Mechanical Reasoning about Families of UTP Theories

Frank Zeyda*, Ana Cavalcanti

Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK

Abstract

The Unifying Theories of Programming (UTP) of Hoare and He is a general framework in which the semantics of a variety of specification and programming languages can be uniformly defined. In this paper we present a semantic embedding of the UTP into the ProofPower-Z theorem prover; it concisely captures the notion of UTP theory, theory instantiation, and, additionally, type restrictions on the alphabet of UTP predicates. We show how the encoding can be used to reason about UTP theories and their predicates, including models of particular specifications and programs. We support encoding and reasoning about combinations of predicates of various theory instantiations, as typically found in UTP models. Our results go beyond what has already been discussed in the literature in that we support encoding of both theories and programs (or their specifications), and high-level proof tactics. We also create structuring mechanisms that support the incremental construction and reuse of encoded theories, associated laws and proof tactics.

Key words: semantic embedding, theorem proving, verification, ProofPower, Z, *Circus*

1. Introduction

The Unifying Theories of Programming (UTP) provides a framework for the semantic integration of a variety of programming languages from different computational paradigms within a unified relational model. It captures the meaning of imperative, functional and concurrent languages, for example, and helps to identify common features. It also provides a unified semantic presentation of programming theories, and shows how links between them can be formulated and reasoned about. The semantics of a variety of integrated programming and modelling languages are based on the UTP [MD00, SJ02, BSW07]. Crucially, the UTP unifies different notions of refinement, and enables us to formally state and prove that a program acts in accordance with a specification of its required behaviour.

The UTP models a program (or specification) as a relation capturing the observations that can be made of its behaviour. The calculus of the UTP is based on alphabetised relations similar to Tarski's [Tar41], but presented in a predicative style. For example, $x' = x + 1$ encodes the relation that increments the value of x. By convention, undashed variables are used to denote the state (values of the variables in the alphabet) before the computation, and corresponding dashed variables the values of the variables at a later point in the computation. A UTP theory is characterised by an alphabet and a collection of healthiness conditions that identify the predicates that are valid models of computation in that theory.

In [OCW07], Oliveira presents an embedding of the UTP for the ProofPower-Z theorem prover. ProofPower is a versatile and extendable prover for higher-order logic that has been successfully used in the avionics industry [AC05]. ProofPower-Z is an extension of ProofPower that provides additional reasoning support for the Z language [WD96]. (ProofPower itself is based on the logic of HOL [Gor88].)

In the approach of [OCW07], the static notion of a ProofPower theory is used to capture the definitions, operators, axioms and laws of various UTP theories, namely the theories of relations, designs, reactive

*Corresponding Author.
Email address: frank.zeyda@cs.york.ac.uk (Frank Zeyda)

designs, CSP, and *Circus*. The mechanisation has been successfully used to prove laws that are *generally* valid within these UTP theories; hereby a repository of more than 500 verified theorems was created.

A question of practical interest which Oliveira's work did not address in full, however, is reasoning about particular UTP specifications, especially in the presence of types. We consider, for example, the proof of the following refinement conjecture within the UTP theory of relations.

$$x := x + 1 \lhd x = 1 \rhd \mathbf{I\!I} \quad \sqsubseteq \quad x := 2$$

The notation $P \lhd b \rhd Q$ is used for conditionals: the program that behaves as P if b holds, otherwise as Q. $\mathbf{I\!I}$ denotes the computation that has no effect (Skip). The above refinement is valid under the assumption that x ranges over the values of the set $\{1, 2\}$. Since the UTP acknowledges strict typing, it is sensible and, as illustrated here, in certain cases even necessary to exploit assumptions about the types of variables.

The alphabet of each of the UTP theories described in [HJ98] includes a set of variables w, whose particular names are left unspecified. They are included to represent the programming variables and are named after them. In the case of our example above, this would be the single variable x and its dashed counterpart x' used to denote the final value of x. Therefore, we can conclude that the theory descriptions in [HJ98] define *families* of theories rather than particular instances which fix the set of programming variables of interest and their types. Our refinement conjecture cannot be expressed and does not hold in all instantiations, only in those which include x and x' in their alphabet, and specify $\{1, 2\}$ as their type.

With the existing semantic encoding, there are a few subtle complications related to reasoning about refinement statements such as the above. They mostly arise from the fact that neither a dynamic notion of UTP theory nor of instantiation of a theory is provided. Instead, that encoding introduces a global universe of variable names with no restrictions imposed on their types. Concretely, a type of bindings (records) that associate names to values is introduced and predicates of all theories are modelled as sets of bindings.

$$\mathit{BINDING} \mathrel{\widehat=} \mathit{NAME} \nrightarrow \mathit{VALUE}$$

To capture type-constraints on variables, restrictions on *BINDING* have to be placed *a posteriori* by virtue of suitable axiomatic constraints. For the previous example, we would have to specify

$$\vdash \forall\, b : \mathit{BINDING} \mid x \in \operatorname{dom}\ b \bullet b\,(x) \in \{1, 2\}$$

as an additional axiom. Here, x refers to a global constant that represents the variable $\boldsymbol{x}$.

We identify a number of problems with this approach. Firstly, such axioms would not merely restrict the predicates of a singular UTP theory but in fact all UTP theories in the current ProofPower theory scope. This is due to UTP theories being organised in a static hierarchical manner, and ultimately each theory being characterised by further restrictions on the general theory of relations whose underlying predicates would be constrained by axioms such as the above. (The *BINDING* type is indirectly shared by all theories.) For this reason, it would not be possible for two predicates in which the variable x has different types to coexist within the same definitional scope of the prover, a drawback that we overcome in this work.

A second difficulty is that when reasoning about particular programs, it is frequently necessary to work with predicates of different *instances* of a theory. A trivial example is a variable block, whose body is a predicate in a theory whose alphabet is enriched by the declared variables, and so different from the theory of the block itself. The presence of other encapsulation mechanisms, in languages like *Circus* [CSW03] and TCOZ [MD00], for example, whose semantics are based on the UTP, raises more issues of this nature. *Circus* specifications contain a series of processes, and a TCOZ specification contains a series of classes. The states of the processes and the attributes of the classes define different instances of theories that need to be handled in a single specification. In other words, a ProofPower theory defining these specifications involves predicates of several UTP theories, so that the static association is appropriate for families of theories, but less accurate when we reason about their instances.

At this point it is worth noting the significant distinction between UTP theories and ProofPower theories. Whereas ProofPower theories are database-like entities of the host environment, carrying the definitions and theorems for the semantic encoding, UTP theories are the abstract mathematical entities that we model and reason about. In the existing treatment, UTP theories are 'instantiated' by importing the appropriate ProofPower theory; due to the static structure of theory hierarchies in ProofPower, this can only be done once and for all. Hence it is in principle possible to formulate and prove the above refinement conjecture even in the existing encoding by creating a designated ProofPower theory for it, but this approach is not viable for verification techniques where multiple specifications have to reside in the same definitional reasoning scope.

The third difficulty with the current encoding is that the introduction of subsequent constraints for constants could potentially result in the model becoming inconsistent and thus vacuous. To establish consistency, ProofPower-Z has facilities to generate existential proof obligations, but only for *newly* introduced constants and not for subsequently added constraints on existing ones.

To solve these problems, we introduce a new approach to the mechanisation of the UTP in which we provide a semantic characterisation of theories, and a means for dynamic instantiation. This reduces the risk for inconsistency, and creates opportunities for formulating and proving properties which were previously beyond the scope of mechanical reasoning. These are, for example, theorems about links between UTP theory instantiations such as isomorphisms, Galois connections, and many others.

Our contributions in this paper can be summarised as follows.

1. We propose a method for encoding specific UTP theories which lends itself to integration into verification strategies based on a UTP semantics. The hierarchical presentation of UTP theories, which is explored in [OCW07], has crucial benefits in terms of reusing definitions and laws, hence we do not to abandon it, but rephrase it in a more dynamic and modular way. We illustrate our ideas through many examples, and use *Circus* as a major case study (see Section 4). The mechanisation and its approach, however, are of a much wider applicability, and can support, for instance, reasoning in TCOZ.
2. A second contribution is a strategy to encode particular specifications. In doing so we outline some of the opportunities for reasoning about concrete specifications and their refinements. We use a *Circus* specification as a motivating example, and also explain how the theory of *Circus* is embedded, how modularity is exploited in its presentation, and how different theory instantiations arise. This illustrates the benefits of recasting Oliveira's original model because our, and other similar scenarios, are difficult to deal with in the original approach.
3. A third additional contribution of the work is the implementation of the majority of the high-level proof tactics we suggested in [ZC09]. This produced further insight into how we create an infrastructure of tactics that can dynamically be extended as the ProofPower theory hierarchy encoding specific UTP theories unfolds. Altogether three layers of organisation can be identified: the Z definitions for particular UTP theories, a collection of associated theorems, and extensions to the automatic proof tactics. The organisation and dependency of these layers is another aspect which we address.

We have to a certain extent tackled these problems previously in [ZC08]. In this paper, we significantly improve on that work by presenting new ideas and practical results motivated by experience since gained. A major new feature of the encoding described here is the adoption of a more concise model for typing. It is isomorphic to our previous model, but more convenient for proof. The motivation in our previous work was to render the semantic model of alphabetised predicates as simply as possible, but it has been shown to complicate many proofs of theorems about universes, including their simplification. We, therefore, present a new model (different from that in [ZC08]), an encoding strategy like that described in [ZC08] and illustrated for *Circus* in [ZC09], but adapted for the new model, and, as already said, the proof tactics that are only suggested in [ZC09], but here are discussed in detail and considered in the context of the new model.

In summary, we provide a model that can cover all the features in [OCW07], and in addition supports reasoning about particular specifications of a theory, as well as its general laws. Furthermore, we provide a model that improves on our own previous results [ZC08], in that it is better suited for (automated) proof.

The structure of the paper is as follows. First, in Section 2 we further detail the main principles and ideas of the UTP. Section 3 presents the relevant parts of our semantic encoding defining the notion of alphabetised predicate, UTP theory, and instantiation. Section 4 then explains the encoding of more elaborate theories using *Circus* as an example. Here, we also present and discuss the UTP model of a simple vending machine described in *Circus*; it is used for illustration in subsequent sections. Section 5 surveys how we reason about UTP theories in general, formulate properties of theory links, and specify and apply refinement laws. Section 6 discusses the encoding of particular specifications making use of the vending machine example, and Section 7 reports on techniques and experiences regarding proof automation. In Section 8 we finally draw our conclusions, raise some discussion of our design decisions, and outline future work.

2. The Unifying Theories of Programming

The UTP provides a unified framework for semantics definitions which aims to be independent of the particular language or paradigm studied. The motivation of the UTP is to support direct comparison of a variety of differing programming and modelling languages, isolate their semantic features, and, additionally, facilitate their combination. First, it instructs that general concepts such as sequential composition or nondeterminism should be equally expressed across semantic theories. Another important aspect is a common notion of refinement and associated calculus that permits the verification of implementations with respect to some specified behaviour. A common notation is used for both, programs and their specifications.

To achieve these goals, the UTP follows a denotational approach that represents the elements of a semantic model as relations over some agreed alphabet; hence the calculus of the UTP is essentially one of relations, presented in a predicative style. Relations are used to describe the observable behaviour of a particular computation, process or system under consideration, and the sets of relations that represent meaningful computations in some paradigm, including their operators, are called theories.

As already mentioned, we use predicates to define relations, and therefore assume every predicate to be implicitly (or explicitly) associated with an alphabet. For this reason, predicates in the UTP are customarily referred to as alphabetised predicates. The alphabet of a predicate P is given by αP. As a simple example, the predicate $x' > x$ with alphabet $\{x, x'\}$ specifies the computation that nondeterministically increases the value of x by some unknown quantity. In general, undecorated variables are used to represent initial observations, and corresponding dashed variables to represent subsequent or final observations. The *in* operator yields all undashed variables of some alphabet, and the *out* operator all (single-)dashed variables.

Alphabets in general define the variables that record a relevant observable property. In programming theories these could be, for example, state variables, but also auxiliary variables that may record termination of the program ($okay$), traces of events while the program executes (tr), and indeed any further ones that are required to fully characterise computations in the paradigm under consideration.

Apart from alphabetised predicates we also consider UTP theories to have alphabets. The alphabet of a theory predetermines the alphabet of the predicates belonging to that theory. It is possible to dash individual variables as well as alphabets; in case of the latter, dashing applies to each variable of the alphabet. The notion of an alphabetised predicate is slightly more general than that of a relation: whereas predicates permit any sets of variables, relations are restricted to undecorated ones and those with a single dash.

Standard predicate calculus operators can be used to combine alphabetised predicates; for example, $x' = x + 1 \vee x' = x - 1$ specifies a program that either increments or decrements the value of x. Nondeterminism here is modelled by disjunction of predicates, and this is indeed a common feature across different UTP theories. The sequence of computations $P\,;\ Q$, similarly, is generally characterised by relational composition. Other standard operators include Skip ($\mathbf{II}_A$), the assignment $x :=_A e$, the conditional $P \lhd b \rhd Q$, and local variable blocks. The subscript A in some of these operators is an alphabet that needs to be given as a parameter for the construct. Every UTP operator must specify the alphabet of the resulting predicate; where the alphabet cannot be implicitly determined from the operand(s), it must be explicitly given.

Table 1 lists the core operators of the UTP, including their definitions, alphabets and possible caveats. Skip is the computation that leaves the state unchanged. Assignment changes the value of a variable by constraining its final value, but leaves other variables in the alphabet unaffected. The UTP conditional takes a less familiar form as an infix operator, leading to a more symmetric presentation of its algebraic properties; the more familiar analogue is **if** b **then** P **else** Q. For sequential composition to be well-defined, the output alphabet of the first predicate has to match the input alphabet of the second predicate; it is defined by existentially quantifying over the intermediate states. It is also possible to extend the alphabet of a predicate; namely P_{+x} extends the alphabet of P with the variable x and its dashed version. The

Name	**Syntax**	**Definition**	**Alphabet**	**Caveat**
Skip	$\mathbf{II}_A$	$v = v'$	A	$A = \{v, v'\}$
Assignment	$x :=_A e$	$x' = e \wedge a' = a \wedge b' = b \wedge \ldots$	A	$A = \{x, x', a, a', b, b', \ldots\}$
Conditional	$P \lhd b \rhd Q$	$(b \wedge P) \vee (\neg\, b \wedge Q)$	αP	$\alpha P = \alpha Q$ and $\alpha b \subseteq \alpha P$
Sequential Composition	$P\,;\ Q$	$\exists\, v_0 \bullet P[v'\backslash v_0] \wedge Q[v\backslash v_0]$	$in\,\alpha P \cup out\,\alpha Q$	$out\,\alpha P = \{v'\} = (in\,\alpha Q)'$
Nondeterministic Choice	$P \sqcap Q$	$P \vee Q$	αP	$\alpha P = \alpha Q$
Alphabet Extension	P_{+x}	$P \wedge x = x'$	$\alpha P \cup \{x, x'\}$	$\alpha P \cap \{x, x'\} = \varnothing$
Declaration	$\mathbf{var}\, x$	$\exists\, x \bullet \mathbf{II}_A$	$A \setminus \{x\}$	$\{x, x'\} \subseteq A$
Undeclaration	$\mathbf{end}\, x$	$\exists\, x' \bullet \mathbf{II}_A$	$A \setminus \{x'\}$	$\{x, x'\} \subseteq A$
Weakest Fixed Point	$\mu\, X \bullet F(X)$	$\bigsqcap \{P \mid P \Rightarrow F(P)\}$	$\alpha F(X)$	$\forall\ P_1, P_2 \bullet \alpha F(P_1) = \alpha F(P_2)$
Strongest Fixed Point	$\nu X \bullet F(X)$	$\bigsqcup \{P \mid F(P) \Rightarrow P\}$	$\alpha F(X)$	$\forall\ P_1, P_2 \bullet \alpha F(P_1) = \alpha F(P_2)$
Iteration	$b * P$	$\mu\, X \bullet (P\,;\ X) \lhd b \rhd \mathbf{II}_{\alpha P}$	αP	

Table 1: Definition of common UTP Operators

definition ensures the variable retains its value. Lastly, the **var** and **end** constructs support the use of local variables. This is achieved by sequentially composing them with the body of the variable block as in $\mathbf{var}\, x\,;\ P\,;\ \mathbf{end}\, x$. Composition with **var** hides the initial value of the local variable, and composition with **end** hides its final value. Thus, **var** is used to open the scope of a local variable, and **end** to close it. The body P of the declaration must have the declared variable and its dashed counterpart in its alphabet.

As already mentioned, the UTP is founded on a unified notion of refinement. Refinement intuitively captures that all behaviours of some concrete specification (intermediate refinement or implementation) are also possible behaviours of some abstract specification. Hence, refinement establishes the formal correctness of an implementation with respect to a specification, and therefore is used to construct verification arguments. For example, the previous specification $x' = x + 1 \vee x' = x - 1$ is refined by $x := x + 1$; we use the notation $x' = x + 1 \vee x' = x - 1 \sqsubseteq x := x + 1$ to formally state this. The example illustrates how refinement may resolve possible nondeterminism in the specification. Mathematically, refinement is characterised by universal (reverse) implication: $S \sqsubseteq P \ \widehat{=}\ [P \Rightarrow S]$. Here, $[\_]$ is the universal closure of a predicate; it is defined by $[P] \ \widehat{=}\ \forall\, w \bullet P$ where w are the variables in the alphabet of P.

Relational theories are complete lattices under refinement, with bottom element *true* and top element *false*. It then follows that monotonic functions on their predicates have weakest and strongest fixed points with respect to that order, and the operators $\mu\, X \bullet F(X)$ and $\nu X \bullet F(X)$ are introduced to denote them. Their definition is also in Table 1; we observe that $\bigsqcap$ and $\bigsqcup$ are meet and join operators. Recursion is modelled using fixed points, and iteration is treated as a special case of recursion. Intuitively, $b * P$ repetitively executes P unless b becomes *false*; its more familiar syntax is the **while** b **do** P statement.

In the UTP approach, specific features of a paradigm are usually captured by custom relational operators, and further validation and understanding is provided by a set of algebraic laws that relate them to the other operators. One such law is, for instance, that non-deterministic choice distributes through conditional, expressed as $P \lhd b \rhd (Q \sqcap R) = (P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)$. This, and many other theorems relating the operators in Table 1 are presented in [HJ98], and they constitute a repository for proving further algebraic properties of new operators, and refinement conjectures of relational predicates.

In defining UTP theories, we are usually only interested in a *subset* of the predicates over a given alphabet, namely those that represent meaningful computations. The most general theory is the one of relations as it includes *all* predicates of a certain alphabet. It is at the top of the theory hierarchy. In more concrete theories we use healthiness conditions to identify the predicates that belong to the theory. These embody facts about the computational models by restricting the set of permissible predicates. For example, the theory of designs is a restriction of the theory of relations that supports reasoning about program termination. It is useful since the general theory of relations is not expressive enough to capture nonterminating behaviour, as required in a total-correctness program semantics.

The theory of designs handles nontermination by introducing additional boolean variables $okay$ and $okay'$ which record whether a program has started or finished. To filter out predicates that make assumptions about the effect of a program before it has started, all predicates P are required to satisfy the healthiness condition $P = okay \Rightarrow P$. Predicates satisfying this condition evaluate to $true$ whenever $okay = false$. This condition, and healthiness conditions in general, can be expressed using idempotent functions that map a possibly unhealthy predicate to a valid healthy one; here, it would be the function $\mathbf{H1}(P) = okay \Rightarrow P$. The predicates of a theory are exactly the cumulative fixed points of its healthiness functions.

A second healthiness condition **H2** for a design P is that $P[okay' \backslash false] \Rightarrow P[okay' \backslash true]$; it requires that any observable behaviour under nontermination must also be a possible terminating behaviour. This implies that it is not possible to actually rely or insist on nontermination. **H2** can be written as an idempotent by defining an auxiliary predicate $J \mathrel{\widehat{=}} okay \Rightarrow okay' \land v = v'$ with v being the state alphabet of the design theory under consideration. With it we can define $\mathbf{H2}(P) = P \,;\, J$. That the fixed points of **H2** are exactly those predicates that satisfy the property above is proved in [HJ98].

Together, **H1** and **H2** characterise valid designs. It can be proved that they are exactly the predicates that can be written in the form $P \land okay \Rightarrow Q \land okay'$ where P and Q specify the familiar pre and postcondition pair of a computation. They are described using the design notation $P \vdash Q$.

In summary, the UTP adopts a very general approach in which theories are predicate sets characterised by alphabets and healthiness conditions. Unification is achieved by the ability to freely combine predicates of different theories, a common notion of refinement, and fundamental operators like sequential composition and nondeterminism being equivalently defined across theories. New theories may be specified by specialising existing theories with additional healthiness conditions, and naturally laws that have been proved for the less restrictive theories carry over to the more restrictive ones. The theory of designs, for example, restricts the theory of relations over a certain alphabet. Additionally, new operators may be introduced, and their soundness with respect to a theory can be examined by showing they are closed under its predicates.

An area on which the UTP literature is not very explicit is types. Generally, we assume any theory to be strongly typed, and this implies that variables in alphabets implicitly have type constraints associated with them. In the next section we explain how we encode aspects of the UTP that concisely capture the notion of alphabetised predicate and UTP theory, and doing so moreover deal with the issue of typing.

3. Semantic Encoding of the UTP

In this section we present and explain the core definitions of our semantic encoding of the UTP. Our main objective is to develop a semantic model for alphabetised predicates and UTP theories that, unlike existing mechanisations, takes into account type restrictions on variables. Because of the extensiveness of the encoding, we are not able to present and explain all definitions in detail. The complete ProofPower theory scripts can be found on the web at `http://www.cs.york.ac.uk/circus/tp/tools.html`.

In this section we particularly look at the contents of the ProofPower-Z theories utp-lang, utp-pred, and utp-theory, establishing the fundamental language definitions and semantic model for alphabetised predicates and of UTP theories. An overview of the entire theory hierarchy is presented in Figure 1.

3.1. Expressions and Values

Before examining the particulars of modelling alphabetised predicates, we need to establish a few fundamental definitions that allow us to embed the syntax and semantics of expressions and values. The definitions necessary for this part of the mechanisation are provided in the ProofPower-Z theory utp-lang.

First, we introduce a new type $NAME$ to represent variable names. Names are characterised by a triple of natural numbers whose first component is a unique identifier, the second component records the number of dashes, and the third component is used to specify a possible subscript. We seldom need to refer to the detailed representation of names. We usually introduce them as arbitrary elements from $NAME$ with some additional constraints; it is however important to have this amount of detail in the semantic model to prove

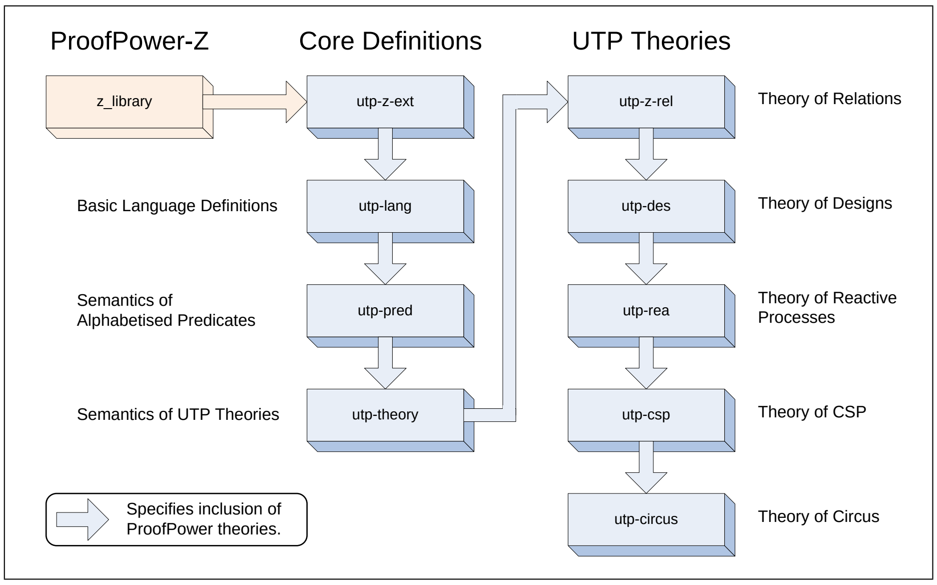

Figure 1: Hierarchy of ProofPower theories of the mechanisation.

distinctiveness properties such as $x' \neq x''$ and $x_1 \neq x_2$ in places where we need them. A purely axiomatic characterisation that did not define a concrete model for $NAME$ is used in [OCW07]. It acknowledges some incompleteness, and assumes that whenever additional properties of names and their operators are required, they are added as new axioms. The reason we deviate from this approach is because of a technical intricacy in ProofPower-Z concerning the consistency proofs for operators. This also lessens the risk of invalidating proofs by future alterations to definitions.

Several operators are provided of which the most important one is the *dash* function. Its effect is to decorate names by appending them with a dash. This is achieved by incrementing the second component of the name. We omit the formal definition here, but it can be inspected in Appendix A.1. We point out that *dash* is injective, a property which is relied upon in the sequel.

Using *dash* we also define the constant $dashed \mathrel{\widehat{=}} \operatorname{ran} dash$ as the set of decorated names, and furthermore $undashed \mathrel{\widehat{=}} NAME \setminus dashed$ as the set of undecorated names. We support the encoding of names that carry multiple dashes, though in the following we only consider decorated names with a single dash. The corresponding set *dashed_once* is introduced for them and specified as $dash \llparenthesis undashed \rrparenthesis$.

Alphabets are elements of a type $ALPHABET$ and are introduced as subsets of $NAME$. They are *homogeneous* if they contain only matching pairs of dashed and undashed names. Two alphabets a_1 and a_2 are *composable* if the dashed names of the first correspond the undashed names of the second, that is, $\forall n : undashed \bullet n \in a_2 \Leftrightarrow dash\ n \in a_1$. Composability is crucial for relational composition to be well-defined. Both properties are specified by the relational constants *homogeneous* and *composable*.

We now turn to the definition of $VALUE$ and $EXPRESSION$, which capture the semantic domain of values and the syntax of expressions. They are both introduced by Z declarations of suitable free types. $VALUE$, whose definition is given below, represents all values in the semantic universe; its specification by virtue of a new ProofPower-Z type allows for a uniform treatment of values in the host logic (HOL). It also permits reasoning about types within the model of the object language (UTP). Although we generally assume that encoded expressions and predicates are type correct, it is nevertheless necessary to be able to reason about types in the semantics, for example, to prove theorems relying on such assumptions.

$$\begin{aligned}&VALUE ::= Int(\mathbb{Z}) \mid Bool(\mathbb{B}) \mid Real(\mathbb{R}) \mid Channel(NAME) \mid \\ &\quad Set(\mathbb{P}\ VALUE) \mid Pair(VALUE \times VALUE) \mid Seq(\operatorname{seq} VALUE) \mid Sync\end{aligned}$$

Simple values can be integers, booleans, real numbers, or channel names, and are obtained by the respective constructor functions *Int*, *Bool*, *Real*, or *Channel*. *Sync* is a special value used in synchronising communications without data exchange. *Set*, *Pair* and *Seq* are constructor functions to create more complex values such as sets, pairs, and sequences; for example, $Seq(\langle Int(1), Int(2), Int(3)\rangle)$ encodes the sequence $\langle 1, 2, 3\rangle$. Alongside, we introduce the sets *INT_VAL*, *BOOL_VAL*, and so on to refer to the values representing integers, booleans, etc. We can think of them as modelling carrier sets for particular value types.

The abstract syntax of expressions is encoded by virtue of a free type *EXPRESSION*. It considers the case of value constants, variables, relations, and unary as well as binary function applications.

$$\begin{array}{l} EXPRESSION ::= Val(VALUE) \mid Var(NAME) \mid \\ \quad Rel(REL \times EXPRESSION \times EXPRESSION) \mid \\ \quad Fun_1(UNARY\_FUN \times EXPRESSION) \mid \\ \quad Fun_2(BINARY\_FUN \times EXPRESSION \times EXPRESSION) \end{array}$$

The three sets *REL*, *UNARY_FUN* and *BINARY_FUN* model the types for relations $VALUE \leftrightarrow VALUE$, unary functions $VALUE \nrightarrow VALUE$, and binary functions $VALUE \times VALUE \nrightarrow VALUE$ on values. Such functions have to be individually provided to give a semantic interpretation for all relations and operators of the syntax. For example, a function $(\_ +_V \_) : BINARY\_FUN$ is required to encode the expression $1 + 2$, whereby the encoding is $Fun_2((\_ +_V \_), Val(Int(1)), Val(Int(2)))$. Analogously, to encode $x \leq y$ we would use $Rel((\_ \leq_V \_), Var(x), Var(y))$ with $(\_ \leq_V \_)$ being suitably defined. Although many such concrete relations and operators are defined in utp-lang, their specific definition shall not be relevant here. As a convention, we agree on a general rule to use a subscript V for functions and relations on values.

Two important operators on expressions are $FV(e)$ and $Eval(b, e)$. The first infers the free variables of an expression e, and the second evaluates an expression with respect to some binding b. Intuitively, bindings associate (a subset of) names with values. Since functions on values are partial, evaluation in general is partial too. As a restriction for $Eval(b, e)$ to be well-defined, we first require the binding b to equate each free variable in e with a value. Secondly, it has to be a value of the correct type depending on the operators occurring in the expression. For example, to evaluate $x + y$ a binding would have to associate both x and y with integer values; namely, because the result is eventually determined by the function $(\_ +_V \_)$, which is only defined for integer pairs. Although, fundamentally, well-definedness of evaluation in our model is subject to verification, initial type checking of the syntax of expressions and predicates justifies to postulate it as an axiom. This eases the burden on proofs having to discharge respective provisos.

The next sections introduce the semantic model of alphabetised predicates and UTP theories.

3.2. Alphabetised Predicates and Universes

The semantic model for an alphabetised predicate is a set of bindings describing the valuations that render it true. For example, the predicate $x = 1 \lor x = 2$ is characterised by the set of bindings including just $x \mapsto 1$ and $x \mapsto 2$ if we assume the only variable of the alphabet is x. The potential bindings that can be used in representing predicates are, however, subject to type restrictions. The formal characterisation of an alphabetised predicate is a pair defined as follows.

$$ALPHA\_PREDICATE \mathrel{\widehat=} \{bs : BINDINGS;\ u : UNIVERSE \mid bs \subseteq Bindings_U\ u\}$$

In this definition, *BINDINGS* is the set of all binding sets irrespective of type constraints, and *UNIVERSE* the set of all valid typing definitions specifying the predicate's alphabet and the types of the variables included in it. Individual bindings are represented by partial functions from variable names to values, that is $BINDING \mathrel{\widehat=} NAME \nrightarrow VALUE$, and $BINDINGS \mathrel{\widehat=} \mathbb{P}\,BINDING$ is the type of all binding *sets*. Universes are modelled by partial functions from names to non-empty sets of values. We introduce a new set $TYPE \mathrel{\widehat=} \mathbb{P}_1\ VALUE$, and with it define *UNIVERSE* as given below.

$$UNIVERSE \mathrel{\widehat=} NAME \nrightarrow TYPE$$

The variables (or alphabet) of a universe is simply its domain. For clarity, we introduce the function $Alphabet_U$ to infer it. For each name $n \in Alphabet_U\ u$, the application $u(n)$ yields the set of values over which n may range in the universe u. The functional requirement ensures that variables can have at most one type associated with them, and the restriction to non-empty subsets of values avoids the case of degenerate types that contain no values at all. The problem with empty types is that the respective universe has no bindings, and does not allow the differentiation of the predicates *true* and *false*. For that, at least one binding needs to exist, since otherwise both *true* and *false* correspond to the empty set of bindings.

The purpose of the function $Bindings_U$ used in the definition of $ALPHA\_PREDICATE$ is to construct the binding set that corresponds to a given universe. It contains all the bindings that assign values to all the variables in the universe, and no others, that agree with the type restrictions imposed by it. Clearly, we do not want to mention variables outside the universe, but also we do not consider bindings that may valuate only a subset of the universe's alphabet. For the definition of $Bindings_U$ see Appendix A.1.

The universe model we specify here is different from the earlier one proposed in [ZC08], where universes were characterised directly by sets of well-typed bindings, including bindings that may not valuate all alphabet variables (Appendix B provides the definition of $UNIVERSE$ in that model). It, however, shall be noted that this does not affect the expressiveness of the model: it was merely a design decision to simplify universe combinators. Our new model in fact turns out to be isomorphic to this model. On one hand, the $Bindings_U$ function enables us to extract the binding set of a universe. Vice versa, it is always feasible to extract the type of a variable in the former model.

The motivation behind the earlier approach was to simplify the definition of $ALPHA\_PREDICATE$: its constraint was simply $bs \subseteq u$. We expected this also to facilitate proofs which either exploit or have to show this property. However, it turned out that a high price had to be paid for this design, for proofs about universes became much harder; firstly, as universe operators required more elaborate definitions, and secondly since essential properties of $UNIVERSE$, like for example subset-closure, had to be established to prove closure of operators. To show that some specific value is a universe required more proof effort too.

The present model is evidently more concise, and a considerable simplification in the proof of laws about universes mirrors this. Interestingly, it is also not more complicated in terms of proofs of properties of alphabetised predicates; by introducing $Bindings_U$, we can formulate theorems about the bindings of universes in a modular and general way. Mathematically, the two models have equal expressive power. A formal proof of the underlying isomorphic relationship is possible, but would distract us from the main objectives of the paper. We now look at some of the operators for manipulating universes.

3.3. Universe Operators

The two central universe operators are universe extension $u_1 \oplus_U u_2$ and universe restriction $u \ominus_U a$. Universe extension is simply defined as the union of u_1 and u_2, yielding a universe whose alphabet contains the variables of both u_1 and u_2. It is only applicable if the universes agree on the types of the shared variables. If so, we say that the universes are *compatible*. We formally define compatibility as follows.

$$Compatible_U : UNIVERSE \times UNIVERSE \rightarrow \mathbb{B}$$

$$\forall\, u_1, u_2 : UNIVERSE \bullet$$
$$\quad Compatible_U(u_1, u_2) \;\Leftrightarrow\; (\mathrm{dom}\, u_1 \;\cap\; \mathrm{dom}\, u_2) \lhd u_1 = (\mathrm{dom}\, u_1 \;\cap\; \mathrm{dom}\, u_2) \lhd u_2$$

Complementary, universe restriction removes the variables in the alphabet a from the universe u; its result is defined by the domain subtraction $a \ntriangleleft u$.

A few further operators are provided, namely $Merge_U(us)$ which merges a set of universes, $Rename_U(f, u)$ which renames the variables in a universe, and $typeof(n, u)$ and $typeof_E(e, u)$ which yield the type of a variable and of an expression in a universe, respectively. The simple definitions of $Merge_U$ and $Rename_U$ are omitted in the narrative here, but can be inspected in Appendix A.1; the former is just a generalisation of $u_1 \oplus_U u_2$ using generalised set union, and the latter carries out a renaming according to a given injective function f on names. The function $typeof(u, n)$ is defined as the application of u to n, providing n is in the alphabet of u. For $typeof_E$ we have the following definition. Here, $WF\_EXPRESSION\_UNIVERSE$ encapsulates restrictions on the arguments e and u. Specifically, $typeof_E\,(e, u)$ is only meaningful and hence applicable if the free variables in e are also included in u.

$$
\begin{array}{|l}
\mathit{typeof}_E : \mathit{WF\_EXPRESSION\_UNIVERSE} \to \mathbb{P}\ \mathit{VALUE} \\
\hline
\forall\, e : \mathit{EXPRESSION};\ u : \mathit{UNIVERSE} \mid \\
\quad (e, u) \in \mathit{WF\_EXPRESSION\_UNIVERSE} \bullet \mathit{typeof}_E(e, u) \ = \ \{b : \mathit{Bindings}_U\ u \bullet \mathit{Eval}\,(b, e)\}
\end{array}
$$

Operationally, we evaluate the expression e in all bindings of u and thereby construct the set of values e may take. The typeof_E function is useful to formulate semantic constraints in which the type of an expression must be compatible with the type of a variable; a typical example of this is substitution or assignment.

In the previous two sections we have developed a semantic model for alphabetised predicates that takes into account type restrictions. For this purpose, we have introduced and formalised the notion of (typing) universes, which conceptually is an extension of alphabets in the UTP. Indeed we shall think of universes as alphabets that additionally record type information. Such must be taken into account when, for example, defining the operators in Table 1 that expect alphabets — implicitly they all take universes now. In the following section, we discuss some of the core operators on alphabetised predicates.

3.4. Core Predicate Operators

The core predicate operators are contained in the ProofPower theory utp-pred. They include alphabet extension and restriction, the common logical connectives, equality of expressions, substitution, refinement, lattice-theoretic operations, and operators that allow one to bridge between object and host language logic. Our definitions are based on the ones in [Oli05, OCW07] with the difference that operators have to construct the *universe* of the result rather than its alphabet, in addition to the binding set.

As a first example, alphabet extension P_{+a} of predicates is specified. In our encoding, this operator does not take an alphabet a as its argument, but a universe u which indirectly defines an alphabet, but also the types of its variables. The ProofPower-Z definition is given below.

$$
\begin{array}{|l}
\_ \oplus_P \_ : \mathit{WF\_ALPHA\_PREDICATE\_UNIVERSE} \to \mathit{ALPHA\_PREDICATE} \\
\hline
\forall\, p : \mathit{ALPHA\_PREDICATE};\ u : \mathit{UNIVERSE} \mid \\
\quad (p, u) \in \mathit{WF\_ALPHA\_PREDICATE\_UNIVERSE} \bullet \\
\quad\quad p \oplus_P u \ = \ (\{b : \mathit{Bindings}_U(p.2 \oplus_U u) \mid (\mathit{Alphabet}_P\ p \lhd b) \in p.1\},\ p.2 \oplus_U u)
\end{array}
$$

The semantic restriction $\mathit{WF\_ALPHA\_PREDICATE\_UNIVERSE}$ on the domain of the function requires the universe of the predicate p and the argument universe u to be compatible (they need not be disjoint). The universe of the resulting predicate is obtained by extending the universe of p with u. The binding set contains all bindings of the extended universe which, if restricted to the alphabet of p, are original bindings of p. The effect of this is that no constraints are placed on those variables in u which are not in the alphabet of p. The definition of alphabet restriction is even simpler: we apply domain subtraction to all bindings of the predicate to obtain the bindings of the restricted predicate. We observe that the above alphabet extension is not the alphabet extension of relations defined in Table 1. For the latter, we provide an analogue operator ($\_ \oplus_R \_$) in utp-rel that also constrains variables to retain their value.

It is useful noting that subscripts are used to highlight membership of operators to specific UTP theories. Above, for instance, the subscript P indicates that we are dealing with operators of the (most general) theory of plain predicates. Other subscripts R, D, REA, CSP and C are later employed when specifying operators in more concrete theories, namely those of relations, designs, reactive processes, CSP processes and *Circus*. Additionally, we use the subscript V for functions on values, E for functions on expressions, and U for universe operators. We later also make use of the subscript T for general operators on theories.

The logical constants *true* and *false* are characterised by the operators $\mathit{True}_P\ u$ and $\mathit{False}_P\ u$; both have a universe argument. The bindings of *true* are exactly the ones of the universe (given by $\mathit{Bindings}_U\ u$), and the binding set of *false* is always empty. The exclusion of empty types guarantees that there is at least one binding in $\mathit{Bindings}_U\ u$. Thus $\mathit{True}\,u \neq \mathit{False}\,u$ and furthermore $P \neq \neg\, P$ are universally valid theorems, as one would expect; otherwise this would not generally be the case.

Negation is defined by constructing the complementary set of bindings with respect to the bindings of the predicate's universe. (The $\mathit{Bindings}_U$ function is again used to infer them.)

$$\begin{array}{|l}
\neg_P : \mathit{ALPHA\_PREDICATE} \rightarrow \mathit{ALPHA\_PREDICATE} \\
\hline
\forall\, p : \mathit{ALPHA\_PREDICATE} \bullet \neg_P\ p\ =\ (\mathit{Bindings}_U\, p.2 \setminus p.1,\ p.2)
\end{array}$$

The universe is not altered by the operator. To provide an example of a binary logical connective, we present below the definition of conjunction.

$$\begin{array}{|l}
\_ \wedge_P \_ : \mathit{WF\_ALPHA\_PREDICATE\_PAIR} \rightarrow \mathit{ALPHA\_PREDICATE} \\
\hline
\forall\, p_1, p_2 : \mathit{ALPHA\_PREDICATE} \mid (p_1, p_2) \in \mathit{WF\_ALPHA\_PREDICATE\_PAIR} \bullet \\
\qquad p_1 \wedge_P p_2\ =\ ((p_1 \oplus_P p_2.2).1 \cap (p_2 \oplus_P p_1.2).1,\ p_1.2 \oplus_U p_2.2)
\end{array}$$

The domain type $\mathit{WF\_ALPHA\_PREDICATE\_PAIR}$ imposes restrictions that require the two arguments to be compatible predicates, meaning their universes must be compatible. We obtain the binding set of the result by intersecting the binding sets of the predicates after extension of their universes to the joint universe. The universe of the result emerges from combining the universes of the individual predicates. Disjunction is defined analogously using union of the binding sets instead of intersection.

Two other operators of special interest are equality and substitution. Equality constructs a predicate from two expressions and needs to be supplied with a universe in whose context the equality is considered.

$$\begin{array}{|l}
=_P : \mathit{WF\_Equals}_P \rightarrow \mathit{ALPHA\_PREDICATE} \\
\hline
\forall\, u : \mathit{UNIVERSE};\ e_1, e_2 : \mathit{EXPRESSION} \mid (u, e_1, e_2) \in \mathit{WF\_Equals}_P \bullet \\
\quad =_P (u, e_1, e_2)\ =\ (\{b : \mathit{Bindings}_U\ u \mid \mathit{Eval}(b, e_1) = \mathit{Eval}(b, e_2)\}, u)
\end{array}$$

The semantic restriction $\mathit{WF\_Equals}_P$ includes all triples (u, e_1, e_2) where (u, e_1) and (u, e_2) are well-defined universe and expression pairs. This means that the universe must include the free variables of each of the expressions. We then select all bindings from the binding set of u in which evaluation of the expressions yield the same value. We assume that the expressions are type-correct with respect to u; this ensures that $\mathit{Eval}(b, e_1)$ and $\mathit{Eval}(b, e_2)$ are well-defined for any binding of u.

The definition of substitution is slightly more complicated. Again, we have to provide a universe u, the name n of the variable to be substituted, and an expression e. Substitution is an example of an operator for which we need to restrict applicability to cases where type-correctness is guaranteed. For this we first define the domain of the corresponding semantic function as follows.

$$\begin{array}{|l}
\mathit{WF\_Subst}_P\ \widehat{=}\ \{p : \mathit{ALPHA\_PREDICATE};\ n : \mathit{NAME};\ e : \mathit{EXPRESSION} \mid \\
\quad n \in \mathit{Alphabet}_P\ p \wedge \\
\quad (\mathit{Alphabet}_P\ p, e) \in \mathit{WF\_ALPHABET\_EXPRESSION} \wedge \\
\quad \mathit{typeof}_E(e, p.2) \subseteq \mathit{typeof}(n, p.2)\}
\end{array}$$

The first two conjuncts establish that the substituted variable as well as the free variables of the expression are contained in the universe. The third conjunct establishes that the substituted expression denotes a value of the right type, that is the type of n. Below the definition of substitution is presented.

$$\begin{array}{|l}
/_P : \mathit{WF\_Subst}_P \rightarrow \mathit{ALPHA\_PREDICATE} \\
\hline
\forall\, p : \mathit{ALPHA\_PREDICATE};\ n : \mathit{NAME};\ e : \mathit{EXPRESSION} \mid \\
\quad (p, n, e) \in \mathit{WF\_Subst}_P \bullet \\
\qquad /_P\ (p, n, e)\ =\ (\{b : \mathit{Bindings}_U\ p.2 \mid b \oplus \{n \mapsto \mathit{Eval}(b, e)\}\ \in\ p.1\}, p.2)
\end{array}$$

The intuition behind this definition is that, for b to be a binding of the substituted predicate, the expression e has to evaluate to some value which the variable might have had in the original predicate to render it true.

To give an example, the binding set that represents the predicate $x = 2$ has only one binding $x \mapsto 2$. The substitution $(x = 2)[x \backslash x + 1]$, which is equivalent to the predicate $x + 1 = 2$, thus contains those bindings where $x + 1$ evaluates to 2, that is $x \mapsto 1$. The functional override is necessary to accommodate additional variables of the predicate unaffected by the substitution.

As a final remark we discuss why precisely we need the restriction $\mathit{typeof}_E(e, p.2) \subseteq \mathit{typeof}(n, p.2)$ in order to apply substitution. One reason is that it is not possible to prove, for example, distributivity of substitution through negation without it. The conjecture $(\neg\, \mathit{okay})[\mathit{okay} \backslash 1] = \neg\, (\mathit{okay}[\mathit{okay} \backslash 1])$ illustrates the problem, if we assume *okay* to be of boolean type. The following proof sketch shows how this leads to a contradiction, if the type restrictions are not enforced like we do.

$$(\neg\, \mathit{okay})[\mathit{okay} \backslash 1] \;=\; \neg\, (\mathit{okay}[\mathit{okay} \backslash 1])$$

$\equiv$ “*okay* is an abbreviation for $\mathit{okay} = \mathbf{true}$ where **true** is a boolean value.”

$$(\neg\, \mathit{okay} = \mathbf{true})[\mathit{okay} \backslash 1] \;=\; \neg\, (\mathit{okay} = \mathbf{true})[\mathit{okay} \backslash 1]$$

$\equiv$ “*okay* is of boolean type”

$$(\mathit{okay} = \mathbf{false})[\mathit{okay} \backslash 1] \;=\; \neg\, (\mathit{okay} = \mathbf{true})[\mathit{okay} \backslash 1]$$

$\equiv$ “substitution”

$$(1 = \mathbf{false}) \;=\; \neg\, (1 = \mathbf{true})$$

$\equiv$ “evaluation of equalities; in what follows *false* and *true* are predicates.”

$$\mathit{false} \;=\; \neg\, \mathit{false}$$

$\equiv$ “simplification of negation”

$$\mathit{false} \;=\; \mathit{true}$$

This kind of problem is overcome if we substitute *okay* with a value of its type. Moreover, it is possible to prove the general distributivity law for substitution through negation with the additional assumptions on the type of the substituted expression. This exemplifies how in our encoding some of the operators have to integrate assumptions about typing to support proofs of essential laws and reasoning in general.

The last operator we consider is refinement. Unlike the operators we have encountered before, the result of $P \sqsubseteq Q$ directly constitutes a predicate of the host logic (ProofPower), and thus it is not the encoding of some alphabetised predicate of the object language (UTP). For its definition, it is useful to introduce a function *Tautology* which determines if an encoded predicate is universally true. This corresponds to the $[\_]$ operator used in the UTP, which universally quantifies over all variables of the alphabet.

$$\begin{array}{|l}
\mathit{Tautology} : \mathit{ALPHA\_PREDICATE} \rightarrow \mathbb{B} \\
\hline
\mathit{Tautology}\ p \;\Leftrightarrow\; p = \mathit{True}_P\,(p.2)
\end{array}$$

The definition is in fact equivalent to demanding that the bindings of the predicate p have to be equal to the binding set of its universe. With this definition we can conveniently define refinement as follows.

$$\begin{array}{|l}
\_ \sqsubseteq \_ : \mathbb{P}\ \mathit{WF\_ALPHA\_PREDICATE\_PAIR} \\
\hline
\forall\, p_1, p_2 : \mathit{ALPHA\_PREDICATE} \mid (p_1, p_2) \in \mathit{WF\_ALPHA\_PREDICATE\_PAIR} \bullet \\
\quad p_1 \sqsubseteq p_2 \;\Leftrightarrow\; \mathit{Tautology}\,(p_2 \Rightarrow_P p_1)
\end{array}$$

Implication is defined in the usual way as $P \Rightarrow Q =_{df} \neg\, P \vee Q$. The encoding of refinement in the mechanised semantics enables us to state and mechanically prove refinement conjectures about particular specifications; this is further discussed in Section 5.2.

Further operators are specified to characterise, for example, universal and existential quantification, the least upper and greatest lower bound of predicate sets, and the weakest and strongest fixed points of functions on predicates. For their inspection we point to `http://www.cs.york.ac.uk/circus/tp/tools.html` where all ProofPower scripts are published. In the next section we will look at the characterisation of UTP theories.

3.5. Characterisation of UTP Theories

UTP theories are modelled as records: elements of a schema type whose components define the theory's universe, and a set of healthiness conditions.

$$\begin{array}{|l}
\textit{UTP\_THEORY} \\ \hline
\textit{THEORY\_UNIVERSE} : \textit{UNIVERSE} \\
\textit{HEALTH\_CONDS} : \mathbb{P}\ \textit{HEALTH\_COND} \\ \hline
\end{array}$$

The alphabet of the theory can be inferred from its universe using the previously introduced $Alphabet_U$ function, hence there is no need to record it separately. The theory universe determines the universe of the predicates of the theory. The next section explains how they can be determined.

Healthiness conditions are elements of a type *HEALTH_COND* containing all idempotent, partial functions from *ALPHA_PREDICATE* to *ALPHA_PREDICATE*. To specify it we first introduce another set *ALPHA_FUNCTION*. It contains all partial (unary) functions on *ALPHA_PREDICATE* that are valid in the sense that they preserve compatibility of predicates.

$$\begin{array}{|l}
\textit{ALPHA\_FUNCTION}\ \widehat{=}\ \{f : \textit{ALPHA\_PREDICATE} \nrightarrow \textit{ALPHA\_PREDICATE} \mid \\
\quad \forall\, p_1, p_2 : \textit{ALPHA\_PREDICATE} \mid (p_1, p_2) \in \textit{WF\_ALPHA\_PREDICATE\_PAIR} \land \\
\qquad \{p_1, p_2\} \subseteq \operatorname{dom} f \bullet (f(p_1), f(p_2)) \in \textit{WF\_ALPHA\_PREDICATE\_PAIR}\}
\end{array}$$

We recall that for (p_1, p_2) to be an element of *WF_ALPHA_PREDICATE_PAIR*, the predicates have to be compatible. The definition thus states that for f to be a member of *ALPHA_FUNCTION*, any two compatible predicates in the domain of f have to be mapped to predicates which are also compatible.

The justification for this restriction is that when, for example, specifying properties of functions, there is a risk of undefinedness. To illustrate this, we consider the set $\textit{monotonic} \subseteq \textit{ALPHA\_FUNCTION}$ which records whether a function is monotonic. Formally, we have to establish that $[f(p_1) \Rightarrow f(p_2)]$ holds under the assumption $[p_1 \Rightarrow p_2]$. First, for $p_1 \Rightarrow p_2$ to be meaningful, we only consider predicates p_1 and p_2 that are compatible; however, for $f(p_1) \Rightarrow f(p_2)$ to be meaningful the above properties is additionally required. Additionally, observe that restricting *ALPHA_FUNCTION* to total functions would be too strong. An example case is the healthiness function $\mathbf{H1}(p) = \textit{okay} \Rightarrow p$. It is only defined for predicates p in which the type of *okay* is boolean if it occurs, therefore it is not total on *ALHPA_PREDICATE*.

We give below the definition for *HEATH_COND*, which captures the requirement for idempotence.

$$\begin{array}{|l}
\textit{HEALTH\_COND} : \textit{ALPHA\_FUNCTION} \\ \hline
\forall\, h : \textit{HEALTH\_COND} \bullet h \mathbin{;} h\ =\ h
\end{array}$$

Because the elements of *HEALTH_COND* are partial functions, any concrete definition of a function as a member of *HEALTH_COND* must also specify the function's domain. Partiality has another pragmatic advantage here: it enables us to define healthiness conditions that only apply to predicates with specific alphabets, for example homogeneous predicates. Idempotence is captured by h being invariant with regards to relational composition with itself. This is equivalent to $h(x) = x$ for any x in the range of h.

The type *UTP_THEORY* allows us to represent arbitrary instantiations of UTP theories within the same ProofPower reasoning scope. To make the process of constructing theories more convenient, we provide functions for generic instantiation, instantiation through strengthening existing theories, or specific instantiation of common UTP theories. The inherent hierarchy of various types of UTP theories is directly reflected by the ProofPower definitions which provide their instantiation means.

Figure 1 on page 7 presents an overview of all ProofPower theories that are part of the mechanisation. The theories in the middle column encapsulate the core definitions previously explained, and do not contribute specific definitions for UTP theory encodings. An exception is utp-theory, which beyond providing the types and functions for handling theory instances also specifies the instantiation means for the theory of plain predicates explained in the sequel. In the right column we find a ProofPower theory for each UTP theory that

Encoding of **H1** in utp-des.

$$\begin{array}{|l} H1 : \mathit{HEALTH\_COND} \\ \hline \operatorname{dom} H1 = \mathit{DES\_COMPATIBLE} \land \\ \quad (\forall p : \mathit{DES\_PREDICATE} \bullet H1\ p = \mathit{OKAY} \Rightarrow_P p) \end{array}$$

Encoding of **H2** in utp-des.

$$\begin{array}{|l} H2 : \mathit{HEALTH\_COND} \\ \hline \operatorname{dom} H2 = \mathit{DES\_COMPATIBLE} \land \\ (\forall p : \mathit{DES\_PREDICATE} \bullet H2\ p = p\ ;_R\ J\,(\mathit{Contract}_U\,(p.2, \mathit{out\_a}\,(\mathit{Alphabet}_P\ p)))) \end{array}$$

Figure 2: Encoding of the healthiness conditions **H1** and **H2** for designs.

is encoded. Arrows between boxes indicate the static inclusion dependency between ProofPower theories, which ensures that more specific theories can access the definitions and theorems of more general ones.

The theory of relations (utp-rel) is the most general one placing no restrictions on the predicates other than requiring their alphabets to be homogeneous. The other theories encapsulate specific computational paradigms. Designs (utp-des) are, as previously mentioned, used to model computations that may exhibit non-termination. Reactive designs (utp-rea) describe reactive processes which can have sequential behaviour while continuously interacting through communication events with the environment. The theory of CSP (utp-csp) provides an enriched model of failures and divergences, the canonical semantics for CSP [Hoa85, Ros97, CW04]. Finally the theory of *Circus* (utp-circus) provides a suitable model for the *Circus* language [CSW03], which, as said before, is an integration of Z and CSP. The left-most theory z_library is not part of the encoding. We include it to import the default laws and utilities for the ProofPower encoding of Z, provided by ProofPower-Z, and utp-z-ext is a custom extension of these laws.

At the bottom of this hierarchy resides the general theory of alphabetised predicates, which has no healthiness conditions or restrictions on the theory universe.

$$\begin{array}{|l} \mathit{InstPredTheory} : \mathit{UNIVERSE} \to \mathit{UTP\_THEORY} \\ \hline \forall u : \mathit{UNIVERSE} \bullet \mathit{InstPredTheory}\ u \ = \ \mathit{InstTheory}\,(u, \varnothing) \end{array}$$

Here $\mathit{InstTheory}\,(u, hs)$ yields an element of $\mathit{UTP\_THEORY}$ whose theory universe and healthiness functions are trivially determined by u and hs.

We can strengthen an existing theory by adding further healthiness functions while maintaining its alphabet and typing universe using the function $\mathit{SpecialiseTheory}\,(th, hs)$. The following definition illustrates how we exploit it in building UTP theory hierarchies. In particular, we provide a function to instantiate a theory of designs as discussed in Section 2; its definition is based on an instantiation of a theory of relations.

$$\begin{array}{|l} \mathit{InstDesTheory} : \mathit{DES\_UNIVERSE} \to \mathit{UTP\_THEORY} \\ \hline \forall u : \mathit{DES\_UNIVERSE} \bullet \mathit{InstDesTheory}\ u \ = \ \mathit{SpecialiseTheory}\,(\mathit{InstRelTheory}\ u, \{H1, H2\}) \end{array}$$

The encoding of the healthiness conditions **H1** and **H2** is given in Figure 2. $\mathit{DES\_COMPATIBLE}$ is the set of all predicates whose universe is compatible with constraining $okay$ and $okay'$ to boolean values. $\mathit{Contract}_U(u, a)$ contracts a universe u to the variables in a, and the encoding of J is in Appendix A.4.

$\mathit{InstRelTheory}(u)$ actually yields $\mathit{InstPredTheory}(u)$ as a result, but imposes further restrictions on the alphabet of u; relational theories must contain only undashed and single-dashed names. Besides, dashed variables, if present, must have the same types as their corresponding undashed counterparts. These restrictions are captured by the set *REL_UNIVERSE*, which acts as the domain of *InstRelTheory*.

$$\begin{array}{l} \mathit{REL\_UNIVERSE} \mathrel{\widehat=} \{u : \mathit{UNIVERSE} \mid \mathit{Alphabet}_U\ u \in \mathit{REL\_ALPHABET} \land \\ \quad (\forall n : \mathit{NAME} \mid \{n, \mathit{dash}\ n\} \subseteq \mathit{Alphabet}_U\ u \bullet \mathit{typeof}(n, u) = \mathit{typeof}(\mathit{dash}\ n, u))\} \end{array}$$

REL_ALPHABET is defined as *undashed* ∪ *dashed_once*, the set of names which are either undashed or have a single dash. For generality, relational alphabets do not necessarily have to be homogeneous.

By equating the domain of the instantiation functions *InstRelTheory*, *InstDesTheory*, and so on with *REL_UNIVERSE*, *DES_UNIVERSE*, and so on, we are able to impose suitable restrictions on the alphabet and types of variables of the respective UTP theories. For instance, *DES_UNIVERSE*, used in our example above specifies restrictions requiring all design theories to incorporate the boolean variables *okay* and *okay′*.

$$\begin{array}{l} \mathit{DES\_UNIVERSE} \mathrel{\widehat=} \\ \quad \{u : \mathit{REL\_UNIVERSE} \mid \mathit{Alphabet}_U\ u \in \mathit{DES\_ALPHABET} \land \mathit{typeof}(\mathit{okay}, u) = \mathit{BOOL\_VAL}\} \end{array}$$

This exemplifies how modularity is exploited in defining not only instantiation functions, but also the alphabets and universes of theories within the hierarchy. Here, the definition of the type *DES_UNIVERSE* imposes additional restrictions on the elements of *REL_UNIVERSE*. Similarly, *DES_ALPHABET*, whose definition we omit, restricts *REL_ALPHABET* so that *okay* and *okay′* are present.

With the instantiation functions, we can now define theory families: sets that contain all the theories of a particular kind, albeit with different universes. They are given by the range of the corresponding instantiation functions. For example, *REL_THEORY*, the family of relational theories, is given by ran *InstRelTheory*; *DES_THEORY*, the family of design theories, by ran *InstDesTheory*, and so on. The sets *REL_THEORY*, *DES_THEORY*, ... effectively allow us to reason about the possible instantiations of the respective theories. We will make use of them in formulating laws for particular theory families.

The approach we present here is not just an effort towards supporting dynamic instantiation of theories, but a uniform mechanisation of UTP theories. Uniformity facilitates the development of reusable laws and proof tactics, and therefore automation. For example, by strengthening theories we exploit the fact that any predicate of the new theory fulfils the healthiness conditions of the extended theory, and so its laws apply.

3.6. Theory Predicates

One of the main motivations for instantiation is to permit reasoning about the predicates of particular UTP theories, and construct verification arguments based on refinement. Although *UTP_THEORY* has the ingredients to distinguish various theories, we have to provide further means to characterise the predicates of these theories. The predicates of a *UTP_THEORY* object are determined by the function *TheoryPredicates*.

$$\mathit{TheoryPredicates} : \mathit{UTP\_THEORY} \rightarrow \mathbb{P}\ \mathit{ALPHA\_PREDICATE}$$

$$\begin{array}{l} \forall\, \mathit{th} : \mathit{UTP\_THEORY} \bullet \mathit{TheoryPredicates}\ \mathit{th}\ = \\ \quad \{p : \mathit{ALPHA\_PREDICATE} \mid p.2 = \mathit{th}.\mathit{THEORY\_UNIVERSE} \land \\ \qquad (\forall\, h : \mathit{th}.\mathit{HEALTH\_CONDS} \bullet p \in \operatorname{dom}\ h \land h\,(p) = p)\} \end{array}$$

The definition implies that for predicates to belong to a particular theory, they have to share the theory's universe, and fulfil its healthiness conditions. Using this function, we define the sets of all relational predicates, design predicates, and so on. For example, the set of all designs can be characterised as follows.

$$\mathit{DESIGN} = \{p : \mathit{ALPHA\_PREDICATE} \mid (\exists\, \mathit{th} : \mathit{DES\_THEORY} \bullet p \in \mathit{TheoryPredicates}\ \mathit{th})\}$$

Membership of some p to *DESIGN* implies that there is an instantiation of a design theory to which p belongs, in other words p possesses a permissible universe and fulfils the healthiness condition for design theories. Beyond this, we also introduced two further, less restrictive sets: *DES_PREDICATE* and *DES_COMPATIBLE*. The first contains all predicates that have an adequate universe for a design predicate, but need not fulfil the healthiness conditions. The second one only requires compatibility with some design theory universe, hence if $okay$ and $okay'$ occur they have to be of boolean type. $DES\_COMPATIBLE$ is, for instance, used in specifying the domain of the healthiness condition **H1** in Figure 2. Analogue sets are defined for all UTP theories in the hierarchy.

Finally, it is possible to take an arbitrary predicate and apply the healthiness conditions of a UTP theory to obtain a healthy one with respect to that theory (or in fact any arbitrary set of healthiness functions). The corresponding function is $ApplyHealthConds\,(p, hs)$, which expects a predicate p and a sequence of elements from $HEALTH\_COND$. The result is obtained by folding the application of the functions in hs. The reason for using sequences rather than sets of healthiness functions is to accommodate cases in which the healthiness functions do not commute, and hence their order of application is significant. The definition of $ApplyHealthConds$ is included in Appendix A.2 for inspection.

3.7. Theory-specific Operators

It is sometimes necessary to impose additional restrictions on the arguments of operators; for example, sequence, or relational composition, is only defined if the dashed variables in the alphabet of the first relation match the undecorated variables of the alphabet of the second relation. Besides, there exist operators whose application only makes sense in the context of particular UTP theories and their predicates.

Therefore, operator definitions may specify restrictions on the arguments. In our encoding, the most fundamental restriction is that predicates must have compatible universes, which agree on the types of the common variables. Additionally, functions representing operators of specific UTP theories may only be partially defined on $ALPHA\_PREDICATE$: the argument has to be a predicate of the respective theory. Similarly, the range may be specified to identify predicates of specific theories. An example is the definition of the Skip operator for designs, which is different from the relational Skip $\mathbf{II}_R$.

$$\mathbf{II}_D : DES\_UNIVERSE \rightarrow DESIGN$$

$$\forall\, u : DES\_UNIVERSE \bullet \mathbf{II}_D\,(u) = True_P\,(u) \vdash_D \mathbf{II}_R\,(u)$$

Using total functions with a more specific characterisation of their domain and range simplifies proofs of theorems involving their application: it factors the proof of properties of the range into the consistency proof of the functions. For example, the total function axiom of $\mathbf{II}_D$ allows us to easily prove $\mathbf{II}_D\,(u) \in DESIGN$ by showing $u \in DES\_UNIVERSE$. Section 7 returns to this issues in the light of automating proofs.

To illustrate how operators are used to encode specifications, we present a program that nondeterministically chooses to toggle the value of a variable x between 0 and 1, or leave it unchanged. Assuming the type of x is $\{0, 1\}$, the program $(x := 1 \lhd x = 0 \rhd x := 0) \sqcap \mathbf{II}_{\{x\}}$ is encoded as follows.

$$(Assign_R\,(u, \langle x \rangle, \langle\, Val(Int(1)) \rangle) \lhd_R \;=_P\,(u,\, Var(x),\, Val(Int(0))) \rhd_R\; Assign_R\,(u, \langle x \rangle, \langle\, Val(Int(0)) \rangle))$$
$$\sqcap_R\,(\mathbf{II}_R\; u)$$

The various semantic functions used here are '$Assign_R$', '$=_P$', '$(\_ \lhd_R \_ \rhd_R \_)$', '$\sqcap_R$' and '$\mathbf{II}_R$' which are apart from equality all defined in the corresponding ProofPower theory `utp-rel` for relational predicates. A suitable universe u must moreover be provided to apply some of them. ProofPower-Z definitions for these operators are included in Appendix A.3 for scrutiny but are not discussed in detail here.

In the next section we look at the encoding of more elaborate theories by examining the mechanisation of *Circus*. Although we use a similar approach to the one discussed in the current section, it gives rise to more interesting healthiness conditions and the possibility of structuring theories in hierarchies. We thereby besides develop the preliminaries for later encoding concrete *Circus* specifications.

4. More Elaborate Theories: *Circus*

In this section we review some of the fundamental elements of *Circus* and its semantic model, and illustrate its encoding in ProofPower as a theory using our mechanisation.

The *Circus* language combines aspects of both sequential and reactive programming. It is therefore especially suitable for the specification and development of state-rich reactive systems. As in CSP, the key

```
**channel** $Insert5pCoin, Insert10pCoin, DispenseItemBtn, ReturnChangeBtn$
**channel** $GiveItems, GiveChange : \mathbb{N}$
| $capacity, item\_price : \mathbb{N}_1$
**process** $SimpleVendingMachine \mathrel{\widehat{=}}$ **begin**
**state** $State == [credit : \mathbb{N};\ stock : \mathbb{N}]$
$InitState == [State' \mid credit' = 0 \land stock' = capacity]$
$InsertMoney \mathrel{\widehat{=}}$
    $Insert5pCoin \longrightarrow credit := credit + 5 \ \Box\ Insert10pCoin \longrightarrow credit := credit + 10$
 ___ $CalcDispense$ ___
| $\Xi State;\ items!, left\_credit! : \mathbb{N}$
|___
| $items! \leq credit \operatorname{div} item\_price \land left\_credit! = credit - items! * item\_price]$
|___
$DispenseItem \mathrel{\widehat{=}}$
    $DispenseItemBtn \longrightarrow$ **var** $items, left\_credit : \mathbb{N} \bullet CalcDispense;$
        $((items \neq 0 \land items \leq stock)\ \&$
            $(GiveItems!items \longrightarrow credit, stock := left\_credit, stock - items))$
                $\Box$
        $(items = 0 \lor items > stock)\ \&\ Skip$
$ReturnChange \mathrel{\widehat{=}}$
    $ReturnChangeBtn \longrightarrow ((credit \neq 0)\ \&\ GiveChange!credit \longrightarrow credit := 0)\ \Box\ (credit = 0)\ \&\ Skip$
    $\bullet\ InitState;$
        $\mu X \bullet InsertMoney\ \Box\ DispenseItem\ \Box\ ReturnChange\ ;\ X$
**end**
```

Figure 3: *Circus* process of the simple vending machine.

concept in *Circus* is that of a process; it encapsulates state, and actions which operate on the state while at the same time interacting with the environment by means of communication events. The state of a process is specified as a Z (state) schema, and actions are defined through a mixture of Z operation schemas, CSP constructs, and guarded commands [Dij76]. The state and internal actions are local to the process and thus not externally visible; the behaviour is exposed by a designated main action of the process. *Circus* also defines CSP-like operators such as parallelism, interleaving and hiding to combine processes.

4.1. Circus: Notation and Example

Figure 3 depicts a simple example of a *Circus* process. It specifies the behaviour of a minimalistic vending machine. We assume there is only one type of item dispensed by the machine whose price is *item_price*. It is possible to either insert a 5p or a 10p coin, press the dispense button, or press the return change button. An idealising assumption we make is that there is no limit to the amount of credit that may be inserted. Pressing the dispense button results in one or more items being dispensed, provided sufficient money has been entered and the machine has enough items in stock to serve the request. The actual number of items dispensed is left unspecified, therefore an implementation can make a choice here. Possible resolutions are, for example, to dispense exactly one item, or alternatively dispense as many items as the current credit warrants. Pressing the return change button returns whatever credit currently resides in the machine. The buttons may be pressed at any time. Initially, the number of items in stock is given by the constant *capacity*.

The *Circus* model first declares the communication channels. The four typeless channels $Insert5pCoin$, $Insert10pCoin$, $DispenseItemBtn$, and $ReturnChangeBtn$ represent the events of either inserting coins into the machine or pressing one of its buttons. The next two channels $GiveItems$ and $GiveChange$ are both of type natural, and communicate the number of items dispensed by the machine, and the amount of change returned, if so. The global constants *capacity* and *item_price* respectively determine the stock capacity and price of an item sold, and are loosely introduced as positive natural numbers.

The process body first declares the state of the process in a **state** paragraph. In the example, we have two state components, *credit* and *stock*, which record the amount of credit in the machine and the number of items in stock, respectively. They are both represented by natural numbers. The rôle of *InitState* is to suitably initialise the state; here, the initial value of *credit* is set to zero, and the value of *stock* to *capacity*.

The following three auxiliary actions *InsertMoney*, *DispenseItem*, and *ReturnChange* specify the behaviour for inserting money or pressing one of the two buttons. In their definition we use a mixture of CSP constructs, including prefixing and external choice, as well as sequential commands, such as assignments. The guarded action $P \mathbin{\&} A$ proceeds if predicate P holds, and otherwise is blocked.

A fourth action *CalcDispense* is specified by a Z schema and its purpose is to calculate the number of items dispensed, and the amount of credit subsequently remaining in the machine. The schema components here include the state of the process, as well as two extra output variables *stock*! and *left_credit*! that are used to hold the computed values. $\Xi State$ in the declarative part of the schema ensures that the state variables are not changed by the operation (the values of dashed variables are equated with their undashed counterparts). Regarding the operation's behaviour, the predicate $items! \leq credit \operatorname{div} item\_price$ encapsulates the nondeterminism in dispensing any number of items for which the credit suffices; they may also be zero. The second conjunct $left\_credit! = credit - items! * item\_price$ determines the amount of credit remaining in the machine after the items are actually dispensed.

The *DispenseItem* action first waits for the *DispenseItemBtn* event to occur (press of button). It then invokes *CalcDispense* to determine the number of items to be dispensed and the remaining credit. This information is stored in the two local variables *items* and *left_credit*, introduced by the **var** statement. Following this, we specify a choice between two actions with complementary guards; due to the choice being external, it selects whichever one is enabled. In the case of sufficient items in stock and the number of items not being zero, the delivery of the items is signalled by communicating their number on the *GiveItems* channel. Afterwards, it updates the *credit* and *stock* state components accordingly. In the complementary case of insufficient items in stock, it terminates immediately without changing the state (*Skip*).

ReturnChange similarly waits for an occurrence of the *ReturnChangeBtn* event, and then outputs the current value of *credit* on the *GiveChange* channel, thereafter setting it to zero. This only happens if *credit* is not equal to zero, otherwise the operation does nothing. The behaviour we specify permits the button to be pressed at any time, but money only to be returned when there is some credit in the machine.

The main action of the process, following the '•', consists of two sequential commands: the first one invokes *InitState* to initialise the state, while the second one keeps letting the environment choose among the three auxiliary actions. The latter is realised by recursion: the name X is introduced to refer to the body of the recursion, and a recursive call after the choice ensures that it is offered over and over again.

4.2. The Circus UTP model

Reactive Processes. To model interactions with the environment and intermediate states, four auxiliary variables, and their dashed counterparts, are needed. They are *okay*, *tr*, *ref* and *wait*. The boolean variable *okay* is used to distinguish stable from divergent states. The initial observation *okay* indicates that the previous process has not diverged, and $okay'$ indicates that the current process has not reached a divergent state. Regarding the stable states of a process, a further distinction between intermediate and final states is recorded by the boolean variable *wait*; if *wait* is true, the previous process has reached an intermediate state, and $wait'$ records that the current observation is that of an intermediate state. So, if $wait'$ is false (and $okay'$ is true), we have reached a final state (termination of the process).

To record the interaction with the environment while the process executes, *tr* and *ref* are used. Specifically, *tr* records event traces that have already taken place when the process is started, and tr', by extending it, the events the process has engaged in ($tr' - tr$ yields the actual events contributed by the process). In any intermediate state ref' records the set of events which are refused by the process in that state; the refusal set is not relevant after termination, and therefore the initial variable *ref* is merely added as a technicality

Name	Definition	Informal Description
R1	$\mathbf{R1}(P) \mathrel{\widehat=} P \wedge tr \leq tr'$	A process cannot change the previous history of events.
R2	$\mathbf{R2}(P) \mathrel{\widehat=} P[tr, tr' \setminus \langle\rangle, tr' - tr]$	The process behaviour must be oblivious to events that happened prior to its activation.
R3	$\mathbf{R3}(P) \mathrel{\widehat=} \mathbf{II_{rea}} \lhd wait \rhd P$	A process is only activated when its predecessor has finished; intermediate stable states do not progress.

Table 2: Healthiness conditions for reactive processes.

to make predicates of the theory homogeneous, and therefore, for their relational composition to be well-defined. The auxiliary variables are needed to reason about the reactive behaviour, but we also allow the presence of arbitrary state variables as in plain sequential computations.

Reactive processes are characterised by the three healthiness conditions **R1** to **R3** listed in Table 2. **R1** captures that a process cannot alter any events that took place prior to its activation, hence the trace of any subsequent state has to be an extension of the initial trace tr. **R2** specifies that the process behaviour does not depend on events before its activation. This is so if it can be described solely in terms of $tr' - tr$. Lastly, **R3** requires that if the previous process is in an intermediate state, then the behaviour must be that of $\mathbf{II_{rea}}$, the reactive identity, defined as $\mathbf{II_{rel}} \lhd okay \rhd tr \leq tr'$. It is an identity for relational composition whenever $okay$ is true, and otherwise permits any subsequent behaviour as long as trace extension is maintained. The latter captures that after divergence any behaviour may be observable.

We encapsulate the UTP theory of reactive designs in the ProofPower theory utp-rea. The encoding approach is similar to the one for the theory of designs already explained in Section 3. We note that the theory of reactive designs is not a restriction of the theory of designs. This is so since **H1**-healthy predicates do not satisfy **R1**. In particular, **H1** requires the absence of any constraint on the behaviour when $okay$ is false; however, for a predicate to obey **R1** it has to guarantee $tr \leq tr'$ in all circumstances.

The encoding of utp-rea first introduces the three auxiliary variables $wait$, tr and ref and their dashed counterparts. Since we statically include utp-okay, we do not need to introduce $okay$ again.

$$wait, tr, ref : NAME$$
$$\{wait, tr, ref\} \subseteq undashed \wedge distinct\ \langle okay, wait, tr, ref\rangle$$

The $distinct$ relation states that the elements of a sequence of names are mutually distinct. This is conveniently expressed by requiring the sequence to be injective: $distinct\ s \Leftrightarrow s \in \mathrm{iseq}\, NAME$. The dashed versions of the variables are introduced via individual definitions $wait' \mathrel{\widehat=} dash\ wait$, and so on. The alphabet consisting of the four auxiliary variables, and their dashed counterparts, is finally introduced through the constant $ALPHABET\_OWTR \mathrel{\widehat=} \{okay, okay', wait, wait', tr, tr', ref, ref'\}$.

Next the two constants $REA\_ALPHABET$ and $REA\_UNIVERSE$ are introduced to specify the types of valid alphabets and universes for reactive process theories. We follow a similar approach to that in Section 3, however reusing the definitions of design alphabets and universes.

$$REA\_ALPHABET \mathrel{\widehat=} \{a : DES\_ALPHABET \mid ALPHABET\_OWTR \subseteq a\}$$

$$\begin{aligned} REA\_UNIVERSE \mathrel{\widehat=} \{&u : DES\_UNIVERSE \mid Alphabet_U\ u \in REA\_ALPHABET \wedge \\ &typeof\ (wait, u) = BOOL\_VAL \wedge \\ &typeof\ (tr, u) = SEQ\_EVENT\_VAL \wedge \\ &typeof\ (ref, u) = SET\_EVENT\_VAL\} \end{aligned}$$

Above, SEQ_EVENT_VAL and SET_EVENT_VAL are particular types (subsets of $VALUE$) that represent sets and sequence of events. Events are in turn characterised by channel / value pairs.

Additionally, we define $REA\_UNIVERSE\_MIN$ as the minimal reactive universe that contains no other than the auxiliary variables, that is $Alphabet_U\ REA\_UNIVERSE\_MIN = ALPHABET\_OWTR$.

$$\boxed{\begin{array}{lcl} \mathit{TReqTR'} & \mathrel{\widehat{=}} & =_P (\{tr, tr'\} \times \{\mathit{SEQ\_EVENT\_VAL}\}, \mathit{Var}(tr), \mathit{Var}(tr')) \\ \mathit{TRprfxTR'} & \mathrel{\widehat{=}} & =_P (\{tr, tr'\} \times \{\mathit{SEQ\_EVENT\_VAL}\}, \mathit{Rel}((\_ \leq_V \_), \mathit{Var}(tr), \mathit{Var}(tr')), \mathit{True}_E) \\ \mathit{TRdiffTR'} & \mathrel{\widehat{=}} & \mathit{Fun}_2((\_\mathit{SeqDiff}_V\_), \mathit{Var}(tr'), \mathit{Var}(tr)) \end{array}}$$

Figure 4: Utility definitions for the encoding of the reactive healthiness conditions.

As explained in the previous section, two predicate sets $\mathit{REA\_PREDICATE}$ and $\mathit{REA\_COMPATIBLE}$ are introduced to define sets of predicates that impose the correct type restrictions on the auxiliary variables, but not necessarily fulfil the healthiness conditions. Whereas predicates in $\mathit{REA\_PREDICATE}$ must also have a valid alphabet from $\mathit{REA\_ALPHABET}$, $\mathit{REA\_COMPATIBLE}$ merely requires the auxiliary variables to be of the correct type if they occur. These sets are needed to specify the domain of the healthiness functions **R1** to **R3**. In [OCW07], such is the set of all relational predicates, for the types of variables were statically determined. In our recast model we have to be more discriminating to avoid type conflicts.

To make the encoding of healthiness functions more readable, we provide a few supplementary definitions. For example, $\mathit{WAIT} \mathrel{\widehat{=}} =_P (\{\mathit{wait} \mapsto \mathit{BOOL\_VAL}\}, \mathit{Var}(\mathit{wait}), \mathit{True}_E)$ encodes the predicate wait (or more accurately $\mathit{wait} = \mathbf{true}$). In this construction, the universe is explicitly provided, and True_E is the encoding of the syntactic expression **true**, namely $\mathit{Val}(\mathit{Bool}(\mathit{True}))$. Other useful constants include $\mathit{TReqTR'}$ encoding $tr = tr'$, $\mathit{TRprfxTR'}$ encoding $tr \leq tr'$, and $\mathit{TRdiffTR'}$ encoding $tr' - tr$. They are given in Figure 4. Using these utility definitions, **R2**, for example, is encoded as follows.

$$\begin{array}{|l} R2 : \mathit{HEALTH\_COND} \\ \hline \mathrm{dom}\, R2 = \mathit{REA\_COMPATIBLE} \wedge \\ (\forall\, p : \mathit{REA\_COMPATIBLE} \bullet \\ \quad /_P\ (/_P\ (p \oplus_P \{tr, tr'\} \times \{\mathit{SEQ\_EVENT\_VAL}\}, tr, \mathit{Val}(\mathit{Seq}(\langle\rangle))), tr', \mathit{TRdiffTR'})) \end{array}$$

Since p, the predicate to which the function is applied, does not necessarily mention tr and tr', we extend its universe within the inner substitution to ensure that it does. This is always possible as compatibility of p with the universe $\{tr, tr'\} \times \{\mathit{SEQ\_EVENT\_VAL}\}$ is implied by p being an element of $\mathit{REA\_COMPATIBLE}$. The fact that we use $\mathit{REA\_COMPATIBLE}$ and not $\mathit{REA\_PREDICATE}$ for the domain of **R2** enables us to apply the function to a larger set of predicates, whose universe does not need to include all auxiliary variables for reactive predicates. For example, **R2** can be applied to the predicates of some design theory. We show how this is useful when defining linking functions between theories in Section 5.1.

The two other healthiness idempotents **R1** and **R3** are encoded in the same way. Their corresponding definitions are given in Appendix A.5. For **R3** we additionally need the encoding of $\mathbf{II_{rea}}$, the reactive Skip, which, as explained above, slightly differs from the relational Skip. This throws up a minor problem since we prefer to define theory-specific operators *after* the healthiness conditions and instantiation functions for the theory. The advantage of this order is that we are able to introduce sets that comprise the healthy predicates of some theory first, and subsequently refer to them in the definition of operators. Our strategy to break circularity is to introduce **R1**, **R2** and **R3** loosely first by only stating their type and domain. In this way, we can refer to them in the definition of operators, and complete their exact specification by means of supplementary constraints once all operators of the theory have been specified.

We define the instantiation function $\mathit{InstReaTheory}$ by strengthening a relational theory that has the correct universe. The set $\mathit{REA\_THEORY}$, defined as ran $\mathit{InstReaTheory}$, contains all reactive theories, and $\mathit{REA\_PROCESS}$ the set of predicates that belong to some reactive process theory. Their formal definitions, including the one of $\mathit{InstReaTheory}$ can be found in Appendix A.5 as well.

The remainder of the encoding is concerned with theory-specific operators. For example, **R** is introduced as the composition $\mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3}$ of the healthiness conditions: it takes any $\mathit{REA\_COMPATIBLE}$ predicate and turns it into a healthy one. A reactive design is the result of the application of **R** to some design, in general $\mathbf{R}(P \vdash Q)$. It is extensively used to define operators in the more specialised theories utp-csp for CSP processes and utp-circus for *Circus* actions; otherwise utp-rea does not define any further notable operators.

Name	Definition	Informal Description
CSP1	$\mathbf{CSP1}(P) \mathrel{\widehat=} P \lor (\lnot\, okay \land tr \le tr')$	No other guarantees are made on divergence than preservation of the history; $tr \le tr'$ is the *only* guarantee.
CSP2	$\mathbf{CSP2}(P) \mathrel{\widehat=} P\,;\, J$ where $J \mathrel{\widehat=} (okay \Rightarrow okay') \land \mathbf{II_{rel}}$	Nontermination may not be specified as a requirement. This is an analogue of **H2**, recast for reactive designs.
CSP3	$\mathbf{CSP3}(P) \mathrel{\widehat=} SKIP\,;\, P$	The value of *ref* holding the refused events of the previous process after termination is ignored.
CSP4	$\mathbf{CSP4}(P) \mathrel{\widehat=} P\,;\, SKIP$	A process does not restrict ref' after termination; refused events are irrelevant thereafter.
CSP5	$\mathbf{CSP5}(P) \mathrel{\widehat=} P \mathbin{\vert\vert\vert} SKIP$	Refusals of a process are subset closed. Hence, if a process in some circumstances refuses a set of events, any subset of those events must be refused as well.

Table 3: Healthiness conditions for CSP processes.

The paragraphs above have illustrated a general pattern for presenting the definitions of new UTP theories. Although the theory of reactive processes is not an extension of the theory of designs, we can nevertheless make use of some of the definitions in utp-des. Since the encoding does not impose global constraints, for example to capture typing, there is crucially no risk of interference when including a ProofPower theory for the sake of using its definitions to encode another UTP theory. This notably leads to a better exploitation of modularity and reuse of aspects of existing theory encodings.

CSP Processes. We follow the pattern already illustrated to encode the UTP theory for CSP (and *Circus* later on in this section). While reactive processes provide the most general notion of reactive behaviour, CSP processes require further constraints in the form of healthiness conditions that more specifically deal with the case of divergence. Table 3 includes the complete set of additional healthiness conditions for CSP processes of which only **CSP1** and **CSP2** are considered essential. The encoding of the ProofPower theory utp-csp for CSP processes follows the same schema as the one for reactive processes. The alphabet *CSP_ALPHABET* and universe *CSP_UNIVERSE* are simply equated with *REA_ALPHABET* and *REA_UNIVERSE*. We encode the healthiness conditions **CSP1** and **CSP2**, and define the instantiation function *InstCSPTheory* by specialising a reactive theory. The predicate sets *CSP_PREDICATE* and *CSP_COMPATIBLE* are defined as before to refer to predicates that have a compatible alphabet, and *CSP_PROCESS* is the cumulative set of predicates representing valid CSP processes.

The theory of CSP introduces a few more theory-specific operators. These are, for example, the constant processes *Skip*, *Stop* and *Chaos*, representing immediate termination, deadlock and divergence, as well as external choice $P_1 \Box P_2$ and prefixing $a \longrightarrow P$. We do not discuss the semantics and encoding of these operators in detail, which are available from [HJ98, CW04]. It is nevertheless useful to illustrate how the presence of universes affects the definitions. As an example, we consider the deadlocked process *Stop*.

$$STOP : WF\_Skip_{REA} \rightarrow CSP\_PROCESS$$

$$\forall\, u : WF\_Skip_{REA} \bullet STOP\ u = R\,(Assign_R\,(\{wait, wait'\} \times BOOL\_VAL, \langle wait\rangle, \langle False_E\rangle) \oplus_P u)$$

First, *STOP* takes a universe argument u. The predicate that defines *Stop* is $R(wait := \mathbf{false})$ where *ref* is assumed not to be in the alphabet of the assignment, to leave ref' unconstrained and thereby establish that all events are refused. This is reflected by the *ad hoc* construction of a universe for the assignment which only contains *wait* and $wait'$ (with the right type). For the resulting predicate to have the correct universe, we need to extend it with u, either before or after applying R. WF_Skip_{REA} is a semantic restriction which requires the alphabet of u to be homogeneous and respect the types of auxiliary variables.

Name	Definition	Informal Description
C1	$\mathbf{C1}(A) \mathrel{\widehat=} A\,;\ \mathit{Skip}$	Analogue of **CSP4** for *Circus* actions. The *Circus* *Skip* moreover takes into account state variables.
C2	$\mathbf{C2}(A) \mathrel{\widehat=} A \mathbin{[\![\![} v \mid \varnothing \mathbin{]\!]\!]} \mathit{Skip}$	Analogue of **CSP5** for *Circus* actions. $[\![\![\ldots]\!]\!]$ is the *Circus* interleaving and v the state alphabet of A.
C3	$\mathbf{C3}(A) \mathrel{\widehat=} R(\neg A^f_f\,;\ \mathit{true} \vdash A^t_f)$	The precondition P of a *Circus* process expressed as a reactive design $\mathbf{R}(P \vdash Q)$ contains no dashed variables.

Table 4: Healthiness conditions for *Circus* actions and processes.

Circus. Finally, at the most concrete layer we have the theory of *Circus*. It is constructed by further strengthening the theory of CSP with the healthiness conditions **C1**, **C2** and **C3** listed in Table 4. For the *Circus* theory we define *CIRCUS_ALPHABET* and *CIRCUS_UNIVERSE* to be exactly the same as *CSP_ALPHABET* and *CSP_UNIVERSE*. The encoding of the healthiness conditions requires several theory-specific operators such as the *Circus* version of *Skip* and interleaving. The instantiation function *InstCircusTheory* specialises a theory obtained through *InstCSPTheory* with the healthiness functions **C1**, **C2** and **C3**. *CIRCUS_PREDICATE* and *CIRCUS_COMPATIBLE* similarly provide the sets of compatible predicates, and *CIRCUS_ACTION* all predicates that fulfil the healthiness conditions of *Circus*.

We additionally introduce a set *CIRCUS_PROCESS* which contains exactly those predicates from *CIRCUS_ACTION* whose alphabet only includes the auxiliary variables. Whereas *CIRCUS_ACTION* characterises main or auxiliary actions of a process, *CIRCUS_PROCESS* is used to characterise a process by assuming any state components are concealed.

To summarise, as a general recipe for encoding new UTP theories we first determine whether the theory to be defined is the specialisation of an existing theory, and if so import the corresponding ProofPower theory to reuse types and, in particular, the instantiation function of the existing theory. The sets, healthiness functions, operators, and so on, of the new theory are then presented in a uniform order. We start with the definition of auxiliary variables, alphabet and universe types, and with these subsequently specify the sets of compatible predicates. We then introduce healthiness functions and the theory's instantiation function, which usually is defined in terms of the instantiation function of the existing theory. Next, we introduce sets characterising the theory's instantiation family. The remaining definitions are concerned with theory-specific operators and corresponding semantic sets specifying domains of the operator functions. Where we have to defer the exact definitions of healthiness functions, because they require theory-specific operators, they are given last by virtue of supplementary constraints.

5. Reasoning about UTP Theories

In this section we examine how our mechanisation enables us to reason about UTP theory families in a general manner. We discuss separately our approaches for reasoning about general properties of theory families, and about properties of particular instantiations of theories and their predicates. Theory families can be, for example, all instances of design theories, reactive design theories, CSP theories, and so on, but also more abstract families characterised solely by certain properties their healthiness conditions must possess. Laws about more general theory families may be specialised to more specific theory families. Dealing with instances of theories, on the other hand, is important to formulate soundness and verification properties of particular specification and program encodings. To prove these, we usually have to resort to general laws that are valid across instantiations. Reasoning about instantiations is the subject of Section 6.

To illustrate our approach to reasoning about properties that are valid in families of theories, we consider the family of design theories. A general law that may be formulated is the following.

$$\vdash \forall\, th : \mathit{DES\_THEORY} \bullet \mathit{True}_P\,(th.\mathit{THEORY\_UNIVERSE}) \in \mathit{TheoryPredicates}\ th$$

It states that *true* (over the correct universe) is a valid predicate in any design theory instantiation. To prove this law we appeal to the healthiness conditions for designs. In the mechanical proof this is done when rewriting *TheoryPredicates th* into its definition (see Section 3.6); we can make use of the theory universe and healthiness conditions to characterise the predicates of *th*. The healthiness conditions of *th*, namely the encoding of **H1** and **H2**, are determined by membership to *DES_THEORY*.

To express this law (and others) more concisely, we provide an alternative definition for $True_P$ (and other operators) that takes as a parameter a theory rather than a universe. Conceptually, this allows us to speak of predicates such as *true*, $x := 1$, $x' = 2$, $\mathbf{II}$, and so on within specific theories. The required universe of these operators is inferred from that of the theory. Below we show how this results in a more compact rendition of the above law.

$$\vdash \forall\, th : DES\_THEORY \bullet True_T\ th \in TheoryPredicates\ th$$

This law is indeed not more complicated than a corresponding theorem would have been in the original treatment in [OCW07]. Providing the theory *th* is defined as illustrated in the previous section, using an instantiation function, the requirement to prove $th \in DES\_THEORY$, raised by the use of the above law, can be trivially discharged. In the original treatment we can similarly prove that for any alphabet, $True_R\,(a)$ is a valid design; the corresponding theorem in that treatment would be

$$\vdash \forall\, a : ALPHABET\_DES \bullet True_R\,(a) \in DESIGN$$

where *DESIGN* refers to the predicates of all possible instantiations of design theories, as it is also the case in our treatment. Clearly, the above law does not capture predicate membership to a specific instance of design theory. The fact that we can do so in our treatment allows for a more specialised reasoning.

Closure theorems for general and theory-specific operators are another example of laws whose validity is expressed in terms of the theory context. The following theorem establishes that any theory of designs is closed under disjunction.

$$\vdash \forall\, th : DES\_THEORY \bullet \forall\, p_1, p_2 : TheoryPredicates\ th \bullet p_1 \vee_P p_2 \in TheoryPredicates\ th$$

To apply this law, we first have to establish that there is a member of the family of design theories to which the predicates p_1 and p_2 in question belong. This law is not equivalent to stating that if p_1 and p_2 are elements of the set *DESIGN*, as defined in Section 3.6, so is $p_1 \vee_P p_2$. The latter is how closure laws would have been formulated in the original treatment, but here this would have a different interpretation, namely that if we combine any two design predicates of possibly different design theory instances, we obtain a design predicate (of some design theory). This is false due to the restrictions on alphabets and universes, and the associated compatibility requirements of operators. Specifically, if p_1 and p_2 are incompatible, the application $p_1 \vee_P p_2$ is not well-defined and we should not be able to deduce any properties from it.

Beyond proving laws about the predicates of specific UTP theories as shown above, it is also possible in our encoding to prove more general laws about UTP theories that, for example, exploit their relationship. A very intuitive law establishes that the predicates obtained by extending an existing theory with additional healthiness functions form a subset of the original theory's predicates. We state this theorem as follows.

$$\begin{array}{l}\vdash \forall\, th : UTP\_THEORY;\ hs : \mathbb{P}\ HEALTH\_COND \bullet \\ \quad TheoryPredicates\,(SpecialiseTheory\,(th, hs)) \subseteq TheoryPredicates\ th\end{array}$$

Here *th* can be any instance of a UTP theory, underpinning the generality of the law. Although this property is not particularly surprising, it exemplifies how we can state universal facts about UTP theories independently of theory families with specific sets of healthiness conditions.

A more interesting and practically relevant scenario arises when expressing and proving laws about families of UTP theories for which the healthiness functions possess certain properties. For example, [HCW08] discusses theories in which the healthiness functions are expressed in terms of conjunctions. We can prove certain theorems, for example, closure under conjunction, disjunction, sequence and so on, for the predicates of all such theories. If we define, for example, a predicate $\mathbf{CH}(h)$ that tells us whether a healthiness function

h is expressible in this way, the theorem

$$\begin{aligned}&\vdash \forall\, th : \mathit{UTP\_THEORY} \mid (\forall\, h : th.\mathit{HEALTH\_CONDS} \bullet \mathit{CH}(h)) \bullet\\ &\quad \forall\, p_1, p_2 : \mathit{TheoryPredicates}\ th \bullet p_1 \mathbin{;_R} p_2 \in \mathit{TheoryPredicates}\ th\end{aligned}$$

asserts that the predicates of all such theories are closed under relational composition. The possibility of expressing such properties of classes of theories distinguishes our approach from the existing one, and adds to its expressive power. The general theorem above may be particularised to any theory whose healthiness conditions have the required property, and the effort invested in proving it once is effectively reused.

So far we have confined our attention to properties of theory families and their predicates at different levels of abstraction. Below we address reasoning about links between families of theories.

5.1. Linking Theories

Theory links are functions mapping the predicates of one theory into (a subset of) the predicates of another. We have already encountered such functions, namely the healthiness functions **H1** and **H2** which map predicates from the more general theory of relations into the more restrictive theory of designs, providing the appropriate assumptions are met on the types of *okay* and *okay′* in the relational theory.

Linking functions often enjoy properties like idempotence, monotonicity, or weakening and strengthening as described in [HJ98]. These properties allow us to deduce further characteristics of links and their predicates. In our encoding we formalise links between theories by partial functions on *ALPHA_PREDICATE*. For this we use the type *ALPHA_FUNCTION* as introduced in Section 3.5.

As an introductory example, we consider the link that maps a relation to a (terminating) design.

$$L(Q) \mathrel{\widehat=} \mathit{true} \vdash Q$$

We recall that the turnstile operator $P \vdash Q$ yields a design predicate with precondition P and postcondition Q; as explained in Section 2 it is defined as $\mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge Q$ and encoded by

$$\begin{array}{|l}\_ \vdash_D \_ : \mathit{DES\_COMPATIBLE} \times \mathit{DES\_COMPATIBLE} \rightarrow \mathit{DESIGN}\\ \hline \forall\, p, q : \mathit{DES\_COMPATIBLE} \bullet p \vdash_D q = (\mathit{OKAY} \wedge_P p) \Rightarrow_P (\mathit{OKAY}' \wedge_P q)\end{array}$$

Here, OKAY and OKAY' are constants which represent the predicates *okay* and *okay′*, respectively. Their definition is included in Appendix A.4. We recall that *DES_COMPATIBLE* is the set of all alphabetised predicates whose universes are compatible with the typing restrictions on design predicates. By contrast, *DESIGN* is the set of predicates belonging to some instantiation of a design theory as explained in Section 3.6.

We now can encode the linking function L above as

$$\begin{array}{|l}L : \mathit{ALPHA\_FUNCTION}\\ \hline \operatorname{dom} L = \mathit{DES\_COMPATIBLE} \wedge (\forall\, p : \mathit{DES\_COMPATIBLE} \bullet L(p) \ = \ \mathit{True}_P\ \varnothing \vdash_D p)\end{array}$$

Because *ALPHA_FUNCTION* only entails partial functions, the first conjunct defining the domain of L is crucial to determine where application of L is defined. We now express the property that for every relational theory $th_1 : \mathit{REL\_THEORY}$ with suitable typing on the auxiliary variables, there exists a corresponding theory of designs where L maps each predicate of the former theory to a predicate of the latter.

$$\begin{aligned}&\vdash \forall\, th_1 : \mathit{REL\_THEORY} \mid \mathit{Compatible}_U\,(th_1.\mathit{THEORY\_UNIVERSE}, \mathit{DES\_UNIVERSE\_MIN}) \bullet\\ &\quad \exists\, th_2 : \mathit{DES\_THEORY} \bullet L \ (\!|\, \mathit{TheoryPredicates}\ th_1 \,|\!) \subseteq \mathit{TheoryPredicates}\ th_2\end{aligned}$$

A similar technique can be used to express the law that a particular theory of CSP processes is an image of a certain theory instance of designs under the function **R** which is the composition of healthiness functions

R1 to **R3** for reactive processes as defined in Table 2.

$$\vdash \forall\, th_1 : CSP\_THEORY \bullet \exists\, th_2 : DES\_THEORY \mid TheoryPredicates\ th_1 = R\ (\!|\ TheoryPredicates\ th_2\ |\!)$$

This law asserts that, for very every CSP theory, there is a theory of designs such that R, the linking function in this context, maps the predicates of the design theory surjectively into those of the CSP theory.

Even more concretely, we can formulate and prove laws for particular theory instantiations. If theory instances $inst\_th_1$ and $inst\_th_2$, not necessarily with compatible universes, are given, we can try to establish, for example, that for those particular values

$$L\ (\!|\ TheoryPredicates\ inst\_th_1 |\!) \subseteq TheoryPredicates\ inst\_th_2$$

holds.

As a closing remark, we observe that this section did not aim to explore in all detail the possibilities for reasoning about theory links. We content ourselves with providing evidence for its feasibility, and give an indication of which properties may be expressible. Particular properties of UTP theories are refinement laws that hold for the predicates of those theories. We look at them in more detail in the following section.

5.2. Refinement Laws

A standard step in formal verification consists in proving that a given specification is refined by some implementation. In essence, refinement is a property of alphabetised predicates that can be established independently of their particular UTP theory membership. In the mechanical proof environment this shows in the fact that every refinement can be proved by appealing to the definition of operators involved, as well as axioms and laws specified in the lowest level of the theory hierarchy.

In practice, however, proofs unfolding the definition of all operators involved, and thereby expanding predicates in terms of their semantic representation, are tedious and require a lot of low-level proof steps. We consider, for example, the simple refinement

$$x := 1 \sqcap x := 2 \quad \sqsubseteq \quad x := 1$$

in the context of the design theory instantiation presented in Section 5. Rewriting the operator definitions yields the following sequence of steps.

$$\begin{aligned}
& x := 1 \sqcap x := 2 \quad \sqsubseteq \quad x := 1 \\
\equiv\ & [x := 1 \sqcap x := 2 \quad \Leftarrow \quad x := 1] \\
\equiv\ & [x := 1 \vee x := 2 \quad \Leftarrow \quad x := 1] \\
\equiv\ & [(true \vdash x' = 1) \vee (true \vdash x' = 2) \ \Leftarrow\ (true \vdash x' = 1)] \\
\equiv\ & [(okay \wedge true \Rightarrow okay' \wedge x' = 1) \vee (okay \wedge true \Rightarrow okay' \wedge x' = 2) \\
& \quad \Leftarrow\ (okay \wedge true \Rightarrow okay' \wedge x' = 1)]
\end{aligned}$$

To continue the proof in the context of our encoding, we have to unfold the definition of the logical operators and equalities yielding a purely semantic representation of the alphabetised predicate, which by extensional means has to be proved equal to the alphabetised predicate *true*. This is feasible but not practical, as the unfolding of functions can produce very complex terms.

An alternative approach is to formulate and prove a collection of algebraic (refinement) laws. This is achieved by explicitly stating the family of theories within which they hold. In the case of nondeterministic choice we can formulate the following law that allows us to easily prove the above refinement.

$$\vdash \forall\, th : DES\_THEORY \bullet \forall\, d_1, d_2 : TheoryPredicates\ th \bullet d_1 \sqcap_D d_2 \ \sqsubseteq\ d_1$$

The only proviso for this law is that the two predicates have to belong to the same theory, otherwise the $\sqcap_D$ operator would not be well-defined, namely due to possible incompatibility of universe. Applying the refinement law hence requires the proof that constituent operators belong to a certain theory of designs; this kind of requirement is in fact common to the application of most algebraic laws. In our particular case this obliges us to show that the predicates $x := 1$ and $x := 2$ are predicates of some theory of designs.

The required proofs are partly based on the definitions of the programming operators. For our example, the constructor function for design assignments guarantees that $x := 1$ and $x := 2$ are elements of DESIGN, from which easily follows that there is *a* theory of designs to which they belong. That it is the theory the law is specialised with partly follows from the universe of the predicates. These extra proofs are an additional cost that we have to pay for our more expressive semantics. We can, however, largely automate them for typical cases by supplying suitable lemmas and tactics.

To clarify this point, we consider the application of the nondeterminism law to predicates involving other operators, for example, sequence.

$$(x := 1;\ x := 2) \sqcap x := 3 \quad \sqsubseteq \quad (x := 1;\ x := 2)$$

To apply the law, we need to establish that all constituting predicates are within the design theory of discourse; this involves showing that $x := 1;\ x := 2 \in \mathit{TheoryPredicates}\ th$ for some $th : \mathit{UTP\_THEORY}$. The proof of properties like these cannot be sensibly captured by a single law; it is first necessary to prove that $x := 1 \in \mathit{TheoryPredicates}\ th$, then $x := 2 \in \mathit{TheoryPredicates}\ th$, and finally exploit the closure property of sequence. The structure of the predicate guides the proof. In summary, we can reduce the proof effort to discharge refinement conjectures considerably by providing algebraic laws, but their application requires further theorems, and importantly, high-level tactics for automation which we address in Section 7.

Another type of law which is useful to reason algebraically about refinements are identity laws such as

$$\vdash \forall\, th : \mathit{DES\_THEORY} \bullet \forall\, d_1, d_2 : \mathit{TheoryPredicates}\ th \bullet d_1 \sqcap_D d_2 = d_2 \sqcap_D d_1\ ,$$

here exploiting the commutativity of nondeterministic choice. Again, to fundamentally apply such laws we have to establish membership of the predicates involved to a particular UTP theory.

To conclude, we observe that our encoding enables us to prove the refinement conjecture presented in the introduction, namely $x := x + 1 \lhd x = 1 \rhd \mathbf{II} \sqsubseteq x := 2$. To do so we first define an alphabet containing x and x', and a universe in which they range over the set $\{1, 2\}$. Using the instantiation function for relational theories, we then instantiate the corresponding UTP theory, as explained in Section 5; we call it th here. The programs can be directly expressed as predicates of th and thereby acquire their universes from it.

$$\mathit{Assign}_R\,(th, \langle x\rangle, \langle x+1\rangle) \lhd_R x = 1 \rhd_R \mathbf{II}_R\,(th) \sqsubseteq \mathit{Assign}_R\,(th, \langle x\rangle, \langle 2\rangle)$$

The proof is carried out by unfolding the definition of the conditional into primitive logical operators on alphabetised predicates. We can then use an algebraic law that rewrites the implication $P \Rightarrow (Q \wedge R)$ originating from the refinement and conditional into a conjunction, and another law that allows us to prove the conjuncts separately. The interesting case is where $\neg_R\,(x = 1)$ appears in the antecedent of the implication. Here, we exploit the fact that the universe only permits bindings mapping x to either 1 or 2; the semantic definition of negation takes this into account. This yields the necessary assumption $x = 2$ required to complete this branch of the proof as $\mathbf{II}_R\,(th)$ has no influence on the value of x.

The formulation and proof of general algebraic laws of designs, reactive designs, *Circus*, and so on has already been explored in Oliveira's work. In this section we contemplated how these laws may be rephrased and used to reason about particular specifications. In the next section we investigate how to reason about theory instances and their predicates in a more specific way.

6. Reasoning about Theory Instances

Beyond general laws, our semantic encoding enables us to reason about particular UTP theories and their predicates. In this section we illustrate this by giving a few examples.

6.1. Instantiating a Theory

First, we consider a UTP theory of designs with alphabet $\{x, x', okay, okay'\}$. The auxiliary variables $okay$ and $okay'$ are introduced in the ProofPower theory `utp-des` encapsulating common definitions for design theories, but x and x' are specific variables which have to be introduced, for example, in a separate ProofPower theory accommodating the definitions for the instantiation as follows.

$$\begin{array}{|l} x, x' : NAME \\ \hline x \in undashed \land x' = dash\ x \land distinct\ \langle x, okay \rangle \end{array}$$

The first two conjuncts establish that x is an undecorated name with x' being its dashed version. To exclude the case where $x = okay$, we once again use the *distinct* function, and formalise this requirement as $distinct\ \langle x, okay \rangle$. Because of the injectivity of the *dash* function and disjointness of the sets of undashed and single-dashed names, this implies too that x, x', $okay$ and $okay'$ are all distinct.

The alphabet of our example theory instance can now be specified as follows.

$$\begin{array}{|l} INST\_ALPHABET : DES\_ALPHABET \\ \hline INST\_ALPHABET = \{x, x', okay, okay'\} \end{array}$$

In the above, $DES\_ALPHABET$ includes all alphabets that contain $okay$ and $okay'$, amongst other possible variables. Discharging the existential consistency proof obligation for this definition establishes that the alphabet we provide is a valid alphabet for a theory of designs.

Generally, consistency proof obligations in Z are used to establish that definitions are free of contradictions, and hence guarantee the existence of a model for the constants. Formally, the proof has to provide a witness that can be used in place of the defined constant, and moreover renders its predicate true. Should this predicate, as here, be an equality in which the left-hand side matches the constant defined, the only sensible witness is the right-hand value. Thus what we effectively prove for consistency is that $\{x, x', okay, okay'\} \in DES\_ALPHABET$. An alternative way of specifying $INST\_ALPHABET$ would be

$$\begin{array}{|l} INST\_ALPHABET \ \widehat{=}\ \{x, x', okay, okay'\} \end{array}$$

Consistency of this conservative definition is trivial, but we cannot immediately infer from it that the defined $INST\_ALPHABET$ is an element of $DES\_ALPHABET$; this would have to be specified as a separate theorem. The former style of defining constants as member of some suitable semantic set is the preferable method for two reasons. First, a failure in the consistency proof reveals a conceptual error in the specification which otherwise would be harder to localise, and second the membership of the defined constant to some type becomes an axiom that can immediately be used in other proofs, facilitating automation.

We are next required to provide a theory universe. The instantiation function for design theories obliges us to type the variables $okay$ and $okay'$ as boolean. By contrast, we can choose any type for x and x'. By reusing the set $DES\_UNIVERSE$ which already captures the appropriate type constraints for the auxiliary variables, the following definition specifying the types of x and x' uniquely determines the theory universe.

$$\begin{array}{|l} INST\_UNIVERSE : DES\_UNIVERSE \\ \hline Alphabet_U\ INST\_UNIVERSE = INST\_ALPHABET \land \\ typeof\ (x, INST\_UNIVERSE) = INT\_VAL \end{array}$$

The same style of specification is used as before, yielding immediately that $INST\_UNIVERSE$ is a valid universe for designs (member of $DES\_UNIVERSE$), providing consistency of the definition has been proved. No predicate is included for explicitly constraining the type of x'. Such is redundant since the properties of $REL\_UNIVERSE$, of which $DES\_UNIVERSE$ is a restriction, ensure that the dashed counterparts of undecorated variables, if present, have identical types. INT_VAL is defined as the subset of the semantic domain of values that includes all integers (see Section 3.1). We observe that the additional constraint

$\mathit{typeof}\,(\mathit{okay}, \mathit{INST\_UNIVERSE}) = \mathit{BOOL\_VAL}$, namely to type-constrain okay (and okay') to boolean values, is not required as it is already implied by the definition of $\mathit{DES\_UNIVERSE}$.

The previous example can be used to illustrate why it is important to ensure distinctiveness of x and okay. Under the assumption $x = \mathit{okay}$, the above definition would by referential transparency imply that

$$\mathit{typeof}\,(x, \mathit{INST\_UNIVERSE}) = \mathit{typeof}\,(\mathit{okay}, \mathit{INST\_UNIVERSE})$$

which is a contradiction because of $\mathit{typeof}\,(x, \mathit{INST\_UNIVERSE}) = \mathit{BOOL\_VAL}$; hence, in that case we cannot find a model for $\mathit{INST\_UNIVERSE}$. Indeed, the only possible model fulfilling all constraints is

$$\{\mathit{okay} \mapsto \mathit{BOOL\_VAL}, \mathit{okay}' \mapsto \mathit{BOOL\_VAL}, x \mapsto \mathit{INT\_VAL}, x' \mapsto \mathit{INT\_VAL}\}$$

Clearly, it is only a universe (partial function) if x and okay are distinct. Distinctiveness of the variables for this reason becomes a necessary proviso for the consistency proof.

The UTP theory is conveniently obtained using an instantiation function for design theories.

$$\mathit{INST\_THEORY} \;\widehat{=}\; \mathit{InstDesTheory}\ \mathit{INST\_UNIVERSE}$$

We are now able to prove, for example, that certain alphabetised predicates belong (or do not belong) to the instantiated theory's predicates. For example, we can prove that

$$\mathit{True}_P\ \mathit{INST\_UNIVERSE} \in \mathit{TheoryPredicates}\ \mathit{INST\_THEORY}$$

as well as

$$\mathit{False}_P\ \mathit{INST\_UNIVERSE} \notin \mathit{TheoryPredicates}\ \mathit{INST\_THEORY}$$

That is, the predicate *true*, or more accurately *true* of our particular alphabet and universe, is a design predicate, whereas *false* over that same alphabet and universe is not. This is done, for example, by using the law presented in Section 5. Proofs of this kind are necessary to verify that the concrete computations we encode and reason about belong to the UTP theory in which they are considered.

6.2. Handling Multiple Theory Instances: the Circus Example

We proceed to present a more elaborate example illustrating *Circus* theory instantiation and encoding of their process actions. Our objective is to show how we can handle examples written in languages with an elaborate scope structure. Within a *Circus* program, we can have a collection of processes, each with a local state that defines a particular alphabet and universe. In addition, inside a process, it is possible to introduce extra variables with limited and possibly nested scopes. In [ZC09], we have given an overview of the main issues raised by this problem and discussed our proposed solution. Below, we review and complement this work in the context of our revised model by presenting a more elaborate example that in addition provides interesting opportunities for refinement.

6.3. Models of Circus Programs

We now explore how the mechanisation of *Circus* is used to encode concrete processes. For this we resort to the example of the simple vending machine presented in Figure 3.

To accommodate the ProofPower-Z definitions, we create a new ProofPower theory circus-vm as a child of utp-circus. We begin by creating definitions that introduce channel names, state components and local variables. They are introduced through axiomatic definitions as unique elements of the NAME type. For the channel names it does not matter whether channels are typeless or communicate values as we are only interested in their identifier. Types are important when we communicate values over these channels, and such communication events are represented separately by name / value pairs.

The $\mathit{SimpleVendingMachine}$ process has two state components, credit and stock, which are introduced (together with their dashed counterparts) as distinct values from the set $\mathit{Z\_VAR\_NAME}$; this implicitly ensures distinctiveness from the auxiliary variables $okay$, $wait$, tr, and ref. The constant Z_VAR_NAME is simply defined as $NAME \setminus ALPHABET\_OWTR$.

The minimal alphabet of process actions includes auxiliary variables as well as state components and is added as a separate definition $VM\_ALPHABET$.

$$VM\_ALPHABET \mathrel{\widehat=} ALPHABET\_OWTR \cup \{credit, stock, credit', stock'\}$$

We refer to it as 'minimal' since it is possible for auxiliary actions to include additional variables; for example, operation schemas can have extra input and output variables, and variable blocks and input communications introduce new variables (with limited scope). This is, for example, the case for the *CalcDispense* action which introduces the additional output variables *items* and *left_credit*. The main action of the process, however, must exactly have the alphabet $VM\_ALPHABET$.

The next step consists of instantiating the *Circus* theory for the main action. To do so we first define the universe $VM\_UNIVERSE$ of the main action. This is a *Circus* universe with alphabet $VM\_ALPHABET$ that imposes suitable type restrictions on the state components.

$$\begin{array}{|l} VM\_UNIVERSE : CIRCUS\_UNIVERSE \\ \hline Alphabet_U\, VM\_UNIVERSE = VM\_ALPHABET \wedge \\ typeof(credit, VM\_UNIVERSE) = INT\_VAL \wedge \\ typeof(stock, VM\_UNIVERSE) = INT\_VAL \end{array}$$

By selecting the universe from $CIRCUS\_UNIVERSE$ we already ensure that auxiliary variables are typed correctly; the only type constraints to be formulated here are the ones restricting the state variables.

We are now able to define the UTP theory for the actions of *SimpleVendingMachine*. It is not just one UTP theory, because, as hinted above, the alphabet of actions may include extra variables. We require a family of *Circus* theories whose universe can be any possible extension of $VM\_UNIVERSE$.

$$\begin{array}{|l} VM\_THEORY \mathrel{\widehat=} \{u : CIRCUS\_UNIVERSE \mid \\ \quad VM\_ALPHABET \subseteq Alphabet_U\ u \wedge Compatible_U(u, VM\_UNIVERSE) \bullet InstCircusTheory\ u\} \end{array}$$

The main benefit of VM_THEORY is that it permits us to state (or verify) that actions such as *InsertMoney*, *CalcDispense*, *DispenseItem*, and so on, are characterised by predicates that belong to (one of) the *Circus* theories for the *VM* process, encapsulating healthiness conditions as well as type constraints on the state components. For this we define the set VM_ACTION which contains all predicates characterising valid actions in the context of the *SimpleVendingMachine* process.

$$VM\_ACTION \mathrel{\widehat=} \bigcup TheoryPredicates \llparenthesis VM\_THEORY \rrparenthesis$$

It is simply the union of all predicates of UTP theories in VM_THEORY. The property that a predicate P is a valid process action of *SimpleVendingMachine* can hence be easily expressed as $P \in VM\_ACTION$. This helps in formulating theorems that formally state the soundness of action encodings. Another definition $VM\_MAIN\_ACTION \subseteq VM\_ACTION$ is included to specify those predicates which are valid main actions of the process; their universe has to be exactly $VM\_UNIVERSE$, not including any extra variables.

We now turn to encoding the actions specified in *SimpleVendingMachine*. We first look at the initialisation action *InitState* which is defined through a Z operation schema. In its definition the corresponding schema $[State' \mid credit' = 0 \wedge stock' = capacity]$ has to be lifted to become a *Circus* action, more accurately a valid predicate of a *Circus* theory in VM_THEORY. The schema itself is encoded by a relational predicate over the universe that contains its components $credit'$ and $stock'$ with the right type. We note that it neither belongs to a *Circus* theory instance, nor does it have the auxiliary variables in its alphabet.

The semantic function $SchemaExp_C$ performs the lifting; it takes a relational predicate and an instance of VAR_DECLS encapsulating the declaration of the schema components. The universe of the predicate has to be compatible with the variable declarations. The latter are encoded by a pair of sequences: the first component listing the variable names, and the second, their types.

$$VM\_InitState\_VAR\_DECLS \mathrel{\widehat=} (\langle credit', stock' \rangle, \langle INT\_VAL, INT\_VAL \rangle)$$

Types are represented as sets of values, that is sets of elements from the unified $VALUE$ type introduced in Section 3.1. Here, INT_VAL is the set of integer values in the semantic model, defined as $\{n : \mathbb{N} \mid Int(n)\}$. It is obtained by applying the type constructor Int for integer values to all elements of its domain $\mathbb{N}$.

The encoding of $VM\_InitState$ is as follows.

$$\begin{array}{l}
VM\_InitState : VM\_ACTION \\
\hline
VM\_InitState = SchemaExp_C\ (VM\_InitState\_VAR\_DECLS, \\
\quad (=_P\ (\{credit' \mapsto INT\_VAL\}, Var(credit'), Val(Int(0)))) \land_P \\
\quad (=_P\ (\{stock' \mapsto INT\_VAL\}, Var(stock'), Var(capacity))))
\end{array}$$

In the above $=_P$ is the semantic function used to construct alphabetised predicates for equalities between variables and expressions as defined in Section 3.4. It must be provided with a universe, namely that of the resulting predicate, a variable, and an expression. The universe is created in an *ad hoc* manner as we need it. The fact that a simpler universe model is used in this paper, as opposed to [ZC09] makes the construction particularly concise. Namely, the earlier approach requires the use of another function $Create_U$ here.

Notably, the universe of the schema predicate has $credit'$ and $stock'$ in its alphabet, since $\land_P$ merges the universes of the constituent predicates. The predicate defined by $SchemaExp_C$ additionally includes in its universe the auxiliary variables and (provably) fulfils the healthiness conditions for *Circus* actions. This more generally illustrates how predicates of different UTP theories coexist in the same definition.

By introducing $VM\_InitState$ as an element of VM_ACTION, we ensure that irrespective of how we define it, that is, using *Circus* operators or, alternatively, plain predicate connectives, it has to characterise a valid action of $SimpleVendingMachine$. This is effectively discharged by the consistency proof of the axiomatic definition generated by ProofPower-Z. A situation in which $VM\_InitState \notin VM\_ACTION$ would result in a contradiction and hence the existential proof to fail.

The encoding of schemas that include extra components, like $CalcDispense$, is similar. To simplify the encoding of the schema predicate, we define a universe $VM\_CalcDispense\_UNIVERSE$ containing exactly the components of the schema. We omit its definition here, and the one of $VM\_CalcDispense\_VAR\_DECLS$. The encoding of the action is presented below.

$$\begin{array}{l}
VM\_CalcDispense : VM\_ACTION \\
\hline
VM\_CalcDispense = SchemaExp_C\ (VM\_CalcDispense\_VAR\_DECLS, \\
\quad (=_P\ (VM\_CalcDispense\_UNIVERSE, Var(credit'), Var(credit))) \land_P \\
\quad (=_P\ (VM\_CalcDispense\_UNIVERSE, Var(stock'), Var(stock))) \land_P \\
\quad (=_P\ (VM\_CalcDispense\_UNIVERSE, \\
\qquad Rel((\_ \leq_V \_), Var(item'), Fun_2((\_\ Div_V\ \_), Var(credit), Var(item\_price))), True_E)) \land \\
\quad (=_P\ (VM\_CalcDispense\_UNIVERSE, Var(left\_credit'), \\
\qquad Fun_2((\_ -_V \_), Var(credit), Fun_2((\_ *_V \_), Var(items'), Var(item\_price)))))
\end{array}$$

The first two conjuncts reflect the inclusion of $\Xi State$ which requires the state components of the *Circus* process to maintain their value; the corresponding implicit constraints are $credit' = credit$ and $stock' = stock$. The next two conjuncts encode the schema predicate. Here, Div_V, $-_V$ and $*_V$ are appropriate functions on values. As before, $SchemaExp_C$ lifts the schema into a *Circus* action, and membership to VM_ACTION ensures that is an action of the correct *Circus* theory family.

The shriek, which introduces an output variable in the operation schema, is generally translated into a corresponding pair of variables to render the alphabet of the action homogeneous (in our case $\{items, items'\}$ and $\{left\_credit, left\_credit'\}$). The same also applies to input variables decorated with question marks should they occur. Generally, we identify variables decorated with a question mark with the corresponding *undashed* name, and those decorated with a shriek with the corresponding *dashed* name.

An example of a more complex encoding of an action that uses a combination of guarded commands and CSP operators is *DispenseItem*. It is given in full below.

$$
\begin{array}{l}
\mathit{VM\_DispenseItem} : \mathit{VM\_ACTION} \\
\hline
\mathit{VM\_DispenseItem} = \mathit{DispenseItemBtn} \longrightarrow_{CSync} \mathit{var}_C\ (\mathit{items}, \mathit{var}_C\ (\mathit{left\_credit}, \\
\quad \mathit{VM\_CalcDispense}\ ;_C \\
\qquad (=_P (\boldsymbol{VM\_UNIVERSE} \oplus_U \{\boldsymbol{items}, \boldsymbol{items'}, \boldsymbol{left\_credit}, \boldsymbol{left\_credit'}\} \times \{\boldsymbol{INT\_VAL}\}, \\
\qquad\quad \mathit{Rel}((\_ \neq_V \_), \mathit{Var}(\mathit{items}), \mathit{Val}(0)), \mathit{True}_E)) \wedge_P \\
\qquad (=_P (\boldsymbol{VM\_UNIVERSE} \oplus_U \{\boldsymbol{items}, \boldsymbol{items'}, \boldsymbol{left\_credit}, \boldsymbol{left\_credit'}\} \times \{\boldsymbol{INT\_VAL}\}, \\
\qquad\quad \mathit{Rel}((\_ \leq_V \_), \mathit{Var}(\mathit{items}), \mathit{Var}(\mathit{stock})), \mathit{True}_E)) \ \&_C \\
\qquad\qquad (\mathit{GiveItems}, \mathit{Var}(\mathit{items})) \rightarrow_{Cout} \mathit{Assign}_C\ (\\
\qquad\qquad\quad \boldsymbol{VM\_UNIVERSE} \oplus_U \{\boldsymbol{items}, \boldsymbol{items'}, \boldsymbol{left\_credit}, \boldsymbol{left\_credit'}\} \times \{\boldsymbol{INT\_VAL}\}, \\
\qquad\qquad\quad \langle \mathit{credit}, \mathit{stock}\rangle, \langle \mathit{Var}(\mathit{left\_credit}), \mathit{Fun}_2((\_ -_V \_), \mathit{Var}(\mathit{stock}), \mathit{Var}(\mathit{items}))\rangle) \\
\qquad\qquad\qquad \Box_C \\
\qquad (=_P (\boldsymbol{VM\_UNIVERSE} \oplus_U \{\boldsymbol{items}, \boldsymbol{items'}, \boldsymbol{left\_credit}, \boldsymbol{left\_credit'}\} \times \{\boldsymbol{INT\_VAL}\}, \\
\qquad\quad \mathit{Rel}((\_ =_V \_), \mathit{Var}(\mathit{items}), \mathit{Val}(0)), \mathit{True}_E)) \vee_P \\
\qquad (=_P (\boldsymbol{VM\_UNIVERSE} \oplus_U \{\boldsymbol{items}, \boldsymbol{items'}, \boldsymbol{left\_credit}, \boldsymbol{left\_credit'}\} \times \{\boldsymbol{INT\_VAL}\}, \\
\qquad\quad \mathit{Rel}((\_ >_V \_), \mathit{Var}(\mathit{items}), \mathit{Var}(\mathit{stock})), \mathit{True}_E)) \ \&_C \\
\qquad\qquad \mathit{Skip}_C\ (\boldsymbol{VM\_UNIVERSE} \oplus_U \{\boldsymbol{items}, \boldsymbol{items'}, \boldsymbol{left\_credit}, \boldsymbol{left\_credit'}\} \times \{\boldsymbol{INT\_VAL}\}) \\
\quad))
\end{array}
$$

The encoding of this action uses the above defined *VM_CalcDispense*, and besides requires a few theory-specific operators of the *Circus* theory. We give a brief explanation of them without elaborating on their semantic encoding in the ProofPower theory utp-circus. The latter can be found in Appendix A.6.

- The operator $c \longrightarrow_{CSync} a$ is used to encode prefixed actions in *Circus* where c is the channel to synchronise on.
- The operator $(c, e) \longrightarrow_{Cout} a$ encodes an output prefix in which the value of the expression e is communicated over the channel c.
- The $\mathit{var}_C\ (n, a)$ construct declares a local variable n whose type is determined by the universe of the action a which constitutes the body of the declaration, and must include n in its alphabet. It is defined in terms of the UTP constructs for variable declarations given in Table 1.
- The $\mathit{Assign}_C\ (u, ns, es)$ construct encodes the reactive assignment. Although parameterised in the same way, it is different from the relational or design assignment. Similarly, $\mathit{Skip}_C\ (u)$ encodes the reactive Skip ($\mathbf{II_{rea}}$), which we have already encountered in Section 4.2.
- The operator $a_1 \Box_C a_2$ encodes external choice between actions a_1 and a_2. For the application to be well-defined, they have to belong to the same *Circus* theory instance.

The action *VM_DispenseItem* first waits for synchronisation on the *DispenseItemBtn* channel, signalling the button press. It then declares the two local variables *items* and *left_credit* which store the results after executing *VM_CalcDispense*. It is important that the body of the chained declarations has the variables *items* and *left_credit*, including their dashed versions, in its universe. We have explained that this is true for *VM_CalcDispense*, however it should also hold for each of the statements following it. This importantly ensures composability and thus well-definedness of the sequential composition. It is why subsequent operators are equipped with universes obtained by suitably extending *VM_UNIVERSE*. We have highlighted these universes in the encoding of the action in bold font.

The afore-mentioned is another example that illustrates how predicates of different UTP theories can coexist in the same ProofPower definitional scope. The encoding of the remaining actions will not be discussed in detail as they follow the same principle of the exemplified action encodings.

Since the main action of the process is anonymous, we introduce a designated constant *VM_MainAction*. It does not, however, truly characterise the process since it still contains the state components in its universe.

Because these are local to the process, they should be hidden it its semantic description. This is achieved by the operator $begin_C \_ end_C$. With it we obtain the following definition for *SimpleVendingMachine*.

$$\begin{array}{|l} SimpleVendingMachine : CIRCUS\_PROCESS \\ \hline SimpleVendingMachine = begin_C\ VM\_MainAction\ end_C \end{array}$$

As said earlier, the set $CIRCUS\_PROCESS$ contains all predicates of the *Circus* theory obtained by instantiation with a minimal universe, which comprises auxiliary variables only and no state components. The hiding of the state components is achieved by existential quantification over non-auxiliary variables.

In this section we have demonstrated how specifications of more elaborate theories can be encoded, and how we can formulate the encoding of specifications in such a way as to verify their soundness. The encoding requires that type information is computed prior to translation and consequently exploited in the construction of universes; this can be easily achieved using the *Circus* type checker [XCS06, FWC07]. We stress that our approach is such that no interference among ProofPower theories encoding different *Circus* specifications can arise. As a consequence we are able to import and reason about them in the same declarative ProofPower theory scope; for example, none of the axiomatic definitions for the vending machine process specify global constraints, and all information about types is captured in local definitions such as $VM\_UNIVERSE$ or $VM\_CalcDispense\_UNIVERSE$. This also enables us to incrementally construct specifications from multiple processes by virtue of process combinators, and thereby paves the way for employing the mechanisation in the verification of more complex systems such as control law diagrams [CCO05].

7. Proof Automation

We have so far limited our discussion to definitions and theorems of the mechanisation. In this section we report on strategies for proof automation. Developing mechanisms to facilitate proof is important for a number of reasons. First, it allows us to prove general theorems efficiently within the various UTP theory encodings. These are, for example, algebraic properties of operators on predicates, universes, or refinement laws. In Oliveira's original work, a large number of such theorems have already been proved, aided by rudimentary use of custom proof tactics. This raises the question whether tactic programming may be further exploited to modularise and shorten proofs.

A second reason for developing such tactics is to automate soundness and refinement proofs for particular specifications in order to support the construction of highly integrated tools that can be used by engineers without expert knowledge of ProofPower. In our case the use of tactics is essential, as we aim at proofs of properties of particular specifications. Instead of just proving laws which are generally useful, we also want to support theorem proving about particular UTP models.

Finally, factoring common functionality into tactics enables us to design proofs in a more robust way. This is important to tame the effect of possible future changes to the encoding with respect to re-establishing proofs of theorems that might thus become invalid: a problem we are currently facing in reusing Oliveira's original proofs and recasting them in the light of modifications to definitions.

In this section we examine two layers of automation. The first section on low-level tactics discusses general facilities for automation that extend those already designed in [OCW07]. They are tactics which we believe are of more general use, and thus are not specific to the mechanisation of the UTP. The second section discusses more specialised, high-level tactics tailored to facilitate proofs of specific properties in the mechanisation. Both low-level and high-level tactics are implemented in Standard ML (SML).

7.1. Low-level Tactics for Automation

Low-level utilities and proof tactics are defined in the ProofPower theory utp-z-ext, the parent of all other ProofPower theories in our encoding (see Figure 1). The purpose of utp-z-ext is to provide a few custom extensions to the embedding of Z in ProofPower, including additional laws for Z operators.

Generic utility tactics and functions have evolved through analysis of repetitive and tedious steps in proofs. We do not discuss all of them, but give a few examples that illustrate their benefit. In doing so the main issue we address is the one of rewriting expressions. Although this is generally a well-established branch of research [VB98], the standard facilities of ProofPower-Z prevent us from taking full advantage of more sophisticated strategies for rewriting. We look at three areas of particular relevance to our application of the UTP encoding to reason about models of particular programs, namely the rewriting of sets, of memberships of function applications to their range, and of applications of semantic functions.

Rewriting of Set Memberships. To prove predicates of the form $x \in S$, it is often necessary to rewrite S into its definition. This is very common in our proofs, arising from the application of semantic functions and laws. Definitions and laws in most cases have provisos that require entities to belong to some semantic set, and for function applications we need to show membership of the argument(s) to the function's domain.

To give an example, in order to rewrite $p_1 \wedge_P p_2$ into its semantic definition given in Section 3.4, we have to show $(p_1, p_2) \in \mathit{WF\_ALPHA\_PREDICATE\_PAIR}$, that is (p_1, p_2) belongs to the domain of the function. Using the definition of $\mathit{WF\_ALPHA\_PREDICATE\_PAIR}$ for rewrite reduces the goal to

$$(p_1, p_2) \in \{p_1 : \mathit{ALPHA\_PREDICATE};\ p_2 : \mathit{ALPHA\_PREDICATE} \mid (p_1.2, p_2.2) \in \mathit{WF\_UNIVERSE\_PAIR}\}$$

We moreover see that expanding the above generates again subgoals of the form $x \in S$, namely to establish membership of p_1 and p_2 to $\mathit{ALPHA\_PREDICATE}$ and $(p_1.2, p_2.2)$ to $\mathit{WF\_UNIVERSE\_PAIR}$. They are respectively provisos for type membership and compatibility of universes.

Conventionally, the default ProofPower tactic to achieve such simple rewrites is ($\mathit{rewrite\_tac\ thms}$), which takes as an argument the list of equational theorems used for rewriting. An inconvenience using it is that we always have to provide the definition of the global constant to be rewritten.

We make such proof steps easier by creating a parameterless tactic ($\mathit{prove\_\in\_tac}$), which first extracts the set S from the expression of the goal (provided the goal is of the above form), then automatically obtains its definition from the theory database, and subsequently uses it for rewriting. The tactic besides performs several other steps to ascertain that the goal is of the correct shape, and that the right-hand operand of the set membership is a global constant. It also performs standard simplification and stripping steps after the rewrite. The advantage of the tactic is that it does not require specific knowledge of the set. This is convenient in manual proof steps, but also in automatic proof tactics as a tentative step if the goal is of the form $x \in S$. In practice, simple tactics like ($\mathit{prove\_\in\_tac}$) do already make manual proof more efficient.

A second possibility that we exploit in rewriting sets is the use of proof contexts. They are structures of ProofPower to store equational theorems used for default rewriting. A problem with proof contexts in our work, and in general, is that they provide no control under what conditions expressions should be rewritten. This problem is investigated in more detail in Section 7.2 on high-level tactics.

Rewriting of Range Memberships. The development of more elaborate tactics directed by the structure of the goal is common place in proof strategy programming, and has also been showed to be generally powerful in reducing the proof effort in our work. To present an example of how this idea is used, a kind of theorem frequently encountered in subgoals is $f(x) \in R$ where $f : D \rightarrow R$ is typically some semantic function in the denotational model. It particularly occurs in rewriting function applications corresponding to theory-specific operators. To prove, for example, associativity of conjunction, stated by the following goal,

$$\{p_1, p_2, p_3\} \in \mathit{WF\_ALPHA\_PREDICATE\_SET} \vdash (p_1 \wedge_P p_2) \wedge_P p_3 = p_1 \wedge_P (p_2 \wedge_P p_3)$$

we aim at entirely eliminating the applications of $\wedge_P$ (see Section 3.4 for its definition). We do so by rewriting the outer conjunctions first. The order of rewrite matters since once an expression has been rewritten into its semantic representation (in terms of a binding set and universe), proving certain properties such as membership to semantic sets can become harder. One of the provisos for rewriting, for instance, $(p_1 \wedge_P p_2) \wedge_P p_3$ is that $p_1 \wedge_P p_2$ and p_3 are elements of $\mathit{ALPHA\_PREDICATE}$. (There is also the requirement for compatibility which we shall ignore here.) Because $\wedge_P$ is a total function whose range is $\mathit{ALPHA\_PREDICATE}$, it is sufficient to prove that (p_1, p_2) is in its domain to establish that $p_1 \wedge_P p_2 \in \mathit{ALPHA\_PREDICATE}$; thus we have a theorem of the form $f(x) \in R$, where f is $\wedge_P$ and R is $\mathit{WF\_ALPHA\_PREDICATE\_PAIR}$.

The proof relies on the trivial law $f : D \rightarrow R \wedge x \in D \Rightarrow f(x) \in R$. We can hereby reduce the proof of $p_1 \wedge_P p_2 \in \mathit{ALHPA\_PREDICATE}$ to $(p_1, p_2) \in \mathit{WF\_ALPHA\_PREDICATE\_PAIR}$ exploiting the axioms of $\wedge_P$ which specify it to be a total function. To automate this step, we define a parameterless tactic (*prove_app_tac*) that automatically reduces goals of the form $f(x) \in R$ to subgoals $x \in D$, and does so independently of the actual definition of the function f as well as of the domain and range sets.

The specification of this tactic is slightly more complicated than the previous one as it makes use of a supplementary theorem, but in principle follows a similar approach of first extracting the name of the function, then obtaining the appropriate axiom that defines its functional type, adding it to the current list of assumptions, and finally using the backward-chaining theorem below to reduce the goal.

$$\vdash \forall f : \mathbb{U};\ X : \mathbb{U};\ Y : \mathbb{U};\ x : \mathbb{U} \mid x \in X \bullet f \in X \rightarrow Y \vee f \in X \rightarrowtail Y \vee f \in X \twoheadrightarrow Y \vee f \in X \rightarrowtail\!\!\!\!\!\twoheadrightarrow Y \Rightarrow f(x) \in Y$$

The backward-chaining theorem in addition implicitly deals with the case of total injections, surjections and bijections. To support backward-chaining using theorems which are written in the Z sub-language of ProofPower, a few other low-level tactics had to be implemented which are not mentioned here. Observe that the theorem by itself is not sufficient to carry out the reduction performed by (*prove_app_tac*) because generated subgoals of the form $f \in X \rightarrow Y$ then would have to be manually discharged.

Rewriting of Function Applications. The previous two tactics have been useful in many practical cases but their application is limited to subgoals of a very specific form. In the sequel, we present a generic tactic that much more substantially automates repetitive steps in proofs that inherently do not require human interaction, and in combination with other (high-level) tactics reduces the size of proof scripts in some cases by a ratio of up to 70% compared to similar ones in [OCW07].

A recurring task when proving properties in our mechanisation, and presumably in any deep semantic embedding, is that the application of semantic functions has to be frequently rewritten or eliminated. This applies, for example, when proving elementary laws that rely on the semantic definition of the operators, but also when applying laws to conduct proofs at a more algebraic level. Many of the laws in the UTP are expressed in terms of equalities, and the main strategy for proof is indeed rewriting of terms.

To prove the associativity law for $\wedge_P$ previously presented, we want to rewrite both sides of the equation purely in terms of binding sets and universes, eliminating all occurrences of $\wedge_P$ and other dependent operators it may unfold into. This kind of rewrite differs from the default rewriting of ProofPower in that the theorems (or axioms) used for rewriting have assumptions to be discharged. ProofPower clearly has expressive mechanisms to deal with term rewriting, but they are not immediately designed to handle equalities qualified by assumptions. In practical terms this meant that in many of the proofs in [OCW07] the standard rewrite facilities could not be used for the purpose of eliminating semantic functions and applying laws.

The process involves several steps for each individual application to be rewritten, and furthermore requires manual instantiation of the defining axioms for the operators. They are in most cases of the form

$$\vdash \forall x_1 : T_1;\ x_2 : T_2;\ \ldots \mid P_1 \wedge P_2 \wedge \ldots \bullet f(x_1, x_2, \ldots) = E$$

Where many applications have to be rewritten in succession, the resulting goal expression can become very large spanning over multiple pages of formulae. For example, equivalence unfolds into implications and conjunction, implication unfolds into negation and disjunction, and so on.

To facilitate this process in backward proofs, we provide a set of tactics and conversions that are able to process rewrite theorems with assumptions, and cumulatively generate subgoals for all provisos to be discharged. This extension gives rise to a framework that reimplements all of the standard functions of ProofPower for rewriting, albeit in a more powerful way to directly process theorems for laws and defining axioms of operators like the one above. The thereby provided rewrite tactics, rules and conversions were given similar names to those in ProofPower, however prefixed with 'z_' to highlight that they are particularly useful in the view of the Z extension of ProofPower. (Theorems in ProofPower-Z often have assumptions, let it be only to establish type membership of variables.) A crucial advantage of this approach is that now more sophisticated rewrite mechanisms can be implemented using all of ProofPower's default tools, for example controlling traversal orders, combining multiple rewrites in one invocation, and many others.

As an example, the tactic (*z_rewrite_fun_tac ops*) takes a list of operator terms *ops*, and in one atomic step rewrites them in the correct (top-down) order within the goal; while doing so, it automatically generates subgoals that need to be discharged for the rewrite to succeed. Another tactic is (*z_rewrite_tac thms*) which instead takes a list of theorems that now may be quantified equalities that may include assumptions. To exercise more control over the order in which functions are rewritten, we also provide respective conversions. Conversions in ProofPower are a convenient mechanism to rewrite subexpressions by means of equality theorems exploiting the axiom of referential transparency. They can be combined in various ways to specify which subexpressions should be rewritten and also the order in which recursive rewrite has to proceed.

The implementation of the new rewrite framework is in essence simply based on a different canonicalisation of the theorems before they are used for rewriting. This means that outer universal quantifications are automatically removed, and provisos are moved into the assumptions of the theorem. Hence, the above defining axiom would be canonicalised into the following theorem prior to being used.

$$x_1 \in T_1 \wedge x_2 \in T_2 \wedge \ldots, P_1 \wedge P_2 \wedge \ldots \vdash f(x_1, x_2, \ldots) = E$$

Although ProofPower can in principle use this theorem for rewriting, a technical problem arises when employing it within structured proofs, for example, as an argument to the standard rewrite and conversion tactics; by default these expect equality theorems without assumptions. To solve the problem, we specified enhanced versions of certain default tactics, such as *conv_tac_sharp* which extends the behaviour of ProofPower's *conv_tac* to properly handle assumptions when performing rewrites in backward proofs.

Our experience showed that by using (*z_rewrite_fun_tac ops*), rewriting of semantic functions can usually be done in only a few lines of proof script, and the only work required is the discharge of the assumptions of the rewrite theorems. Similarly, laws can be applied in a very flexible manner; namely, the same law may be applied multiple times, or interleaved in defined order with other laws. Conversions provide the expressive power to specify such actions. The residual goals are of a more specific nature, and we have an infrastructure of high-level tactics to automate their proof in many cases. The next section discusses them.

7.2. High-level Tactics for Automation

High-level tactics are intended to accomplish proofs whose goals are very specific to the UTP embedding. They typically involve more complex reductions and recursive interaction with other tactics to be effective. To give an example, we consider rewriting the disjunction in $\neg_P (p_1 \wedge_P p_2) \vee \neg_P (p_2 \wedge_P p_3)$. Using the tactic (*z_rewrite_fun_tac* $[⌜_Z (\_ \vee_P \_)⌝]$) for rewriting of the top-level application yields the following subgoals.

1. $\neg_P (p_1 \wedge_P p_2) \in \mathit{ALPHA\_PREDICATE}$
2. $\neg_P (p_2 \wedge_P p_3) \in \mathit{ALPHA\_PREDICATE}$
3. $(\neg_P (p_1 \wedge_P p_2), \neg_P (p_2 \wedge_P p_3)) \in \mathit{WF\_ALPHA\_PREDICATE\_PAIR}$.

We observe that the proof of some of these properties is a recursive process; for example, to show (1) we can apply (*prove_app_tac*) and thereby eliminate $\neg_P$ and reduce the goal to $p_1 \wedge_P p_2 \in \mathit{ALPHA\_PREDICATE}$. Further application of (*prove_app_tac*) to this subgoal yields another triple of subgoals.

1. $p_1 \in \mathit{ALPHA\_PREDICATE}$
2. $p_2 \in \mathit{ALPHA\_PREDICATE}$
3. $(p_1, p_2) \in \mathit{WF\_ALPHA\_PREDICATE\_PAIR}$.

We often know by assumption (or otherwise can easily conclude so via forward-chaining) that individual predicates such as p_1, p_2, and p_3 belong to $\mathit{ALPHA\_PREDICATE}$ and are mutually compatible; this trivially discharges subgoals like (1) and (2). For subgoal (3) a different strategy has to be used, that is, applying (*prove_*$\in$*_tac*). This generates further subgoals that capture the compatibility requirement; they, however, can be eventually discharged by the initial hypotheses.

The example illustrates that the way we proceed in each step depends on the structure of the goal, and further that tactics have to be applied recursively to emerging subgoals. ProofPower provides proof contexts to encapsulate proof strategies, but by default they follow a rigid pattern: first rewriting the goal and hypotheses according to an equational context of rewrite theorems, and then performing a resolution-based proof. It is on the other hand not possible to store a dynamic collection of tactics in a proof context.

What we require, however, is a mechanism that allows exactly this, and to exercise finer control when those tactics are applied. For this we have a framework that automates proofs as the above for arbitrarily complex predicates; it is flexible, modular, and allows the reasoning capabilities to be dynamically extended when new theorems are added and new theories are incorporated into the UTP hierarchy.

The General Proof Tactic. The general proof tactic is $utp\_gen\_prove\_tac$, which first performs several basic initial proof steps such as carrying out default rewriting and stripping of the goal. It then obtains the goal and uses a dictionary of expression patterns to determine the tactic(s) that should be applied to the goal. This dictionary can be dynamically extended to associate new tactics with arbitrary goal patterns. Once the tactics applicable to the current goal are obtained, we apply them in their recorded order until one of them succeeds. Otherwise, if no match is found some finalising actions are performed that usually require the proof to be interactively completed, leaving remaining subgoals on the goal stack.

The tactic dictionary is implemented as a discrimination net [CRMM87] in ProofPower. This is a particular data structure that supports efficient lookup of objects indexed by terms, albeit may deliver spurious results that are not exact matches of the pattern. This does not matter since in such cases the application of the tactic simply fails, and the next candidate is tried. Efficiency is an issue because the dictionary is deemed to become large with more tactics being registered throughout the hierarchy. The general proof tactic is frequently used and thereby becomes a bottleneck for run-time performance.

The registration of new tactics can be performed at any point when new theories, definitions, and theorems are added. Monolithic high-level tactics for particular proof tasks, on the other hand, become very unwieldy and do not do justice to the modularity and interdependency of tactics at different levels of the hierarchy. The current approach also supports recursive calls to $utp\_gen\_prove\_tac$ at any point in the component tactics; recursion is commonly used to discharge subgoals generated within tactics.

Execution of Component Tactics. In the encoding of the ProofPower-Z theory utp-pred, for example, several tactics for automation are configured. One of these, $PROVE\_\in\_ALPHA\_PREDICATE\_TAC$, applies to goals of the general form $p \in ALPHA\_PREDICATE$ where p can be an arbitrary expression of the right type. It is one of the essential components in automating the proof of provisos for rewriting semantic functions as considered above. It first checks that the goal is of the correct form, and afterwards extracts the left-hand side p of the membership. It then determines whether it is a function application, and if so whether the operator belongs to $ALPHA\_PREDICATE\_OPS$, a list that records all operators on alphabetised predicates. If this is the case, the previously explained low-level tactic ($prove\_app\_tac$) is invoked to try and prove the goal, and $utp\_gen\_prove\_tac$ is recursively applied to all subgoals resulting from this application. Proving, for example, $p_1 \wedge_P p_2 \in ALPHA\_PREDICATE$ would result in $utp\_gen\_prove\_tac$ being applied to $(p_1, p_2) \in WF\_ALPHA\_PREDICATE\_PAIR$. By registering another tactic that handles goals of this form we enable the proof to proceed recursively, and in principle to establish membership to $ALPHA\_PREDICATE$ for arbitrarily complex predicates.

If the (recursive) application of $utp\_gen\_prove\_tac$ determines that no recorded tactic for a certain form of goal exists, or the applied tactic did not discharge the goal but instead produced a collection of subgoals, the residual goals are simply left on the goal stack to be tackled manually. The user may then decide to either conduct these proofs by hand, or otherwise enhance $utp\_gen\_prove\_tac$ by registering additional component tactics to improve its capabilities in a reusable manner.

Contention arises if more than one tactic is applicable. Some refinements of the operational behaviour are possible; for example, we could preferably select tactics that leave fewer subgoals. Such extensions can be incorporated if they display practical benefit; experience still needs to be gained in this respect.

In summary, the execution mechanism of the general proof tactic enables us to exercise finer control when, for example, backward and forward-chaining tactics are applied, and effectively promotes the use of a combination of backward-chaining, forward-chaining, and rewriting at a *custom level of abstraction*. The behaviour of the general proof tactic depends largely on the component tactics. By having a single, dynamically growing, high-level tactic as a common entry point for all emerging subgoal proofs, we can incorporate additional proof capabilities as required, and let tactics added earlier take advantage of capabilities incorporated later without having to redefine any of the earlier encoded functions. Our approach moreover breaks down complex proofs into small, manageable component tactics that can be distributed across theories.

Practical Application and Experiences. We have created several component tactics primarily to automate proofs about universes, and membership to semantic sets such as *ALPHA_PREDICATE*, *UNIVERSE*, and related ones. The tactic suite for universes also entails tactics to prove compatibility of universes; this is necessary and exploited, for example, to prove membership to *WF_ALPHA_PREDICATE_PAIR*, hence proofs about predicates and universes are tightly coupled. In fact, we automate proofs much more complex than those presented in this section, such as the equivalence of elaborate universe expressions using a combination of high-level and custom normalisation tactics; this confirms the scalability of the approach.

High-level tactics are in particular useful in combination with the backward-chaining rewrite tactic (*z_rewrite_fun_tac ops*) discussed in the previous section. We can put the two together as follows.

```
    (z_rewrite_run_tac ops) THEN_BUT_FIRST UTP_GEN_PROVE_TAC
```

The effect of this tactic is to rewrite all semantic functions given by `ops`, and then try to automatically discharge all subgoals generated, apart from the original one. *THEN_BUT_FIRST* is an infix tactic combinator that applies the second argument to all subgoals produced by the left-hand tactic, the first goal excluded. Depending on the success rate of *utp_gen_prove_tac*, this can entirely automate the rewrite of semantic functions. More importantly, a similar approach is also feasible for discharging provisos when applying, for instance, refinement laws. The automation of individual law applications is necessary to automate more complex refinement strategies such as the one in [CCO05].

To conclude this section, we note that Oliveira suggests in [OCW07] that specialised tactics may be used to prove typing premises such as $p \in REL\_PREDICATE$ for more complex predicates, and considered this as a potentially significant reduction in proof effort. Our practical experience confirms this. We also claim the approach to be viable to automate (low-level) aspects of reasoning about particular specifications, that is, to discharge the proofs of assumptions for the automated application of laws.

8. Conclusion

We have presented a semantic encoding of the UTP in ProofPower-Z that provides facilities for theory instantiation and thus allows us to mechanically reason about UTP theories in a specific as well as general manner. Previous work on mechanised reasoning in the UTP was geared towards proving laws valid in certain families of theories rather than properties of particular models. In contrast, our approach supports reasoning about (elements of) specific instances of theories, and as almost a side effect, about theories in general. Families can be characterised in a very general way, for example by properties of the their healthiness conditions. We also support succinct reasoning about relationships between theories such as theory links. Our work can be regarded as a recast of Oliveira's encoding that enables us to formulate and discharge refinement conjectures for specifications and implementations within arbitrary UTP theory instantiations by permitting predicates of different theories to coexist in the same declarative scope. We have also examined opportunities for automation in proofs, and illustrated how modularity and reusability are exploited not just at the level of definitions and theorems, but also tactics for automation.

Since the very beginning we tried to minimise changes to the semantic encoding in [OCW07] in order to increase the likelihood of reusing the majority of the existing laws and proofs. This tight-rope walk unfortunately proved to fail, forcing us to open a Pandora's Box by incorporating a notion of theory and instantiation. Consequently, a lot of the existing laws are rephrased as discussed in Section 5 making it much harder to transfer existing mechanical proofs. On the positive side, this provides us with the opportunity to address issues that deserve further attention in the existing work; they are discussed below.

A first problem is consistency. In general, the axiomatic definitions of constants in ProofPower-Z are not consequently checked for introducing contradictions. We can enable and actually enforce such checks, however previous work did not exploit this facility. This did in fact result in an inconsistency: in the introduction we hinted that *BINDING* would have be to specified loosely in order to allow further type constraints being imposed on the variables. Previous work, however, used the unambiguous definition

$BINDING \mathrel{\widehat{=}} NAME \mathrel{\ooalign{$\rightarrow$}\!\!\!\!+} VALUE$. It is unlikely this inconsistency was exploited in any of the proofs, but especially with automated proof tactics there is always a potential risk of doing so without realising.

We are currently working towards establishing consistency of all axiomatic definitions, and reduce or avoid the use of *a posteriori* constraints being placed on existing variables as they are not checked. This has been taken into consideration when recasting the existing definitions. For example, to handle the restrictions on the type of $okay$ and $okay'$ in a theory of designs, we do not impose any constraints on a previously introduced set. Instead, we define a set $DES\_UNIVERSE$, which explicitly specifies the domain of the instantiation function $InstDesTheory$ presented in Section 3.5. To apply $InstDesTheory$ to some universe u, we have to prove that u introduces the correct type restrictions on the auxiliary variables. Otherwise, the result of the function application is undefined, and this can be detected as soon as we attempt to prove properties about $InstDesTheory\ u$ because of the absence of knowledge concerning its value. This is not, however, an inconsistency and does not raise the possibility of vacuous proofs.

A second problem has to do with taming the complexity introduced by formalising theories. It seems inevitable that we have to associate theories with a universe that captures the typing constraints on variables in the alphabet, but besides it proves essential to equip alphabetised predicates themselves with a universe in order to provide sufficient information for operators such as negation or substitution. These operators need to know about the types of variables; for example, negating $b = \mathbf{true}$ should contain the bindings where b equals **false**, but not any other values such as 1, 2, and so on. Associating alphabetised predicates with universes seems to yield a more coherent encoding than, for example, associating them with theories. The latter, besides, does not reflect the fact that a predicate can belong to more than one theory.

Finally, in this paper we adopt a more succinct universe model than the one we proposed in [ZC08, ZC09]. In particular, this facilitates proofs about specifications, where we are often required to verify properties about the universes of the predicates involved; this especially amounts to discharging antecedents of algebraic and refinement laws. The new model does not seem to introduce any additional complication in terms of proving general laws about predicates, but considerably simplifies those formerly mentioned proofs, which are symptomatic for reasoning about particular specifications. This aspect of mechanical proof is what we primarily aim to automate in the long run. Based on universe laws, we have developed normalisation tactics for universe expressions, which further play an important part in simplifying proofs.

A noteworthy piece of related work is Nuka's mechanisation of the alphabetised relational calculus [NW04] and UTP [NW06]. It explores a mechanised semantic model for alphabetised predicates, and the definition of common UTP operators. The work is especially interesting as it assumes an untyped view of predicates, but otherwise shares conceptual similarities with Oliveira's encoding and our own by representing predicates as sets of bindings, and introducing a unified value domain. A problem in that work arises if we represent, for example, predicates such as $x' = x + 1$. Semantically, we construct the set of bindings that render the predicate true, but in an untyped world it is not clear what values x and x' must range over. In [NW06] this is the set of all values, but this can result either in undefinedness when we evaluate expressions like $true = false + 1$, or possibly incompleteness if we force all functions on values to be total in order to guarantee that terms such as $false + 1$ are defined. We observe that the definition of $=_P$ in Section 3.4 solves this by only quantifying over the bindings that are well typed according to the given universe.

Although we used ProofPower-Z as our proof environment, the work could have potentially been done in other theorem provers as well. PVS [SRI], for example, offers specific features for dynamic instantiation of parametrised axiomatic theories. This could be explored as a means for instantiating UTP theories and providing further structuring mechanisms for encapsulating their axioms and theorems. The comparative study [Gor95] suggested that PVS supersedes Isabelle and HOL in user-friendliness and has more powerful built-in decision procedures for proof, but at the same time lacks the openness and extensibility of HOL that is afforded by the LCF approach. In particular, HOL appears to be more suitable for developing special-purpose proof infrastructures; we also profit from this using ProofPower-Z, being at its core based on HOL. An ongoing investigation is how this work could be done in alternative provers such as Coq [Ins].

Future work will first investigate how the large collection of laws proved in Oliveira's original encoding can be transferred to our setting. The work we have done so far on simplifying, for example, the rewriting of semantic functions, and further experience gained with proofs about universes, should make this process manageable. Rather than merely rephrasing the laws, one is challenged to find ways of adopting proofs while making them more robust and maintainable; the existing proofs amount to approximately 80,000 lines of proof script, and it would be desirable to reduce the effort for their recreation.

Further experience needs to be gained with proving properties of particular specifications in different UTP theories. This far we have only carried out toy-example experiments. The wider objective of this research is to use the encoding to do algebraic reasoning, for example, about *Circus* specifications and refinements in a general manner. Automation of such reasoning poses a particular challenge; a benchmark is the ClawZ [AC05] suite of tools, which shows that verification of embedded control systems can be carried out by engineers without in-depth knowledge of the underlying formalism and semantics. To do justice to this goal, we currently investigate the integration of ArcAngel*C* [OC08], a tactic language specifically developed for refinement, into ProofPower. It supports the specification of high-level strategies for refinement, and eventually, we anticipate, the development of verification tools based on the mechanisation.

Current approaches to verify implementations of control systems [AC05] translate the specification of the control law into a Z model, encoded for ProofPower-Z, and use built-in reasoning support for Z to discharge refinement proof obligations — aided by custom, high-level tactics to automate proof procedures. An extension of this approach using *Circus* is presented in [CCO05]. It considers a wider class of models and implementations by capturing and describing parallelism in the control law and supporting the refinement into concurrent programs. One line for future work is to use the mechanisation and embedding of *Circus* to conduct refinement proofs of such control systems. A refinement strategy is presented in [CCO05] that relies upon a collection of *Circus* laws which we aim to prove in the mechanisation. The application of the laws to particular specifications will be facilitated by tactics, building on the principles discussed in Section 7. The refinement strategy, on the other hand, can be expressed in ArcAngel*C* [OC08]. Future research will determine whether and how the combination of the two can in practice effectively automate proofs arising from this approach to verifying control systems. We hope that the experience will give rise to new methods and verification tools based on our mechanisation of the UTP.

9. Acknowledgements

Valuable discussions have taken place with Marcel Oliveira regarding his original encoding, experience with mechanical proofs, and possible ways to overcome the inconsistency issue that was first pointed out by Steve Dunne. We would also like to acknowledge EPSRC for funding this work under the "Programming from Control Laws" research grant EP/E025366/1.

A. Appendix: Relevant Definitions of the Mechanisation

In this appendix we include an extract of the Z definitions that are relevant to the material presented in the paper. It is not intended to give a comprehensive account of the entire mechanisation in ProofPower-Z. For that we refer to the theory source published at `http://www.cs.york.ac.uk/circus/tp/tools.html`.

A.1. ProofPower Theory utp-lang (Common Language Definitions)

ProofPower-Z Definition 1. *Type representing variable names.*

$$NAME \;\widehat{=}\; \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

The first component of the tuple is a unique identifier, the second component indicates the number of dashes, and the third component specifies a possible subscript.

ProofPower-Z Definition 2. *The dash function is used to decorate a name with a dash.*

$$\begin{array}{|l} dash : NAME \rightarrowtail NAME \\ \hline \forall\, i, j, k : \mathbb{N} \bullet dash\,(i, j, k) \;=\; (i, j + 1, k) \end{array}$$

ProofPower-Z Definition 3. *Semantic set that characterises universes.*

$$UNIVERSE \mathrel{\widehat{=}} NAME \pfun TYPE$$

Universes record type information for the names in their alphabets by means of a partial function. The set $TYPE$ *is defined as* $TYPE \mathrel{\widehat{=}} \mathbb{P}_1\ VALUE$*, hence a type must at least include one value.*

ProofPower-Z Definition 4. *Alphabet of a universe.*

$$\begin{array}{|l}
Alphabet_U : UNIVERSE \rightarrow ALPHABET \\
\hline
\forall u : UNIVERSE \bullet Alphabet_U\ u\ =\ \mathrm{dom}\, u
\end{array}$$

The alphabet of a universe is simply its domain.

ProofPower-Z Definition 5. *Bindings of a universe.*

$$\begin{array}{|l}
Bindings_U : UNIVERSE \rightarrow BINDINGS \\
\hline
\forall u : UNIVERSE \bullet \\
\quad Bindings_U\ u\ =\ \{b : BINDING \mid \mathrm{dom}\, b = Alphabet_U\ u \land (\forall n : \mathrm{dom}\, b \bullet b(n) \in u(n))\}
\end{array}$$

The rôle of the first constraint $\mathrm{dom}\, b = Alphabet_U\ u$ *is to ensure that we only consider bindings that exactly associate the variables of the universe with values. Type correctness is guaranteed by the second constraint. It ensures that the value bound to* n *in a binding of the universe belongs to the type that* u *records for* n*.*

ProofPower-Z Definition 6. *Binary merge of universes.*

$$\begin{array}{|l}
\_ \oplus_U \_ : WF\_UNIVERSE\_PAIR \rightarrow UNIVERSE \\
\hline
\forall u_1, u_2 : UNIVERSE \mid (u_1, u_2) \in WF\_UNIVERSE\_PAIR \bullet u_1 \oplus_U u_2\ =\ u_1 \cup u_2
\end{array}$$

Above $WF\_UNIVERSE\_PAIR$ *restricts the arguments to be compatible universes.*

ProofPower-Z Definition 7. *Restriction of universes.*

$$\begin{array}{|l}
\_ \ominus_U \_ : (UNIVERSE \times ALPHABET) \rightarrow UNIVERSE \\
\hline
\forall u : UNIVERSE;\ a : ALPHABET \bullet u \ominus_U a\ =\ a \lhd u
\end{array}$$

Restriction of a universe is simply defined in terms of domain restriction.

ProofPower-Z Definition 8. *Merge of a set of universes.*

$$\begin{array}{|l}
Merge_U : WF\_UNIVERSE\_SET \rightarrow UNIVERSE \\
\hline
\forall us : WF\_UNIVERSE\_SET \bullet Merge_U\ us\ =\ \bigcup us
\end{array}$$

Above $WF\_UNIVERSE\_PAIR$ *restricts the argument to be a set of universes whose members have to be pairwise compatible.*

ProofPower-Z Definition 9. *Renaming of the variables of a universe.*

$$WF\_Rename_U \mathrel{\widehat{=}} \{f : NAME \pinj NAME;\ u : UNIVERSE \mid Alphabet_U\ u \in \mathrm{dom} f\}$$

$$
\begin{array}{|l}
Rename_U : WF\_Rename_U \to UNIVERSE \\
\hline
\forall f : NAME \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;\; \leftrightarrow NAME;\ u : UNIVERSE \mid (f, u) \in WF\_Rename_U \bullet \\
\qquad Rename_U\,(f, u) \ = \ \{n : NAME;\ t : \mathbb{P}\,VALUE \mid (n, t) \in u \bullet f(n) \mapsto t\}
\end{array}
$$

The restriction WF_Rename_U ensures that the variables of the universe to be renamed are in the domain of the remaining function f. The renaming function has to be an injection on names.

A.2. ProofPower Theory utp-theory (UTP Theories)

ProofPower-Z Definition 10. *$ApplyHealthConds\,(p, hs)$ realises the application of a sequence of healthiness functions hs to a predicate p. Its use is to construct a healthy predicate from an unhealthy one.*

$$
\begin{array}{|l}
ApplyHealthConds : ALPHA\_PREDICATE \times seq\,HEALTH\_COND \nrightarrow ALPHA\_PREDICATE \\
\hline
\forall p : ALPHA\_PREDICATE;\ hs : seq\,HEALTH\_COND \bullet \\
\quad ApplyHealthConds\,(p, hs) \ = \ (fold\ hs)\ p
\end{array}
$$

In the above, the application (fold hs) realises the successive application of the functions in hs, that is their folding. The formal Z definition of fold is included with the following definition.

ProofPower-Z Definition 11. *Folding of a sequence of functions.*

$$
\begin{array}{|l|}
\hline
[X] \\
fold : seq\,(X \nrightarrow X) \to (X \nrightarrow X) \\
\hline
(\forall fs : seq\,(X \nrightarrow X) \mid fs = \langle\rangle \bullet fold\ fs = id\ X) \land \\
(\forall fs : seq\,(X \nrightarrow X) \mid fs \neq \langle\rangle \bullet fold\ fs = (head\ fs) \mathbin{;} (fold\,(tail\ fs))) \\
\hline
\end{array}
$$

Here, head and tail are functions that yield the head and tail of a sequence, and id yields the identity relation on a given set. Making the fold function generic promotes its reuse in other possible contexts.

A.3. ProofPower Theory utp-rel (Relations)

ProofPower-Z Definition 12. *Relational Skip.*

Domain of Relational Skip.

$$
\begin{array}{|l}
REL\_UNIVERSE\_HOM \ \widehat{=} \ \{u : REL\_UNIVERSE \mid Alphabet_U\ u \in homogeneous\}
\end{array}
$$

Definition of Relational Skip.

$$
\begin{array}{|l}
\mathbf{II}_R : REL\_UNIVERSE\_HOM \to REL\_PREDICATE \\
\hline
\forall u : REL\_UNIVERSE\_HOM \bullet \\
\quad \mathbf{II}_R\ u \ = (\{b : u \mid dom\,b = Alphabet_U\ u \land \\
\qquad (\forall n : Alphabet_U\ u \mid n \in undashed \bullet b(n) = b(dash\ n))\}, u)
\end{array}
$$

The restrictions imposed by $REL\_UNIVERSE\_HOM$ ensure that the universe only mentions undashed and single-dashed names, and that it is moreover homogeneous. The bindings of the relational Skip are exactly those that associate corresponding undashed and dashed names with the same value.

ProofPower-Z Definition 13. *Assignment.*

Domain of Assignment.

$$\begin{array}{l}
WF\_Assign_R \mathrel{\widehat=} \{u : UNIVERSE;\ ns : \mathrm{iseq}\, NAME;\ es : \mathrm{seq}\, EXPRESSION \mid \\
\quad Alphabet_U\ u \in homogeneous \land \\
\quad (\forall n : \mathrm{ran}\, ns \bullet n \in Alphabet_U\ u \land n \in undashed) \land \\
\quad (\forall e : \mathrm{ran}\, es \bullet (Alphabet_U\ u, e) \in WF\_EXPRESSION \land FV(e) \in undashed) \land \\
\quad \#\, ns = \#\, es \neq 0\}
\end{array}$$

Definition of Assignment.

$$\begin{array}{l}
Assign_R : WF\_Assign_R \to REL\_PREDICATE \\
\hline
\forall u : UNIVERSE;\ ns : \mathrm{iseq}\, NAME;\ es : \mathrm{seq}\, EXPRESSION \mid \\
\quad (u, ns, es) \in WF\_Assign_R \land \#\, ns = 1 \Rightarrow \\
\quad\quad (\exists n : NAME \mid n = head(ns) \bullet \\
\quad\quad\quad Assign_R\,(u, ns, es) \ = \\
\quad\quad\quad\quad =_P (u, Var(dash\ n), head(es)) \land_P \mathbf{II}_R\,(Alphabet_U\ u \ominus_U \{n, dash\ n\})) \land \\
\quad (u, ns, es) \in WF\_Assign_R \land \#\, ns > 1 \Rightarrow \\
\quad\quad (\exists n : NAME \mid n = head(ns) \bullet \\
\quad\quad\quad Assign_R\,(u, ns, es) \ = \\
\quad\quad\quad\quad =_P (u, Var(dash\ n), head(es)) \land_P \\
\quad\quad\quad\quad\quad Assign_R\,(Alphabet_U\ u \ominus_U \{n, dash\ n\}, tail(ns), tail(es)))
\end{array}$$

Since assignment is to be generally defined for lists of variables and expressions, we split the definition into two cases: one for the base case of a singleton list and one for the inductive case. Each variable assigned amounts to the encoding of an equality of the form $n' = e$, and other variables retain their value.

ProofPower-Z Definition 14. *Non-deterministic Choice.*

Domain of Choice.

$$\begin{array}{l}
WF\_REL\_PREDICATE\_PAIR \mathrel{\widehat=} \\
\quad \{p_1 : REL\_PREDICATE;\ p_2 : REL\_PREDICATE \mid \\
\quad\quad \exists th : REL\_THEORY \bullet \{p_1, p_2\} \subseteq TheoryPredicates\ th\}
\end{array}$$

Definition of Choice.

$$\begin{array}{l}
(\_ \sqcap_R \_) : WF\_REL\_PREDICATE\_PAIR \to REL\_PREDICATE \\
\hline
\forall p_1 : REL\_PREDICATE;\ p_2 : REL\_PREDICATE \mid \\
\quad (p_1, p_2) \in WF\_REL\_PREDICATE\_PAIR \bullet p_1 \sqcap_R p_2 \ = \ p_1 \lor_P p_2
\end{array}$$

Non-deterministic choice is simply defined in terms of disjunction, as explained in Section 2. The restriction WF_REL_PREDICATE_PAIR specifying the domain of $\sqcap_R$ captures that the argument predicates have to belong to the same relational theory. That is, they have to be relations and their universe must be the same.

ProofPower-Z Definition 15. *UTP Conditional.*

Domain of UTP Conditional.

$$\begin{array}{l}
WF\_Cond_R \mathrel{\widehat=} \{u_1, b, u_2 : REL\_PREDICATE \mid \\
\quad \{u_1, b, u_2\} \in WF\_REL\_PREDICATE\_SET \land_R \\
\quad Alphabet_P\ b \subseteq Alphabet_P\ u_1 \land Alphabet_P\ u_1 = Alphabet_P\ u_2\}
\end{array}$$

Definition of UTP Conditional.

$$\begin{array}{|l} \_ \lhd_R \_ \rhd_R \_ : WF\_Cond_R \rightarrow REL\_PREDICATE \\ \hline \forall\, p_1, b, p_2 : REL\_PREDICATE \mid p_1 \lhd_R b \rhd_R p_2 \;=\; (b \wedge_P p_1) \vee_P (\neg_P\, b \wedge_P p_2) \end{array}$$

The conditional is defined in the familiar way. WF_Cond_R requires the predicates to have the same universe, and the variables of the condition to be included in the alphabets of the predicate's universes. Observe that the function $Alphabet_P\ p$ simply yields $Alphabet_U\ p.1$: the alphabet of the universe of p.

A.4. ProofPower Theory utp-des (Designs)

ProofPower-Z Definition 16. *The function J is used to encode the healthiness idempotent* ***H2****.*

$$\begin{array}{|l} J : UNIVERSE \rightarrow\!\!\!\!\!+\; DES\_PREDICATE \\ \hline dom\, J = \{u : UNIVERSE \mid Alphabet_U\ u \subseteq dashed\_once\} \wedge \\ (\forall\, u : UNIVERSE \mid Alphabet_U\ u \subseteq dashed\_once \bullet \\ \quad J\ u = (OKAY \Rightarrow_P OKAY') \wedge_P \mathbf{II}_R((Rename_U(undash, u) \oplus_U u) \ominus_U ALPHABET\_OKAY)) \end{array}$$

$OKAY$ and $OKAY'$ encode the predicates okay and okay′, respectively. The $Rename_U$ function, as already explained, is used to rename the variables in a universe. $ALPHABET\_OKAY$ is the set $\{okay, okay'\}$.

ProofPower-Z Definition 17. *The constant $OKAY$ encodes the UTP predicate $okay =$ **true**. The universe of the predicate only includes the variable okay of boolean type.*

$$\begin{array}{|l} OKAY : DES\_COMPATIBLE \\ \hline OKAY \;=\; =_P (\{okay \mapsto BOOL\_VAL\}, Var(okay), True_E) \end{array}$$

The constant $True_E$ abbreviates the expression $Val(Bool(True))$.

ProofPower-Z Definition 18. *The constant $OKAY'$ encodes the UTP predicate $okay' =$ **true**. The universe of the predicate only includes the variable okay′ of boolean type.*

$$\begin{array}{|l} OKAY : DES\_COMPATIBLE \\ \hline OKAY \;=\; =_P (\{okay \mapsto BOOL\_VAL\}, Var(okay), True_E) \end{array}$$

The constant $True_E$ above abbreviates the expression $Val(Bool(True))$.

A.5. ProofPower Theory utp-rea (Reactive Designs)

ProofPower-Z Definition 19. *Instantiation function for reactive design theories.*

$$\begin{array}{|l} InstReaTheory : REA\_UNIVERSE \rightarrow UTP\_THEORY \\ \hline InstReaTheory\ u \;=\; SpecialiseTheory\,(InstRelTheory\ u, \{R1, R2, R3\}) \end{array}$$

Here, $R1$, $R2$ and $R3$ are functions that encode the healthiness idempotents for reactive designs.

ProofPower-Z Definition 20. *Set of reactive design predicates.*

$$\begin{array}{l} REA\_PROCESS \;\widehat{=} \\ \quad \{p : ALPHA\_PREDICATE \mid (\exists\, th : REA\_THEORY \bullet p \in TheoryPredicates\ th)\} \end{array}$$

For p to be a valid reactive design some reactive theory must exist whose predicates include p.

ProofPower-Z Definition 21. *Healthiness function **R1** for reactive designs.*

$$\begin{array}{|l} R1 : HEALTH\_COND \\ \hline dom\, R1 = REA\_COMPATIBLE \land (\forall\, p : REA\_COMPATIBLE \bullet R1\ p \ = \ p \land_P TRprfxTR') \end{array}$$

The constant $TRprfxTR'$ was previously defined in Figure 4.

ProofPower-Z Definition 22. *Healthiness function **R3** for reactive designs.*

$$\begin{array}{|l} R3 : HEALTH\_COND \\ \hline dom\, R3 = \{p : REA\_COMPATIBLE \mid p.2 \in WF\_Skip_{REA}\} \land \\ (\forall\, p : REA\_COMPATIBLE \mid p.2 \in WF\_Skip_{REA} \bullet R3\ p \ = \ (\mathbf{II}_{REA}\ p.2) \lhd_R WAIT \rhd_R p) \end{array}$$

WF_Skip_{REA} is the domain of the $\mathbf{II}_{REA}$ function. It requires the alphabet of the predicate to be homogeneous, with additional compatibility constraints imposed on the types of auxiliary variables, should they occur. WAIT encodes the predicate wait as was explained in Section 4.2.

A.6. ProofPower Theory utp-circus (Circus)

ProofPower-Z Definition 23. *Synchronising prefix for Circus actions.*

$$\begin{array}{|l} \_ \longrightarrow_{CSync} \_ : (VAR\_NAME \times CIRCUS\_ACTION) \to CIRCUS\_ACTION \\ \hline \forall\, n : VAR\_NAME;\ p : CIRCUS\_ACTION \bullet n \longrightarrow_{CSync} p \ = \ (n, Val(Sync)) \longrightarrow_C p \end{array}$$

Here $(n, v) \longrightarrow_C p$ is the general operator for a communication prefix in the theory of Circus. It is parametrised in terms of a channel name n, a value v, and the prefixed action p. We omit its definition which, however, can be found in the ProofPower theory scripts mention at the beginning of the appendix.

ProofPower-Z Definition 24. *Output prefix for Circus actions.*

$$\begin{array}{|l} \_ \longrightarrow_{Cout} \_ : WF\_PREFIXING_C \to CIRCUS\_ACTION \\ \hline \forall\, n : VAR\_NAME;\ e : EXPRESSION;\ p : CIRCUS\_ACTION \mid \\ \quad ((n, e), p) \in WF\_PREFIXING_C \bullet (n, e) \longrightarrow_{Cout} p \ = \ (n, e) \longrightarrow_C p \end{array}$$

Here $(n, v) \longrightarrow_C p$ is the general operator for a communication prefix in the theory of Circus, and its domain $WF\_PREFIXING_C$ identifies the constraints for its applicability. Both definition can be found in the ProofPower theory scripts mention at the beginning of the appendix.

ProofPower-Z Definition 25. *Local variable block in Circus.*

$$\begin{array}{l} WF\_var_C \ \widehat{=} \ \{n : VAR\_NAME;\ p : CIRCUS\_ACTION \mid \\ \quad (p.2, n) \in WF\_var_R\_end_R \land n \in ALPHABET\_OWTR\} \end{array}$$

$$\begin{array}{|l} var_C : WF\_var_C \to CIRCUS\_ACTION \\ \hline \forall\, n : VAR\_NAME;\ p : CIRCUS\_ACTION \mid \\ \quad (n, p) \in WF\_var_C \bullet var_C(n, p) \ = \ var_R(p.2, n) \,;_C\, p \,;_C\, end_R(p.2, n) \end{array}$$

The declaration of a local variable block in Circus directly reuses the functions for declaring local variables in relational theories, apart from additional restriction on the domain of the function that require the predicate to be a Circus action. In particular, $;_C$ is sequential composition of Circus actions, and $WF\_var_R\_end_R$ the domain of the var_R and end_R functions in the ProofPower-Z theory for UTP relations (utp-rel).

ProofPower-Z Definition 26. *Circus Assignment.*

$$\begin{array}{|l} Assign_C : WF\_Assign_C \rightarrow CIRCUS\_ACTION \\ \hline \forall\, u : UNIVERSE;\ ns : \mathrm{seq}\, NAME;\ es : \mathrm{seq}\, EXPRESSION \mid (u, ns, es) \in WF\_Assign_C \bullet \\ \qquad Assign_C(u, ns, es) \ = \ R\,(True_P\ u\ \vdash_D Assign_R(u, ns, es) \wedge_P TReqTR' \wedge_P (\neg_P WAIT')) \end{array}$$

Circus assignment is defined in terms of applying the reactive healthiness idempotent R to a design assignment. As before, $WAIT'$ encodes the predicate wait, and the definition of $TReqTR'$ can be found in Figure 4. Its domain WF_Assign_C ensures that the arguments are actions that belong to the same Circus theory.

ProofPower-Z Definition 27. *External Choice of Circus actions.*

$$\begin{array}{|l} \_ \sqcap_C \_ : WF\_CIRCUS\_ACTION\_PAIR \rightarrow CIRCUS\_ACTION \\ \hline \forall\, p_1, p_2 : CIRCUS\_ACTION \mid (p_1, p_2) \in WF\_CIRCUS\_ACTION\_PAIR \bullet \\ \qquad p_1 \ \sqcap_C \ p_2 = R\,(\\ \qquad\quad \neg_P\ (p_1\ \omega_f\ \sigma_f) \wedge_P \neg_P (p_2\ \omega_f\ \sigma_f) \\ \qquad\qquad \vdash_D \\ \qquad\quad ((p_1\ \omega_f\ \sigma_t) \wedge_P (p_2\ \omega_f\ \sigma_t)) \lhd_R TReqTR' \wedge_P WAIT' \rhd_R ((p_1\ \omega_f\ \sigma_t) \vee_P (p_2\ \omega_f\ \sigma_t))) \end{array}$$

In the above definition, ω_t and ω_f are post-fix operators that perform a substitution of wait with **true** *and* **false***, and σ_t and σ_f are similar operators that perform a substitution of $okay'$ and* **true** *or* **false***. The restriction imposed by $WF\_CIRCUS\_ACTION\_PAIR$ requires the arguments to be actions from the same Circus theory. $WAIT'$ encodes the predicate $wait'$, and the definition of $TReqTR'$ can be found in Figure 4.*

B. Appendix: Universe Model in [ZC08]

ProofPower-Z Definition 28. *Semantic definition of universes in our previous work.*

$$\begin{array}{l} UNIVERSE \mathrel{\widehat=} \{bs : BINDINGS \mid \\ \quad \varnothing \in bs \wedge (\forall\, b_1 : bs;\ b : BINDING \mid b \subseteq b_1 \bullet b \in bs) \wedge (\forall\, b_1, b_2 : bs \bullet b_1 \oplus b_2 \in bs)\} \end{array}$$

The first constraint requires the empty binding to be a member of any universe, the second constraint ensures that the bindings of a universe are subset-closed, and the third orthogonality constraint that type restrictions imposed on one variable cannot be sensitive to the values taken by other variables. Subset closure means that if, for instance, $\{x \mapsto 1, y \mapsto 2\}$ is a universe binding, so is $\{x \mapsto 1\}$.

References

[AC05] M. M. Adams and P. B. Clayton. ClawZ: Cost-Effective Formal Verification for Control Systems. In *Formal Method and Software Engineering: 7th International Conference on Formal Engineering Methods*, volume 3785 of *Lecture Notes in Computer Science*, pages 465–479. Springer, November 2005.

[BSW07] A. Butterfield, A. Sherif, and J. Woodcock. Slotted Circus: A UTP-family of reactive theories. In *Integrated Formal Methods: 6th International Conference*, volume 4591 of *Lecture Notes in Computer Science*, pages 75–97. Springer, July 2007.

[CCO05] A. Cavalcanti, P. Clayton, and C. O'Halloran. Control Law Diagrams in Circus. In *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 253–268. Springer, July 2005.

[CRMM87] E. Charniak, C. Riesbeck, D. McDermott, and J. Meehan. *artificial intelligence programming, 2nd edition)*. Lawrence Erlbaum Associates, 1987.

[CSW03] A. Cavalcanti, A. Sampaio, and J. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2-3):146–181, November 2003.

[CW04] A. Cavalcanti and J. Woodcock. A Tutorial Introduction to CSP in Unifying Theories of Programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*, pages 220–268. Springer, November 2004.

[Dij76] E. Dijkstra. *A Discipline of Programming.* Prentice Hall Series in Automatic Computation. Prentice Hall, 1976.

[FWC07] L. Freitas, J. Woodcock, and A. Cavalcanti. An Architecture for *Circus* Tools. In *SBMF 2007: Brazilian Symposium on Formal Methods*, August 2007.

[Gor88] M. Gordon. HOL: a Proof Generating System for Higher Order Logic. In *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[Gor95] M. Gordon. Notes on PVS from a HOL perspective. Technical report, August 1995.

[HCW08] W. Harwood, A. Cavalcanti, and J. Woodcock. A Model of Pointers for the Unifying Theories of Programming – Extended Version. Technical report, University of York, Department of Computer Science, UK, 2008. Available from `http://www-users.cs.york.ac.uk/alcc/publications/HCW08.pdf`.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall, February 1998.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series In Computer Science. Prentice Hall, April 1985.

[Ins] Institut National de Recherche en Informatique et en Automatique (INRIA). *The Coq Proof Assistant*. Tool and documentation available from `http://coq.inria.fr/`.

[MD00] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.

[NW04] G. Nuka and J. Woodcock. Mechanising the Alphabetised Relational Calculus. *Electronic Notes in Theoretical Computer Science*, 95:209–225, May 2004.

[NW06] G. Nuka and J. Woodcock. Mechanising a Unifying Theory. In *Unifying Theories of Programming, First International Symposium*, volume 4010 of *Lecture Notes in Computer Science*, pages 217–235. Springer, February 2006.

[OC08] M. Oliveira and A. Cavalcanti. ArcAngel*C*: a refinement tactic language for *Circus*. *Electronic Notes in Theoretical Computer Science*, 214:203–229, June 2008.

[OCW07] M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for *Circus*. *Formal Aspects of Computing*, Online First, December 2007.

[Oli05] M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science, University of York, 2005.

[Ros97] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall International Series In Computer Science. Prentice Hall, November 1997.

[SJ02] A. Sherif and He Jifeng. Towards a Time Model for *Circus*. In *Formal Method and Software Engineering: 4th International Conference on Formal Engineering Methods*, volume 2495 of *Lecture Notes in Computer Science*, pages 613–624. Springer, 2002.

[SRI] SRI International Computer Science Laboratory. *PVS Specification and Verification System*. Tool and documentation available from `http://pvs.csl.sri.com/`.

[Tar41] A. Tarski. On the Calculus of Relations. *Journal of Symbolic Logic*, 6(3):73–89, September 1941.

[VB98] E. Visser and Z. Benaissa. A Core Language for Rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422 – 441, 1998.

[WD96] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice-Hall International Series In Computer Science. Prentice Hall, 1996.

[XCS06] M. Xavier, A. Cavalcanti, and A. Sampaio. Type Checking *Circus* Specifications. In *SBMF 2006: Brazilian Symposium on Formal Methods*, pages 105–120, September 2006.

[ZC08] F. Zeyda and A. Cavalcanti. Mechanical Reasoning about Families of UTP Theories. In *SBMF 2008, Brazilian Symposium on Formal Methods*, pages 145–160, August 2008.

[ZC09] F. Zeyda and A. Cavalcanti. Encoding Circus Programs in ProofPower-Z. In *Unifying Theories of Programming 2008*, Lecture Notes in Computer Science. Springer, 2009. Awaiting publication.