, and of our decisions. We aim at providing tool support not only for CSP and *Circus*, but also for further explorations of Hoare and He's unification, and for the mechanisation of languages whose semantics is based on the UTP.

Keywords: Relational semantics; Theorem proving; *Circus*.

1. Introduction

Researchers have recently shown a lot of interest in the combination of programming paradigms, with the design of several languages to describe and reason about different aspects and artifacts of software development. Hoare and He gave one of the most significant contribution towards unification [HJ98]. In the *Unifying Theories of Programming* (UTP), they use Tarski's relational calculus to give a denotational semantics to constructs from several programming paradigms. Relations between an initial and a subsequent observation of computer devices are used to give meaning to specifications, designs, and programs. Observational variables and associated healthiness conditions characterise theories for imperative, communicating, or reactive processes and their designs.

The relations are defined as predicates over observational variables that record information relevant to the characterisation of the program behaviour; examples are the global variables of the program themselves, and a variable to record whether the program has finished or not. The initial observations of the value of each variable are represented in the predicates using references to the undecorated names of these variables, and subsequent observations are represented using the names of the variables decorated with a dash. This follows the style of languages like Z, for example. Precisely, every predicate is a pair $(\alpha P, P)$, where αP is the alphabet: the set of observational variables that can be free in the formula P.

There are several theories in the UTP, which model different (combinations of) programming paradigms. Many theories share common ideas; sequential composition, conditional, nondeterminism, and parallelism are some of them. Refinement is interpreted as inclusion of relations: reverse implication. Healthiness conditions are used to characterise the relations that belong to a theory. They are often expressed in terms of an idempotent function ϕ that makes a program healthy. Every healthy program P is a fixed point of ϕ.

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

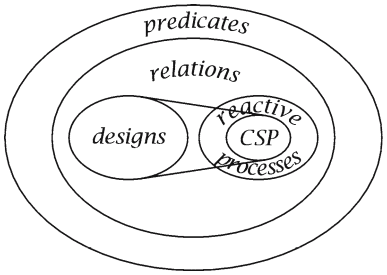

Fig. 1. Theories in the UTP

Figure 1 describes how some UTP theories are related. Relations are predicates whose alphabet includes *undashed* and *dashed* variables. Designs are models for specifications written in terms of pre and postconditions. Reactive processes are programs whose behaviour may depend on interactions with an environment. Finally, the theory of CSP processes is a failures-divergences model for CSP, enriched with state; they can be characterised as relations that result from applying **R** to designs [CW06]. This healthiness condition characterises the reactive processes; it is defined as the composition of three other healthiness conditions, **R1**, **R2** and **R3**, which are described in Sect. 3.4.

The mechanisation of these theories enables a further exploration of the UTP. Our work brings to light some important aspects of the definitions of alphabets and the healthiness conditions. Further work is bound to encourage precision in the definition of new theories, and in the characterisation and proofs of laws.

Following the trend of language integration, *Circus* [CSW03] combines a model-based language, Z [WD96], a process algebra, CSP [Hoa85], Dijkstra's language of commands [Dij76], and specification statements in the style of Morgan [Mor94]. It differs from other combinations [TA97, RWW94, Fis97, TS99] in that it has an associated refinement theory [CSW03, OCW05].

The early *Circus* semantics [WC01] did not allow us to prove the refinement laws. For this reason, we redefined the *Circus* semantics as a deep embedding of *Circus* in Z [Oli05b]. In this approach, the syntax and the semantics of *Circus* were formalised in Z. 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, and is the basis for the development of its theorem prover.

In subsequent work [OCW06b], we presented the first step towards automating the *Circus* semantics and the proof of its refinement laws: the mechanisation of the UTP in the theorem prover ProofPower-Z [PPW]. The definitions of the theories of relations, designs, reactive processes, and CSP, and more than five hundred theorems, is the result of our work. In this paper we take a step further. First, we present the mechanisation of the theories presented earlier [OCW06b] in more detail. Specifically, the theory of CSP is shown in much more detail here. Moreover, we present the mechanisation of the semantics of the *Circus* actions in ProofPower-Z based on the theories presented previously [OCW06b].

In early work [SJ02], Sherif and He present a timed extension of *Circus* whose semantics is based on a UTP theory for CSP processes with a notion of time. Qin et al. used the UTP to formalise the semantics of TCOZ [QDC03]; this is a combination of CSP and Object-Z. This semantics is being used as a reference document in the development of tools for TCOZ and as a semantic foundation for proving soundness of these tools. We have used the UTP model [WH02] in order to give a formal semantics to a programming language that contains shared variables. The work that we present here provides mechanical support not only to *Circus*, but also to any language that has the UTP as its theoretical basis.

The choice of the theorem prover for mechanising *Circus* and its refinement calculus was a major concern. ProofPower-Z is a higher-order tactic-based theorem prover implemented using New Jersey SML [Pau91]; it supports specifications and proofs in Z. It extends ProofPower-HOL [Art], which builds on ideas arising from research at the Universities of Cambridge [GM93] and Edinburgh [GMW79]. Some of the extensions provided by the New Jersey SML were used in ProofPower-Z, in order to achieve features such as a theory hierarchy, extension of the character set accepted by the metalanguage ML, and facilities for quotation of object language (Z or HOL) expressions, and for automatic pretty-printing of such expressions. Since it supports a powerful logic, ProofPower-Z has a lower level of automation than other theorem provers that support, for example, first-order logic. On the other hand, it has been successfully used in industry [KAW96, CCO05].

As it is an extension of ProofPower-HOL, definitions can be made using Z, HOL, and even SML, which is the input command language. ProofPower-Z also offers the possibility of defining proof tactics, which can be used to reduce and modularise proofs. Among other analysis support, it provides syntax and type checking, schema expansion, precondition calculation, domain checking, and general theorem proving. Using the subgoal package, goals can be split in simpler subgoals. The Z notation used into ProofPower-Z is almost the same as that of the Z standard [ISO02]; we explain the specificities of ProofPower-Z's notation as needed.

ProofPower-Z also provides a theory hierarchy, in which each theory is fully populated with appropriate definitions and theorems. For instance, the theory *z_sets* contains information about Z sets and the theory *z_relations* inherits from *z_sets* and adds operators over sets which are specific to sets of ordered pairs, such as dom and ran. In ProofPower-Z, the theory *z_library* is recommended as the parent of any theory that the user creates to work on Z; hence, it is used as the parent of all theories presented in this paper.

The large number of formally verified theories, including algebra, set theory, and many Z related theories, was one of the reasons for the choice of ProofPower-Z as the theorem prover used in the mechanisation of the *Circus* refinement calculus. Furthermore, by providing features like theory hierarchy and proof tactics, ProofPower-Z fosters the reuse of our results. The development of new theories in other theorem provers based on an axiomatisation of our results is yet another possibility of reuse.

In Sect. 2, we discuss design issues and describe the hierarchy of theories that we created. Section 3 describes the mechanisation of the UTP relations, designs, reactive processes, and CSP. The proof of a theorem illustrates our approach. *Circus* and its mechanisation is presented in Sect. 4. Finally, in Sect. 5, we draw our conclusions and describe future work.

2. Design issues

In the automation of the UTP, the first difficulty that we faced was that the name of a variable is used to refer both to the name itself and to its value. For instance, in the relation $(\{x\}, x = 0)$, the left-most x indicates the name x, while the right-most x stands for the value of x. We explicitly differentiate between variable names and values.

One of the options to give semantics to relations is axiomatically: the behaviour of relations is defined using axioms. We discarded this option because we would not be able to use most of the theorems that are built-in in ProofPower-Z to reason about sets and other models. We decided to give a set-based model for relations: for us, they are pairs of sets.

Since we want to prove refinement laws, our mechanisation must offer the possibility of expressing and proving meta-theorems. By way of illustration, we must be able to express and prove the left-unit law for *Skip*: $(\forall a : ALPHABET;\ A : CIRCUS\_ACTION \bullet Skip_a ; A = A)$. A shallow embedding, in which the mapping from language constructs to their semantic representation is part of the meta-language, would not allow us to express such theorems. We use a deep embedding, where the syntax and the semantics of the alphabetised relations is formalised inside the host language [BG95]. This is one of the differences between our embedding and that of [BG95], where Bowen and Gordon present a shallow embedding of Z into HOL.

The syntax of relations and designs could be expressed as a data type (Z free types), say *PRED*, for the relations. In this case, the semantics would be given by a partial ($\nrightarrow$) function $f : PRED \nrightarrow PRED$. For instance, the syntax of relations could be defined as follows.

$$PRED ::= True(ALPHABET) \mid ... \mid And(PRED \times PRED) \mid ...$$

The semantic function would have type $PRED \rightarrow S$, where S would be some structure in the semantic model. If we took this approach, most of the proofs would be by induction over *PRED*. Any extension to the language would require proving most of the laws again. Instead, we express the language constructors as functions; this is a standard approach in functional languages [SS99]. For instance, the function $\wedge_R : (PRED \times PRED) \rightarrow PRED$ can be used to define conjunction. Extensions require only the definition of the new constructors, and that they preserve any healthiness conditions; no proofs need to be redone.

Using SML as a meta-language would not give us a deep embedding. In ProofPower-Z, we were left with the choice of Z or HOL. If we used HOL as meta-language, reusing the definitions of Z constructs would not be possible, because they are written in SML. Because of our knowledge of Z, and the expressiveness of its toolkit, we have used Z to express the syntax and the semantics of alphabetised relations. This approach gave us a deep embedding, which, besides allowing us to prove meta-theorems like our refinement laws, has the additional advantage of providing the possibility of introducing new predicate combinators.

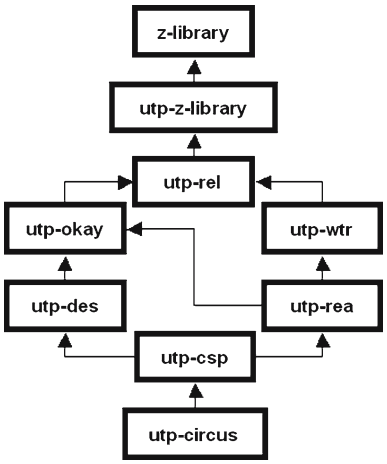

Fig. 2. Theories in the UTP

Figure 2 presents our hierarchy of theories. The theory *utp-z-library* extends the ProofPower-Z theory *z-library* and provides a more elaborate library of theorems about Z sequences. The theory *utp-rel* is that of general UTP relations. It includes basic alphabetised operators like conjunction and existential quantification; relational operators like alphabet extension, sequential composition, and skip; and refinement. Like all our theories, it includes the operators' definitions and their laws. Our proofs of the laws of a theory do not expand definitions of its parent theory; it uses the parent's laws. This provides modularisation and encapsulation.

Two theories inherit from *utp-rel*: *utp-okay* is concerned with the observational variable *okay*, and *utp-wtr* with *wait*, *trace*, and *ref*. These are the main observational variables of the theory of reactive processes. The theory *utp-okay* is the parent of *utp-des*, the theory for designs. Along with *utp-wtr*, *utp-okay* is also one of the parents of the reactive processes theory, *utp-rea*, which redefines part of *utp-rel*. The theory for CSP processes, *utp-csp*, inherits from both *utp-rea* and *utp-des*. The theory for *Circus*, *utp-circus*, inherits from *utp-csp*. Besides CSP, *Circus* also includes Z, specification statements, and guarded commands. Nevertheless, in *Circus* they are given a reactive semantics. For this reason, we have not created a separate theory for each one of them.

In the next section we describe in detail our theories. For the sake of presentation, we do not present the Z as generated by the ProofPower-Z document preparation tool, which has an awkward indentation for expressions. Instead, we present a better indented copy of the pretty-printed ProofPower-Z expressions.

3. Encoding the UTP

The ProofPower-Z theory *z_library* provides the Z toolkit. We enriched this theory with definitions of operations on sequences like prefix and subtraction, and with theorems on sequences and relations; for that, we created a separate theory *utp-z-library*.

3.1. Relations

The first substantial theory in our work is *utp-rel*. It provides a set-based model for alphabetised relations. In this theory, we define alphabets, relations, and basic programming constructs as we describe in the sequel.

Alphabets. A name is an element of the basic type (given set) $[NAME]$. Each relation has an alphabet of type $ALPHABET \mathrel{\widehat=} \mathbb{P}\, NAME$. The Z abbreviation $N == A$ is provided as $N \mathrel{\widehat=} A$ in ProofPower-Z; it gives a name N to the mathematical object A. 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 is the set of *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 define the input (*undashed*) names and the output (*dashed*)

names of a given alphabet. All the definitions and proof scripts for the theories that we present here can be found elsewhere [Oli05a].

An alphabet a in which $n \in a \Leftrightarrow n' \in a$, for all *undashed* names n, is called *homogeneous*. For us, n' is encoded 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.

Values and types. A value is an element of the free-type VAL, which can be an integer, a boolean, a set of values, a sequence of values, a pair of values, a communication channel, or a special synchronisation value.

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

This definition introduces the set of values VAL, which has the constant $Sync$ as one of its elements. In ProofPower-Z, $Bool(\mathbb{B})$ stands for the Z constructor $Bool\langle\!\langle \mathbb{B} \rangle\!\rangle$, which introduces a collection of different constants in VAL, one for each element of the set $\mathbb{B}$, ProofPower-Z's boolean type.

We have chosen not to take the same approach as for the relational predicates because we are not particularly interested in the semantics of values. Furthermore, the type VAL can be extended without any impact on the proofs of our theories.

Although we are defining an untyped theory, some observational variables have types; for instance, the variable *okay* used in the theory of designs and in the theory of reactive processes is a boolean. For this reason, we specify some types in our theory. For example, Booleans are in the set $BOOL\_VAL$ that is defined as $\{Bool(true), Bool(false)\}$, and channels are in the set $CHAN\_VAL \mathrel{\widehat=} \{n : NAME \bullet Channel(n)\}$.

In Z, a set comprehension $\{x : s \mid p \bullet e\}$ denotes the set of all expressions e such that x is taken from s and satisfies the condition p. Usually, p and e contains one or more free occurrences of x; a *true* condition may be omitted.

Events are pairs: the first element is a communication channel and the second element is a communicated value. They are elements of $EVENT\_VAL \mathrel{\widehat=} \{c : CHAN\_VAL;\ v : VAL \bullet Pair(c, v)\}$. In Z, a set-comprehension may declare more than one variable.

Expressions. A couple of definitions allow us to abstract from the syntax of expressions. The set of relations ($\leftrightarrow$) between values is $REL \mathrel{\widehat=} VAL \leftrightarrow VAL$. The set of unary functions is $UN\_F \mathrel{\widehat=} VAL \twoheadrightarrow VAL$; similarly, for binary functions we have the set BIN_F that is defined as the set of partial functions from pairs of values to values. For example, the sum function can be represented as $\{(Int(0), Int(0)) \mapsto Int(0), (Int(0), Int(1)) \mapsto Int(1), \ldots\}$. An expression can be a value, a name, a relation, or a unary or binary function application.

$$\begin{aligned} EXP ::= {} & Val(VAL) \mid Var(NAME) \mid Rel(REL \times EXP \times EXP) \\ & \mid Fun_1(UN\_F \times EXP) \mid Fun_2(BIN\_F \times EXP \times EXP) \end{aligned}$$

The definitions for unary functions, binary functions, and relations deal only with values; $Fun_1(f, e)$ can only be evaluated once e is evaluated to some VAL.

A binding is a partial function from $NAME$ to VAL, and therefore we define the set of all bindings, $BINDING$, as $NAME \twoheadrightarrow VAL$. 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).

Relations. Alphabetised relations are modelled by the type REL_PRED defined below. Basically, a relation is a pair: the first element is its alphabet, and the second element 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. Alternative models which relax this restriction are possible; however, they would lead to more complex definitions as we discuss in Sect. 5.

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

In Z, the omission of a constructor expression in a set-comprehension $\{x_0 : s_0;\ \ldots;\ x_n : s_n \mid p\}$ denotes a set-comprehension $\{x_0 : s_0;\ \ldots;\ x_n : s_n \mid p \bullet (x_0, \ldots, x_n)\}$.

Here, we use Z axiomatic definitions, which introduce constrained objects, to define our constructs. For example, the axiomatic definition below introduces a symbol x, an element of s, satisfying the predicate p.

$$\begin{array}{|l} x : s \\ \hline p \end{array}$$

Our first construct represents the boolean *true*. For a given alphabet a, $True_R(a)$ is defined as the pair with alphabet a, and with all the bindings with domain a.

$$True_R : ALPHABET \to REL\_PRED$$
$$\forall a : ALPHABET \bullet True_R(a) = (a, \{b : BINDING \mid \mathrm{dom}(b) = a\})$$

We subscript the constructs in order to make it easier to identify which theory they belong to; we use R for the theory of relations.

Nothing satisfies *false*: the second element of $False_R(a)$ is the empty set.

$$False_R : ALPHABET \to REL\_PRED$$
$$\forall a : ALPHABET \bullet False_R(a) = (a, \varnothing)$$

This operator is the main motivation for representing relations as pairs. If we had defined relations just as sets 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. These predicates have different alphabets; hence, they are indeed different and must be differentiated. It is also important to distinguish between $True_R(\varnothing)$ and $False_R(\varnothing)$: the second element of the former is the set containing the empty binding, while the second element of the latter is the empty set.

An equality compares the value of a variable with the value of an expression. It can only be applied to triples (a, n, e) where the name n is a member of the alphabet a and the set of free variables of the expression e is a subset of a. If we defined equality as a partial function, domain checks would be required during proofs. Instead, we define a total function on the set of tuples (a, n, e) that satisfy these conditions (WF_Equals_R).

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

The alphabet of an equality is simply the alphabet given as argument. 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, $\mathrm{dom}(b) = a$ and $b(n) = Eval(b, e)$.

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.

A similar equality encoded in this theory compares the values of two expressions. It is encoded as a function from $WF\_ALPHABET\_EXPRESSION$ to UTP relations. The type $WF\_ALPHABET\_EXPRESSION$ is the set of triples (a, e_1, e_2) where the free variables of both expressions e_1 and e_2 are in the alphabet a.

$$=_{+R} : WF\_ALPHABET\_EXPRESSION \to REL\_PRED$$
$$\forall a\_e\_e : WF\_ALPHABET\_EXPRESSION \bullet$$
$$=_{+R} (a\_e\_e) = (a\_e\_e.1, \{b : BINDING \mid \mathrm{dom}(b) = a\_e\_e.1 \land Eval(b, a\_e\_e.2) = Eval(b, a\_e\_e.3)\})$$

The alphabet of this equality is also the alphabet given as argument. For a given alphabet a, and expressions e_1 an e_2, such that the free variables of both e_1 and e_2 are in a, the function $=_{+R} (a, e_1, e_2)$ returns a relational predicate (a, bs), in which for every binding b in bs, $\mathrm{dom}(b) = a$ and $Eval(b, e_1) = Eval(b, e_2)$.

As we are working directly with the semantics of predicates, we are not able to give a syntactic characterisation of free variables. Instead, we have the concept of an unrestricted variable, which is actually not equivalent to that of a free variable.

$$UnrestVar : REL\_PRED \to \mathbb{P}\, NAME$$
$$\forall u : REL\_PRED \bullet UnrestVar(u) = \{n : u.1 \mid \forall b : u.2;\ v : VAL \bullet b \oplus \{n \mapsto v\} \in u.2\}$$

For a relation u, a name n from its alphabet is unrestricted if, for every binding b of u, all the bindings obtained by changing the value of n in b are in u. In Z, $f \oplus g$ stands for the relational overriding of f with g. If a variable is not a free variable of a predicate, then it is unrestricted, but the converse does not hold. For instance, x is unrestricted in $x = x$, but it is a free variable.

Unrestricted variables are necessarily untyped, as they can assume any value in VAL. As previously discussed, however, the observational variables are typed. As a consequence, we need a weaker concept to express that some variables are unrestricted within their types. The function $UnrestTypedVar$ returns a boolean that states if a given variable n is unrestricted within its given type T in a given predicate u.

$$UnrestTypedVar : REL\_PRED \times NAME \times \mathbb{P}\, VAL \to \mathbb{B}$$

$$\forall u : REL\_PRED;\ n : NAME;\ T : \mathbb{P}\, VAL \bullet$$
$$UnrestTypesVar(u, n, t) \Leftrightarrow (\forall b : u.2;\ v : T \bullet b \oplus \{n \mapsto v\} \in u.2)$$

The predicate that checks if a variable is unrestricted within its type is similar to the one used for *UnrestVar*; however, it does not check all possible values v in VAL, but only values v in T, which is a set of VALues.

Operators. 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 operator $A \lhd R$: it restricts the domain of a relation $R : X \leftrightarrow Y$ to a set A, which must be a subset of X, ignoring any member of R whose first element is not a member of A.

$$\_ \oplus_R \_ : REL\_PRED \times ALPHABET \to REL\_PRED$$

$$\forall u : REL\_PRED;\ a : ALPHABET$$
$$\bullet\ u \oplus_R a = (u.1 \cup a, \{b : BINDING \mid (u.1 \lhd b) \in u.2 \land \mathrm{dom}(b) = u.1 \cup a\})$$

Conjunction is the union of the alphabets and the intersection of the extended set of bindings of each relation.

$$\_ \land_R \_ : REL\_PRED \times REL\_PRED \to REL\_PRED$$

$$\forall u1, u2 : REL\_PRED \bullet u1 \land_R u2 = (u1.1 \cup u2.1, (u1 \oplus_R u2.1).2 \cap (u2 \oplus_R u1.1).2)$$

The definition of disjunction is similar, but the second element of the result is the union of the extended set of bindings. We have proved that these definitions are idempotent, commutative, and associative, and that they distribute over each other [Oli05a]. We have also proved that $True_R$ is the zero for disjunction and the unit for conjunction; similar laws were also proved for $False_R$. However, restrictions on the alphabets must be taken into account. As an example, we present below the unit law for conjunction. The ProofPower-Z notation $n \vdash t$ gives name n to a theorem t. In Z, the quantification $\forall x : a \mid p \bullet q$ is equivalent to the predicate $\forall x : a \bullet p \Rightarrow q$.

$$REL\_True\_ \land_R \_id\_thm1 \vdash \forall a : ALPHABET;\ u : REL\_PRED \mid a \subseteq u.1 \bullet u \land_R True_R(a) = u$$

The value of the conjunction $u \land true$ is u, but the alphabet of the truth must be a subset of the alphabet of u. Otherwise, the conjunction may have a larger alphabet and the theorem does not hold.

The negation 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 conciseness, we omit the trivial definitions of implication ($\_ \Rightarrow_R \_$), equivalence ($\_ \Leftrightarrow_R \_$), and conditional expressions ($\_ \lhd_R \_ \rhd_R \_$) in terms of the operators above.

The function $-_R$ removes variables from the alphabet of a relation using domain anti-restriction to remove names from the set of bindings. It is defined as $u -_R a = (u.1 \setminus a, \{b : u.2 \bullet a \lhd b\})$. Complementary to domain restriction, the domain anti-restriction operator $A \lhd R$, ignores any member of R, whose first element is a member of A. Existential quantification $\exists_{\_R}$ simply removes the quantified variables from the alphabet and changes the bindings accordingly.

$$\exists_{\_R} : ALPHABET \times REL\_PRED \to REL\_PRED$$

$$\forall a : ALPHABET;\ u : REL\_PRED \bullet \exists_{\_R}(a, u) = u -_R a$$

Universal quantification $\forall_{\_R}(a, u)$ is defined as $\neg_R \exists_{\_R}(a, \neg_R u)$.

The UTP uses another existential quantification in the CSP theory; it does not remove the quantified variables from the alphabet. This alternative quantifier $\exists_R(a, u)$ is defined as $(\exists_{\_R}(a, u)) \oplus a$: it removes the quantified names from the alphabet and re-includes them with unrestricted values.

Sequential composition is not defined as in the UTP, using an existential quantification on the intermediary state; the motivation is to simplify our proofs. In the UTP, the existential quantification is described using new 0-subscripted names to represent the intermediate state. Its encoding requires two functions: one for creating new names, and one for expressing substitution of names. Any proof on sequential composition would require induction on both functions. Instead, we give an equivalent model-based definition.

Relations can only be combined in sequence if their alphabets are *composable*. The type WF_Semi_R is the set of pairs of relations, which have *composable* alphabets.

$$
\begin{array}{|l}
\_ ;_R \_ : WF\_Semi_R \rightarrow REL\_PRED \\
\hline
\forall\, u1\_u2 : WF\_Semi_R \bullet u1\_u2.1 ;_R u1\_u2.2 = (in\_a(u1\_u2.1.1) \cup out\_a(u1\_u2.2.1), \\
\qquad\qquad \{b1 : u1\_u2.1.2;\ b2 : u1\_u2.2.2 \\
\qquad\qquad \mid (\forall\, n : \mathrm{dom}(b2) \mid n \in undashed \bullet b2(n) = b1(dash(n))) \\
\qquad\qquad \bullet (undashed \lhd b1) \cup (dashed \lhd b2)\})
\end{array}
$$

The alphabet of a sequential composition is composed of the input alphabet of the first relation u_1 and the output alphabet of the second relation u_2. For each pair of bindings (b_1,b_2) from u_1 and u_2, respectively, we make a combination of all input values in b_1 (*undashed* names) with output values in b_2(*dashed* names). However, only those pairs of bindings in which the final values of all names in b_1 are equal to their initial values in b_2 are included in this combination.

The UTP defines an alphabet extension that enables sequential composition to be applied to operands with non-composable alphabets. The function $+_R$ encodes the UTP's alphabet extension P_{+a}; it differs from $\oplus_R$ in that it restricts the value of the new name to be left unchanged. For a given predicate P and alphabet a of *undashed* names, it returns the encoding of the predicate $P \wedge \mathit{II}_{a \cup a'}$, where II is the relational skip that is described in the sequel.

Although useless for practical purposes, II is very useful for reasoning about programs. In our work it is encoded as the function shown below. Given a well-formed alphabet a, it does not change the alphabet and returns all the bindings b with domain a, in which for every *undashed* name n in a, $b(n) = b(n')$. The type WF_Skip_R is the set of all *homogeneous* alphabets.

$$
\begin{array}{|l}
\Pi_R : WF\_Skip_R \rightarrow REL\_PRED \\
\hline
\forall\, a : WF\_Skip_R \bullet \\
\quad \Pi_R(a) = (a, \{b : BINDING \mid \mathrm{dom}(b) = a \wedge (\forall\, n : a \mid n \in undashed \bullet b(n) = b(dash(n)))\})
\end{array}
$$

Variable blocks are also part of *utp-rel*. We present the definitions of variable declaration and undeclaration.

$$
\begin{array}{|l}
\mathbf{var}_R, \mathbf{end}_R : WF\_Var\_End_R \rightarrow REL\_PRED \\
\hline
\forall\, a\_n : WF\_Var\_End_R \bullet \mathbf{var}_R\ a\_n = \exists_{\_R}(\{a\_n.2\}, \Pi_R(a\_n.1)) \\
\qquad\qquad\qquad\qquad \wedge\ \mathbf{end}_R\ a\_n = \exists_{\_R}(\{dash(a\_n.2)\}, \Pi_R(a\_n.1))
\end{array}
$$

The type $WF\_Var\_End_R$ is the set of pairs (a, n), such that a is a *homogeneous* alphabet that contains both n, which must be an *undashed* name, and n'. The operator $\mathbf{var}_R$ begins the scope of a name: it uses the existential quantification to hide this name from programs that precede the variable block. Similarly, $\mathbf{end}_R$ finishes the scope of a name using the existential quantification to hide the *dashed* counterpart of the given name from programs that follow the variable block.

In the UTP, assignments are alphabetised: each variable in the left-hand side receives the value of the corresponding expression in the right-hand side and all the variables that are in the alphabet but not mentioned in the left-hand side remain unchanged. For instance, the meaning of the assignment $x :=_A 0$, where A is the alphabet $\{x, x', y, y'\}$, is $x' = 0 \wedge y' = y$.

The relational assignment receives a *homogeneous* alphabet a, a sequence ns of names and a sequence $exps$ of expressions. All the names in ns and free variables in $exps$ must be *undashed* and belong to a; both lists ns and $exps$ have the same strictly positive length. The set of tuples $(a, ns, exps)$ that satisfy these conditions is WF_Assign_R.

$$
\begin{array}{|l}
Assign_R : WF\_Assign_R \rightarrow REL\_PRED \\
\hline
\forall\, a\_ns\_exps : WF\_Assign_R \bullet \\
\quad \exists\, n : NAME;\ e : EXP;\ a : ALPHABET;\ ns : \mathrm{seq}\ NAME;\ exps : \mathrm{seq}\ EXP \\
\quad \mid a = a\_ns\_exps.1 \wedge ns = a\_ns\_exps.2 \wedge exps = a\_ns\_exps.3 \\
\quad\ \wedge n = head(ns) \wedge e = head(exps) \\
\quad \bullet (\#ns = 1 \wedge Assign_R(a\_ns\_exps) = (=_R (a, dash(n), e) \wedge_R \Pi_R(a \setminus \{n, dash(n)\}))) \\
\quad\ \vee (\#ns > 1 \wedge Assign_R(a\_ns\_exps) = \\
\qquad\qquad (=_R (a, dash(n), e) \wedge_R Assign_R(a \setminus \{n, dash(n)\}, tail(ns), \\
\qquad\qquad\qquad tail(exps))))
\end{array}
$$

```
SML
| set_goal([],
|     ⌜z ∀ F : REL_FUNCTION; Y : REL_PRED
|         | Y ∈ dom F
|         ∧ (F(Y) ⊑_R Y = True_R ∅)
|       • μ_R(F) ⊑_R Y = True_R ∅ ⌝);
|
| a (rewrite_tac[]);
| a (REPEAT z_strip_tac);
| a (all_asm_fc_tac[z_get_spec ⌜z μ_R ⌝]);
| a (asm_rewrite_tac[]);
| a ( (PC_T1 "initial" lemma_tac
|     ⌜z Y ∈ {u : REL_PRED
|       | a = u.1 ∧ F u ⊑_R u = True_R{}} ⌝ )
|   THEN1 (asm_prove_tac[]) );
| a (all_asm_fc_tac[]);
|
|
|
```

```
SML
| a ( (lemma_tac
|     ⌜z{u : REL_PRED
|       | a = u.1 ∧ F u ⊑_R u = True_R{}}
|     ∈ ℙ REL_PRED ⌝ )
|   THEN1 (PC_T1 "z_sets_ext"
|                 asm_prove tac[]) );
| a ( (lemma_tac
|     ⌜z(a, {u : REL_PRED
|         | a = u.1 ∧ F u ⊑_R u = True_R{}})
|     ∈ WF_Glb_R_Lub_R ⌝ )
|   THEN1
|   ((rewrite_tac[z_get_spec ⌜z WF_Glb_R_Lub_R ⌝])
|     THEN
|     (PC_T1 "z_sets_ext" asm_prove_tac[])) );
| a (apply_def REL_lower_bound_thm
|     ⌜z(a ≙ a, u ≙ Y,
|       us ≙ {u : REL_PRED
|           | a = u.1 ∧ F u ⊑_R u = True_R{}}) ⌝ );
```

Fig. 3. Proof script for the weakest fixed point theorem

The function $Assign_R$ presented above is recursive. In its definition, we take n and e as the first elements (*head*) of the given lists of names *ns* and expressions *exps*, respectively, and a to be the given alphabet. If *ns* is a singleton, $Assign_R$ returns the predicate $n' = e \wedge \mathit{II}_{a \setminus \{n,n'\}}$; this corresponds to the original definition of assignment since it states, using the relational skip II, that the values of the remaining variables are unchanged. If, however, *ns* is not a singleton, $Assign_R$ returns the conjunction of the predicate $n' = e$ with the evaluation of $Assign_R$ with argument $(a \setminus \{n, n'\}, tail(ns), tail(exps))$.

We could have used universal quantification to define assignments. Nevertheless, this would lead to an extremely more complicated definition. For this reason, we used Z's recursion.

Further programming constructs are also included in this theory, but are omitted here for conciseness.

Refinement. We now turn to the definition of refinement, which, in the UTP, is the universal implication of relations. The universal closure used in the UTP is defined as follows.

$$\langle \_ \rangle_R : REL\_PRED \to REL\_PRED$$
$$\forall u : REL\_PRED \bullet \langle u \rangle_R = \forall_{\_R}(u.1, u)$$

We have used angled brackets, instead of the square brackets of [HJ98], because of problems with the LaTeX automatically generated by the ProofPower's document preparation tool.

For a pair of relations (u_1,u_2), such that $(u_1, u_2) \in WF\_REL\_PRED\_PAIR$ (both have the same alphabet), we have that u_1 is refined by u_2, if, and only if, for all names in their alphabets, $u_2 \Rightarrow u_1$. This is expressed by the definition below.

$$\_ \sqsubseteq_R \_ : WF\_REL\_PRED\_PAIR \to REL\_PRED$$
$$\forall u1\_u2 : WF\_REL\_PRED\_PAIR \bullet u1\_u2.1 \sqsubseteq_R u1\_u2.2 = \langle u1\_u2.2 \Rightarrow_R u1\_u2.1 \rangle_R$$

We have proved that our interpretation of refinement is, as expected, a partial order. Moreover, the set of relations with alphabet a is a complete lattice.

Only functions $f : REL\_PRED \nrightarrow REL\_PRED$ whose domain is a set of relations with the same alphabet are considered in the theory of fixed points. We call the set of such functions *REL_FUNCTION*. The definition of the weakest fixed point of such functions is standard. The greatest fixed point is defined as the least upper bound of the set $\{X \mid X \sqsubseteq f(x)\}$. This is different from Hoare and He's definition [HJ98], which is not convenient for proofs. However, it is trivial to prove that we have an equivalent definition.

The proof of theorems is the subject of the next section, where we present an example of a proof of a fixed point law.

3.2. Proving theorems

We have built a theory with more than five-hundred laws on alphabets, bindings, relational predicates, and laws from the predicate calculus. In what follows, we illustrate our approach in their proofs.

The proof script for one of our laws, the weakest fixed point law ($\forall F,\ Y \bullet F(Y) \sqsubseteq Y \Rightarrow \mu F \sqsubseteq Y$), is shown in Fig. 3. We set our goal to be the law we want to prove using the SML command *set_goal*. It receives a list of

assumptions and the proof goal. In our case, since we are not dealing with standard predicates, we must explicitly say that relations are $True_R$.

We start our proof by rewriting the Z empty set definition (*rewrite_tac*) and stripping the left-hand side of the implication into the assumptions (*z_strip_tac*). The SML command *a* applies a tactic to the current goal; the tactical *REPEAT* applies the given tactic as many times as possible. The next step is to rewrite the definition of least fixed point in the conclusion: we use forward chaining in the assumptions (*all_asm_fc_tac*), giving our Z definition of least fixed point as argument, and use the new assumption to rewrite the conclusion (*asm_rewrite_tac*).

The application of a previously proved theorem, *REL_lower_bound_thm*, concludes our proof. However, it requires some assumptions, before being applied. We introduce them in the assumption list using the tactic *lemma_tac*. The first condition is that Y is an element of the set of relations u, with an alphabet a, such that $F(u) \sqsubseteq_R u$. We use the tactical PC_T1 to stop ProofPower-Z from rewriting our expression by using the proof context *initial*, which is the most basic proof context. In order to avoid a new subgoal, we use the tactical $THEN1$ that applies the tactic in the right-hand side to the first subgoal generated by the tactic in the left-hand side. Here, this proves that the assumption we are introducing is valid. The validity of the introduction of the first assumption is proved using the tactic *asm_prove_tac*, a powerful tactic that uses the assumptions in an automatic proof procedure. Next, after introducing the first condition shown above in the list of assumptions, we use forward chaining again to state the fact that the alphabet of Y is a.

The next step introduces the fact that the set to which Y belongs is in fact a set of *REL_PRED*. The proof script for the validity of this assumption uses ProofPower-Z's proof context *z_sets_ext*, an aggressive complete proof context for manipulating Z set expressions. The last assumption that is needed is the fact that the pair composed by the alphabet a and the set to which Y belongs, is indeed of type $WF\_Glb_R\_Lub_R$, which contains all set of pairs (a, bs), in which every binding in the set bs has a as its alphabet. Its proof rewrites the conclusion using the Z definition of $WF\_Glb_R\_Lub_R$, and then uses the tactic *asm_prove_tac* in the *z_sets_ext* proof context. Finally, we use a tactic defined by us, *apply_def*, to instantiate the theorem *REL_lower_bound_thm* with the given values. The tactic *apply_def* instantiates the given theorem with the values given as arguments, and tries to rewrite the conclusion, using this instantiation.

ProofPower-Z has provided us with facilities that resulted in a rather short proof, for a quite complex theorem. First, by using proof contexts, we can control the context in which we want the proofs to be carried out. Secondly, tacticals, such as *REPEAT* and *THEN_1* shorten the proof script considerably. Finally, the use of user-defined tactics, such as *apply_def*, and automated proof tactics, such as *asm_rewrite_tac* and *all_asm_fc_tac*, shortens considerably the proofs, saving time and effort.

3.3. Okay and designs

The UTP theory of pre and postcondition pairs (designs) introduces two extra observational boolean variables: *okay* indicates that a program has started, and *okay*′ indicates that the program has terminated. In the theory *utp-okay*, we define *okay* as an *undashed* name ranging over the booleans.

$$\frac{okay : name}{okay \in undashed}$$

We restrict the type *BINDING* by determining that *okay* and *okay*′ are only associated with boolean values.

$$\begin{array}{l} \forall\, b : BINDING \mid okay \in \mathrm{dom}(b) \bullet b(okay) \in BOOL\_VAL \\ \land\ \forall\, b : BINDING \mid dash(okay) \in \mathrm{dom}(b) \bullet b(dash(okay)) \in BOOL\_VAL \end{array}$$

This could have been introduced when we first defined *BINDING*, but as we intend to have modular theories, we postponed the restriction on observational variables used by specific theories.

As expected, *okay* and *okay*′ are not totally unrestricted. They can, however, be unrestricted within *BOOL_VAL*. The functions below indicate whether *okay* and *okay*′ are unrestricted in a given predicate.

$$\begin{array}{l} unrestOKAY, unrestOKAY' : REL\_PRED \to \mathbb{B} \\ \hline \forall\, u : REL\_PRED \bullet unrestOKAY(u) = unrestTypeVar(u, okay, BOOL\_VAL) \\ \qquad\qquad \land\ unrestOKAY'(u) = unrestTypeVar(u, dash(okay), BOOL\_VAL) \end{array}$$

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_PRED*, the set of relations u, such that $u.1 \in ALPHABET\_DES$. 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 we encode as $=_R (a, okay, Val(Bool(true)))$. Designs are defined as follows.

$$\begin{array}{|l} \_ \vdash_D \_ : WF\_DES\_PRED\_PAIR \rightarrow REL\_PRED \\ \hline \forall d : WF\_DES\_PRED\_PAIR \bullet d.1 \vdash_D d.2 = (=_R (d.1.1, okay, Val(Bool(true))) \wedge_R d.1) \Rightarrow_R \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (=_R (d.1.1, dash(okay), Val(Bool(true))) \wedge_R d.2) \end{array}$$

The members of $WF\_DES\_PRED\_PAIR$ are pairs of relations (r_1, r_2) from DES_PRED 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 latter uses it to define designs. We have kept both of them, but we subscript the UTP design turnstile with a D. The most important result for designs, which is the motivation for their definition, has also been proved in our mechanisation: the left-zero law for $True_R$.

In this new setting, new definitions for Π_R and assignment are needed. The skip for designs Π_D is defined in terms of the relational skip Π_R as follows.

$$\begin{array}{|l} \Pi_D : WF\_Skip_D \rightarrow REL\_PRED \\ \hline \forall a : WF\_Skip_D \bullet \Pi_D(a) = True_R(a) \vdash_D (\Pi_R(a)) \end{array}$$

The type WF_Skip_D is formed by all the *homogeneous* alphabets that contain $okay$ and $okay'$. The new definition of assignment uses the relation assignment in a very similar way.

Designs are also characterised as the set of relations that satisfy two healthiness conditions. The first, **H1**, guarantees that observations cannot be made before the program starts. We define $H1(d) = okay \Rightarrow d$ as $H1(d) = (=_R (\{okay\}, okay, Val(Bool(true)))) \Rightarrow_R d$. The set of relations that satisfy a healthiness condition h is the set of relations r such that $h(r) = r$. By way of illustration, the set of **H1**-healthy relations is $H1\_healthy = \{d : REL\_PRED \mid H1(d) = d\}$.

H2-healthy relations do not require non-termination. In previous research [CW06], we presented a way of expressing **H2** in terms of an idempotent function: $\mathbf{H2}(P) = P; J$, where $J \mathrel{\widehat{=}} (okay \wedge okay' \Rightarrow v' = v)$. We express $v' = v$ as the relational skip Π_R on the alphabet containing the names in the lists v and v'. We define J as a function that takes an alphabet a' containing only *dashed* variables, and yields the relation presented below, where $A = a \cup a'$, and a is obtained by undashing all the names in a'.

$$(okay =_A true \Rightarrow_R okay' =_A true) \wedge_R \Pi_R(A \setminus \{okay, okay'\})$$

Our definition of the function $H2$ is as follows.

$$\begin{array}{|l} H2 : REL\_PRED \nrightarrow REL\_PRED \\ \hline \forall d : REL\_PRED \mid dash(okay) \in d.1 \bullet H2(d) = (d ;_R (J(out\_a(d.1)))) \end{array}$$

The function $H2$ is partial because J defines a relation that includes $okay$ and $okay'$ in its alphabet, and hence, the alphabet of a relation d that can be made $H2\_healthy$ must contain $okay'$ in order to be *composable* with J ($out\_a(d.1)$). In order to reuse our previous results [CW06], we use this definition for **H2**.

More than 30 laws from previous works [HJ98, CW06], involving designs and their healthiness conditions, have been included in our theory of designs. Their proofs do not expand any definitions in the relations theory; only laws of that theory are used. Many laws were included in the relations theory in order to support proofs in the designs and in the other theories. Our work provides a base of laws that are useful to support mechanised proof in the UTP in practice.

3.4. WTR and reactive processes

The behaviour of reactive processes cannot be expressed only in terms of their final states; interactions with the environment (events) need to be considered. Besides *okay*, in the theory of reactive processes we have the observational variables *tr*,*wait*, and *ref*. The definitions of these variables are in the theory *utp-wtr*. The variable *wait* records whether the process has terminated or is waiting for interaction with the environment in an intermediate state. Since it is a boolean, the definition of *wait* is similar to that of *okay*. The variable *tr* records the sequence of events in which the process has engaged; it has type SEQ_EVENT_VAL. The variable *ref* is a set of events in which the process may refuse to engage; its type is SET_EVENT_VAL.

In the theory *utp-rea*, we define *REA_PRED* as the set of relations whose alphabet is in *ALPHABET_REA*, the set of reactive alphabets. These are the alphabets that contain at least *okay*, *tr*, *wait*, *ref*, and their *dashed* counterparts.

Healthiness conditions characterise the reactive processes. For instance, **R1** states that the history of events of a process cannot be changed, therefore, the value of *tr* can only get longer. Our definition uses a function $\leqslant_R$, which is the Z sequence-prefixing relation lifted to *VAL*ues.

$$\begin{array}{|l} \_ \leqslant_R \_ : VAL \leftrightarrow VAL \\ \hline (\_ \leqslant_R \_) = \{s1, s2 : SEQ\_VAL \mid ((Seq^{\sim})(s1))\, prefix_Z\, ((Seq^{\sim})(s2))\} \end{array}$$

The type *SEQ_VAL* is defined as $\{s : \text{seq } VAL \mid Seq(s)\}$, and the Z sequence-prefixing operator $prefix_Z$ is defined in *utp-z-library*. Furthermore, in Z, $^{\sim}$ stands for the relational inverse operator.

The definition of $R1$ below encodes the UTP function $\mathbf{R1}(P) = P \wedge tr \leqslant tr'$.

$$\begin{array}{|l} R1 : REL\_PRED \to REL\_PRED \\ \hline \forall r : REL\_PRED \bullet R1(r) = r \wedge_R (=_{+R} (ALPHABET\_OWTR, \\ \qquad\qquad Rel((\_ \leqslant_R \_), Var(tr), Var(dash(tr))), Val(Bool(true)))) \end{array}$$

The set *ALPHABET_OWTR* contains only *okay*, *tr*, *wait*, *ref*, and their *dashed* counterparts. In order to transform $tr \leqslant tr'$ into a relational predicate that can be used in a conjunction, we assert that $Rel((\_ \leqslant_R \_), Var(tr), Var(dash(tr)))$ is equal to $Val(Bool(true))$. We may adopt this strategy to transform any Z relation (for example, $\in, \notin, \subseteq$) and function (using Fun_1 and Fun_2) into relational predicates.

The second healthiness condition establishes that a reactive process $P(tr, tr')$ should not rely on events that happened before it started; it is defined in the UTP as $\sqcap_s(P(s, s \frown (tr' - tr)))$. We encode our previous simpler formulation $R2(P(tr, tr')) = P(\langle\rangle, tr' - tr)$ [CW06]; this requires that P is not changed if *tr* is taken to be the empty sequence, and tr' is taken to be $tr' - tr$, the sequence obtained from tr' by removing its prefix *tr*. The notation $P(\langle\rangle, tr' - tr)$ is encoded using substitution; **R2** is encoded as $R2(P) = P[\langle\rangle/tr][tr' - tr/tr']$.

The final healthiness condition **R3** defines the behaviour of a process that is still waiting for another process to finish: it should not start. It is defined as $\mathbf{R3}(P) = \mathit{II}_{REA} \lhd wait \rhd P$, and is encoded as follows:

$$\begin{array}{|l} R3 : REA\_PRED \nrightarrow REA\_PRED \\ \hline \forall r : REA\_PRED \mid r.1 \in WF\_Skip_{REA} \bullet \\ \qquad R3(r) = (\Pi_{REA}(r.1)) \lhd_R =_R (\{wait\}, wait, Val(Bool(true))) \rhd_R r \end{array}$$

This definition uses a conditional expression and the reactive skip. Conditional expressions are defined only for branches with the same alphabet, and Π_{REA} is defined only for *homogeneous* reactive alphabets (WF_Skip_{REA}). For this reason, our work reveals that **R3** is not total: it can only be applied to *homogeneous* reactive relations (members of *REA_PRED*).

A reactive process (*REA_PROCESS*) is a relation with a reactive alphabet *a*, which is *R_healthy*; the function R is defined as $R(r) = R1(R2(R3(r)))$. Based on these definitions, more than sixty laws, including some we presented previously [CW06], are part of our theory of reactive processes. Among other properties, they prove that the healthiness conditions for reactive processes are idempotent and commutative, and the closure of some of the programming operators with respect to the healthiness conditions. They also explore relations between healthiness conditions for reactive processes and designs.

3.5. CSP processes

The theory of CSP processes includes the predicates of the theory of reactive processes that satisfy two additional healthiness conditions. The first one states that the only guarantee in the case of divergence ($\neg\, okay$) is that the trace can only be extended. Its encoding is as follows.

$$\begin{array}{|l} CSP1 : REL\_PRED \nrightarrow REL\_PRED \\ \hline \forall r : REL\_PRED \bullet CSP1(r) = r \vee_R (=_R (ALPHABET\_OWTR, okay, Val(Bool(false))) \\ \qquad\qquad \wedge_R (=_{+R} (ALPHABET\_OWTR, \\ \qquad\qquad\qquad Rel((\_ \leqslant_R \_), Var(tr), Var(dash(tr))), \\ \qquad\qquad\qquad Val(Bool(true))))) \end{array}$$

This is a direct translation of the function $\mathbf{CSP1}(r) \mathrel{\widehat=} r \vee (\neg\, okay \wedge tr \leqslant tr')$.

The second healthiness condition is a recast of **H2**, presented in Sect. 3.3, with an extended reactive alphabet. The encoding of **CSP2** in ProofPower-Z reveals, as it does for **H2**, that this function is not total: it is only applicable to relational predicates which contain $okay'$, tr', $wait'$, and ref' in their alphabet.

$$CSP2 : REL\_PRED \nrightarrow REL\_PRED$$
$$\forall r : REL\_PRED \mid \{dash(okay), dash(tr), dash(wait), dash(ref)\} \subseteq r.1 \bullet CSP2(r) = r ;_R J(out\_a(r.1))$$

A *CSP_PROCESS* is a *CSP*1*_healthy* and *CSP*2*_healthy* reactive process.

The process *STOP* is incapable of engaging in any events and is always waiting.

$$STOP : CSP\_PROCESS$$
$$STOP = R(Assign_R\,(ALPHABET\_OWTR, \langle dash(wait)\rangle, \langle Val(Bool(true))\rangle))$$

By assigning *true* to $dash(wait)$, we guarantee that $wait'$ is true and hence, the process is always waiting.

In earlier work [CW06], we presented an introduction to CSP in the UTP. Our definitions correspond to those presented by Hoare and He [HJ98], but with a different style of specification: every CSP process is defined as a reactive design of the form $\mathbf{R}(pre \vdash post)$. 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**. The structural uniformity of the semantics (and its encoding) given to the *Circus* operators is reflected in the proofs of the refinement laws, which use the strategy discussed in our earlier work [OCW06a]. This uniformity also facilitates mechanisation since it allows the modularisation of proofs, and the construction and application of more general and effective proof tactics.

The correspondences between Hoare and He's definitions [HJ98] and ours [CW06] are presented in our theory as theorems. For instance, the theorem below states this correspondence for *STOP*.

$$\begin{array}{l} CSP\_STOP\_design\_thm \\ \vdash STOP = R(\, True_R(ALPHABET\_OWTR) \\ \qquad \vdash_D =_R (ALPHABET\_OWTR, dash(tr), Var(tr)) \\ \qquad\quad \wedge_R =_R (ALPHABET\_OWTR, dash(wait), Val(Bool(true)))\,) \end{array}$$

It suggests that *STOP* never diverges since it has a *true* precondition; furthermore, it is always in a waiting state, but never changes the trace.

The *SKIP* process terminates immediately. The value of *ref* is irrelevant; it is quantified in the definition below.

$$SKIP : CSP\_PROCESS$$
$$SKIP = R(\exists_R\,(\{ref\}, \Pi_{REA}\, ALPHABET\_OWTR))$$

The existential quantification does not remove *ref* from the alphabet, as opposed to that used in the definition, for instance, of variable blocks. The theorem *CSP_SKIP_design_thm*, which is part of our theory, states the equivalence between this definition and a reactive design similar to that presented for *STOP*, but with the value *false* for $wait'$.

The external choice of processes P and Q with a common alphabet behaves like the conjunction of P and Q if no progress has been made, that is, no event has been observed and termination has not occurred. Otherwise, it behaves like their disjunction. It is encoded as follows.

$$\_ \,\Box_{CSP}\, \_ : WF\_CSP\_PROCESS\_PAIR \to CSP\_PROCESS$$
$$\forall pp : WF\_CSP\_PROCESS\_PAIR \bullet pp.1 \,\Box_{CSP}\, pp.2 = CSP2((pp.1 \wedge_R pp.2) \lhd_R STOP \rhd_R (pp.1 \vee_R pp.2))$$

The members of *WF_CSP_PROCESS_PAIR* are pairs of CSP processes with the same alphabet.

In our earlier work [CW06], we proved that there is indeed a reactive design that corresponds to the UTP's definition of external choice. When its predecessor has terminated without diverging, an external choice $P_1 \,\Box\, P_2$ does not diverge if neither P nor Q do. We capture this behaviour in the precondition of the following definition of external choice. The postcondition of this reactive design establishes that if the trace has not changed and the choice has not terminated, the behaviour of an external choice is given by the conjunction of the effects of both processes; otherwise, the choice has been made and the behaviour is either that of P or Q.

$$P \,\Box\, Q \mathrel{\widehat{=}} \mathbf{R}((\neg\, P_f^f \wedge \neg\, Q_f^f) \vdash ((P_f^t \wedge Q_f^t) \lhd tr' = tr \wedge wait' \rhd (P_f^t \vee Q_f^t)))$$

The notation P_c^b denotes $P[b/okay'][c/wait]$. Basically, P_f^f is the condition in which P diverges when it is not waiting for its predecessor to finish, and P_f^t defines the result of P on termination without divergence.

Four auxiliary functions encode the substitutions P_c^b; in order to make it more like the textual notation, we use a postfix notation for them. For instance, $P\,\sigma_f\,\omega_f$ encodes the predicate P_f^f.

$$\begin{array}{|l}
\omega_f, \omega_t, \sigma_f, \sigma_t : CSP_PROCESS \to CSP_PROCESS \\
\hline
\forall\, c : CSP_PROCESS \bullet\ c\,\sigma_f = /_R(c, Val(Bool(false)), dash(okay)) \\
\qquad\qquad\qquad\qquad \wedge\ c\,\sigma_t = /_R(c, Val(Bool(true)), dash(okay)) \\
\qquad\qquad\qquad\qquad \wedge\ c\,\omega_f = /_R(c, Val(Bool(false)), wait) \\
\qquad\qquad\qquad\qquad \wedge\ c\,\omega_t = /_R(c, Val(Bool(true)), wait)
\end{array}$$

The expression $/_R(p, e, n)$ encodes the substitution of a variable n by an expression e in a predicate p ($p[e/n]$). Its definition belongs to the theory of relations and has been omitted here for the sake of conciseness. For the same reason as that discussed for free variables, it is not a syntactic definition, but a transformation on the set of bindings of c.

Using these auxiliary functions, we have included the following theorem in our theory to relate the original UTP definition of external choice to the reactive design presented above.

$$\begin{array}{l}
CSP_\,\Box_{CSP}\,_design_thm \\
\vdash \forall\, P, Q : CSP_PROCESS \mid (P, Q) \in WF_CSP_PROCESS_PAIR \\
\quad \bullet\ P \,\Box_{CSP}\, Q = R(\,((\neg\,{}_R P\,\sigma_f\,\omega_f) \wedge_R (\neg\,{}_R Q\,\sigma_f\,\omega_f)) \\
\qquad\qquad\qquad \vdash_D \\
\qquad\qquad\qquad (((\neg\,{}_R P\,\sigma_t\,\omega_f) \wedge_R (\neg\,{}_R Q\,\sigma_t\,\omega_f)) \\
\qquad\qquad\qquad\ \lhd_R =_R (P.1, dash(tr), Var(tr)) \\
\qquad\qquad\qquad\quad \wedge_R =_R (P.1, dash(wait), Val(Bool(true)))) \rhd_R \\
\qquad\qquad\qquad\ ((\neg\,{}_R P\,\sigma_t\,\omega_f) \vee_R (\neg\,{}_R Q\,\sigma_t\,\omega_f))\,)\,)
\end{array}$$

Although long, the right-hand side of this theorem is a direct encoding of the previously presented reactive design representation of external choice.

Besides the definition for prefixing given by the UTP, we encode a simpler definition which was proven equivalent: $e \to_{CSP} SKIP = R(true \vdash do_C(e))$. The function below is a simplified version of the function do_A presented in the UTP to characterise the occurrence of an event. The simplification is possible because we express prefixing as a reactive design.

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

An event has either not happened, and the trace has not changed and the process is willing to engage in e, or it has happened and the trace has been extended.

The encoding of do_C is not straightforward. We have already discussed the encoding of the conditional expressions and its condition $wait'$, and the equality $tr' = tr$. The encoding of $e \notin ref'$ and $tr' = tr \frown \langle e \rangle$ are a little more complex, as we explain now.

In CSP, one might write $n.e \to SKIP$, where e is actually an expression. For this reason, our encoding of do_C presented below receives two arguments: the name n of the channel and the communicated expression e. We assume that the observational variables cannot be used in an expression that specifies a communicated value. The type VAR_NAME contains all names that are not observational variables.

We need to express an event itself as an expression; for this purpose, we define a function $MkPair$ that receives a pair of VALues (v_1, v_2) and returns the VALue $Pair(v_1, v_2)$. The expression that defines the event e as an expression is $Fun_2(MkPair, Val(Channel(n)), e)$; its evaluation will give us a pair where the first element is $Channel(n)$ and the second element is the evaluation of e. In the left-hand side of the conditional expression, we lift the set non-membership relation $\notin_R$ to VALues in the same way we did for $\leqslant_R$. In the right-hand side though, we use yet another function, $MkSingleton$, which receives a value v and returns the singleton sequence value $Seq(\langle v \rangle)$. The expression $Fun_1(MkSingleton, Fun_2(MkPair, Val(Channel(n)), e))$ corresponds to the expression $\langle ev \rangle$, where e is itself an event expression. Finally, the same strategy to lift Z relations is applied to lift the Z concatenation

$$
\begin{array}{l}
\textbf{channel}\ count, reset, ticket, close \\
\textbf{channel}\ result, sold : \mathbb{N} \\
\textbf{process}\ Counter \mathrel{\widehat=}\ \textbf{begin state}\ CState \mathrel{\widehat=} [\, c : \mathbb{N} \,] \\
\qquad\qquad CInit \mathrel{\widehat=} [\, CState' \mid c' = 0 \,] \\
\qquad\qquad Inc \mathrel{\widehat=} [\, \Delta CState \mid c' = c + 1 \,] \\
\qquad\qquad Run \mathrel{\widehat=} count \rightarrow Inc \\
\qquad\qquad\qquad\qquad \Box\ result!c \rightarrow Skip \\
\qquad\qquad\qquad\qquad \Box\ reset \rightarrow CInit \\
\qquad\qquad \bullet\ CInit;(\mu\, X \bullet Run;\ X) \\
\qquad\qquad \textbf{end} \\
\textbf{process}\ Tickets \mathrel{\widehat=}\ \textbf{begin}\ Selling \mathrel{\widehat=} ticket \rightarrow count \rightarrow Skip \\
\qquad\qquad\qquad\qquad\qquad\qquad \Box\ close \rightarrow Closing \\
\qquad\qquad\qquad\qquad Closing \mathrel{\widehat=} result?x \rightarrow sold!x \rightarrow reset \rightarrow Skip \\
\qquad\qquad \bullet\ (\mu\, X \bullet Selling;\ X) \\
\qquad\qquad \textbf{end} \\
\textbf{chanset}\ Sync == \{\!|\ count, result, reset\ |\!\} \\
\textbf{process}\ TicketOffice \mathrel{\widehat=}\ (Counter\ [\![\ Sync\ ]\!]\ Tickets) \setminus Sync
\end{array}
$$

Fig. 4. A ticket office

function; however, we do not need to assert that the expression is equal to *true* because we are dealing with functions, not relations.

$$
\begin{array}{|l}
do\_C : VAR\_NAME \times EXP \rightarrow CSP_PROCESS \\
\hline
\forall n : VAR\_NAME;\ e : EXP \bullet \\
\quad do\_C(n, e) = \\
\qquad =_R\ (ALPHABET\_OWTR, dash(tr), Var(tr)) \\
\qquad \wedge_R\ (=_{+R}\ (ALPHABET\_OWTR, \\
\qquad\qquad Rel((\_ \notin_R \_), Fun_2(MkPair, Val(Channel(n)), e), Var(dash(ref))), \\
\qquad\qquad Val(Bool(true)))) \\
\qquad \lhd_R =_R\ (ALPHABET\_OWTR, dash(wait), Val(Bool(true))) \rhd_R \\
\qquad =_R\ (ALPHABET\_OWTR, \\
\qquad\quad dash(tr), Fun_2((\_ \frown_R \_), Var(tr), \\
\qquad\qquad\qquad Fun_1(MkSingleton, Fun_2(MkPair, Val(Channel(n)), e))))
\end{array}
$$

This function is used in the definition of CSP prefix as a reactive design.

$$
\begin{array}{l}
CSP\_ \rightarrow_{CSP} \_design\_thm \\
\vdash \forall\, ev : VAR\_NAME \times EXP \bullet ev \rightarrow_{CSP} SKIP = R(True_R(ALPHABET\_OWTR) \vdash_D do\_C(ev))
\end{array}
$$

All of our CSP theorems [CW06] and Hoare and He's UTP theorems [HJ98] are part of our *utp-csp* theory, which is the basis of the *Circus* theory presented in the next section.

4. *Circus*

A *Circus* program is 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. 4; it specifies a ticket office that uses a counter to store the number of sold tickets.

Our example has three processes: the process *Counter* counts the number of tickets, the process *Tickets* sells the tickets, and the process *TicketOffice* represents the whole system. Furthermore, six channels are used in the system: *count*, *reset* and *result* are used to request the *Counter* to increment, reset, or return its value, respectively; tickets can be bought via channel *ticket*; channel *sold* communicates the number of sold tickets; finally, channel *close* is used to close the ticket office.

In the next section, we give an overview of the language and explain the details of our example, before discussing the encoding of *Circus* actions in Sect. 4.2.

4.1. The language

In *Circus*, a channel declaration gives its name and type; if the channel is used purely for synchronisation, then no type is needed. A channel declaration may declare more than one channel of the same type. In this case, instead

of a single channel name, we have a comma-separated list of channel names. This is illustrated in Fig. 4 by the declaration of channels *count*, *reset*, *ticket*, and *close*. We introduce sets of channels in a **chanset** paragraph. In our example, we declare the channel set *Sync*, which groups the channels used in the communication between processes *Tickets* and *Counter*.

Processes may be defined explicitly or in terms of other processes (compound processes). An explicit process definition is delimited by the keywords **begin** and **end**: it is formed by a state definition (identified by the keyword **state**), a sequence of paragraphs, and a nameless main action. Process *Counter* in Fig. 4 is defined in this way. The schema *CState* describes the internal state of the process *Counter*: it contains a natural number c that stores the value of the counter. The behaviour of *Counter* is described by the unnamed action after the $\bullet$; it is recursive: after an initialisation, it behaves like *Run*, and then recurses. The state component c can be initialised and incremented using the Z operations *CInit* and *Inc*, respectively. The process *Tickets* does not have a state; it also has a recursive behaviour but does not invoke any initialisation.

Compound processes are defined using the CSP operators for sequence, external and internal choice, parallelism and interleaving, or their corresponding iterated operators, or event hiding. The parallelism follows the alphabetised approach adopted by Roscoe [Ros98], instead of that adopted by Hoare [Hoa85]. The process *TicketOffice* in Fig. 4 receives an indication that a ticket has been sold via channel *ticket* and interacts with a *Counter* through channel *count*. It is a parallel composition of processes *Counter* and *Tickets*; they synchronise on the set of events *Sync*. Furthermore, process *TicketOffice* encapsulates the interaction between these two processes; the only ways to interact with *TicketOffice* are via the channels *ticket*, *close* and *sold*.

An action can be a schema expression, a guarded command, an invocation of any action, or a combination of these constructs using CSP operators. Three primitive actions are available: *Skip*, *Stop*, and *Chaos*. The prefixing operator is standard, but a guard construction may be associated with it. For instance, if the condition p is *true*, the action $p \;\&\; c?x \rightarrow A$ 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. Enabling conditions like p may be associated with any action.

The action *Closing* in the process *Tickets* (Fig. 4) exemplifies the input and output prefix operators. It synchronises with *Counter* on *result*, receiving the total number x of sold tickets, which is sent to the environment using channel *sold*, and requests the *Counter* to reset.

Like the *Circus* processes, *Circus* actions can be composed using the CSP operators of sequence, external and internal choice, parallelism, interleaving, their corresponding iterated operators, and hiding. Furthermore, recursive definitions are also available. Our *Counter*, as previously described, has a recursive behaviour. Its cycle, the action *Run*, is an external choice: its current value may be incremented, using channel *count*; the result may be requested using channel *result*; finally, the counter may be reset through channel *reset*.

To avoid conflicts in access to the variables in scope, parallelism and interleaving of actions declare a synchronisation channel set and two sets that partition all the variables. In the parallelism $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, 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.

It is important to distinguish the parallel composition of processes and the parallel composition of actions. In the former, we do not need any state partition because each process encapsulates its own state; in the latter, however, we need to partition the state.

Finally, an action may be a variable block, specification statement, assumption, coercion, alternation, or 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 theory are presented elsewhere [Oli05b]; its encoding in ProofPower-Z is the subject of the next section.

4.2. Encoding the *Circus* semantics

Although the constructors of CSP do not contain state variables, the set of processes described by the UTP theory of CSP contains processes that might have state components. By definition, a *CSP_PROCESS* is a *CSP*1*_healthy* and *CSP*2*_healthy* reactive process; the only restriction on the alphabet is that it must contain the observational variables and their *dashed* counterparts, 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. Later in this section, we discuss the *Circus* healthiness conditions.

The definitions of the ProofPower-Z theory of *Circus*, *utp-circus*, follow directly from Oliveira's semantics [Oli05b]. In what follows, we present some of the more interesting definitions and discuss important aspects that were raised during this mechanisation.

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.

***Stop*, *Skip*, and *Chaos*.** We start with the definition of *Stop*. For a given *homogeneous* alphabet a that contains $ALPHABET\_OWTR$ ($WF\_hom\_alpha_C$), *Stop* is the reactive design with a *true* precondition, which we encode using the relational $True_R$, and with the conjunction $tr' =_a tr \wedge_R wait'$ as its postcondition.

$$
\begin{array}{|l}
Stop : WF\_hom\_alpha_C \to CSP\_PROCESS \\
\hline
\forall a : WF\_hom\_alpha_C \bullet \\
\quad Stop(a) = R(True_R(a) \vdash_D (=_R (a, dash(tr), Var(tr)) \wedge_R =_R (a, dash(wait), Val(Bool(true)))))
\end{array}
$$

In the postcondition of the reactive design above, state components are not mentioned; this means that state changes do not decide the choice and, hence, *Stop* leaves the values of the state components unconstrained.

The encoding of *Skip* is similar; however, besides leaving the trace unchanged, its postcondition requires termination ($\neg\ wait'$) and leaves the state unchanged as we present below.

$$
\begin{array}{|l}
Skip : WF\_hom\_alpha_C \to CSP\_PROCESS \\
\hline
\forall a : WF\_hom\_alpha_C \bullet Skip(a) = R(True_R(a) \vdash_D =_R (a, dash(tr), Var(tr)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge_R =_R (a, dash(wait), Val(Bool(false))) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge_R\ \Pi_R(a \setminus ALPHABET\_OWTR))
\end{array}
$$

By giving the expression $a \setminus ALPHABET\_OWTR$ as the argument to the relational skip, we keep all the variables in a that are not in $ALPHABET\_OWTR$ unchanged. *Chaos* is encoded as $R(False_R(a) \vdash_D True_R(a))$.

Guards. An important definition is that of predicates that can be used in the syntax of *Circus* specifications, which cannot mention any of the UTP observational variables. In our model, they are represented by the type $CIRCUS\_PRED$, which contains all the predicates in which the observational variables are in the alphabet, but left unrestricted within their types. On the other hand, in the syntax of *Circus*, conditions are predicates that contain no *dashed* variables. The type $CIRCUS\_COND$ contains all the relational predicates whose alphabet contains the UTP observational variables, but in which their values are unrestricted within their types, and the values of the remaining *dashed* variables are unrestricted.

A guarded action is defined in terms of a condition and an action.

$$
\begin{array}{|l}
\_\ \&_C\ \_ : CIRCUS\_COND \times CSP\_PROCESS \to CSP\_PROCESS \\
\hline
\forall g : CIRCUS\_COND;\ a : CSP\_PROCESS \bullet \\
\quad g\ \&_C\ a = R((g \Rightarrow_R \neg_R (a\,\sigma_f\ \omega_f)) \\
\qquad\qquad \vdash_D \\
\qquad\qquad ((g \wedge_R (a\,\sigma_t\ \omega_f)) \vee_R (\neg_R g \wedge_R =_R (a.1, dash(tr), Var(tr)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge_R =_R (a.1, dash(wait), Val(Bool(true))))))
\end{array}
$$

This definition derives directly from Oliveira's semantics [Oli05b], but uses the new notation used in the encoding for substitution. If the guard g is *false*, this definition can be reduced to *Stop*. However, if the guard g is *true*, we are left with the reactive design $\mathbf{R}(\neg\ A^f_f \vdash A^t_f)$. In [HJ98], Hoare and He show that this reactive design is exactly A itself.

Prefixing. Simple prefix has a very similar definition to that in the CSP theory; however, since *Circus* processes have state, the postcondition must guarantee that it is left unchanged. Also, instead of defining two different functions, one for simple prefix followed by *Skip*, and another for simple prefix followed by any other action, we define a single function as presented below. This is motivated by the convenience of proofs in ProofPower-Z and by our wish to keep the theory *utp-csp* as faithful as possible to the UTP.

$$
\begin{array}{|l}
\_ \to_C \_ : (VAR\_NAME \times EXPR) \times CSP\_PROCESS \to CSP\_PROCESS \\
\hline
\forall c : VAR\_NAME;\ e : EXPR;\ A : CSP\_PROCESS \bullet \\
\quad (c, e) \to_C A = R(True_R(A.1) \vdash_D do\_C(c, e) \wedge_R \Pi_R(A.1 \setminus ALPHABET\_OWTR)); A
\end{array}
$$

This function receives a pair that contains a channel name c and an expression e, and an action A; it gives the pair (c, e) to the function do_C. We use the relational skip to state that the state components are left unchanged. If no value is being communicated, we have yet another function $\_ \rightarrow_{CSync} \_$, which receives only a channel name c as its first argument and is trivially defined as $(c, Val(Sync)) \rightarrow_C A$.

The encoding of variable blocks is done in terms of the relational operations that can be used to introduce and remove a variable from scope. Variable declaration is used in the expected way in the encoding of the input prefix.

Parallelism. 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. We assume that references to channel sets have already been expanded using their corresponding definitions. We present the encoding of the parallel composition as a reactive design in two parts: first we discuss its precondition, and then, we discuss its postcondition. In both parts, we first discuss its original semantics and then we present its encoding.

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

$$\exists 1.tr', 2.tr' \bullet (A_{1_f}^{f}; (1.tr' = tr)) \wedge (A_{2f}; (2.tr' = tr)) \wedge 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs$$

Basically, if there exist two traces $1.tr'$ and $2.tr'$, that are equal to the final traces of A_1 after divergence and of A_2, respectively, and if these two traces are equal modulo cs, then it is possible for A_1 to reach divergence. First, the relation $A_{1_f}^{f}; (1.tr' = tr)$ assigns to $1.tr'$ the trace on which A_1 diverges. The first predicate of the sequence $A_{1_f}^{f}$ give us the conditions under which A_1 diverges; we record the final trace in $1.tr'$ in the second predicate of the sequence $1.tr' = tr$, which does not restrict the 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 communications on the channels in cs. These can occur independently, but for the communications that require synchronisation, $1.tr'$ and $2.tr'$ have to agree: they have to be equal modulo cs. We use the sequence filtering function $\upharpoonright$.

Given two processes $a1$ and $a2$ and a channel set cs, the function $DivPar$ defines the predicate that describes the condition on which $a1$ may diverge.

$$\begin{array}{|l} DivPar : CSP\_PROCESS \times CSP\_PROCESS \times SET\_EVENT\_VAL \rightarrow REL\_PRED \\ \hline \forall a1, a2 : CSP\_PROCESS;\ cs : SET\_EVENT\_VAL \bullet \\ \quad DivPar(a1, a2, cs) = \exists_R(\{dash(one(tr)), dash(two(tr))\}, \\ \qquad\qquad ((a1\ \sigma_f\ \omega_f);_C(=_R\ (a1.1, dash(one(tr)), Var(tr)))) \\ \qquad\qquad \wedge_R\ ((a2\ \omega_f);_C(=_R\ (a2.1, dash(two(tr)), Var(tr)))) \wedge_R\ (MSync(cs))) \end{array}$$

We encode the expression $1.tr$ and $2.tr$ as the application of two functions $one, two : NAME \rightarrowtail NAME$ which have disjoint ranges. The expression $1.tr \upharpoonright cs = 2.tr \upharpoonright cs$ is encoded as $MSync(cs)$. This encoding and some others that follow have been omitted here, for conciseness; they have been informally describe and can be found in [Oli05a].

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

$$\neg_R(DivPar(a1, a2, cs)) \wedge_R \neg_R(DivPar(a2, a1, cs))$$

The postcondition uses Hoare and He's parallel by merge [HJ98]. Conceptually, it runs both actions independently and merges their results afterwards:

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

Their independent executions are expressed using a relabelling function Ul: the result of applying Ul to an alphabet $\{v'_1, \ldots, v'_n\}$ is the predicate $l.v'_1 = v_1 \wedge \cdots \wedge l.v'_n = v_n$. Before the merge, however, we extend the alphabet of the predicate that expresses the independent execution of both actions with v' and tr'; this records the initial values of the trace tr and of the state components and local variables v in tr' and v', respectively. As explained

in Sect. 3.1, for a predicate P and an alphabet a of *undashed* names, P_{+a} (encoded as $P \ +_R \ a$) is equivalent to $P \wedge \Pi_{a \cup a'}$. The initial values of *tr* and *v* are used by the merge function $M_{\|_{cs}}$, as we explain later in this section.

The relabelling is done by the function U.

$$
\begin{array}{|l}
U : (NAME \rightarrowtail NAME) \times ALPHABET \nrightarrow REL\_PRED \\
\hline
\forall f : (NAME \rightarrowtail NAME);\ a' : ALPHABET \mid a' \subseteq dashed \\
\quad \bullet\ (\exists\, a : ALPHABET \mid a \subseteq undashed \wedge a' = dash(\!|\ a\ |\!) \\
\qquad \bullet\ U(f, a') = (\{n : NAME \mid n \in a \bullet dash(f(n))\} \cup a, \\
\qquad\qquad \{b : BINDING \mid \mathrm{dom}(b) = \{n : NAME \mid n \in a \bullet dash(f(n))\} \cup a \\
\qquad\qquad\qquad \wedge\ (\forall n : NAME \mid n \in a \bullet b(dash(f(n))) = b(n))\}))
\end{array}
$$

It receives a renaming function f (for instance, *one*) and an alphabet a' containing only *dashed* names, and returns a relational predicate whose alphabet is the union of a' with the corresponding *undashed* alphabet a. We use the Z relational image $(\!|\ |\!)$ to retrieve the *undashed* version of a'; for a given relation $D : X \leftrightarrow Y$, and a subset A of X, $D(\!|\ A\ |\!)$ returns the set of all elements in Y to which some element of A is related via D. The bindings of the resulting relational predicate are those whose domain is the same as the relation's alphabet, and in which the values of the final (*dashed*) values of the relabelled names $b(dash(f(n)))$ are the same as the values of their corresponding *undashed* original names.

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

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

The trace is extended with the merge of the new events that happened in both actions. The function $\|_{cs}$ takes the individual traces and gives a set containing all the possible combinations of these two traces taking *cs* into consideration. The expression that antecedes the merge $M_{\|_{cs}}$ in (1) gives us all the possible behaviours of running A_1 and A_2 independently; however, only those combinations that are feasible regarding the synchronisation on *cs* should be considered ($1.tr \upharpoonright cs = 2.tr \upharpoonright cs$). The definition of $\|_{cs}$ is omitted here but can be found elsewhere [Oli05b]; it is similar to that presented by Roscoe [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. We merge the states (*MSt*) when both actions terminate: for every local variable and state component *v*, if it is declared in ns_1, its final value is that of A_1; if, however, it is declared in ns_2, its final value is that of A_2; finally, if it is declared in neither ns_1 nor ns_2, its value is left unchanged.

The function *MTrPar* (parallel trace merge) presented below encodes the function $\|_{cs}$. It receives a pair of traces $Pair(tr_1, tr_2)$ and a set of events $Set(cs)$ and returns a set $Set(s)$ containing all the possible sequences of events $Seq(e)$ in which e is in the set of combinations of tr_1 and tr_2 according to *cs*, that is, e is a possible synchronisation sequence of events. It uses a similar function $\_ \llbracket \__Z \rrbracket_Z \_$ that is defined for Z sequences, rather than pairs of traces; it is defined in the theory *utp-z-library*.

$$
\begin{array}{|l}
MTrPar : PAIR\_SEQ\_EVENT\_VAL \times SET\_EVENT\_VAL \rightarrow SET\_SEQ\_EVENT\_VAL \\
\hline
\forall ps : PAIR\_SEQ\_EVENT\_VAL;\ cs : SET\_EVENT\_VAL \bullet \\
\quad MTrPar(ps, cs) = \\
\qquad Set(\{\, e : ((Seq^{\sim})((Pair^{\sim})\, ps).1) \llbracket_Z ((Set^{\sim})cs) \rrbracket_Z ((Seq^{\sim})((Pair^{\sim})\, ps).2) \bullet Seq(e) \,\})
\end{array}
$$

Another function, *MTrParPred*, has a rather long, but trivial, encoding. It receives a set of events *cs* and returns the encoding of the predicate $tr' - tr \in (1.tr - tr \parallel_{cs} 2.tr - tr)$.

Two other predicates *BranchesWaiting* and *BranchesNotWaiting* are defined in order to make the final definition of the merge function more readable: the former encodes the predicate $1.wait \vee 2.wait$ and the latter encodes the predicate $\neg\, 1.wait \wedge \neg\, 2.wait$. Yet another function, which has a rather long but simple definition, is *MRefPar*: it receives a set of events *cs* and returns the encoding of the following predicate:

$$tr' - tr \in (1.tr - tr) parallel_{cs} 2.tr - tr$$

Finally, the recursive function *MSt* returns a predicate that corresponds to the state merge. It receives three sets of names: the set *st* corresponds to the state components, and the sets *ns*1 and *ns*2 correspond to the

names in the left side and right side partitions of the parallel composition, respectively. By way of illustration, given a state $st = \{x, y, z\}$ and partitions $ns1 = \{x\}$ and $ns2 = \{y\}$, the predicate $MSt(st, ns1, ns2)$ is $x' = 1.x \land y' = 2.y \land z' = z$. In summary, the conditional expression presented in the merge function $M_{\parallel_{cs}}$ is encoded as follows.

$$
\begin{array}{|l}
MWtRefStPar : SET\_EVENT\_VAL \times ALPHABET \times ALPHABET \times ALPHABET \\
\qquad\qquad \to REL\_PRED \\
\hline
\forall ns1, ns2, st : ALPHABET;\ cs : SET\_EVENT\_VAL \bullet \\
\quad MWtRefStPar(cs, st, ns1, ns2) = BranchesWaiting \land_R MRef Par(cs) \\
\qquad\qquad\qquad\qquad \lhd_R =_R (\{dash(wait)\}, dash(wait), Val(Bool(true))) \rhd_R \\
\qquad\qquad\qquad\qquad BranchesNotWaiting \land_R MSt(st, ns1, ns2)
\end{array}
$$

It receives the synchronisation channel set cs, the set of names st of the state components, and the sets of names that correspond to the partitions $ns1$ and $ns2$. If the parallel combination is still waiting, then at least one of the branches is still waiting (*BranchesWaiting*), and the refusal set is defined by the function *MRefPar*; otherwise, both branches have terminated (*BranchesNotWaiting*) and the state is merged accordingly (*MSt*).

The merge function $M_{\parallel_{cs}}$ is encoded as follows.

$$
\begin{array}{|l}
MPar : SET\_EVENT\_VAL \times ALPHABET \times ALPHABET \times ALPHABET \to REL\_PRED \\
\hline
\forall ns1, ns2, st : ALPHABET;\ cs : SET\_EVENT\_VAL \bullet \\
\quad MPar(cs, st, ns1, ns2) = MTrParPred(cs) \land_R MSync(cs) \land_R MWtRefStPar(cs, st, ns1, ns2)
\end{array}
$$

It receives the same arguments as the function *MWtRefStPar* and returns the conjunction of the trace merge (*MTrParPred*), the predicate $MSync(cs)$, and the conditional expression described above.

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

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

The function that encodes the parallel composition receives two processes $a1$ and $a2$ with the same alphabet, the two partitions $ns1$ and $ns2$, which must be disjoint and contain only *undashed* names, and the synchronisation channel set cs. The parallel composition diverges if it is possible for either of the actions to diverge; this is expressed in the precondition of the resulting reactive design by using *DivPar* as follows.

$$
\begin{array}{|l}
\_ \llbracket \_ \rrbracket_C \_ : CSP\_PROCESS \times (ALPHABET \times SET\_EVENT\_VAL \times ALPHABET) \times \\
\qquad\qquad CSP\_PROCESS \to CSP\_PROCESS \\
\hline
\forall a1, a2 : CSP\_PROCESS;\ cs : SET\_EVENT\_VAL;\ ns1, ns2 : ALPHABET \bullet \\
\quad a1 \llbracket_C (ns1, cs, ns2) \rrbracket_C a2 = \\
\qquad R(\ (\neg\ _R(DivPar(a1, a2, cs)) \land_R \neg\ _R(DivPar(a2, a1, cs))) \\
\qquad\quad \vdash_D \\
\qquad\quad (((((a1\ \sigma_t\ \omega_f);_C U(one, out\_a(a1.1))) \land_R ((a2\ \sigma_t\ \omega_f);_C U(two, out\_a(a2.1)))) \\
\qquad\qquad +_R(\{tr\} \cup (a1.1 \setminus (ALPHABET\_OWTR \cup dashed)))\ );_C \\
\qquad\quad (MPar(cs, a1.1 \setminus (ALPHABET\_OWTR \cup dashed), ns1, ns2)))\ )
\end{array}
$$

We use the function U to relabel the final values of the execution of actions $a1$ and $a2$; the relabelling functions *one* and *two*, respectively, are used as argument. Furthermore, we extend the alphabet of the resulting predicate with *tr* and the state components; these are all the names that are in the alphabet of $a1$ which are neither a UTP observational variable nor *dashed*. Finally, we sequentially compose the parallel execution of both actions with the merge function *MPar*.

Although rather long, the encoding of the parallel composition has a direct correspondence to its semantics. What we needed to do was to clarify and make explicit the alphabet restrictions on the components.

Assignments. The *Circus* assignment is reactive, and hence, it needs a new definition different from that of the relational assignment or that given purely as a design. The *Circus* assignment also receives a *homogeneous* alpha-

bet a that contains at least all the UTP observational variables and their *dashed* counterparts, a sequence ns of names and a sequence $exps$ of expressions. All the names in ns and free variables in $exps$ must be *undashed*, belong to a, but cannot be an observational variable; both lists ns and $exps$ have the same length. The set of tuples $(a, ns, exps)$ that satisfy these conditions is WF_Assign_C.

The reactive design that is returned in the definition below has $True_R(a)$ as its precondition; its postcondition states that the trace is left unchanged and that the final value of *wait* is *false*. Furthermore, we use the relational assignment to express the change of the state components.

$$
\begin{array}{|l}
Assign_C : WF\_Assign_C \rightarrow CSP\_PROCESS \\
\hline
\forall a : ALPHABET;\ ns : \mathrm{seq}\ VAR\_NAME;\ exps : \mathrm{seq}\ EXP \mid (a, ns, exps) \in WF\_Assign_C \\
\quad \bullet\ Assign_C(a, ns, exps) = R(\ True_R(a) \vdash_D =_R (a, dash(tr), Var(tr)) \\
\qquad\qquad\qquad\qquad \wedge_R =_R (a, dash(wait), Val(Bool(false))) \\
\qquad\qquad\qquad\qquad \wedge_R\ Assign_R(a, ns, exps))
\end{array}
$$

Assignments could change the trace, if the observational variables were allowed in assignments, which is not the case. This also applies to specification statements and schema expressions as we explain in the sequel.

Specification statements. The function that encodes the specification statement $f : [preC, postC]$ receives a *homogeneous* alphabet a that contains, among other variables, all the UTP observational variables and their *dashed* counterparts, a finite set f of names, the precondition $preC$, and the postcondition $postC$. The set $WF\_SpecStatement_C$ is the set of all quadruples $(a, f, preC, postC)$ such that: every name in f is *undashed*, different from all of the UTP observational variables, and belongs to a; and the alphabets of $preC$ and $postC$ are equal to a.

$$
\begin{array}{|l}
SpecStatement_C : WF\_SpecStatement_C \rightarrow CSP\_PROCESS \\
\hline
\forall a : ALPHABET;\ f : \mathbb{F}\ VAR\_NAME;\ preC : CIRCUS\_COND;\ postC : CIRCUS\_PRED \\
\quad \mid (a, f, preC, postC) \in WF\_SpecStatement_C \\
\quad \bullet\ SpecStatement_C(a, f, preC, postC) = \\
\qquad R(\ preC \vdash_D =_R (a, dash(tr), Var(tr)) \\
\qquad\qquad \wedge_R =_R (a, dash(wait), Val(Bool(false))) \wedge_R postC \\
\qquad\qquad \wedge_R \Pi_R(a \setminus ((f \cup \{n : f \bullet dash(n)\}) \cup ALPHABET\_OWTR)))
\end{array}
$$

The reactive design that is returned has $preC$ as its precondition; on termination, the specification statement does not change the trace. Furthermore, it also establishes the postcondition $postC$. Finally, the specification statement cannot change any variable that is not in the frame f; we encode this property using the relational skip on the alphabet that does not contain any observational variable or any variable that is in the frame (and their *dashed* counterparts).

Schema expressions. In *Circus*, we can also describe operations using schema expressions; they can be used in the same context as specification statements. The reason for offering both is that specification statements are important for the refinement calculus and schema expressions gives us the advantages inherent to using Z. As a matter of fact, the semantics of schema expressions is given by normalisation and transformation into a specification statement [Oli05b].

We assume that schema expressions have already been normalised. Furthermore, a very important aspect is implicitly considered by Oliveira's original semantics [Oli05b] and must be made explicit in the automation: the typing of the declared variables. The function *Typing* receives a list of variable declarations and an alphabet, and returns the encoding of a conjunction of predicates: for each variable declaration $n : T$, it contains the encoding of a predicate $n \in T$. The first argument of the function *Typing*, the variable declaration, is a pair of lists: the first element is the list of variable names, which must be elements of the alphabet a, and the second element is the list of types; both lists must have the same size. These restrictions are captured by the type VAR_DECLS, whose cartesian product with $ALPHABET$ is the domain of *Typing*.

$$Typing : VAR\_DECLS \times ALPHABET \nrightarrow REL\_PRED$$

The set of well-formed schema expressions, $WF\_SchemaExp_C$, contains all pairs $(decls, p)$, where $decls$ are well-formed variable declarations, and p is a predicate, such that the set of variables that are declared in $decls$ is equal to the alphabet of p after removing the names in $ALPHABET\_OWTR$. A schema expression is defined

using a specification statement: the alphabet contains all the declared variables and the observational variables, the frame contains the *undashed* versions of all the *dashed* declared variables, the precondition is the existential quantification of the *dashed* variables, where the predicate also includes the typing restrictions of the variables, and the postcondition is the conjunction of the typing restrictions of the variables and p.

$$
\begin{array}{|l}
SchemaExp_C : WF\_SchemaExp_C \to CSP\_PROCESS \\
\hline
\forall\, decls : VAR\_DECLS;\ p : REL\_PRED \mid (decls, p) \in WF\_SchemaExp_C \\
\quad \bullet\ \exists f : \mathbb{F}\, VAR\_NAME \mid f \subseteq undashed \land \mathrm{ran}(decls.1 \rhd dashed) = dash(\!| f |\!) \\
\qquad \bullet\ SchemaExp_C(decls, p) = SpecStatement(\ \mathrm{ran}(decls.1) \cup ALPHABET\_OWTR, f, \\
\qquad\qquad\qquad\qquad \exists_R(\mathrm{ran}(decls.1) \setminus undashed, Typing(decls, p.1) \land_R p), \\
\qquad\qquad\qquad\qquad Typing(decls, p.1) \land_R p\ )
\end{array}
$$

Alternation. Three auxiliary recursive functions are used in the encoding of the last command presented in this section, the alternation. The three of them receive an element of the type *G_ACTIONS* as argument. This type contains all the pairs of finite lists with the same length, in which the first element is a list of *Circus* conditions and the second element is a list of actions. For example, the guarded actions $g_1 \to A_1 [\!] g_2 \to A_2$ is represented in our mechanisation as the pair $(\langle g_1, g_2\rangle, \langle A_1, A_2\rangle)$. The first function, *TrueGuards*, encodes $\bigvee i \bullet g_i$; it returns the disjunction of all guards in the first list. The function *NonDivActions* encodes $\bigwedge i \bullet g_i \Rightarrow \neg\ A_{if}^{f}$. Finally, *ExecActions* encodes $\bigvee i \bullet g_i \land A_{if}^{t}$.

An alternation does not diverge if at least one of the guards is *true* and if every action guarded by a *true* guard does not diverge (functions *TrueGuards* and *NonDivActions*, respectively). When it terminates, it establishes the result of executing one of the actions that are guarded by a *true* guard (*ExecActions*).

$$
\begin{array}{|l}
if_C \_ fi_C : G\_ACTIONS \to CSP\_PROCESS \\
\hline
\forall\, gactions : G\_ACTIONS \bullet if_C\ gactions\ fi_C = R(\ TrueGuards(gactions) \land_R NonDivActions(gactions) \\
\qquad\qquad\qquad\qquad\qquad\qquad \vdash_D ExecActions(gactions)\ )
\end{array}
$$

In order to simplify proofs, we also provide a simpler binary alternation.

Further constructs. The encoding of external and internal choice is trivial and derive directly from their corresponding CSP operators presented in Sect. 3.5.

It is a direct and important consequence of our definition of external choice 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 $(x := 0; c_1 \to Skip) \Box (x := 1; c_2 \to Skip)$. This choice is not resolved instantly; it is only resolved when either c_1 or c_2 happens. The final value of x depends on which communication 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 encoding of interleaving ($\_ [\![[\![\_]\!]]\!]_C \_$), hiding ($\_ \setminus_C \_$), and parameterised actions ($param_C$) have a direct correspondence to their semantics. Furthermore, the encoding of recursion (μ_C) is trivially defined in terms of the weakest fixed point described in Sect. 3.1.

The semantics of all *Circus* processes are given as syntactic transformations from the process definition to some *Circus* action. As discussed in Sect. 3.1, we are working directly with the semantics of terms; hence, we may not be able to express some of the syntactic transformations directly. Therefore, in order to encode the semantics of *Circus* processes, it may be the case that new concepts like the unrestricted variables in Sect. 3.1 need to be investigated and used to encode the semantics of processes; this is left as future work.

Healthiness conditions. Processes that can be defined using the notation of CSP satisfy other healthiness conditions. One of them, **CSP3**, requires that the behaviour of a process does not depend on the initial value of *ref* ($\neg\ wait \Rightarrow P = \exists ref \bullet P$); 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, the value of ref' should have no relevance after termination of **CSP4** processes. Finally, a deadlocked **CSP5** process that refuses some events offered by its environment will still be deadlocked in an environment that offers even fewer events.

Both, **CSP4** and **CSP5**, are expressed in terms of CSP constructs that have a slightly different definition in *Circus*[1]: **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 $\mathbf{C1}(P) \mathrel{\widehat=} P$; *Skip* and $\mathbf{C2}(P) \mathrel{\widehat=} P \llbracket \alpha(P) \mid \varnothing \rrbracket$ *Skip* have a direct correspondence with them; they lift these two healthiness conditions to state-rich *Circus* processes. For instance, we present below the encoding of **C1**.

$$\frac{C1 : CSP\_PROCESS \to CSP\_PROCESS}{\forall a : CSP\_PROCESS \bullet C1(a) = a \mathbin{;_R} Skip(a.1)}$$

As for the other healthiness conditions, the set of relations that satisfy **C1** is the set of relations a such that $a = C1(a)$, as we present below.

$$\frac{C1\_healthy : \mathbb{P}\, CSP\_PROCESS}{C1\_healthy = \{a : CSP\_PROCESS \mid a = C1(a)\}}$$

The encoding of **C2** and the set of **C2**-healthy processes follow a similar approach.

The encoding of the last of the *Circus* healthiness conditions, $\mathbf{C3}(A) \mathrel{\widehat=} \mathbf{R}(\neg\ A^f_f;\ true \vdash A^t_f)$, is presented below. It guarantees that every *Circus* action, when expressed as a reactive design, has no dashed variables in the precondition.

$$\frac{C3 : CSP\_PROCESS \to CSP\_PROCESS}{\forall a : CSP\_PROCESS \bullet C3(a) = R((\neg\ a\, \sigma_f\, \omega_f) \mathbin{;_R} True_R(a.1) \vdash_D (a\, \sigma_t\, \omega_f))}$$

Since *Circus* actions are **CSP1**-**CSP2** healthy, we can transform them into reactive designs [HJ98]; if they are originally already expressed so, this transformation has no effect whatsoever. 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 $C3$.

5. Conclusions

In this paper we give a set-based model for UTP relations, and use it as a basis for the development of five theories: relations, designs, reactive processes, CSP, and *Circus*. This is a basis for theorem provers for the UTP, for *Circus* and its extensions, and for other languages whose semantics is based on the UTP. We have proved, in total, over 500 theorems as part of the various theories. The theorem prover that we used, ProofPower-Z, has a very expressive tactic language, an extensive library of tactics, and has been successfully used in industry.

For us, a relation is a pair, whose first element is a set of names (alphabet) and whose second element is a set of functions from names to values. This is not the only possible model for relations. Our choice was based on the fact that any restriction that applies to the relations has a direct impact on the complexity of the proofs. Our model imposes a simple restriction that results in simpler definitions, and hence proofs.

In related work [CWD06], we defined a relation as a pair formed by an alphabet and a set of pairs of bindings: for every pair (b_1, b_2) of bindings, the domain of b_1 has only *undashed* names and that of b_2 only *dashed* names. Such a restriction has to be enforced by the definition of every operator. There is, however, an isomorphism between our current model and this earlier one [CWD06]. By joining and splitting the sets of bindings, we can move from one model to another; our concern is only with the practicality of mechanical theorem proving.

We also could have used bindings whose domains could be different from the alphabet of the relation. However, the alphabet is the set of names about which the relation describes something. Hence, the alphabet a of a relation would have to be either a subset or equal to the domain of each binding b. Values of names that were not in the alphabet would have no meaning. We chose bindings whose domain is the alphabet because, by taking the other approach, we have a more complex definition for alphabet extension: bindings for names that are not in the alphabet need to be removed before being left unrestricted. Alphabet extension is at the heart of the definitions of conjunction, disjunction, and parallelism.

If, in the hope of finding simplifications in other points, we accepted the more complex definition of alphabet extension, then we would need to determine how to handle the names that are not in the alphabet of the

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

Table 1. Theories mechanisation

	Number of theorems	Lines of proof script
utp-z-library	17	702
utp-rel	306	16,874
utp-okay	19	908
utp-des	46	2,397
utp-wtr	27	1,158
utp-rea	81	3,199
utp-csp	13	893
utp-circus	6	1,473
Total	515	27,604

relation: bindings could be total functions that map these names to an undefined value; or we could leave these names unrestricted. These restrictions are more complex than ours and lead to more complex definitions and proofs. We have an isomorphism between our model and each of these; by applying a domain restriction to the bindings in these models and extending our model's bindings, we can change the representations.

As an industrial theorem prover, ProofPower-Z proved to be powerful (and helpful). The support provided by hundreds of built-in tactics and theories, such as libraries for Z constructs and set theory, made our work much simpler. The axiomatisation of the theorems proved in our work in other theorem provers, like Z/Eves [Saa97], and the development of new theories based on these axioms makes the use of our results in different theorem provers possible. In ProofPower-Z, the tactics that can be created are more powerful than in Z/Eves; however, the level of expertise needed for initial users of Z/Eves is not as high as for ProofPower-Z.

This discussion about alternative models is based on our experience with ProofPower-Z; some of them could make proofs easier in another theorem prover. This investigation is a topic for future research.

In other work, Nuka and Woodcock [NW04] formalised the alphabetised relational calculus in the theorem prover Z/EVES. They did not restrict the set of bindings in the same way that 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. Here, we extend these previous definitions [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 investigations. In another paper [NW06], Nuka and Woodcock [NW04] present the same mechanisation as in their previous work but in ProofPower-Z. They also extend their previous work [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; as briefly discussed in Sect. 2, this makes it harder to extend their specification language.

Hoare and He [HJ98], although dealing with alphabetised predicates, often leave the alphabet quite implicit. For example, *true* is often seen unalphabetised, while in fact, it is alphabetised. This abstraction simplifies things, but is not suitable for theorem provers. With the obligation to deal with alphabets, our work gives more details on how the alphabets are handled within the UTP.

The UTP's alphabet extension constrains the values of the new variables: they cannot be changed. However, our set-based model for relations needs a different alphabet extension that leaves their values unconstrained. Furthermore, in the UTP, existential quantifications are used in two different ways: in the definition of variable blocks, the authors explicitly state that the quantified variables are removed from the alphabet; and in the definition of *SKIP*, the alphabet is, implicitly, left unchanged. Our encoding defines two existential and two universal quantifications: one of them removes the quantified variables from the alphabet, and the other one does not. The redefinition of some UTP definitions resulted in easier proofs.

Our work also reveals details that are left implicit in the UTP regarding the domain of the healthiness conditions. By mechanising **R3**, for instance, we make it explicit that this healthiness condition, and consequently **R**, is a partial function that can only be applied to *homogeneous* reactive relations.

We expressed the language constructors as functions. For this reason, they can be extended without losing the previous proofs; the syntax of expressions was abstracted by using five simple definitions: values, variables, relations, unary functions, and binary functions. Furthermore, the strategy that we adopted for lifting Z functions and relations to relational predicates, for instance $\leqslant_R$, makes the Z toolkit directly available in our theory.

In the mechanisation of the CSP and *Circus* theories, some expressions proved to be non-trivial. For instance, in the encoding of the function do_C, the expressions regarding the refusal set and the increment of the trace, and

the representation of events were not trivial. Our strategy for lifting Z functions and relations to values proved to be of much use in both cases.

The current number of laws on sequential composition may need to be expanded to allow users of our theory of relations not to expand its definition in the proof of theorems. The proof of more laws on sequential composition that will make this possible is an important piece of future work.

We are currently developing the mechanical proofs of the *Circus* refinement laws. A few proofs have already been done; they are based on the encoding of the *Circus* semantics presented in this paper. Table 1 presents a summary of the effort needed so far. During these proofs, we are investigating and implementing proof tactics that can be used to reduce considerably the proof scripts. Furthermore, the proofs done so far raised subtleties on alphabets and free variables that were left implicit during the hand proofs, but needed to be made explicit during mechanical proofs. Mostly, they involve properties on the healthiness conditions (specially on reactive and csp processes). For instance, properties of expressions like $tr \leqslant tr'$, that are used in the definitions of the healthiness conditions and the *Circus* operators, properties of substitutions, conditional expressions, 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.

Circus is not the first specification language of concurrent systems that has its semantics mechanised in a theorem prover. The CSP traces model and failures-divergences model have already been encoded in HOL [Cam90a, Cam90b]. In both cases, Camilleri also proved some standard CSP laws based on his encodings. The CSP trace semantics has also been embedded in PVS [DS97] and used to verify the correctness of a verification protocol. Their CSP variant also differs: Dutertre and Schneider [DS97] mechanise Roscoe's CSP [Ros98] whereas Camilleri [Cam90a] mechanises Hoare's CSP [Hoa85]. Because of the state-rich nature of *Circus* processes, the encoding of its semantics requires the encoding of a semantic model that is able to combine the notions of refinement for CSP and for imperative programs. For this reason, we encoded the *Circus* semantics based on our previous mechanisation of the UTP [OCW06b]. In [CW06], we relate our model to Roscoe's standard model.

Our aim is to provide a mechanisation of the UTP that can support the development of other languages whose semantics is based on the UTP. *Circus* is such a language, and is the first one to use our mechanisation of the UTP. In the future, we intend to complete the automation of the proofs of the *Circus* refinement laws. This will provide *Circus* with a mechanised refinement calculus that can be used in the formal development of state-rich reactive programs; it will be the basis for a refinement editor and a theorem prover for *Circus*.

Acknowledgements

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.

References

[Art] Arthan R PowerProof Reference Page. http://www.lemma-one.com/ProofPower/index/index.html

[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

[CCO05] Cavalcanti ALC, Clayton P, O'Halloran C (2005) Control law diagrams in *Circus*. In: Fitzgerald J, Hayes IJ, Tarlecki A (eds) FM 2005: formal methods symposium, Vol 3582 of LNCS. Springer, Heidelberg, pp 253–268

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

[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

[CWD06] Cavalcanti ALC, Woodcock JCP, Dunne S (2006) Angelic nondeterminism in the unifying theories of programming. Formal Asp Comput 18(3):288–307

[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, Murray Hill, August 1997. 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, New York, pp 423–438
[GM93] Gordon MJC, Melham TF (eds) (1993) Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, Cambridge
[GMW79] Gordon M, Milner R, Wadsworth C (1979) Edinburgh LCF, Vol **78** of LNCS. Springer, Heidelberg
[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
[ISO02] ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics, 2002. International Standard
[KAW96] King DJ, Arthan RD, Winnersh ICL (1996) Development of practical verification tools. ICL Syst J 11(1)
[Mor94] Morgan C (1994) Programming from specifications. Prentice-Hall, Englewood Cliffs
[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) Refine—international refinement workshop, eletronic notes in theoretical computer science. Springer, Heidelberg (to appear)
[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, Heidelberg, pp 123–140
[Oli05a] Oliveira MVM (2005) Formal derivation of state-rich reactive programs using *circus*—additional material. 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
[Pau91] Paulson LC (1991) ML for the Working Programmer. Cambridge University Press, Cambridge
[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
[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
[SJ02] Sherif A, Jifeng H (2002) Towards a time model for *Circus*. In: George C, Miao H (eds) Formal methods and software engineering: 4th international conference on formal engineering methods, ICFEM 2002, Vol 2495 of LNCS. Springer, Heidelberg, pp 613–624
[SS99] Seres S, Spivey MJ (1999) Embedding prolog into haskell. In: Haskell Workshop'99, Sep 1999
[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, New York, 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
[WC01] Woodcock JCP, Cavalcanti ALC (2001) *Circus*: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building,
[WD96] Woodcock JCP, Davies J (1996) Using Z—specification, refinement, and proof. Prentice-Hall University Press, Englewood Cliffs
[WH02] Woodcock JCP, Hughes A (2002) Unifying theories of parallel programming. In: George C, Miao H (eds) Formal methods and software engineering: 4th international conference on formal engineering methods, ICFEM 2002, Vol 2495 of LNCS. Springer, Heidelberg, pp 24–37

Received 31 October 2006
Revised 9 May 2007
Accepted 25 June 2007 by S E Dunne and T S E Maibaum
Published online 3 August 2007