Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 240 (2009) 239–257
www.elsevier.com/locate/entcs

Mechanical Reasoning about Families of UTP Theories

Frank Zeyda[1] and Ana Cavalcanti[2]

Department of Computer Science
University of York
Heslington, York, YO10 5DD, U.K.

Abstract

In this paper we present a semantic embedding of Hoare and He's Unifying Theories of Programming (UTP) framework into the ProofPower-Z theorem prover; it concisely captures the notion of UTP theory, theory instantiation, and, additionally, type restrictions on the alphabet. 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 elements of collections of theory instantiations, as typically found in UTP models of particular specifications and programs.

Keywords: theorem proving, ProofPower, Z, semantic embedding

1 Introduction

The Unifying Theories of Programming (UTP) [5] provide a framework that allows for the integration of a variety of programming languages with different computational paradigms within a unified relational model. It captures the meaning of imperative, functional, declarative and concurrent languages, for example, and identifies common features. It also provides a uniform 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 [7,12,2].

In [10,11] a deep embedding of the UTP tailored for the ProofPower-Z theorem prover is presented. In that work 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 designs, CSP processes, and the *Circus* [3]

[1] Email: zeyda@cs.york.ac.uk
[2] Email: ana.cavalcanti@cs.york.ac.uk

doi:10.1016/j.entcs.2009.05.055

language. This mechanisation has been successfully used to prove laws that are *generally* valid within the UTP theories.

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 relations similar to Tarski's [13] presented in a predicative style. Each predicate is associated with an alphabet: that of the underlying relation. A UTP theory is characterised by an alphabet determining the alphabet of its predicates, and a collection of healthiness conditions that identify the predicates that are valid models of computation.

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

$$x := x + 1 \lhd x = 1 \rhd I\!I \sqsubseteq x := 2$$

The notation $P \lhd b \rhd Q$ is used for conditionals: the program that behaves as P if b holds, else as Q. $I\!I$ 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, sometimes even necessary to exploit assumptions about the types of variables.

The alphabet of each of the UTP theories described in [5] 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, including its dashed counterpart used to denote the final value of x. Therefore, we can conclude that the theory descriptions in [5] 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 as the above. They mostly arise from the fact that neither a dynamic notion of UTP theory nor of instantiation of a theory are 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.

$$BINDING \mathrel{\widehat=} NAME \pfun VALUE$$

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

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

as an additional axiom.

We identify two 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 partly 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. For this reason it would not be possible for two predicates in which variable x has different types to coexist within the same ProofPower theory scope, a crucial limitation that we overcome in this work. The hierarchical presentation of UTP theories, which is explored in [11], has crucial benefits in terms of reusing definitions and laws, hence our aim is not to abandon it but rephrase it in a dynamic way.

The second difficulty is that defining subsequent constraints on constants might easily result in the model becoming 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 variables.

Finally, when reasoning about particular specifications, it is frequently necessary to work with predicates of different instances of a theory. A trivial example is provided by a variable block, whose body is a predicate in a theory whose alphabet is enriched by the variables declared in the block, and so different from the theory of the block itself. The availability of other encapsulation mechanisms, in languages like *Circus* [3] and TCOZ [7], for example, whose semantics are based on the UTP, raise more pervasive 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 instances of the theories.

At this point it is worth noting the significant distinction between UTP theories and ProofPower theories. Whereas ProofPower theories are 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 ProofPower reasoning scope.

To solve these problems, we introduce a semantic characterisation of UTP theory, 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 homomorphisms, Galois

connections, and so on. A third contribution is a method for encoding specific UTP theories which lends itself to integration into verification strategies based on a UTP semantics.

Section 2 further details the main principles and ideas of the UTP; the following Section 3 then presents the relevant parts of our semantic encoding defining the notion of alphabetised predicate, UTP theory and instantiation. Section 4 surveys how we reason about UTP theories in general, Section 5 is concerned with refinement laws and reasoning about particular specifications, and Section 6 addresses the formulation and proof of properties about theory links. Section 7 discusses a few relevant design issues and revisits the introductory refinement conjecture, and in Section 8 we finally draw our conclusions.

2 Unifying Theories of Programming

The predicates of UTP theories are customarily referred to as *alphabetised* predicates, and the alphabet of a predicate P is given by αP. As a simple example, we have the predicate $x' = x + 1$ with alphabet x and x', describing the computation that increments the value of the programming variable x. In general, undecorated variables are used to represent initial observations, and corresponding dashed variables to represent final observations.

The *alphabet* of a theory defines the variables that correspond to 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), for example. Healthiness conditions identify the set of predicates with the right alphabet that belong to the theory.

Standard predicate calculus operators can be used to combine alphabetised predicates; for example $x' = x + 1 \lor x' = x - 1$ specifies a computation that either increments or decrements the value x. A common feature across different UTP theories is nondeterministic choice $P \sqcap Q$ being defined as disjunction, and sequence of computations $P \,;\, Q$ characterised by relational composition.

Another important feature of UTP theories is a common notion of refinement which is universal (reverse) implication: $S \sqsubseteq P \mathrel{\widehat{=}} [P \Rightarrow S]$. Intuitively refinement guarantees that any behaviour exhibited by P may also be a possible behaviour described by the specification S. Here, $[\_]$ is the universal closure operator, namely $[P] \mathrel{\widehat{=}} \forall\, w \bullet P$ where w are the variables in the alphabet of P.

Operators include Skip ($I\!I_A$), the assignment $x :=_A e$ and the conditional $P \lhd b \rhd Q$. The subscript A of these operators is an alphabet that needs to be given as a parameter. Any construct in the UTP must specify the alphabet of the corresponding predicate; where the alphabet cannot be determined from the operand(s), it has to be explicitly provided.

Healthiness conditions embody facts about the computational models. For example, the theory of designs introduces additional boolean variables $okay$ and $okay'$, which record the respective observations of whether a program has started or fin-

ished. 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 condition $P = okay \Rightarrow P$. Healthiness conditions can be expressed using idempotent functions which turn a possibly unhealthy predicate into a healthy one; in our example, we have the function $H1(P) = okay \Rightarrow P$. The healthy predicates are exactly the fixed points of the healthiness function.

In the next section we explain how we encode aspects of the UTP that concisely capture the notion of alphabetised predicate and UTP theory.

3 Semantic Encoding of the UTP

As already said, the semantic model for an alphabetised predicate is a set of bindings describing the valuations that render it true. 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 tuple defined as follows.

$$
\begin{array}{l}
ALPHA\_PREDICATE \mathrel{\widehat=} \\
\quad \{bs : BINDINGS;\ u : UNIVERSE \mid \\
\qquad (\forall\, b : bs \bullet \mathrm{dom}\ b = Alphabet_U\ u) \land bs \subseteq u\}
\end{array}
$$

In this definition *BINDINGS* is the type of all binding sets *irrespective* of type constraints, and *UNIVERSE* the type of all subsets of bindings that are valid universes. A universe contains all the well-typed bindings, and in this way defines indirectly the types of all variables in the alphabet.

$$
\begin{array}{l}
UNIVERSE \mathrel{\widehat=} \{bs : BINDINGS \mid \emptyset \in bs \land \\
\quad (\forall\, b1 : bs;\ b : BINDING \mid b \subseteq b1 \bullet b \in bs) \land \\
\quad (\forall\, b1, b2 : bs \bullet b1 \oplus b2 \in bs\}
\end{array}
$$

A universe must be subset-closed, since every subset of a well-typed binding is well-typed, and closure under functional override ensures orthogonality, that is, type restrictions imposed on one variable cannot be sensitive to the values taken by other variables.

The constraints specified on the components of *ALPHA_PREDICATE* are first that the domain of each binding of the predicate has to be equal to the alphabet of the predicate's universe. The alphabet of a universe comprises the variables which the universe constraints and formally is defined by the union of the domains of all its constituting bindings: $Alphabet_U\ u \mathrel{\widehat=} \bigcup \{b : u \mid \mathrm{dom}\ b\}$. The first constraint ensures that the universe of a predicate does not contain information that is not relevant to its meaning. This could result in anomalies when combining predicates of different theories that have incompatible universes. The second constraint ensures that the bindings of a predicate respect the type constraints imposed by the universe.

3.1 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 functions.

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

Note that 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.

Healthiness conditions are elements of a type *HEALTH_COND* which contains all idempotent, partial functions from *ALPHA_PREDICATE* to *ALPHA_PREDICATE*.

$$
\begin{array}{|l}
\textit{HEALTH\_COND} : \mathbb{P}\,(\textit{ALPHA\_PREDICATE} \nrightarrow \textit{ALPHA\_PREDICATE}) \\
\hline
\forall\, h : \textit{HEALTH\_COND} \bullet h \fatsemi h = h \\
\end{array}
$$

Since some healthiness functions may only be sensibly defined for predicates with specific alphabets, we confine ourselves to *partial* functions. This potentially could raise an issue with undefinedness, therefore specific definitions of healthiness functions, as a general rule, always have to explicitly state the function's domain.

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.

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

$$
\begin{array}{|l}
\textit{InstPredTheory} : \textit{UNIVERSE} \rightarrow \textit{UTP\_THEORY} \\
\hline
\forall\, u : \textit{UNIVERSE} \bullet \\
\quad \textit{InstPredTheory}\ u = \textit{InstTheory}\,(u, \emptyset) \\
\end{array}
$$

$\textit{InstTheory}\,(u, hs)$ yields a *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; for this purpose we use the function $\textit{StrengthenTheory}\,(th, hs)$. 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 (specifications or programs that can be written as

pairs of preconditions and postconditions); its definition is based on an instantiation of a theory of relations.

$$InstDesTheory : DES\_UNIVERSE \to UTP\_THEORY$$

$$\forall u : DES\_UNIVERSE \bullet InstDesTheory\ u = StrengthenTheory\,(InstRelTheory\ u, \{H1, H2\})$$

$InstRelTheory\,(u)$ is itself defined by extending an instantiation of the theory of alphabetised predicates $InstPredTheory\,(u)$. The extension here does not introduce further healthiness conditions, but additional restrictions on the alphabet which for relational predicates must only contain undashed and dashed names. It also enforces dashed variables, if present, to have similar types to their corresponding undashed counterparts. By equating the domain of the instantiation function to $DES\_UNIVERSE$ we impose further restrictions requiring all design theories to incorporate the boolean variables $okay$ and $okay'$. The definition of the healthiness functions $H1$ and $H2$ for designs are omitted. With these instantiation functions, we can now define theory families: sets that contain all the theories of a particular kind, albeit with different universes. For example, families of design theories.

$$DES\_THEORY \mathrel{\widehat=} \{u : DES\_UNIVERSE \mid InstDesTheory\ u\}$$

Such sets effectively allow us to reason about the possible instantiations of design theories. We will make use of this to address laws and proof issues.

The approach we take here is not just an effort towards supporting dynamic instantiation *per se*, but a uniform presentation of UTP theories. Uniformity is important in terms of automation to facilitate the development of reusable laws and proof tactics. 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.

3.2 Theory Predicates

One of the 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 given UTP theory object are determined by the function $TheoryPredicates$.

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

$$\forall th : UTP\_THEORY \bullet TheoryPredicates\ th = \{p : ALPHA\_PREDICATE \mid p.2 = th.THEORY\_UNIVERSE \land (\forall h : th.HEALTH\_CONDS \bullet p \in \mathrm{dom}\ h \land h\,(p) = p)\}$$

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

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

Finally, it is possible to take an arbitrary predicate and apply the healthiness conditions e.g. of a UTP theory to obtain a healthy one with respect to that theory. The corresponding function is defined as follows.

$$ApplyHealthCond : (ALPHA\_PREDICATE \times \operatorname{seq} HEALTH\_COND) \pfun ALPHA\_PREDICATE$$

$$\forall\, p : ALPHA\_PREDICATE;\ hs : \operatorname{seq} HEALTH\_COND \bullet ApplyHealthCond\,(p, hs) = foldApp\,(hs, p)$$

The *foldApp* function, whose definition we do not include, successively applies all functions of a sequence given by its first argument to some value given by its second argument.

3.3 Encoding of Operators

It is sometimes necessary to make assumptions about the arguments of operators fulfilling certain provisos; 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 would only make sense in the context of particular UTP theories.

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 be predicates of specific theories. An example is the definition of the Skip operator for designs, which is different from the relational Skip $I\!I_R$.

$$I\!I_D : DES\_UNIVERSE \to DESIGN$$

$$\forall\, u : DES\_UNIVERSE \bullet I\!I_D\,(u) = True_P\,(u) \vdash_D I\!I_R\,(u)$$

Using total function definitions, which, as above, give a specific characterisation of their domain and range simplifies proofs of theorems involving its application: it factors the proof of properties of the range into the consistency proof of the function. The function $True_P\,(u)$ constructs the alphabetised predicate *true* over a given

universe, that is the predicate whose bindings are the ones of u having exactly the alphabet of u as their domain.

To illustrate how operators are used to encode specifications, we present the encoding of a program which nondeterministically chooses to toggle the value of a variable x from 0 to 1 (and vice versa) or not, namely $(x := 1 \lhd x = 0 \rhd x := 0) \sqcap \Pi_{\{x\}}$.

$$\begin{aligned} &ToggleOrNot \mathrel{\widehat=} \\ &\quad (Assign_R\,(u, \langle x\rangle, \langle 1\rangle) \lhd_R =_R (u, x, 0) \rhd_R Assign_R\,(u, \langle x\rangle, \langle 0\rangle)) \sqcap_R \Pi_R\,(u) \end{aligned}$$

The various semantic functions used here are '$=_R$', '$Assign_R$', '$\sqcap_R$', '$(\_ \lhd_R \_ \rhd_R \_)$' and 'Π_R' which are 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.

4 Reasoning about UTP Theories

Our semantic encoding enables us to reason about particular UTP theories and their specifications. In this section we will explore how this is done by giving a few examples.

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

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

This definition does not exclude the case where, for example, $x = okay$. To avoid it, we introduce a predicate on sequences of variables, $distinct\ s \Leftrightarrow s \in \mathrm{iseq}\ NAME$, and formalise this requirement as $distinct\ \langle x, x', okay, okay'\rangle$. Strictly, demanding $distinct\ \langle x, okay\rangle$ suffices since the $dash$ function is injective, and its domain and range are disjoint.

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

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

Discharging the existential consistency proof obligation for this definition establishes that the alphabet we provide is a valid alphabet for a theory of designs.

We are now required to provide a theory universe. The instantiation function for design theories obliges us to type the variables $okay$ and $okay'$ as boolean, however we can choose any type for x and x'. By reusing the definition $DES\_UNIVERSE$ 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.

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

No predicate is included for explicitly constraining the type of x' in the above definition. Such is redundant since the properties of *REL_UNIVERSE*, of which *DES_UNIVERSE* is a subset, ensure that the dashed counterparts of undecorated variables, if present, have identical types. Again, the consistency proof for *INST_UNIVERSE* establishes that there exists a universe with the desired properties; this effectively removes the eminent risk of inconsistencies when applying contradictory type constraints on the universes' variables.

The UTP theory is conveniently obtained by invoking an instantiation function.

$$INST\_THEORY \mathrel{\widehat=} InstDesTheory\ INST\_UNIVERSE$$

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

$$True_P\ INST\_UNIVERSE \in TheoryPredicates\ INST\_THEORY$$

that is, that the predicate *true*, or more accurately *true* of our particular alphabet and universe, is a healthy design predicate. The function $True_P$ that defines the predicate *true* takes the universe of the resulting predicate as a parameter.

When reasoning about predicates of UTP theories, however, we do not want to prove laws as the above for each instantiation; instead, we formulate more general laws that hold for *all* possible instantiations. A general law that we can prove is as follows.

$$\vdash \forall\, th : DES\_THEORY \bullet \\ True_P\,(th.THEORY\_UNIVERSE) \in TheoryPredicates\ th$$

To express this law more concisely, we provide an alternative definition which parameterises $True_P$ with a UTP theory. Conceptually, this allows us to speak of predicates such as *true*, $x := 1$, $y' = 2$, Π, and so on within specific theories. The following illustrates how this results in a more compact rendition of the above law.

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

This law is indeed not more complicated than a corresponding theorem would have been in the original treatment in [11], assuming that for its application the membership $th \in DES\_THEORY$ is trivially discharged by exploiting the actual definition

of *th* in terms of the respective instantiation function. In the original treatment we can analogously prove that for any alphabet $True_R\,(a)$ is a valid design, however we could not express predicate membership to a particular instance of design theory as above.

Similarly, closure theorems for theory operators have to be formulated in terms of the theory context in which they hold; as an example, the following proved law establishes that any theory of designs is closed under disjunction.

$$\vdash \forall\, th : DES\_THEORY \bullet \\ \forall\, p1, p2 : TheoryPredicates\ th \bullet p1 \vee_P p2 \in TheoryPredicates\ th$$

To apply this law we first have to establish that there is a member of the family of design theories to which the predicates $p1$ and $p2$ in question belong. This is *not* equivalent to saying that if $p1$ and $p2$ are elements of the set *DESIGN*, as defined in Section 3.2, so is $p1 \vee_P p2$. 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* design predicates of possibly different design theory instances we obtain a design predicate (of some design theory). This is not true due to the restrictions on alphabets and universes, and the associated compatibility requirements of operators.

Beyond proving laws about the predicates of specific UTP theories as shown above, it is also possible in our encoding to prove general laws about UTP theory instantiations. A very intuitive law is 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.

$$\vdash \forall\, th : UTP\_THEORY;\ hs : \mathbb{P}\ HEALTH\_COND \bullet \\ TheoryPredicates\,(StrengthenTheory\,(th, hs)) \subseteq TheoryPredicates\ th$$

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 general facts about UTP theories independently of particular instantiations. 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, [4] discusses theories in which the healthiness functions are expressed using 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 assume, for example, the existence of a predicate $CH(h)$ that tells us whether a healthiness function h is expressible in this way, the following theorem

$$\vdash \forall\, th : UTP\_THEORY \mid (\forall\, h : th.HEALTH\_CONDS \bullet CH(h)) \bullet \\ \forall\, p1, p2 : TheoryPredicates\ th \bullet p1 ;_R p2 \in TheoryPredicates\ th$$

asserts that the predicates of all such theories are closed under relational composi-

tion. The possibility of expressing such properties of classes of theories segregates our approach from the existing one, and adds to its expressive power.

5 Refinement Laws

To carry out formal verification, it is required to prove a given specification is refined by some implementation. Fundamentally, refinement is a property of alphabetised predicates that can be established independently to their membership to particular UTP theories. 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 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 proving steps. We consider, for example, the refinement

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

in the context of the design theory instantiation that was presented in Section 4. Rewriting it in terms of the underlying operator definitions yields the following.

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

To continue the proof at the mechanical layer, 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* with appropriate universe. This is feasible but not practical.

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

$$\vdash \forall\, th : \mathit{DES\_THEORY} \bullet \forall\, d1, d2 : \mathit{TheoryPredicates}\ th \bullet d1 \sqcap d2 \sqsubseteq d1$$

The only proviso for this law is that the two predicates have to belong to the same theory, otherwise the $\sqcap$ operator would not be well-defined, and the law could not be proved. 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 would oblige us to show that the predicates $x := 1$ and $x := 2$ belong to the predicates of $INST\_THEORY$.

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 given as argument to the assignment constructor indirectly follows from its definition. These extra proofs is 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 constructs, 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 $x := 1;\ x := 2 \in TheoryPredicates\ th$ for some $th : UTP\_THEORY$. The proof of properties like these cannot be sensibly captured by a single law; it is first necessary to prove that $x := 1 \in TheoryPredicates\ th$, then $x := 2 \in TheoryPredicates\ th$, 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, however their application requires further theorems, and importantly, high-level tactics for automation.

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

$$\vdash \forall\, th : DES\_THEORY \bullet \forall\, d1, d2 : TheoryPredicates\ th \bullet d1 \sqcap d2 = d2 \sqcap d1$$

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

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 were contemplating how these laws may be *applied* to reason about particular specifications.

6 Linking Theories

Theory links are functions mapping the predicates from one theory into (a subset of) the predicates from 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.

The linking functions often enjoy specific properties, for example, idempotence, monotonicity, or weakening and strengthening as described in [5]. These properties allow us to deduce further characteristics of the link and the predicates it describes. In our semantic encoding we can formalise links between theories by means of partial functions on the type *ALPHA_PREDICATE*. Consider, for example, the following link which maps a relational predicate to a (terminating) design.

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

We use the turnstile operator $P \vdash Q$ yielding a design predicate with precondition P and postcondition Q; it is defined as $okay \wedge P \Rightarrow okay' \wedge Q$.

A familiar theorem in the UTP states that the $H1$ and $H2$-healthy predicates are exactly those that can be written in the form $P \vdash Q$. We can formulate this law concisely in our treatment; first, assume the following ProofPower definition for $\vdash_D$ embedding the above operator into the semantics.

$$\_ \vdash_D \_ : DES\_COMPATIBLE \times DES\_COMPATIBLE \to DESIGN$$

$$\forall p, q : DES\_COMPATIBLE \bullet \\ p \vdash_D q = (OKAY \wedge_P p) \Rightarrow_P (OKAY' \wedge_P q)$$

Here, *OKAY* and *OKAY′* are constants which represent the predicates *okay* and *okay′*, respectively. *OKAY* is defined as follows, and the definition for *OKAY′* is analogous.

$$OKAY : DES\_COMPATIBLE$$

$$OKAY = =_P (Create_U (\langle okay \rangle, \langle BOOL\_VAL \rangle), okay, Val(Bool(true)))$$

DES_COMPATIBLE is the set of all alphabetised predicates whose universes are compatible with the typing restrictions on design predicates: if *okay* and *okay′* occur they have to be boolean. Opposed to that, *DESIGN* is the set of predicates belonging to some instantiation of a design theory as explained in Section 3.2. The function $Create_U\,(vs, ts)$ constructs a universe from a sequence of variables vs and a sequence of types ts.

We now can define the linking function L above as

$$L : DES\_COMPATIBLE \to ALPHA\_PREDICATE$$

$$\forall q : DES\_COMPATIBLE \bullet L(q) = True_P(\emptyset) \vdash_D q$$

and express the property that for every relational theory $th1 : REL\_THEORY$ 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.

$$\begin{array}{l} \vdash \forall\, th1 : REL\_THEORY \mid \\ \quad Compatible_U\,(th1.THEORY\_UNIVERSE, DES\_UNIVERSE) \bullet \\ \quad\quad \exists\, th2 : DES\_THEORY \bullet \\ \quad\quad\quad L\,(\!|\, TheoryPredicates\ th1 \,|\!) \subseteq TheoryPredicates\ th2 \end{array}$$

This theorem does not explicitly provide information regarding the alphabet and universe of the target theory $th2$, but it is possible to rephrase it in a way that both are captured explicitly i.e. by deriving them from $th1$.

Even more concretely, we can formulate and prove laws for particular theory instantiations $inst\_th1$ and $inst\_th2$ that have compatible universes, e.g. as defined in Section 4. Similarly, we can establish, for example, that

$$L\,(\!|\, TheoryPredicates\ inst\_th1 \,|\!) \subseteq (or =)\ TheoryPredicates\ inst\_th2$$

but other more elaborate properties are conceivable too.

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 satisfy ourselves with making a case of its feasibility, and give an indication of which properties may be expressible.

7 Discussion

It is clearly not possible to survey all aspects of the mechanised UTP semantics we presented, or justify all design decisions in detail. They are sometimes the result of following blind alleys, or realising through experimentation what appears to be the best compromise between generality and simplicity that allows us to carry out the reasoning we require.

Initially, our attempt was to limit alterations to the existing semantic encoding in [11] to a minimum to increase the chance of reusing the majority of the existing laws and proofs. This tight-rope walk unfortunately proved to fail, forcing us to open the 'Pandora's Box' by incorporating a proper notion of theory and instantiation. Consequently, a lot of the existing laws will have to be rephrased as discussed in Section 4, and it will besides be difficult to transfer existing mechanical proofs. On the positive side, this provides us with the opportunity to address and improve issues which might deserve further attention in existing work; we will discuss a few of them in this section.

A first problem is the one of consistency. In general, the axiomatic definitions of constants in ProofPower-Z are not consequently checked whether they might not introduce a contradiction. It is possible to enable and thus enforce such checks, however previous work did not exploit it. This indeed resulted 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, however previous work in fact used the exact definition $BINDING \mathrel{\widehat=} NAME \pfun VALUE$. It

is very 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 and, more disastrously, not realising.

Our aim is to establish consistency of *all* axiomatic definitions, and avoid the use of *a posteriori* constraints being imposed on existing variables as they are not checked; this has been taken into considerations when recasting the existing definitions and encoding. 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, as in [11]. Instead, we define a set *DES_UNIVERSE* which explicitly specifies the domain of the instantiation function *InstDesTheory* presented in Section 3.1. To apply *InstDesTheory* to some universe u, we have to prove that u introduces the correct type restrictions on the auxiliary variables. If this is not the case, 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 about its value. This is not, however, an inconsistency and does not raise the possibility of vacuous proofs.

A second problem is to tame the complexity introduced by formalising theories. It seems inevitable that we have to associate theories with the universe of bindings capturing the typing constraints of variables in the alphabet, but besides it proved essential to equip alphabetised predicates themselves with a universe in order to provide sufficient information to perform operators such as negation or substitution. These operators need to know about the types of variables; for example, negating $b = TRUE$ 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.

Our approach also permits to appropriately handle variable blocks in program specifications. Alphabetised predicates P that occur within some local variable declaration $\mathbf{var}\ v \,;\, P \,;\, \mathbf{end}\ v$ must have universes which introduce suitable type restrictions on v and v' for the construction to be well-defined. Since the variable block removes the corresponding variables from the alphabet of the predicate, P and $\mathbf{var}\ v \,;\, P \,;\, \mathbf{end}\ v$ effectively belong to different UTP theories. The theory of the variable block is obtained by contracting the universe of the theory which P belongs to; such *ad hoc* transformations of theories can be formalised and reasoned about in our encoding as well, and give rise to a collection of specialised laws. Furthermore, it is permissible to associate the same local variable name with different types, providing there is no interferences or name clashes; e.g. in $(\mathbf{var}\ v \,;\, P \,;\, \mathbf{end}\ v) \sqcap (\mathbf{var}\ v \,;\, Q \,;\, \mathbf{end}\ v)$ the predicates P and Q are free to individually specify different type restrictions for v.

Finally, our encoding enables us to prove the refinement conjecture presented as a motivating problem in the introduction, namely $x := x+1 \lhd x = 1 \rhd I\!I \ \sqsubseteq\ x := 2$. To do so we first define an alphabet containing the variables x and x', and a universe in which they range over the values in the set $\{1, 2\}$. Using the instantiation function

for relational theories, we then instantiate the corresponding UTP theory, let us call it *th*, as explained in Section 4. The programs can now be directly expressed as predicates of the instantiated theory and thereby acquire their alphabets and universes from *th*.

$$Assign_R\,(th, \langle x\rangle, \langle x+1\rangle) \lhd_R x =_R 1 \rhd_R \Pi_R\,(th) \sqsubseteq Assign_R\,(th, \langle x\rangle, \langle 2\rangle)$$

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 $\neg_R\,(x =_R 1)$ 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 $x =_R 2$ required to complete this branch of the proof as $\Pi_R\,(th)$ has no influence on the value of x.

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 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.

Our work can be regarded as a recast of Oliveira's encoding that on one hand allows us to formulate and discharge refinement conjectures for specifications and implementations within arbitrary UTP theory instantiations, and on another provides a higher level of confidence the semantic model is consistent and non-vacuous.

Related Work

Related work apart from Oliveira's semantic encoding is Nuka's formalisation of the alphabetised relation calculus [8,9]. This work explores the development of a mechanised semantic model for alphabetised predicates, and within it the definition of common UTP operators. Our semantic model shares conceptual similarities with Nuka's encoding such as representing relations through sets of bindings, however Nuka did not address the issue of typing or mechanise UTP theory instances.

ProofPower-Z has been successfully used for verification of safety-critical systems in the avionics industry [1], hence there is evidence for its suitability in an industrial context. Current approaches to verify implementations of control systems using tools that exploit ProofPower-Z translate the specification of the control system into a Z model, and use built-in reasoning support for Z to discharge refinement proof obligations — aided by custom, high-level tactics to automate proof

procedures [6]. Whereas Z models are mostly suitable to specify sequential programs, the UTP provides a semantic model for a much wider class of languages. Mechanisations of the UTP provide the basis for the construction of further methods and tools which take advantage of the efficiency, power and configurability of ProofPower-Z. Although *Circus*, reconciling elements from the sequential as well as concurrent programming, is our main focus for future work, it is not the only possible application of our encoding.

Future Work

Future work will investigate how the large collection of laws proved in Oliveira's original encoding can be transferred to our setting. The challenge of this is not merely to rephrase the laws as we already indicated in Section 4 and 5, but to find ways of adopting proofs as well; the existing proofs amount to approximately 80,000 lines of proof script, and it would be desirable to reduce the effort for their recreation. Our ProofPower-Z theories, including the definitions and proofs presented or discussed in this paper are available from

`http://www.cs.york.ac.uk/circus/tp/tools.html`.

Subsequently, experience needs to be gained with proving properties of particular specifications in UTP-based theories, and besides how such proofs can be automated. To this point we have only carried out experiments utilising toy examples. The wider objective of this investigation is however to use the encoding to do algebraic reasoning about *Circus* specifications and refinements. Automation of such reasoning will pose a particular challenge; a benchmark here is the ClawZ [1] system 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.

Acknowledgement

Valuable discussions have taken place with Marcel Oliveira regarding his original encoding, experience with proofs, and possible ways to overcome the inconsistency 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.

References

[1] Adams, M. M. and P. B. Clayton, *ClawZ: Cost-Effective Formal Verification for Control Systems*, in: K. Lau and R. Banach, editors, *ICFEM 2005: Formal Methods and Software Engineering*, Lecture Notes in Computer Science **3785** (2005), pp. 465–479.

[2] Butterfield, A., A. Sherif and J. C. P. Woodcock, *Slotted Circus: A UTP-family of reactive theories*, in: *International Conference on Formal Enginneering*, Lecture Notes in Computer Science (2007).

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

[4] Harwood, W., A. L. C. Cavalcanti and J. C. P. Woodcock, *A Model of Pointers for the Unifying Theories of Programming – Extended Version*, Technical report, University of York, Department of Computer Science, UK (2007), www-users.cs.york.ac.uk/~alcc/publications/HCW07.pdf.

[5] Hoare, C. A. R. and H. Jifeng, "Unifying Theories of Programming," Prentice Hall Series in Computer Science, Prentice Hall, 1998.

[6] Jones, R. B., *ICL ProofPower*, BCS FACS FACTS **Series III, 1** (1992), pp. 10–13.

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

[8] Nuka, G. and J. C. P. Woodcock, *Mechanising the Alphabetised Relational Calculus*, Electronic Notes in Theoretical Computer Science **95** (2004), pp. 209–225.

[9] Nuka, G. and J. C. P. Woodcock, *Mechanising a Unifying Theory*, in: *Unifying Theories of Programming, First International Symposium*, Lecture Notes in Computer Science **4010** (2006), pp. 217–235.

[10] Oliveira, M., A. Cavalcanti and J. C. P. Woodcock, *Unifying Theories in ProofPower-Z*, in: *Unifying Theories of Programming, First International Symposium*, Lecture Notes in Computer Science **4010** (2006), pp. 123–140.

[11] Oliveira, M., A. Cavalcanti and J. C. P. Woodcock, *A UTP semantics for Circus*, Formal Aspects of Computing **Online First** (2007).

[12] Sherif, A. and H. Jifeng, *Towards a time model for circus*, in: C. George and H. Miao, editors, *International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science **2495** (2002), pp. 613–624.

[13] Tarski, A., *On the Calculus of Relations*, Journal of Symbolic Logic **6** (1941), pp. 73–89.