Unifying Theories in ProofPower-Z

Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock

Department of Computer Science, The University of York
Heslington, York, YO10 5DD, United Kingdom

Abstract. The increasing interest in the combination of different computational paradigms is very well represented by Hoare & He in the *Unifying Theories of Programming.* 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. An account of how this mechanisation is done, and more interestingly, of what issues were raised and of our decisions, is presented here. We aim at providing tool support for further explorations of Hoare & He's unification, and for the mechanisation of languages based on this unification. More specifically, *Circus*, a specification language that combines Z, CSP, specification statements, and Dijkstra's guarded command language is our final target.

Keywords: Unifying Theories of Programming, theorem prover.

1 Introduction

Researchers have concentrated their interest in the combination of programming paradigms, which consider different aspects and stages of software development. Hoare & He did one of the most significant works towards unification [9]. 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 sequential processes and their designs.

Following this trend of research, *Circus* [21,2] combines a model-based language, Z [22], a process algebra, CSP [8], Dijkstra's language of commands, and specification statements [10]. It differs from other combinations [19,16,5,20] in that it has an associated refinement theory [2,14,13]. The mechanical proof of more than one hundred refinement laws requires the mechanisation of the *Circus* semantics, and will be the basis for its theorem prover. In previous work [21], we define a Z semantics for *Circus*. Although usable for reasoning about systems specified in *Circus*, it is not appropriate to prove properties of the language itself.

In early work [18], Sherif and He present a time model for *Circus*. Qin *et.al.* [15] used the UTP to formalise the semantics of TCOZ and capture some of its new features for the first time. This semantics is being used as a reference document in the development of tools for TCOZ and as a semantics foundation for proving soundness of these tools. Woodcock and Hughes use the UTP model [23] in order to give a formal semantics to a programming language that contains shared variables. Our work provides mechanical support not only to *Circus*, but also to any language that has the UTP as its theoretical basis.

In recent work [3], we summarise the alphabetised relational calculus, and the theory of precondition-postcondition specifications, called designs. A detailed theory for reactive processes is presented, and then combined with the theory of designs, to provide the model for CSP. By mechanising the theories of reactive processes and CSP, we enable a further exploration on these results.

We present here the first step towards mechanising the *Circus* semantics and the proof of its refinement laws: the mechanisation of the UTP in the theorem prover ProofPower-Z [1]. The definitions of the theories of relations, designs, reactive processes, and CSP, and more than three-hundred and seventy theorems, is the result of our work. Many issues arose from the existence of an alphabet and from our intention of proving refinement laws; we discuss them here.

Section 2 presents the UTP and ProofPower-Z. In Section 3, we discuss design issues and describe the theory hierarchy we created. Section 4 describes the mechanisation of the UTP relations, designs, reactive processes, and CSP. The proof of one theorem illustrates our approach. Finally, in Section 5, we draw our conclusions and describe future work.

2 Preliminaries: UTP and ProofPower-Z

The UTP is a framework based on an alphabetised extension of Tarski's relational calculus. Every program, design, and specification is interpreted in the UTP as a relation between an initial observation and a single subsequent observation, which may be either an intermediate or a final observation of the behaviour of a program execution. The relations are defined as predicates over observational variables. The initial observations of each variable are undecorated, and subsequent observations are decorated with a dash.

Several theories share common ideas; sequential composition, conditional, nondeterminism, and parallelism are some of them. Refinement is interpreted as inclusion of relations: reverse implication. Every relation is a pair $(\alpha P, P)$, where αP is the alphabet: set of observational variables that can be free in the predicate P. Healthiness conditions are used to test a specification or design for feasibility, and reject it, if it makes implementation impossible in the target language. They are often expressed in terms of an idempotent function ϕ that makes a program healthy. Every healthy program P must be a fixed point $P = \phi\, P$).

Figure 1 presents how some UTP theories [9] are related. Relations are predicates with an input and an output (dashed) alphabet. Designs are specifications written in terms of pre and postconditions. Reactive processes are programs whose behaviour may depend on interactions with an environment. Finally, 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.

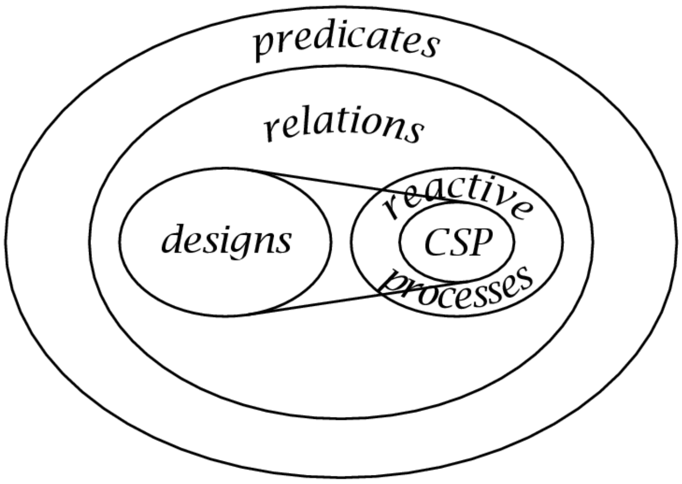

Fig. 1. Theories in the UTP

ProofPower-Z is a higher-order tactic based theorem prover implemented using New Jersey SML, that supports specifications and proofs in Z. It extends ProofPower-HOL, which builds on ideas arising from research at the Universities of Cambridge [7] and Edinburgh [6]. 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 the representation of such expressions.

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, ProofPower-Z 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 in ProofPower-Z is almost the same as that of the Z standard. We explain the differences as needed.

ProofPower-Z comes with a large number of verified theories. However, as it supports a powerful logic, the level of automation is lower than in theorem provers that support, for example, first-order logic. On the other hand, it has been successfully used in industry, and was a natural choice as a basis for a *Circus* theorem prover, as it is routinely used by our industrial partner: QinetiQ.

3 Design Issues

This section describes the issues raised during 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 make explicit the difference between a variable name and a variable value.

We discarded the option of giving an axiomatic semantics to relations, since 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. Our relations are pairs of sets.

Since we want to prove refinement laws, our mechanisation gives the possibility of expressing and proving meta-theorems. 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. The deep-embedding has the additional advantage of providing the possibility of introducing new predicate combinators.

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 as a partial ($\nrightarrow$) function $f : PRED \nrightarrow PRED$. 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. 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. 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 as our meta and target language.

In Figure 2, we present our hierarchy of theories. In order to handle sequences, we extend the ProofPower-Z's theory *z-library*; the result is *utp-z-library*. 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 operator definitions and their laws.

Two theories inherit from *utp-rel*: *utp-okay* is concerned with an observational variable *okay*, and *utp-wtr* with *wait*, *trace*, and *ref*. These are the main 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 the parent 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*; it is under development. Our proofs of the laws of a theory does not expand definitions of its parent theory; it uses the parent's laws. This provides modularisation and encapsulation.

4 Mechanisation

In this section we describe in detail our ProofPower-Z theories. For the sake of

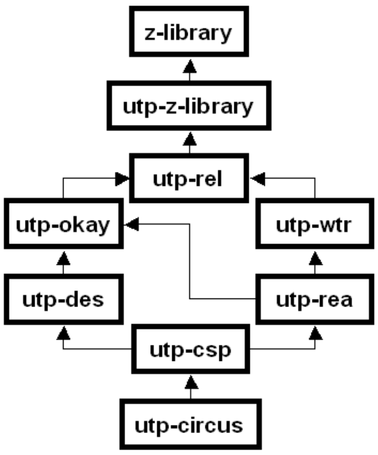

Fig. 2. Theories in the UTP

presentation, we do not present the Z 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.

4.1 Relations

A name is an element of the 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 return the input and the output alphabets of a given alphabet. All the definitions and proof scripts can be found elsewhere [12].

An alphabet a in which $n \in a \Leftrightarrow n' \in a$, for every undashed name n, is called *homogeneous*. For us, n' is mechanised as $dash\, n$. Similarly, a pair of alphabets $(a1, a2)$ is *composable* if $n \in a2 \Leftrightarrow n' \in a1$, for every undashed name n.

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

$$\begin{aligned} VAL ::= {} & Int(\mathbb{Z}) \mid Bool(BOOL) \mid Set(\mathbb{P}\ VAL) \mid Seq(\mathrm{seq}\ VAL) \\ \mid {} & Pair(VAL \times VAL) \mid Channel(NAME) \mid Sync \end{aligned}$$

In ProofPower-Z, $Bool(BOOL)$ stands for the Z constructor $Bool\langle\!\langle BOOL\rangle\!\rangle$, which introduces a collection of constants, one for each element of the set $BOOL$. The ProofPower-Z type $BOOL$ is the booleans. The type VAL can be extended without any impact on the proofs.

Although we are defining an untyped theory, the observational variables have types; for instance, *okay* is a boolean. For this reason, we specify some types; for instance, booleans are in the set $BOOL\_VAL \mathrel{\widehat=} \{Bool(true), Bool(false)\}$, channels are in the set $CHANNEL\_VAL \mathrel{\widehat=} \{n : NAME \bullet Channel(n)\}$, and events are in the set $EVENT\_VAL \mathrel{\widehat=} \{c : CHANNEL\_VAL;\ v : VAL \bullet Pair(c, v)\}$.

Three definitions allow us to abstract from the syntax of expressions. The set of relations ($\leftrightarrow$) between values is $RELATION \mathrel{\widehat=} VAL \leftrightarrow VAL$. The set of unary functions is $UNARY\_F \mathrel{\widehat=} VAL \nrightarrow VAL$; similarly, for binary functions we have the set $BINARY\_F \mathrel{\widehat=} (VAL \times VAL) \nrightarrow VAL$, which defines the set of partial functions from pairs of values to values. For instance, the sum function is $\{(Int(0), Int(0)) \mapsto Int(0), (Int(0), Int(1)) \mapsto Int(1), \ldots\}$. An expression can be a value, a name, a relation, or a unary or binary function application.

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

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

A binding is defined as $BINDING \mathrel{\widehat=} NAME \nrightarrow VAL$, and $BINDINGS$ is the set of bindings. Given a binding b and an expression e with free-variables in the domain (dom) of b, $Eval(b, e)$ gives the value of e in b (beta-reduction). A relation is modelled in our work by the type REL_PRED defined below. Basically, a relation is a pair: the first element is its alphabet, and the second is a set of bindings, which gives us all bindings that satisfy the UTP predicate modelled by the relation. The domain of the bindings must be equal to the alphabet. Optional models in which this restriction could be relaxed are possible; however, they would lead us to more complex definitions as we discuss in Section 5. The 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, e contains one or more free occurrences of x. The *true* condition and the constructor x may be omitted.

$$\begin{aligned} & REL\_PRED \mathrel{\widehat=} \\ & \quad \{a : ALPHABET;\ bs : BINDINGS \mid (\forall b : bs \bullet \operatorname{dom} b = a) \bullet (a, bs)\} \end{aligned}$$

This follows directly from the definition of alphabetised predicates of the UTP.

In our work, we use Z axiomatic definitions, which introduce constrained objects, to define our constructs. For instance, let us consider the following axiomatic definition.

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

It introduces a new symbol x, an element of s, satisfying the predicate p.

Our first construct represents the truth. For a given alphabet a, $True_R\ a$ is defined as the pair with alphabet a, and with all the bindings with domain a.

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

In our work, we subscript the constructs in order to make it easier to identify to 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.

$$\begin{array}{|l} False_R : ALPHABET \rightarrow REL\_PRED \\ \hline \forall a : ALPHABET \bullet False_R\ a = (a, \varnothing) \end{array}$$

This operator is the main motivation for representing relations as pairs. If we had defined relations just as a set of bindings with the same domain a, which would be considered as the alphabet, we would not be able to tell the difference between $False_R\ a_1$ and $False_R\ a_2$, since both sets would be empty. Besides, it is important to notice the difference between $True_R\ \varnothing$ and $False_R\ \varnothing$: the former has a set that contains one empty set of bindings as its second element, and the latter has the empty set as its second element.

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.

$$\begin{array}{|l} UnrestVar : REL\_PRED \rightarrow \mathbb{P}\ NAME \\ \hline \forall u : REL\_PRED \bullet \\ \quad UnrestVar\ u = \{n : u.1 \mid \forall b : u.2;\ v : VAL \bullet b \oplus \{n \mapsto v\} \in u.2\} \end{array}$$

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; furthermore, $t.n$ refers to the *n-th* element of a tuple t.

All usual predicate combinators are defined. Conjunctions and disjunctions extend the alphabet of each relation to the alphabet of the other. The function $\oplus_R$ is alphabet extension; the values of the new variables are left unconstrained. In the following definition we make use of the Z domain restriction $A \lhd R$: it restricts 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.

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

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

$$\_ \wedge_R \_ : REL\_PRED \times REL\_PRED \rightarrow REL\_PRED$$

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

The definition of disjunction is similar, but the union of the extend set of bindings is the result. We have proven that these definitions are idempotent, commutative, and associative, and that they distribute over each other. We have also proven 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. For example, we have the unit law for conjunction. The ProofPower-Z output notation $n \vdash t$ gives name n to a theorem t. Besides, in Z, the quantification $\forall\, x : a \mid p \bullet q$ corresponds to the predicate $\forall\, x : a \bullet p \Rightarrow q$.

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

As expected, the conjunction of a relation u with $True_R$ is u, but the alphabet of $True_R$ must be a subset of the alphabet of u. Otherwise, the conjunction may have an alphabet other than that of u 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 the sake of conciseness, we omit the trivial definitions of implication ($\_ \Rightarrow_R \_$), equivalence ($\_ \Leftrightarrow_R \_$), conditional ($\_ \lhd_R \_ \rhd_R \_$), that can be trivially be defined in terms of the previously defined operators.

The function $-_R$ removes variables from the alphabet of a relation using domain anti-restriction (domain removal) to remove names from the set of bindings. It is defined as $u -_R a = (u.1 \setminus a, \{b : u.2 \bullet a \ntriangleleft b\})$. Complementary to domain restriction, the domain anti-restriction $A \ntriangleleft R$, 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.

$$\exists_{-R} : (ALPHABET \times REL\_PRED) \rightarrow REL\_PRED$$

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

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

In the definition of the CSP *SKIP*, Hoare and He seem to use another existential quantification, in which the quantified variables are not removed from the alphabet. We define this new quantifier $\exists_R(a, u)$ as $(\exists_{-R}(a, u)) \oplus a$. Basically, we remove the quantified variables from the alphabet and include them again, leaving their values unrestricted.

Our sequential composition $u1;\ u2$ is not defined as in the UTP [9], an existential quantification on the intermediary state; the motivation is simplifying our proofs. In the UTP definition [9], the existential quantification is described using new 0-subscripted names to represent the intermediate state. Its mechanisation requires two functions: one for creating new names, and another one for expressing substitution of names. Any proof on sequential composition would require induction on both functions.

Relations can only be combined in sequence if their alphabets are *composable*. If we defined sequential composition as a partial function, domain checks would be required during proofs. Instead, we define a total function on well-formed pairs of relations, $\mathit{WF\_Semi}_R$, which have composable alphabets.

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

The alphabet of a sequential composition is composed of the input alphabet of the first relation and the output of the second relation. 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 correspond to their initial values in b_2 are taken into consideration in this combination.

The UTP defines an alphabet extension that enables sequential composition to be applied to operands with non-composable alphabets. The function $+_R$ differs from $\oplus_R$ in that it restricts the value of the new name to be left unchanged. For a given predicate P and name n, it returns the predicate $P \wedge_R (n' =_{\{n',n\}} n)$.

Although useless for practical purposes, the $\mathit{\Pi}$ (skip) is very useful for reasoning about programs. In our work it is defined as the function defined 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 $\mathit{WF\_Skip}_R$ is the set of all *homogeneous* alphabets.

$$
\begin{array}{|l}
\mathit{\Pi}_R : \mathit{WF\_Skip}_R \to \mathit{REL\_PRED} \\
\hline
\forall\, a : \mathit{WF\_Skip}_R \bullet \\
\quad \mathit{\Pi}_R\; a = (a, \{b : \mathit{BINDING} \\
\qquad\qquad \mid \mathrm{dom}\, b = a \\
\qquad\qquad\; \wedge (\forall\, n : a \mid n \in \mathit{undashed} \bullet b(n) = b(\mathit{dash}\; n))\})
\end{array}
$$

Other programming constructs like variable blocks and assignments are also included in this theory; their definitions can also be found in [12].

We now turn to the definition of refinement as the universal implication of relations. The universal closure used in UTP [9] is defined $\langle_R\, u\, \rangle_R = \forall_{-R}(u.1, u)$. For a pair of relations (u_1,u_2), such that $(u_1, u_2) \in \mathit{WF\_REL\_PRED\_PAIR}$ (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.

$$\_ \sqsubseteq_R \_ : WF\_REL\_PRED\_PAIR \to REL\_PRED$$

$$\forall u1\_u2 : WF\_REL\_PRED\_PAIR \bullet$$
$$u1\_u2.1 \sqsubseteq_R u1\_u2.2 = \langle_R (u1\_u2.2 \Rightarrow_R u1\_u2.1) \rangle_R$$

We have proved that our interpretation of refinement is, as expected, a partial order [12]. 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 a function $f : REL\_FUNCTION$ 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 [9], which is not convenient for proofs. However, it is trivial to prove that we have an equivalent definition.

4.2 Proving Theorems

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

The proof of one of our laws is shown in Figure 3: the weakest fixed point law $(\forall F, Y \bullet F(Y) \sqsubseteq Y \Rightarrow \mu F \sqsubseteq Y)$. 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. Furthermore, 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. In our case, 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 explained above in the list of assumptions, we use forward chaining again to state the fact that the alphabet of Y is a.

```
SML
set_goal([], ⓩ∀ 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 ⓩμ_R ⌝]);
a (asm_rewrite_tac[]);
a ( (PC_T1 "initial"
        lemma_tac
        ⓩ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
        ⓩ{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
        ⓩ(a, {u : REL_PRED
              | a = u.1 ∧ F u ⊑_R u = True_R{}})
        ∈ WF_Glb_R_Lub_R ⌝ )
   THEN1
   ((rewrite_tac[z_get_spec ⓩWF_Glb_R_Lub_R ⌝])
        THEN
        (PC_T1 "z_sets_ext" asm_prove_tac[])) );
a (apply_def REL_lower_bound_thm
        ⓩ(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 next step introduces the fact that the set to which Y belongs is in fact a set of *REL_PRED*. The proof of 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. Some of the facilities we highlight are forward chaining, use of existing and user-defined tactics, proof contexts, and automated proof tactics, such as *asm_rewrite_tac*.

4.3 Okay and Designs

The UTP theory of pre and postcondition pairs (designs) introduces an extra observational variable *okay*: it indicates that a program has started, and *okay'* indicates that the program has terminated. In our theory *utp-okay*, we define *okay* as an undashed name ($okay : NAME \mid okay \in undashed$) ranging over the booleans. We restrict the type *BINDING* by determining that *okay* and *okay'* are only associated with boolean values.

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

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

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 $okay \land p \Rightarrow okay' \land q$. The expression *okay* is the equality $okay =_a true$, which is mechanised in our work as $=_R (a, okay, Val(Bool(true)))$. For a given alphabet a, name n, and expression e, such that $n \in a$ and the free-variables of e are in a, the function $=_R (a, n, e)$ returns a relational predicate (a, bs), in which for every binding b in bs, $b\, n = Eval(b, e)$. A design is defined as follows.

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

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 later uses it to define designs. In our work, we have kept both of them, but we underscore the UTP design turnstile with a D.

The most important result for designs, which is the motivation for its 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 and is omitted here.

Designs are also characterised by 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 $h(r) = r$. For instance, $H1\_healthy = \{d : REL\_PRED \mid H1(d) = d\}$.

An $H2\_healthy$ relation does not require non-termination. In previous research [3], we presented a way of expressing $H2$ in terms of an idempotent function: $H2(P) = P;\ J$, where $J \mathrel{\widehat{=}} (okay \land okay' \Rightarrow v' = v)$. 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 presented below.

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

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 [3], we use this definition for $H2$.

More than thirty laws from previous work [9,3], involving design and their healthiness conditions, have been included in our theory of designs. Their proofs do not expand any definition in the relations theory. Many laws were included in the relations theory, in order to carry out proofs in the designs theory.

4.4 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 variable $wait$ records whether the process has terminated or is interacting 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. The definitions of these variables are in the theory *utp-wtr*. In the theory *utp-rea*, we define REA_PRED, the set of relations whose alphabet is a member of $ALPHABET\_REA$. This is the set of alphabets that contain $okay$, tr, $wait$, ref, and their dashed counterparts.

As for designs, healthiness conditions characterise the reactive processes. The first healthiness condition $R1$ states that the history of interactions of a process cannot be changed, therefore, the value of tr can only get longer. Our definition uses a function $\leq_R$ (sequence prefixing), which, is the Z prefixing relation lifted to VALues.

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

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

The definition of $R1$ below mechanises the function $R1(P) = P \wedge tr \leq tr'$.

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

In order to transform the expression $tr \leq tr'$ into a relational predicate, we assert that the expression $\mathit{Rel}((\_ \leq_R \_), \mathit{Var}(tr), \mathit{Var}(\mathit{dash}\, tr))$ is equals to $\mathit{Val}(\mathit{Bool}(\mathit{true}))$. We adopt the same strategy to lift all needed Z relational operators ($\in$,$\notin$,$\subseteq$, ...) and functions (using Fun_1 and Fun_2) to relational predicates.

The second healthiness condition establishes that a reactive process should not rely on events that happened before it started. We mechanise the formulation $R2(P(tr, tr')) = P(\langle\rangle, tr' - tr)$ [3]; 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 implemented using substitution; $R2$ is defined 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. In UTP [9], $R3$ is defined as $R3(P) = \mathit{\Pi}_{REA} \lhd \mathit{wait} \rhd P$, and is mechanised in our work as follows.

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

This definition of $R3$ uses a conditional and the reactive skip $\mathit{\Pi}_{REA}$. Conditionals are defined only if both branches have the same alphabet and $\mathit{\Pi}_{REA}$ is only defined for *homogeneous* reactive alphabets ($\mathit{WF\_Skip}_{REA}$). For this reason, our definition reveals that $R3$ is not a total function: it can only be applied to *homogeneous* reactive relations.

A reactive process is a relation with a reactive alphabet a, which is $\mathit{R\_healthy}$; the function R is defined as $R(r) = R1(R2(R3(r)))$. Based on these definitions, more than sixty laws, including those we presented previously [3], 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 program operators with relation to the healthiness conditions. They also explore relations between healthiness conditions for reactive processes and designs.

4.5 CSP Processes

Our mechanisation of the CSP theory is based on our earlier research [3]. Basically, CSP processes are reactive processes that satisfy two other healthiness conditions; they can all be expressed as reactive designs: the result of applying

R to a design. The first healthiness condition states that the only guarantee in the case of divergence ($\neg$ *okay*) is that the trace can only be extended. It is mechanised as $CSP1\, r \mathrel{\widehat=} r \lor (\neg\, okay \land tr \leq tr')$.

The second healthiness condition is a recast of $H2$, presented in Section 4.3, with an extended reactive alphabet. The mechanisation 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.

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

A $CSP\_PROCESS$ is a $CSP1\_healthy$ and $CSP2\_healthy$ reactive process.

The $SKIP$ process terminates immediately. The initial value of ref is irrelevant, and it is quantified in the definition of $SKIP$.

$$
\begin{array}{|l}
SKIP : CSP\_PROCESS \\
\hline
SKIP = R(\exists_R\, (\{ref\}, \Pi_{REA}\, ALPHABET\_CSP))
\end{array}
$$

The $ALPHABET\_CSP$ contains only *okay*, *tr*, *wait*, *ref*, and their dashed counterparts. The existential quantification does not remove *ref* from the alphabet, as opposed to that used in the definition, for instance, of variable blocks.

Besides the definition for simple prefixing ($e \rightarrow_{CSP} SKIP$, where e is an event) originally given by the UTP, we mechanise a simpler definition which was proven equivalent: $e \rightarrow_{CSP} SKIP = R(true \vdash do_C\ e)$. The following function is a simplified version of do_A presented in the UTP.

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

The simplification was possible because we express prefixing as a reactive design. 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.

By expressing all operators as reactive designs, we bring uniformity to proofs, and foster reuse of existing results. All of our CSP theorems [3] and Hoare and He's UTP theorems [9] are part of our *utp-csp* theory. It is currently being used as a basis of a *Circus* theory.

5 Conclusions

In this paper we give a set-based model to relations, and use it as a basis for the development of four theories: relations, designs, reactive processes, and CSP. For us, a relation is a pair, whose first element is a set that represents its 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: the domain of the bindings must be equal to the alphabet. This restriction results in simpler definitions, and hence proofs. For instance, in [4], 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 in a relation, 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 model and this one. 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 relation alphabet. 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 actually 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 and disjunction.

If, in the hope to find 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 relation. For example, bindings could be total functions which map these names to an undefined value $\perp$; or we could leave these names unrestricted. These restrictions on relations are in fact more complex than that in our model, and lead to more complex definitions and proofs. We also 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, 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 [17], 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.

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

Nuka and Woodcock [11] formalised the alphabetised relational calculus in Z/EVES. We extend that work by including many other operations, such as sequencing, assignment, refinement, and recursion. The hierarchical mechanisation of the theories of designs, reactive processes, and CSP is also a contribution of our work that provides a powerful tool for further investigations on them.

Hoare and He [9], although dealing with alphabetised predicates, often leave it 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 alphabet extension used in the UTP constrains the values of the new variables: they cannot be changed. However, our set-based model for relations needed 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 the reactive $SKIP$, the alphabet is, implicitly, left unchanged. Our implementation defines two existential and two universal quantifications: one of them removes the quantified variables from the alphabet, and the other one does not. We also redefined some of the UTP definitions in order to facilitate our proofs.

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

We expressed the language constructors as functions. For this reason, they can be simply extended without loosing the previous proofs; the syntax of expressions was abstracted by using three simple definitions. Furthermore, the strategy that we adopted for lifting Z functions and relations to relational predicates, for instance $\leq_R$, makes the Z toolkit directly available in our theory. Our work provides a mechanical support not only to *Circus*, but to any other work theoretically based on any of the UTP theories.

The current number of laws on sequential composition may need to be increased to allow users of our theory of relation 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 aim at providing a mechanisation of the UTP that can support the development of other languages theoretically based on the UTP; *Circus* is such a language. Our next step is to mechanise the *Circus* theory, which will be based on the CSP theory, and will mechanise not only the final version of the semantics of *Circus*, but also all the refinement laws proposed so far. This will provide *Circus* with a mechanised refinement calculus that can be used in the formal development of State-Rich Reactive Programs.

Acknowledgements

We are grateful for the financial support of QinetiQ and the Royal Society. Philip Clayton, Rob Arthan, Roger Bishop Jones, Mark Adams, and Will Harwood provided valuable advice for our work.

References

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

2. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A refinement strategy for *Circus*. *Formal Aspects of Computing*, **15**(2–3):146–181, 2003.
3. A. L. C. Cavalcanti and J. C. P. Woodcock. A tutorial introduction to CSP in Unifying Theories of Programming. In *Proceedings of the Pernambuco Summer School on Software Engineering: Refinement 2004*, 2004.
4. A. L. C. Cavalcanti and J. C. P. Woodcock. Angelic nondeterminism and Unifying Theories of Programming. Technical Report 13-04, Computing Laboratory, University of Kent, June 2004.
5. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, volume **2**, pages 423–438. Chapman & Hall, 1997.
6. M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. volume **78** of *LNCS*. Springer-Verlag, 1979.
7. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
8. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
9. C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming.* Prentice-Hall, 1998.
10. C. Morgan. *Programming from Specifications.* Prentice-Hall, 1994.
11. G. Nuka and J. C. P. Woodcock. Mechanising the alphabetised relational calculus. In *WMF2003: 6th Braziliam Workshop on Formal Methods*, volume **95**, pages 209–225, Campina Grande, Brazil, October 2004.
12. M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus – Additional Material*, 2006. At http://www.cs.york.ac.uk/circus/refinement-calculus/oliveira-phd/.
13. M. V. M. Oliveira and A. L. C. Cavalcanti. From *Circus* to JCSP. In J. Davies *et al.*, editor, *Sixth International Conference on Formal Engineering Methods*, volume **3308** of *LNCS*, pages 320–340. Springer-Verlag, November 2004.
14. M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Refining industrial scale systems in *Circus*. In Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures*, volume **62** of *Concurrent Systems Engineering Series*, pages 281–309. IOS Press, 2004.
15. S. C. Qin, J. S. Dong, and W. N. Chin. A semantic foundation of TCOZ in Unifying Theories of Programming. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, volume **2805** of *LNCS*, pages 321–340. Springer-Verlag, September 2003.
16. A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through Determinism. In D. Gollmann, editor, *ESORICS 94*, volume **1214** of *LNCS*, pages 33–54. Springer-Verlag, 1994.
17. M. Saaltink. The Z/EVES System. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume **1212** of *LNCS*, pages 72–85, Reading, April 1997. Springer-Verlag.
18. A. Sherif and H. Jifeng. Towards a time model for *Circus*. In C. George and H. Miao, editors, *Formal Methods and Software Engineering: 4th International Conference on Formal Engineering Methods, ICFEM 2002*, volume **2495** of *LNCS*, pages 613–624. Springer-Verlag, June 2002.
19. K. Taguchi and K. Araki. The state-based CCS semantics for concurrent Z specification. In M. Hinchey and Shaoying Liu, editors, *International Conference on Formal Engineering Methods*, pages 283–292. IEEE, 1997.

20. H. Treharne and S. Schneider. Using a process algebra to control B operations. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 437–456. Springer, June 1999.
21. J. C. P. Woodcock and A. L. C. Cavalcanti. *Circus*: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK, July 2001.
22. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.
23. J. C. P. Woodcock and A. Hughes. Unifying Theories of Parallel Programming. In H. Miao C. George, editor, *Formal Methods and Software Engineering: 4th International Conference on Formal Engineering Methods, ICFEM 2002*, volume **2495** of *LNCS*, pages 24–37. Springer-Verlag, June 2002.