Encoding *Circus* Programs in ProofPower-Z

Frank Zeyda and Ana Cavalcanti

Department of Computer Science, University of York, U.K.
{zeyda,ana}@cs.york.ac.uk

Abstract. *Circus* combines elements from sequential and reactive programming, and is especially suited for the development and verification of state-rich, reactive systems. In this paper we illustrate, by example, how a mechanisation of the UTP, and of a *Circus* theory, more specifically, can be used to encode particular *Circus* specifications. This complements previous work which focused on using the mechanised UTP semantics to prove general laws. We propose a number of extensions to an existing mechanisation by Oliveira to deal with the problems of type constraints and theory instantiation. We also show what the strategies and practical solutions are for proving refinement conjectures.

Key words: UTP, semantics, Z, theorem proving, refinement

1 Introduction

The *Circus* language combines elements from sequential programming and process algebra [4]. Its key notion is that of a process, which encapsulates state and behaviour, defined by actions that operate on the state and communicate with the environment. Actions may be specified as Z operation schemas, Dijkstra's guarded commands, or constructs of the CSP language. *Circus* is particularly suitable for reasoning about state-rich, reactive systems [13,6] using refinement.

In [16] Oliveira presents a semantics for *Circus* based on the UTP, and an extensive encoding of definitions and laws in the ProofPower-Z theorem prover. ProofPower is a versatile and powerful mechanical theorem prover based on HOL that has been successfully used in industry. ProofPower-Z is an extension of ProofPower that additionally embeds the theory of Z. The work involved the creation of a hierarchy of UTP theory encodings, namely for the theories of relations, designs, reactive designs, CSP, and, on top of the hierarchy, *Circus*. Each embedded UTP theory gives rise to a collection of axiomatic Z definitions, and a ProofPower theory is used in each case to hold the definitions.

The motivation for this work was primarily to prove equality and refinement laws which are *generally* valid within the various UTP theories. To this end it has been successfully employed in proving a large repository (≥ 500) of such laws. Little experience has, however, been gained so far in encoding and proving properties of *particular* *Circus* specifications and programs.

As a motivating example, we consider the process presented in Fig. 1 whose purpose is to compute and output the series of Fibonacci numbers on a channel

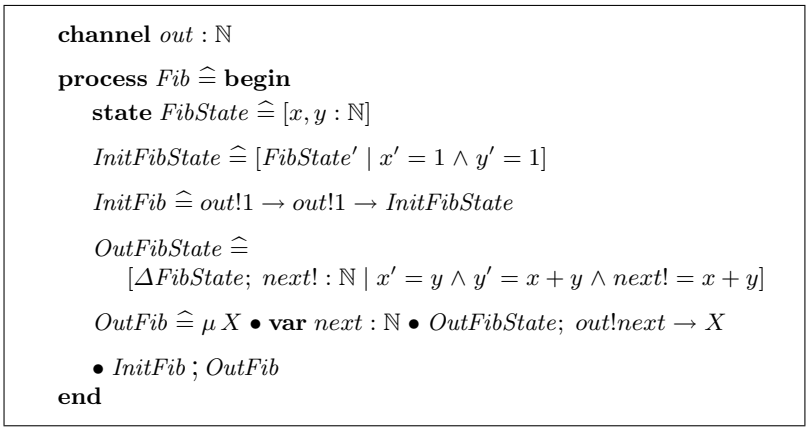

Fig. 1. Specification of the *Circus* process *Fib*.

out. After declaration of the channel in a **channel** paragraph, the process first declares its state components by means of a **state** process paragraph; here, they consist of the variables x and y both of integer type. This is followed by a sequence of actions: first to initialise the state of the process (*InitFibState* and *InitFib*), and further to calculate and communicate the next Fibonacci number (*OutFibState* and *OutFib*). A special action is the main action at the end, following '•', which defines the process behaviour.

Initialisation in *InitFibState* is specified by a Z operation that assigns 1 to the state components. *InitFib* outputs the first two Fibonacci numbers prior to initialising the state. *OutFibState*, again defined by a Z schema, computes the next Fibonacci number, stores it in the local variable *next*, and updates the state. The shriek is merely syntactic sugar for output variables in schemas, and when interpreting a schema as an action treated as referring to the after-state variable (here $next'$). The rôle of *OutFib* is then to invoke *OutFibState* and output the calculated number over *out*; it does so repetitively being defined using recursion. The variable block conceals the *next* and $next'$ components introduced by *OutFibState* making them local to the action. The main action first calls *InitFib*, and afterwards *OutFib*; it does not terminate.

The encoding of *Fib* in Oliveira's mechanisation of the UTP framework raises a few problems. One of them is that it does not support well the case where predicates of different UTP theories coexist in the same ProofPower scope of definitions. For example, the UTP characterisation of the actions of *Fib* are predicates that belong to a specific *family* of UTP theories that fulfil certain healthiness conditions (those of *Circus*), but at the same time have possibly different alphabets (of programming variables). An example is the variable block that defines *OutFib*, whose body is a predicate including *next* and $next'$ in its alphabet, whereas the resulting predicate does not have these variables in the alphabet. In addition, the predicates of Z operation schemas used in the definition of *Fib* actions belong to another family of UTP theories (namely that of relations) with no healthiness conditions, but possibly different variables.

To address this problem, we require a dynamic notion of UTP theory instantiation. In the original work of Oliveira, the closest we get is the by nature static inclusion of the corresponding ProofPower theory. This is problematic mostly since constraints imposed by such 'instantiations' apply globally, and thus affect *all other UTP theories in scope*. Typically, the theory of designs may require the auxiliary variables *okay* and $okay'$ to be of boolean type; however, such a constraint would in the existing treatment *a priori* affect instances of all other UTP theories, as, for example, the ones of plain predicative or relational theories. In the *Fib* process this problem equally arises if we assume the local variable *next* is reused somewhere else but with a different type. An illustrating example for this is the construct $(\mathbf{var}\ next \bullet next := 1 ; P) \sqcap (\mathbf{var}\ next \bullet next := true ; Q)$ in which the type of *next* differs in each branch of the choice operator, and for this reason cannot be statically fixed.

In this paper, we first present a revised mechanisation of the UTP *Circus* theory that deals with the problem of instantiation and local type constraints. We then show how the new framework can be used to encode *Circus* processes in a way that the problems mentioned above largely disappear. Finally, we discuss some practical aspects of refinement proofs.

The revisions that we propose follow the approach discussed and justified in detail in [18]. The agenda in this paper is mainly to view them in the light of the *Circus* theory, and apply them to the concrete encoding of a *Circus* specification such as *Fib*. We also address the concern of refinement proofs.

The structure of the paper is as follows. In Section 2 we present the extensions we propose to the original mechanisation of Oliveira. Section 3 explains the embedding of the UTP theory of *Circus* in our modified settings; Section 4 discusses the encoding of *Circus* processes using that embedding; and Section 5 addresses refinement proofs. In Section 6 we draw our conclusions.

2 Extended Mechanised UTP Semantics

In the UTP [9], the fundamental objects are alphabetised predicates representing observable behaviour. We represent them as a set of bindings (functions) that map variable names to values, and a universe, used to define type constraints.

z

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

BINDINGS is the set of all binding sets, and *UNIVERSE* the set of all binding sets which are valid universes. A universe contains all well-typed bindings and so determines the types of the variables in the bindings. Its definition is as follows.

z

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

The empty set of bindings is a valid universe. Additionally, universes have to be

subset closed, formalising our intuition that the binding resulting from restricting the domain of a well-typed binding remains well typed. Finally, the type of one variable cannot be sensitive to values taken by another variable, that is type restriction has to be orthogonal to binding (function) overriding.

In the specification of *ALPHA_PREDICATE* we state that the binding set has to be a subset of the universe in order to respect the type constraints imposed by it. Moreover, the universe must not retain information about types of variables outside the alphabet of the predicate. (The alphabet $Alphabet_U\ u$ of a universe u is the union of the domains of all its bindings.) This avoids anomalies when combining predicates with different universes. In general, this is only possible if the predicates agree on the types of their shared variables, and we do not want variables *irrelevant* to the predicate's meaning to cause clashes. In our model, alphabets are identified with universes; namely, we may conceptually think of universes as alphabets with additional type information attached to them.

A UTP theory is defined by a new schema type as follows.

z

UTP_THEORY

$$THEORY\_UNIVERSE : UNIVERSE;$$
$$HEALTH\_CONDS : \mathbb{P}\ HEALTH\_COND$$

UTP theory instances are too associated with universes: the universe of the predicates of the theory. The type *HEALTH_COND* comprises all partial idempotent functions on the set *ALPHA_PREDICATE*, and *HEALTH_CONDS* accordingly records the healthiness conditions of the theory. We have to restrict ourselves to partial functions here since some healthiness conditions may not be applicable to predicates with certain variables or type constraints. For example, applying $\mathbf{H1}(P) = okay \Rightarrow P$ is only sensible if the type of *okay* in P is boolean.

UTP_THEORY does not record the predicates of the theory. We can derive them from the universe and healthiness conditions using the function below.

z

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

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

As mentioned before, the predicates of a theory share its universe, and are the common fixed points of all the healthiness functions.

The instantiation of UTP theories is simply carried out by constructing a binding of *UTP_THEORY*. To achieve modularity we provide instantiation functions for every encoding of a UTP theory, for example *InstRelTheory u* for the plain theory of relations, *InstDesTheory u* for the theory of designs, and so on. The functions are solely parameterised in terms of a universe since the healthiness conditions for specific UTP theory families are usually fixed.

Our encoding provides further useful functions which allow for modular construction of a UTP theory hierarchy. For example $StrengthenTheory\,(th, hs)$ enriches an existing theory instance th with a set of additional healthiness conditions hs. The main benefit of constructing theories in such a manner is that proofs about lower-level predicates and operators can be easily reused in higher-level theories, and moreover interesting properties can be formulated regarding theory dependencies. It is also an approach we will adopt discussing the encoding of the UTP theory of *Circus* in the following section.

3 Semantic Embedding of *Circus*

Our encoding of *Circus* is in essence a recast of Oliveira's *Circus* encoding [16] that takes into account the alterations of the previous section. Our version of the ProofPower-Z theory for *Circus* acts solely as a carrier for the various definitions for instantiating *Circus* theories. When instantiating concrete UTP theories we pursue a uniform approach that suggests a certain *structure* in the definitions. It is mirrored by the order in which definitions are presented here.

We first define the sets $CIRCUS\_ALPHABET$ and $CIRCUS\_UNIVERSE$ of possible alphabets and universes of theory instances. Since they are similar to those for reactive designs, we equate them with $REA\_ALPHABET$ and $REA\_UNIVERSE$ — the corresponding sets for the theory of reactive processes.

The set $REA\_ALPHABET$ includes all alphabets that contain the auxiliary variables $okay$, $wait$, tr and ref, including their primed versions. The variables themselves are introduced as distinct elements of the type $NAME$ which represents variable names. Since $REA\_ALPHABET$ is a restriction of $REL\_ALPHABET$ (the type of relational alphabets), we only consider alphabets consisting of input (undashed) and output (single dashed) variables.

The set $REA\_UNIVERSE$ is defined as shown below.

z

$$\begin{array}{l} \boldsymbol{REA\_UNIVERSE} \mathrel{\widehat=} \\ \quad \{u : DES\_UNIVERSE \mid \\ \qquad Alphabet_U\ u \in REA\_ALPHABET \land \\ \qquad typeof\ (wait, u) = BOOL\_VAL \land \\ \qquad typeof\ (tr, u) = SEQ\_EVENT\_VAL \land \\ \qquad typeof\ (ref, u) = SET\_EVENT\_VAL\} \end{array}$$

The alphabet of the universe of an instance of a reactive process theory has to be in $REA\_ALPHABET$, and the types of the auxiliary variables are as we expect. To express the type constraints, we use the function $typeof$.

z

$$\begin{array}{l} \boldsymbol{typeof} : (NAME \times UNIVERSE) \to \mathbb{P}\ VALUE \\ \hline \forall\ n : NAME; u : UNIVERSE \bullet typeof\ (n, u) = \{b : u \mid n \in dom\ b \bullet b\ n\} \end{array}$$

It takes a variable name and a universe, and returns the type of the variable: the

set of values it can have in the respective universe.

In a $DES\_UNIVERSE$, *okay* has type $BOOL\_VAL$, so we do not need to constrain it. For the primed variables, as $DES\_UNIVERSE$ is a $REL\_UNIVERSE$, a constraint on relational universes ensures that dashed variables, if present, have similar types to their undashed counterparts.

In order to define the instantiation function for *Circus* theories, we need to encode their healthiness conditions. The UTP theory for *Circus* is a restriction of the theory of CSP requiring additional healthiness conditions *C1*, *C2* and *C3*, which we omit here. Assuming their encoding, the theory instantiation function yielding elements of UTP_THEORY is defined as follows.

z

$$\mathbf{InstCircusTheory} : CIRCUS\_UNIVERSE \rightarrow UTP\_THEORY$$

$$\forall\ u : CIRCUS\_UNIVERSE \bullet$$
$$InstCircusTheory\ u =$$
$$StrengthenTheory\ (InstCSPTheory\ u, \{C1, C2, C3\})$$

Instantiation is performed by strengthening a corresponding instance of the UTP theory of CSP. Instantiation is defined only if a suitable universe is provided; here, it must be one from the set $CIRCUS\_UNIVERSE$. The instantiation function easily allows us to define the set $CIRCUS\_THEORY$ containing all possible instantiations of *Circus* theories: it is the range of $InstCircusTheory$.

We can now define the subset of alphabetised predicates that characterise valid *Circus* actions and processes: all predicates that belong to some instantiation of a *Circus* theory. They satisfy the healthiness conditions of *Circus* and all subordinate UTP theories i.e. those for CSP, reactive processes, and relations.

z

$$\mathbf{CIRCUS\_ACTION} \mathrel{\widehat=}$$
$$\{p : ALPHA\_PREDICATE \mid$$
$$(\exists\ th : CIRCUS\_THEORY \bullet p \in TheoryPredicates\ th)\}$$

The semantics of a *Circus* process is given by hiding the state components in its main action. Therefore, models of processes are actions whose alphabets include *only* the auxiliary variables *okay*, *wait*, *tr* and *ref*, and their dashed counterparts. The definition is obtained by further restricting $CIRCUS\_ACTION$.

z

$$\mathbf{CIRCUS\_PROCESS} \mathrel{\widehat=}$$
$$\{p : CIRCUS\_ACTION \mid Alphabet_P\ p = ALPHABET\_OWTR\}$$

This concludes the presentation of the core definitions that support instantiation of *Circus* theories. In defining the operators on *Circus* actions, we reuse the definitions of [16], but adapt them to take into consideration universes where required. An example is the function that encodes *Skip*, which takes a *universe*

as a parameter: the universe of the state components of the process.

z

$$\mathbf{Skip_C} : WF\_Skip_C \rightarrow CIRCUS\_ACTION$$

$$\forall\ u : WF\_Skip_C \bullet$$
$$Skip_C\ u = R\ (True_P\ u \vdash_D TReqTR' \wedge_P (\neg_P\ WAIT')) \wedge_P \Pi_R\ u)$$

Skip is defined in terms of applying R, the healthiness condition for reactive processes, to a design that determines the behaviour of the action. The design has a true precondition $True_P\ u$, meaning that it never diverges. The postcondition specifies that *Skip* immediately terminates: the only observable behaviour is that *Skip* is not in an intermediate state ($\neg_P\ WAIT'$) while the trace is not altered ($TReqTR'$ encodes the predicate $tr = tr'$). The relational $Skip\ \Pi_R\ u$ on the state universe u ensures that the state variables are not changed. The rôle of WF_Skip_C is to restrict the domain of the function as to require homogeneous universes that must only mention state variables but not auxiliary ones.

A second example is the function $\rightarrow_C$ which encodes prefixing of *Circus* actions. Prefixing is used, for example, in the *InitFib* action of the *Fib* process.

z

$$\_ \rightarrow_C \_ : WF\_PREFIXING_C \rightarrow CIRCUS\_ACTION$$

$$\forall\ n : CHANNEL\_NAME;\ e : EXPRESSION;\ p : CIRCUS\_ACTION \mid$$
$$((n,\ e),\ p) \in WF\_PREFIXING_C \bullet$$
$$(n,\ e) \rightarrow_C p = R\ (True_P\ p.2 \vdash_D$$
$$(do\_C\ (p.2,\ n,\ e)) \wedge_P$$
$$(\Pi_R\ (p.2 \ominus_U ALPHABET\_OWTR))) ;_C p$$

Prefixing requires a channel name n, an expression e whose value is output on the channel, and the prefixed action predicate p. $WF\_PREFIXING_C$ captures the restriction on the arguments that the free variables in e must be contained in the universe of the predicate to ensures evaluation of e is well-defined. The operator is specified by sequentially composing a reactive process with p that carries out the communication and then terminates. This process as before is specified by applying R to a design. The true precondition of the design indicates again the absence of divergence, and the postcondition makes use of a function $do_C\ (u, n, e)$ being the encoding of the predicate

$$(tr = tr' \wedge (n, e) \notin ref') \lhd WAIT' \rhd tr' = tr \frown \langle (n, e) \rangle.$$

The reactive behaviour is thus to be initially ($tr = tr'$) in a waiting state refusing all events other than (n, e), and to terminate when the process has engaged in the communication event (n, e). The conjunction with *Skip* on the universe of the state components, obtained by removing the auxiliary variables from the universe of p, ensures that values of state variables remain unchanged.

The presence of universes results in many places in additional restrictions on the domains of semantic functions characterising the various UTP theory

operators of *Circus* and subordinate theories. In the lower-level theory of alphabetised predicates and relations (utp-alpha and utp-rel), binary operators such as conjunction, disjunction, and so on are defined for predicate pairs whose universes are compatible, but not necessarily the same. This enables us to combine predicates from different theories. In specific theories, we require the arguments of operators in most cases to be of the same theory instance. Consequently, we need new definitions describing such argument restrictions. For example, $WF\_CIRCUS\_ACTION\_PAIR$ is the set of all predicate pairs for which there exists a *Circus* theory to which both predicates belong. It is used, for instance, as the domain of action operators modelling internal and external choice.

The amendments to definitions were mainly motivated by the need to prove properties and laws, and, as a minimal requirement, to ensure well-definedness of the underlying function applications. We did not try and identify the strongest condition for sensibly applying operators, but one which is consistent without incurring too heavy a burden on proofs. This is justified by assuming that processes and actions are well typed. If proofs later require stronger restrictions, we will tighten the domain definitions appropriately.

Besides we specify operators in a way that allows us to infer easily that their application yields a predicate within the correct UTP theory: we restrict their range to $CIRCUS\_ACTION$. The proof of this is pushed into the consistency theorem for the definition (see Section 5), and closure properties follow trivially from operator definitions and need not be separately formulated as theorems.

4 Encoding of *Circus* Programs

In this section we illustrate how we use the semantic encoding of *Circus* described in the previous section to encode the *Fib* process given in Fig. 1. The ProofPower theory source for all definitions presented here and elsewhere in the paper can be downloaded from http://www.cs.york.ac.uk/circus/tp/tools.html.

To accommodate the ProofPower-Z definitions, we create a new ProofPower theory utp-fib as a child of utp-circus. We begin by creating definitions that introduce channel names, state components and local variables. For our example, we use an axiomatic definition to declare a name $out : CHANNEL\_NAME$ for the *out* channel. $CHANNEL\_NAME$ provides an inexhaustible supply of names (elements from the $NAME$ type) to represent channel identifiers. This set is disjoint from Z_VAR_NAME which contains names of Z schemas as well as local variables and state components. We declare all the names used in the process description to be of type Z_VAR_NAME.

z

$$\boldsymbol{x}, \boldsymbol{x'}, \boldsymbol{y}, \boldsymbol{y'} : Z\_VAR\_NAME$$

$$x' = dash\ x \land y' = dash\ y \land distinct\ \langle x, y \rangle$$

When introducing new variables, it is important to ensure they are mutually distinct from any existing ones that may be used in the same predicate. We

achieve this by virtue of a predicate *distinct* on sequences of names which holds true for all *injective* sequences. Since the *dash* function modelling decoration is injective too, it is sufficient to enforce uniqueness of the undashed names. By selecting names from Z_VAR_NAME we already ensure that they are distinct from any of the auxiliary variables for *Circus* actions. We introduce the local variables exactly in the same way as illustrated in the above definition.

Next, we instantiate the *Circus* theory for the main and auxiliary actions. To do so we define the universe of the main action. It is as follows in our example.

z

$$
\begin{array}{|l}
\boldsymbol{FIB\_UNIVERSE} : CIRCUS\_UNIVERSE \\
\hline
Alphabet_U\ FIB\_UNIVERSE = FIB\_ALPHABET \land \\
typeof\ (x,\ FIB\_UNIVERSE) = INT\_VAL \land \\
typeof\ (y,\ FIB\_UNIVERSE) = INT\_VAL
\end{array}
$$

$FIB\_ALPHABET$ is the set containing both the auxiliary and state variables. Since universes which are selected from $CIRCUS\_UNIVERSE$ 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 theories for the actions of *Fib*. It is not just one UTP theory that is used to model them, since actions like *OutFibState* mention extra variables apart from the state components. We need a theory instance for each possible *extension* of $FIB\_UNIVERSE$.

z

$$
\begin{array}{|l}
\boldsymbol{FIB\_THEORY} \mathrel{\widehat=} \\
\quad \{u : CIRCUS\_UNIVERSE \mid \\
\qquad Compatible_U\ (u,\ FIB\_UNIVERSE) \bullet InstCircusTheory\ u\}
\end{array}
$$

Intuitively, this definition constructs the family of all instances of the *Circus* theory that have a universe which *at least* contains the state components of *Fib*, and imposes the correct type constraints on them. We recall that two universes are compatible if they agree on the types of their shared variables.

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

z

$$
\boldsymbol{FIB\_ACTION} \mathrel{\widehat=} \bigcup (TheoryPredicates\ (\!|FIB\_THEORY|\!))
$$

It is simply the union of all predicates of UTP theories in FIB_THEORY. A more specific action of *Fib* is its main action, because it only has the auxiliary variables and state components in the universe. We include another definition FIB_MAIN_ACTION, comprising the potential predicates for main actions.

We now turn to encoding the actions actually used in *Fib*. We first look at the ones which are defined through Z operation schemas. Considering, for example, the action *InitFibState*, the corresponding schema $[FibState' \mid x' = 1 \land y' = 1]$ has to be lifted to become a *Circus* action, and a valid predicate of a *Circus* theory in *FIB_THEORY*. The schema itself is encoded by a relational predicate over the universe that contains its components x' and y' with the right type.

The semantic function $SchemaExp_C$ performs this 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.

z

$$\mathbf{Fib\_InitFibState\_VAR\_DECLS} \mathrel{\widehat=} (\langle x, y, x', y' \rangle, \langle INT\_TYPE, INT\_TYPE, INT\_TYPE, INT\_TYPE \rangle)$$

Types are represented as values (elements of *VALUE*). Here, *INT_TYPE* is the set of integer values. The encoding of *InitFibState* is as follows.

z

$$\mathbf{Fib\_InitFibState} : FIB\_ACTION$$

$$Fib\_InitFibState = SchemaExp_C\ (Fib\_InitFibState\_VAR\_DECLS, (=_P\ (Create_U\ \{x' \mapsto INT\_VAL\}, x', Val(Int(1)))) \land_P (=_P\ (Create_U\ \{y' \mapsto INT\_VAL\}, y', Val(Int(1)))))$$

In the above $=_P$ is the semantic function used to construct alphabetised predicates for equalities between variables and expressions. It needs to be provided with a universe, namely that of the resulting relation. For this purpose, $Create_U$ ad-hocly creates a universe from a set of name/type pairs.

Notably, the universe of the schema predicate has x' and y' in its alphabet, since $\land_P$ merges the universes of the constituent predicates. The universe of the schema itself comprises x, y, x', and y'. Finally, the predicate defined by $SchemaExp_C$ additionally includes in its universe the auxiliary variables and fulfils the healthiness conditions for *Circus* actions. This illustrates how predicates of different UTP theories coexist in the same definition.

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

The encoding of schemas that include extra components, besides those of the state, like *Fib_OutFibState* for instance, is similar. To simplify the encoding of the schema predicate, we define a universe *Fib_OutFibState_UNIV* (used by all three equalities) containing exactly the variables of the schema. A function *UnivFromVAR_DECLS* defines the conversion of a *VAR_DECL* to an element

of *UNIVERSE*; we omit its definition and that of *Fib_OutFibState_UNIV* itself.

z

$$\boldsymbol{Fib\_OutFibState} : FIB\_ACTION$$

$$Fib\_OutFibState = SchemaExp_C\ (Fib\_OutFibState\_VAR\_DECLS,\ (=_P\ (Fib\_OutFibState\_UNIV,\ x',\ Var(y)))\ \wedge_P\ (=_P\ (Fib\_OutFibState\_UNIV,\ y',\ Fun_2((\_ +_V \_),\ Var(x),\ Var(y))))\ \wedge_P\ (=_P\ (Fib\_OutFibState\_UNIV,\ next',\ Fun_2((\_ +_V \_),\ Var(x),\ Var(y)))))$$

Terms such as $Var(y)$ and $Fun_2((\_ +_V \_), Var(x), Var(y))$ encode expressions in the semantics, here y and $x+y$. 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. The same applies to input variables decorated with a question mark should they occur.

Not all encoded actions are required to be equipped with an action-specific universe. Examples of actions that do not require a universe are *InitFib* and *OutFib*; the encoding of the latter is given below.

z

$$\boldsymbol{Fib\_OutFib} : FIB\_ACTION$$

$$Fib\_OutFib = \mu_C\ (\lambda\ X : CIRCUS\_ACTION \bullet var_C\ (next,\ Fib\_OutFibState\ ;_C\ (out,\ Var(next)) \rightarrow_C X))$$

Here, the universe of the sequential composition is inferred from its arguments, *Fib_OutFibState* and $out!next \rightarrow X$. This case again illustrates how predicates of different UTP theories coexist in the same ProofPower definitional scope. The body of the variable block is an action whose universe includes the extra variables *next* and $next'$, which are concealed by the var_C construct. Hence the universe of *Fib_OutFib* only comprises the auxiliary variables and the state variables.

Crucially, we could indeed have another variable block declaring *next* within the same predicate but *with a different type*. In the original work [15] such would not have been possible since type constraints are *globally* attached to variable names. Above, the types of *next* and $next'$ are deduced from the universe of the underlying relation of the body of the variable block. Thus the association of types and variable names takes place upon construction of the predicate and, in fact, is local with respect to the encoded predicate, and thus not static.

z

$$\boldsymbol{Fib\_MainAction} : FIB\_MAIN\_ACTION$$

$$Fib\_MainAction = Fib\_InitFib\ ;_C\ Fib\_OutFib$$

Again this action does not require a universe as it is inferred by $;_C$ from those of *Fib_InitFib* and *Fib_OutFib*. Since *Fib_MainAction* is declared to be an element

of *FIB_MAIN_ACTION*, that universe has to be *FIB_UNIVERSE*, though.

Fib_MainAction does not truly characterise the process as such since it still contains the state components in its universe. Since these are local to the process they should be hidden it its semantic description. This is achieved by the operator $begin_C \setminus end_C$. Utilising it we obtain the following definition for the process *Fib*.

z

$$\boldsymbol{Fib\_Process} : CIRCUS\_PROCESS$$

$$Fib\_Process = begin_C \; Fib\_MainAction \; end_C$$

The set *CIRCUS_PROCESS* defined in utp-circus contains all predicates of the *Circus* theory obtained by instantiation with a minimal universe; this is the universe which only comprises auxiliary variables, but no state components.

In this section we have demonstrated how particular *Circus* processes can be encoded using our embedding of the UTP theory of *Circus*. It is possible to automate all steps involved, and that is the next step in our work agenda. The encoding requires that type information is deduced prior to translation and consequently exploited in the construction of universes; this can be easily achieved using the *Circus* type checker [17,7]. In the next section we will examine how properties of the encoded process may be formulated and proved.

5 Reasoning about *Circus* Specifications

In our investigation so far, we have considered two primary possibilities in which mechanical reasoning about *Circus* processes may be exploited. First, the encoding strategy which was informally presented in the previous section gives rise to a number of consistency proof obligations that establish the soundness of the encoding. ProofPower-Z is capable of generating the proof obligations automatically, and more importantly prevents axiomatic definitions from being unconditionally used unless their respective consistency theorem has been discharged. The second possibility is to carry out refinement proofs of actions and processes. We address these two opportunities separately in this section.

5.1 Soundness of Process Encodings

Most of our encoding is based on functions that are defined using Z axiomatic descriptions. In general, an axiomatic description introducing a new global constant *DefName* is of the following form, where S is a set and P a predicate.

z

$$DefName : S$$

$$P(DefName)$$

The notation $P(DefName)$ highlights that *DefName* is assumed to be free in P.

Consistency proofs ensure that there exists some element $DefName \in S$ for which the predicate $P(DefName)$ holds. If this is not true, we would be able to conclude *false* from the axiom of the definition allowing us to prove anything. The corresponding consistency proof obligation hence establishes that $\exists\, DefName : T \bullet DefName \in S \land P$. In Z, arbitrary sets S can be used in declarations (of constants), but their types do not include any constraints embodied in these sets. For example, if we declare a function f of type $A \pfun B$, where A and B are given sets, then the type of f is $\mathbb{P}(A \times B)$; the functional property is a constraint on f, rather than part of its type. A process called normalisation provides axiomatic descriptions whose declarations define types, and all constraints are given in the predicate. Assuming that the axiomatic description initially is not normalised, T, instead of S, is the actual type of *DefName* after normalisation. The given proof obligation is not exactly how ProofPower expresses the consistency goal when first generated. For brevity we omit the less concise ProofPower goal since it can be easily reduced to this one.

All encoded actions of a process are of the general form below.

z

$$\begin{array}{|l} ActionName : PROC\_ACTION \\ \hline \\ ActionName = ActionExpr \end{array}$$

ActionName is the name of the action. $PROC\_ACTION$ is the set of predicates of the instances of the UTP *Circus* theory that have a universe compatible with the state components declaration; for *Fib*, this is FIB_ACTION. Finally, *ActionExpr* is the alphabetised predicate that models the action defined by applying the functions for *Circus* operators in our encoding. The consistency proof obligations for action encodings are, therefore, always of the following form.

$$\begin{array}{l} \exists\, ActionName : \mathbb{P}\,(NAME \leftrightarrow VALUE) \times \mathbb{P}\,(NAME \leftrightarrow VALUE) \bullet \\ \quad ActionName \in PROC\_ACTION \land ActionName = ActionExpr \end{array}$$

Proving this subgoal is achieved by providing an existential witness for the quantified variable *ActionName*. From the shape of the predicate it is apparent that there can only be one choice of witness that possibly render it true; that is precisely *ActionExpr* — the right hand side of the action definition. Using *ActionExpr* as a witness, the consistency proof reduces to merely showing that $ActionExpr \in PROC\_ACTION$ which verifies the encoded action belongs to the set of valid actions for the process; it thus is guaranteed to fulfil the healthiness conditions of *Circus*, and possesses a permissible alphabet that includes the state components and imposes the correct type constraints on them.

To give a concrete example, we consider the consistency proof for *InitFib*.

z

$$\begin{array}{|l} \textbf{\textit{Fib\_InitFib}} : FIB\_ACTION \\ \hline \\ Fib\_InitFib = (out,\ Val(Int(1))) \rightarrow_C (out,\ Val(Int(1))) \rightarrow_C Fib\_InitFibState \end{array}$$

As previously explained, $(c, e) \rightarrow_C p$ encodes the prefixing operator outputting

the value of e on channel c, and $Val(Int(1))$ simply encodes the expression 1. *Fib_InitFibState* refers to the encoding of the *InitFibState* action presented in Section 4. The proof obligation which hence needs to be discharged is

$$(out,\ Val(Int(1))) \rightarrow_C (out,\ Val(Int(1))) \rightarrow_C Fib\_InitFibState \in FIB\_ACTION$$

We establish the truth of this goal in essence by exploiting closure properties of the operators. The base case $Fib\_InitFibState \in FIB\_ACTION$ we, notably, get from its definition as it follows from $Fib\_InitFibState : FIB\_ACTION$ in the declaration of the encoding of the *InitFibState* schema action. It is verified by the consistency proof of that definition.

In general, the suggested proof strategy relying on closure laws is symptomatic for *any* consistency proof that arises from the action encodings. (Up to the point where we have to show $ActionExpr \in FIB\_ACTION$ all steps are very easily automated.) The core part of the proof requires nevertheless more sophisticated mechanisms (but we claim that it can be automated too!).

More specifically, after unfolding the definition of FIB_ACTION, another witness needs to be provided to supply the *universe* of the *Circus* theory of which the action is deemed to be a member. $FIB\_UNIVERSE$ can be directly used in this instance since *InitFib* does not include any extra variables. If it did, typing information can be used to determine the right universe. The proof reduces then to showing that the given predicate belongs to the set of predicates of the *Circus* theory $InstCircusTheory\ FIB\_UNIVERSE$. The general closure of *Circus* operators establishes that the defined action is a member of *some* *Circus* theory instance, and properties of universes of the applied operators establish that it is exactly the theory under consideration.

As a concluding remark, it is worth noting that we are not constrained to use exclusively *Circus* operators in defining actions. For example, we may use plain predicative constructs instead if we desire, reflecting the unified view of any computation being a predicate in the UTP. An advantage of the discussed approach is that soundness is established *irrespective* of the specific way of representing actions, but automation of the proof may only be feasible if closure theorems are available for the underlying operators. If not, the alternative approach is to show membership to the theory of *Circus* by explicitly proving the predicate is a fixed point of the healthiness functions; due to the complexity of this proof in particular for *Circus* theory instances this normally requires human interaction.

5.2 Action and Process Refinement

Refinement is uniformly characterised by (universal) reverse implication in the UTP; consequently it is a property of alphabetised predicates and therefore can be established independently of any particular UTP theory membership. A simple proof approach involves unfolding theory-specific operators in terms of their underlying lower-level relational and predicative operator definitions.

Although in principle feasible, this is not a practical approach which lends itself easily for proof automation, in particular if we deal with more complex operator definitions such as parallelism or interleaving of *Circus* actions. To manage

the complexity of refinement proofs involving *Circus* action and process predicates, we have pursued two alternative approaches.

The first approach is to formulate and prove a collection of algebraic refinement and equality laws applying in situations where the predicates are of a certain form. In practice, the provisos that need to be established for application of the laws are (a) memberships to some *Circus* theory, and (b) other restrictions guarding the application of the law which are usually syntactic. The purpose of the provisos in (a) is to guarantee that applications of functions for theory-specific operators are well defined. For example, the following law establishes distribution of *Circus* guarded actions through conjunctions of their guards.

$$\begin{array}{l} \vdash \forall\ g1,\ g2 : CIRCUS\_CONDITION;\ p : CIRCUS\_ACTION \mid \\ \quad (g1,\ p) \in WF\_Guard_C \land (g2,\ p) \in WF\_Guard_C \bullet \\ \quad\quad (g1 \lor_P g2)\ \&_C\ p = (g1\ \&_C\ p) \ominus_C (g2\ \&_C\ p) \end{array}$$

WF_Guard_C is the domain of the function that encodes the guarded action construct, namely $(g\ \&_C\ p)$; it is restricted to guards and actions on the same universe. The symbol $\ominus_C$ denotes external choice.

If we apply laws like the above to actions a of Fib, we obtain the proof obligation $a \in CIRCUS\_ACTION$ from the stronger condition $a \in FIB\_ACTION$; the definition of the actions allows us to discharge such proof obligations directly. If, however, we apply laws to sub-expressions, membership to $CIRCUS\_ACTION$ has to be shown and depends on the particular sub-expression.

The alphabetised predicates of a UTP theory are not characterised syntactically, but by the healthiness conditions of the corresponding theory, and indeed we do not embed the syntax of the operators in our encoding. Therefore, there is no generic theorem that can be formulated to establish membership of arbitrary predicates to particular theories based on their syntax. Instead, we tackle this problem using specialised, high-level recursive tactics. The tactics selectively apply the closure theorems for the various *Circus* operators, and then proceed recursively on the generated subgoals. Similarly, the definition of laws often require the universes of the involved predicates to be compatible, giving rise to proof obligations asserting compatibility of the underlying universes.

Refinement laws can be equality laws as the above, or genuine refinements. ProofPower facilitates the rewriting of terms through the application of equality laws using its in-built rewrite and conversion mechanisms. On the other hand, we also want to be able to replace sub-expressions of a predicate if the law is not an identity but genuine refinement. In this case, however, we have to justify the application of the law by monotonicity of operators with respect to refinement. This, once again, is a process which cannot be encapsulated by a single theorem but needs to be performed by high-level tactics, guided by the structure of *Circus* actions. Monotonicity also gives rise to a second approach to establish action refinement which in particular exploits the monotonicity of R, the healthiness function for reactive designs.

The underlying idea is to express both actions of a refinement $A_1 \sqsubseteq A_2$ as applications of R, so that the proof reduces to $R(P_1 \vdash Q_1) \sqsubseteq R(P_2 \vdash Q_2)$ which, because of monotonicity, is implied by $P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2$, and in turn can

be reduced to $P_1 \Rightarrow P_2$ and $(P_1 \wedge Q_2) \Rightarrow Q_1$. The semantic definition of the *Circus* operators supports this approach by expressing most operators in terms of applying R to some design. The uniformity fosters automation.

To illustrate this approach in the context of *Fib*, we consider the refinement $Fib\_InitFibState \sqsubseteq Fib\_InitFibState\_Ref$ where $Fib\_InitFibState\_Ref$ is the assignment $x, y := 1, 1$. Its encoding is shown below.

z

$$\begin{array}{|l} \boldsymbol{Fib\_InitFibState\_Ref} : FIB\_ACTION \\ \hline \\ Fib\_InitFibState\_Ref = \\ \quad Assign_C\ (FIB\_UNIVERSE,\ \langle x,\ y\rangle,\ \langle Val(Int(1)),\ Val(Int(1))\rangle) \end{array}$$

After rewriting the definitions of $Fib\_InitFibState$ and $Fib\_InitFibState\_Ref$, the initial refinement goal is expressed as follows.

ProofPower Output

$$\begin{array}{|l} (*\ ?\vdash\ *)\ {}^{\llbracket}_{\mathrm{Z}}\!SchemaExp_C\ (Fib\_InitFibState\_VAR\_DECLS, \\ \quad (=_P\ (Fib\_InitFibState\_UNIV,\ x',\ Val\ (Int\ 1))) \wedge_P \\ \quad (=_P\ (Fib\_InitFibState\_UNIV,\ y',\ Val\ (Int\ 1)))) \\ \qquad \sqsubseteq\ Assign_C\ (FIB\_UNIVERSE,\ \langle x,\ y\rangle,\ \langle Val\ (Int\ 1),\ Val\ (Int\ 1)\rangle)^{\urcorner} \end{array}$$

Unfolding the semantic functions for *Circus* operators then yields the following.

ProofPower Output

$$\begin{array}{|l} {}^{\llbracket}_{\mathrm{Z}}(*\ ?\vdash\ *)\ R\ (\exists_P\ (ran\ Fib\_InitFibState\_VAR\_DECLS.1 \cap dashed, \\ \qquad (=_P\ (Fib\_InitFibState\_UNIV,\ x',\ Val\ (Int\ 1)) \wedge_P \\ \qquad\ =_P\ (Fib\_InitFibState\_UNIV,\ y',\ Val\ (Int\ 1))) \oplus_P \ldots) \\ \qquad\quad \vdash_D \\ \qquad ((=_P\ (Fib\_InitFibState\_UNIV,\ x',\ Val\ (Int\ 1)) \wedge_P \\ \qquad\ =_P\ (Fib\_InitFibState\_UNIV,\ y',\ Val\ (Int\ 1))) \oplus_P \ldots) \wedge_P \\ \qquad\quad TReqTR' \wedge_P \neg_P\ WAIT' \wedge_P \Pi_R\ (\ldots)\ )) \\ \qquad\quad \sqsubseteq \\ \quad R\ (True_P\ FIB\_UNIVERSE \\ \qquad \vdash_D\ Assign_R\ (FIB\_UNIVERSE,\ \langle x,\ y\rangle,\ \langle Val\ (Int\ 1),\ Val\ (Int\ 1)\rangle) \wedge_P \\ \qquad\quad TReqTR' \wedge_P \neg_P\ WAIT')^{\urcorner} \end{array}$$

Upon closer inspection we see that both sides of the refinement were rewritten into expressions of the form $R(P \vdash_D Q)$. The precondition of the first design originates from calculating the precondition of the corresponding schema, hence the existential quantification over the dashed variables of the schema corresponding to $ran\ Fib\_InitFibState\_VAR\_DECLS.1 \cap dashed$. The postcondition is simply the predicate of the schema with some additional conjuncts to correctly render the behaviour of the defined reactive process.

The formulas in place of the ellipses have been omitted; their purpose is merely to make some adjustments in order to homogenise universes in cases where the universe of the schema predicate is non-homogeneous. The right hand of the refinement corresponds to the definition of *Circus* assignment; the underly-

ing design has a *true* precondition since assignment always terminates, and the postcondition conjoins the relational assignment with the predicates $tr = tr'$ and $\neg\ wait'$, again to appropriately establish the reactive behaviour.

A sketch of the proof first verifies that the precondition of the second design is $True_P\ FIB\_UNIVERSE$. The fact that the preconditions have different universes does not compromise the proof since the universes are compatible. In general, we use laws within the lower-level theories of relations and plain predicates, or otherwise unfold the operators further into the underlying semantic model of alphabetised predicates. Regarding the postcondition, the approach is similar. Here, this requires some rewriting of the $Assign_R$ operator which results in unfolding it into a predicate resembling the postcondition of the first design. A minor simplification is proving that $\Pi_R\ (...)$ above has no effect.

Process Refinement Process refinement is simply a special case of action refinement where the involved (process) actions only contain auxiliary variables, hence we do not need a special treatment here. An alternative way of establishing process refinement is by reducing it to action refinement of the respective main actions providing their state variables are disjoint; this can be formulated as a theorem and effectively exploited in proofs.

In summary there are at least three different conceptual approaches towards proving *Circus* action refinement which operate on different semantic levels, and vary in terms of the effort that has to be invested. The most convenient is, not surprisingly, to work at the most abstract level — that is the level of high-level algebraic laws. Whether this is possible in specific cases depends on how specialised the conjecture is. The difference between previous work, which encoded *Circus* mostly for the sake of proving general laws, and our present work is that we cannot consider proofs as static entities that have to be established once and for all. Instead we need to provide generic means that automate all aspects of a proof not requiring human interaction so that the user may solely focus on those aspects which are difficult or beyond automation.

6 Conclusions

We have illustrated, by example, how a modified version of Oliveira's mechanisation of the UTP, including its embedding of the *Circus* language can be used to encode particular *Circus* specifications. The encoding is uniform and transparent, and automatically produces consistency proof obligations which guarantee the soundness of encoded programs on a *per case* basis. Although we did not present a formal translation strategy, the principles we outlined are indeed generalisable to semantically encode arbitrary *Circus* specifications using our extension of the mechanised *Circus* semantics. We have also discussed issues regarding the refinement proof of actions and processes, in particular in the light of automation. The latter is important to affirm feasibility for the development of scalable techniques and industrial tools using our extended mechanisation to verify the correctness of realistic, safety-critical systems. A good example of this is the ClawZ system [1] which has been successfully used in the formal verification of non-trivial

control systems in the avionics sector.

To solve the problem of predicates from different UTP theories being present in the same ProofPower theory scope (or even in the same definition), we formalised and integrated the notions of UTP theory and typing universes in the semantic model. Revisiting one of the motivating examples given in the introduction, the following predicate

$$(\mathbf{var}\ next \bullet next := 1 \,;\, P) \sqcap (\mathbf{var}\ next \bullet next := true \,;\, Q)$$

declares different types for *next* in the branches of the choice, namely $\mathbb{N}$ and $\mathbb{B}$. Whereas in the original work such predicates could not be represented since the type of *next* would have to be statically (and globally) identified with either $\mathbb{N}$ or $\mathbb{B}$, our encoding of the predicate creates a suitable universe for each of the bodies of the variable blocks in which the type of *next* is dynamically bound. The type information for *next* is erased by the **var** blocks, which hide *next* by contracting the universes; thus no clashes arise in the overall encoding.

An alternative approach to solve the above problem is to eliminate conflicting uses of variables with clashing types through renaming. This would, however, still require the facility to constrain the type of the variable. A main restriction of the original encoding is the inability to take into account typing constraints in the general theory of relations. Constraints introduced *a posteriori* are, unlike our definitions, not automatically checked in ProofPower-Z for consistency. In addition, elimination of naming clashes would complicate the translation of *Circus* syntax into its semantic characterisation making it more susceptible to errors, and produce less readable and tractable encodings. Another challenge with this approach is to ensure uniqueness of names across separate translations in order to be able to combine them without interference on a semantic level. Even more generally, we could think of a shallow embedding of alphabetised predicates themselves. This, however, could be problematic because they do not naturally map to predicates of the host logic (HOL) carrying *more* information, namely, an alphabet and associated types.

What became clear though in attempting proofs for particular refinements is that even for simple conjectures as the one considered in Section 5, the theorems incidentally become very large to a point where they are not manageable anymore. Our experience suggests that it is crucial to interleave certain simplification steps for alphabets and universes with steps performing a deeper unfolding (rewriting) of functions representing operators. The simplifications can in many cases eliminate operator invocations exploiting certain theorems. As an example, the alphabet of a conjunction may be rewritten as the union of the alphabets of the conjuncts. We are currently experimenting with the development of simplification tactics for automating the rewrite of semantic functions.

Related Work Closely related work is Nuka's mechanisation of the alphabetised relation calculus and UTP [11,12]. It presents a deep semantic embedding of alphabetised predicates and core operators mechanised in ProofPower-Z. Nuka's semantic model shares commonalities with ours in that predicates are represented as sets of bindings which themselves are partial functions from names to values. It mostly differs in that no type information is attached to predicates. The

lack of type information prevents it, for example, from proving type-dependant properties such as $\neg\,(okay = TRUE) \Leftrightarrow okay = FALSE$[1]. Furthermore, to our knowledge Nuka's embedding has not been used so far in proving properties of particular UTP specifications; doing so would be interesting, in particular to investigate possible ramifications of its untyped view.

In [3,2] Camilleri reports on a mechanisation of the CSP traces and failure-divergence model in HOL. Its primary focus is on proving standard CSP laws that are valid within the two semantic models, and another concern is to deeply encode the syntax of CSP. In these publications, however, similarly no account is given on how the mechanisation performs in proving particular CSP process refinements. An alternative embedding of the CSP traces model into PVS is presented in [5] where the authors additionally illustrate its application in verifying robustness properties of an authentication protocol. In doing so they realise the need for specific proof tactics (strategies in PVS) to conduct the proof at a more abstract level, and the scope for proof automation via tactics driven by the structure of proof goals. This coincides with our experience.

In [8] Groves et al. report on a tactic-driven tool implemented in Prolog that aids in performing program derivation in Morgan's refinement calculus [10]. The tool is illustrated by applying it to the example of a simple sorting algorithm. Interestingly, the authors postulate a hierarchy of refinement tactics which categorises them into "derived rules", "goal-directed rules" and high-level "strategies", corresponding to different levels of automation at which subsequent refinements are constructed. We currently do not consider the automation of refinement proofs at a comparably high level, but the experience gained in this work could ultimately be useful when tackling similar goals in the future.

Future Work Future work will focus on two primary aspects. First, the translation of *Circus* processes such as *Fib* into the semantic encoding shall be automated. There is no fundamental reason why this may not be possible, however the automation would need to type check the specification in order to infer the necessary information to construct universes of predicates and actions where needed. In [13] a step into this direction is made by defining a set of formal translation rules. The aim will be to extend and recast these in the light of the informal strategy we propose, and thereby formalise the translation.

A second important area for future work is the development of tools assisting refinement proofs. We already explained the usefulness of powerful ProofPower tactics for this purpose, but in certain cases a proof mostly consists of the application of high-level algebraic refinement laws. In this case, the ArcAngel*C* tactic language provides a more abstract and expressive notation for specifying tactics for *Circus* refinements [14]. It is a tactic language that supports backtracking through angelic choice, and is in this aspect superior to ProofPower's tactic language which does not entail backtracking. We have already developed a prototype implementation of ArcAngel*C* in ProofPower-Z giving some encouraging results; subsequent publications will report on this work.

[1] Note that this does not invalidate the law of the excluded middle in the relational algebra of Nuka's encoding; $okay = TRUE \vee \neg\,(okay = TRUE)$ is still provable.

Acknowledgements We would like to thank Marcel Oliveira for useful discussions and feedback on our revisions to his original mechanisation of the UTP, as well as the anonymous referees for their suggestions. We also is acknowledge EPSRC for funding this work under research grant EP/E025366/1.

References

1. M. Adams and P. Clayton. ClawZ: Cost-Effective Formal Verification of Control Systems. In *Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 465–479. Springer, October 2005.
2. A. Camilleri. A Higher Order Logic Mechanisation of the CSP Failure-Divergence Semantics. Technical Report HPL-90-194, HP Laboratories, September 1990.
3. A. Camilleri. Mechanizing csp trace theory in higher order logic. *IEEE Transactions on Software Engeneering*, 16(9):993–1004, 1990.
4. A. Cavalcanti, A. Sampaio, and J. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2–3):146–181, November 2003.
5. B. Dutertre and Steve Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136. Springer, August 1997.
6. L. Freitas, A. Cavalcanti, and Woodcock J. Taking Our Own Medicine: Applying the Refinement Calculus to State-Rich Refinement Model Checking. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006*, volume 4260 of *Lecture Notes in Computer Science*, pages 697–716. Springer, November 2006.
7. L. Freitas, J. Woodcock, and A. Cavalcanti. An Architecture for *Circus* Tools. In *SBMF 2007: Brazilian Symp. on Formal Methods*, pages 6–21, August 2007.
8. L. Groves, R. Nickson, and M. Utting. A Tactic Driven Refinement Tool. In *5th Refinement Workshop*, pages 272–297. Springer, January 1992.
9. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall, February 1998.
10. C. Morgan. *Programming from Specifications*. Prentice-Hall International Series In Computer Science. Prentice Hall, 1998.
11. G. Nuka and J. Woodcock. Mechanising the Alphabetised Relational Calculus. *Electronic Notes in Theoretical Computer Science*, 95:209–225, May 2004.
12. 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.
13. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science, University of York, UK, 2005.
14. M. Oliveira and A. Cavalcanti. ArcAngelC: a Refinement Tactic Language for *Circus*. *Electronic Notes in Theoretical Computer Science*, 214:203–229, June 2008.
15. M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying Theories in ProofPower-Z. In *Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*, pages 123–140, February 2006.
16. M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for *Circus*. *Formal Aspects of Computing*, Online First, 2007.
17. M. Xavier, A. Cavalcanti, and A. Sampaio. Type Checking *Circus* Specifications. In *SBMF 2006: Brazilian Symposium on Formal Methods*, pages 105–120, 2006.
18. F. Zeyda and A. Cavalcanti. Mechanical Reasoning about Families of UTP Theories. In *SBMF 2008: Brazilian Symp. on Formal Methods*, pages 145–160, 2008.