Automating Refinement of *Circus* Programs

Frank Zeyda and Ana Cavalcanti

University of York, Heslington, York, YO10 5DD, U.K.
{zeyda,ana}@cs.york.ac.uk

Abstract. In previous work, we have presented a mechanisation of *Circus* for the theorem prover ProofPower-Z. *Circus* is a refinement language for state-rich reactive systems that combines Z and CSP. In this paper, we present techniques to automate the discharge of proof obligations typically generated by the *Circus* refinement laws. They eliminate most of the proofs that are imposed by the fact that the encoding has to be precise about typing and well-definedness issues, and leave just those that are expected in a pen-and-paper refinement. This allows us to concentrate on the proof of properties that are significant for the problem at hand, while benefiting from the increased assurance and efficiency afforded by the use of a theorem prover as well as high-level tactic languages for refinement. Our case study is a refinement strategy for verification of control systems; we present the result of several experiments.

Key words: theorem proving, tactics, ArcAngel*C*, ProofPower

1 Introduction

Circus [4] is a process algebra that captures state as well as behavioural aspects of a system. Its key concept is that of a *Circus* process, which, like a CSP [15] process, communicates with the environment via channels, but also aggregates local state that can be internally accessed and modified. A *Circus* process specification contains a Z schema that specifies its state, and a list of dependent local actions used by the process, of which a designated action, the main action, defines the process behaviour. Actions may be specified using a mixture of CSP constructs, Z operation schemas [16], and guarded commands of Dijkstra and Morgan [5,8]. Processes can also be combined using CSP constructs.

A notable feature of the *Circus* language is its formal semantics and associated refinement calculus [4]. This permits the derivation or verification of executable programs. The flexibility of the language to handle both sequential behaviour and parallelism in a unified way makes it especially suitable for the description and formal derivation of state-rich concurrent systems [9].

A semantic embedding of *Circus* for ProofPower-Z, a theorem prover based on HOL that also supports Z, was first presented in [12]. We have extended that encoding to handle types and *Circus* programs (rather than the *Circus* semantics [19]). We have also developed a ProofPower implementation of ArcAngel*C* [18]. This is a refinement-tactic language to formulate strategies to automate the derivation and verification of programs from *Circus* specifications.

A general issue with the mechanisation is the large number of provisos raised by the application of refinement laws. They are mostly conditions that we would not deal with in a pen-and-paper proof. For example, the associativity law $a_1 \; ; \; (a_2 \; ; \; a_3) \; \equiv \; (a_1 \; ; \; a_2) \; ; \; a_3$ for actions does not require any provisos to be discharged when applied in a pen-and-paper proof. This is not so in its encoding in the semantic embedding of *Circus*, which is given below.

$$\begin{array}{l} \vdash \forall a_1, a_2, a_3 : \mathit{CIRCUS\_ACTION} \mid \\ \quad \alpha \; a_1 = \alpha \; a_2 \land \alpha \; a_2 = \alpha \; a_3 \bullet a_1 \; ;_C \; (a_2 \; ;_C \; a_3) \; \equiv \; (a_1 \; ;_C \; a_2) \; ;_C \; a_3 \end{array}$$

First, we have to introduce the provisos that a_1, a_2 and a_3 belong to the set $\mathit{CIRCUS\_ACTION}$, the semantic domain for actions. Secondly, two provisos are needed to ensure that all actions have the same alphabet. The α operator gives the alphabet of an action: the variables on which it operates.

These provisos establish well-formedness constraints that are typically taken for granted given the syntactic and type correctness of the actions. A mechanised model, however, forces us to make them explicit to establish the necessary assumptions for provability of the laws. Each *Circus* operator, like sequential composition of actions above, is encoded by a semantic function [12], and these functions are total on $\mathit{CIRCUS\_ACTION}$, but only partial with respect to the corresponding maximal set. This gives rise to non-trivial constraints, since $\mathit{CIRCUS\_ACTION}$, in our model, is not a type in the sense of being maximal.

In a deep semantic embedding, it is possible to formalise well-formedness as a property of the program syntax. (The difference between a shallow and deep embedding is that the latter also formalises the syntax of the embedded language.) We can then prove that (syntactic) well-formedness implies membership to the aforementioned (semantic) domains. Additionally, a collection of laws may be used to deduce well-formedness of compound expressions from the well-formedness of its components, or conversely, deduce well-formedness of components from the well-formedness of the expression as a whole.

There are, however, important reasons for not pursuing a deep approach in our mechanisation of the UTP and *Circus* to encode alphabetised predicates. The most crucial one is that the language of the UTP is not static: in higher-level theories new operators may be introduced that effectively become part of the syntax. In a deep embedding such extensions would be difficult to handle.

In combination with our implementation of ArcAngel*C* and more complex refinement tactics, the problem of provisos is exacerbated by the fact that tactic application relies on so-called 'model theorems', monotonicity of operators, and reflexivity and transitivity of refinement. Those theorems are frequently applied as part of many of the core tactics and suffer from the same problem of introducing conceptually superfluous provisos. Executing the tactics, the provisos quickly accumulate and result in theorems that become unmanageable.

Our contribution is a novel treatment of the notion of well-formedness at the semantic level. It allows us to eliminate most of the inherent provisos in law applications with minimal incisions to the embedding. To take advantage of that, we have made changes to the implementation of ArcAngel*C* as given in [18]; we also discuss those alterations here. We evaluate our new technique by encoding tactics that perform part of a refinement strategy for control laws [3].

Section 2 introduces the relevant preliminary material: core aspects of our semantic embedding of *Circus*, ArcAngel*C*, and its implementation. Section 3 explains our solution to managing well-definedness, and Section 4 discusses the accompanying extensions to the ArcAngel*C* implementation. Section 5 reports on a case study, and in Section 6 we present our conclusions and future work.

2 Preliminaries

In this section, we first present relevant details of our mechanisation of *Circus*. The mathematical notation we use is standard Z [16], but the ideas in Section 3 and Section 4 also exploit the higher-order support afforded by ProofPower being based on HOL. We secondly briefly introduce ArcAngel*C* and its implementation.

2.1 Mechanisation of *Circus*

The mechanisation of *Circus* is based on its denotational semantic model [12], which is formulated in terms of the Unifying Theories of Programming (UTP) [6]. The UTP is a general framework in which the semantics of a variety of modelling and programming languages can be uniformly expressed. It is founded on a relational calculus like Tarski's, but presented in a predicative style.

In the UTP, relations represent computational behaviours. To encode them, we use *alphabetised predicates*, which are predicates equipped with an alphabet of variables. The predicate $x' = x + 1 \lor x' = x - 1$, for example, encodes the computation that either increments or decrements the value of x; its alphabet includes x and x'. We use undecorated names to denote initial observations of the value of a variable, and dashed names to denote subsequent ones. In the UTP theory for *Circus*, alphabetised predicates describe actions and processes.

In our mechanisation, an alphabetised predicate is a pair.

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

Its first component is a set of bindings (records) describing the valuations of the variables of the alphabet that render it true. The second component records the types of the alphabet variables. $BINDINGS$ is the type of all binding sets, and $UNIVERSE$ consists of all partial functions from VAR to $TYPE$, where VAR is the semantic domain for variables, and $TYPE$ ($\mathrel{\widehat=} \mathbb{P}_1\ VALUE$) contains all non-empty sets of values. The condition $bs \subseteq Bindings_U\ u$ effectively establishes that the bindings of a predicate have to be well typed. (The function $Bindings_U$ constructs the complete set of bindings according to a given typing universe.)

Alphabetised predicates are embedded shallowly in the mechanisation: we characterise their semantics, but do not formalise the syntax of the UTP and the *Circus* language. Instead we provide a collection of semantic functions that correspond to the various syntactic constructs.

We define in [19] operators that provide the logical connectives, equality, substitution, including all *Circus* constructs, and importantly refinement. The latter allows us to state that some (concrete) program P behaves according to its (abstract) specification S. Formally, this is expressed by $S \sqsubseteq P$, and given the mechanisation we can, by aid of a collection of refinement laws, prove whether a given refinement holds. To automate this process, we have implemented a bespoke tactic language ArcAngel*C*; it is explained in the next section.

2.2 ArcAngelC

ArcAngel*C* is a tactic language for the derivation of *Circus* programs from specifications [10]. (Hereafter, we will use the word 'program' synonymously for both specifications and programs.) A salient feature of ArcAngel*C* is first that it supports backtracking through angelic choice, namely from failure of tactic applications. This it inherits from its kin Angel [7], which is more generally concerned with proving arbitrary goals. Secondly, it has a formal semantics that has been specified in the Z notation, and permits the reasoning about tactics.

Basic literal tactics provided by the ArcAngel*C* are **skip**, which leaves the program unchanged, **fail**, which always fails, and **abort**, whose application is not guaranteed to terminate. To apply refinement laws, the tactic **law** name$(args)$ takes the name of the law and a list of arguments. Compound tactics may be declared using the **Tactic** name $(args) \mathrel{\widehat=} body$ **end** construct.

We further have the binary tacticals $t_1 ; t_2$ for sequence and $t_1 \mid t_2$ for alternation. Sequence executes the tactics one after another, and alternation first attempts to apply t_1, and if that fails, applies t_2. Other tactics are $!\,t$ which acts like a cut on the backtracking search of finding a successful path of tactic execution, and **applies to** p **do** t which guards the application of a tactic t by successful matching of the program against a pattern p. Finally, recursive tactics are supported via the fixed-point operator $\mu X \bullet t(X)$.

To apply tactics to the operands of a *Circus* action or process construct, a set of structural combinators is provided. They are boxed versions of the respective *Circus* operators. For example, the combinator $t_1 \boxed{\sqcap} t_2$ applies to actions of the form $a_1 \sqcap a_2$. Its behaviour is to apply t_1 to a_1 and t_2 to a_2. The application is justified by the monotonicity of *Circus* operators with respect to refinement.

We have implemented ArcAngel*C* in ProofPower [18]. The fundamental design of the implementation supports tactics as theorem-generating functions that apply to program expressions A and return lists of refinement theorems $\Gamma \vdash A \sqsubseteq B$. The construction of refinement theorems is necessarily sound because of the LCF approach that prevents invalid theorems from being derived. More specifically, ProofPower-Z uses the type system of the prover's implementation language (ML) to differentiate between (unproved) conjecture and (proved) theorems.

3 Managing well-definedness

Most of the laws of our *Circus* semantic encoding specify well-definedness provisos for their applicability. The provisos ensure that relational expressions, such as $p_1 \sqsubseteq p_2$, as well as operator applications, like $p_1 \sqcap_C p_2$, are well-defined, that is, the underlying semantic functions are applied inside their domain.

As already said, these well-definedness constraints give rise to proof obligations that accumulate through the application of ArcAngel$C$ tactics. Sources for the provisos are model theorems automatically applied in the mechanics of the ArcAngel$C$ implementation, monotonicity theorems applied when invoking structural combinators, and user-defined laws. In practice, even after applying just the first three of the seven phases of the refinement tactic NB for control laws [13], the resulting theorem already includes more than 130 assumptions. Thereafter it becomes unmanageable, slowing down or even bringing to a stall further application of tactics. To tackle this problem we have pursued a combination of two approaches. They are explained separately in the following sections.

3.1 Reducing constraints in the semantic encoding

To tame the complexity of generated assumptions, we have used a novel treatment of typing . This reduced the number of provisos but has retained soundness.

By way of illustration, the semantic function for $\wedge$ is defined as follows.

$$
\begin{array}{|l}
(\_ \wedge_P \_) : \mathit{WF\_ALPHA\_PREDICATE\_PAIR} \rightarrow \mathit{ALPHA\_PREDICATE} \\
\hline
\forall p_1, p_2 : \mathit{ALPHA\_PREDICATE} \mid \\
\quad (p_1, p_2) \in \mathit{WF\_ALPHA\_PREDICATE\_PAIR} \bullet p_1 \wedge_P p_2 = (t_B, t_U)
\end{array}
$$

Here, membership of (p_1, p_2) to $\mathit{WF\_ALPHA\_PREDICATE\_PAIR}$ encapsulates an additional constraint for the compatibility of the universes of p_1 and p_2. The terms t_B and t_U respectively abbreviate the binding set and universe of the result; their particular shape is not relevant here and for brevity is omitted.

Opposed to this, in HOL, the types of variables are part of their identity, meaning that in a term like $n \geq 1 \wedge n = \mathbf{false}$ (that may result from conjoining $\Gamma_1 \vdash n \geq 1$ and $\Gamma_2 \vdash n = \mathbf{false}$), we are effectively talking about two different variables n distinguished by their types. We adopt a similar approach by moving type information from the universe directly into the entities representing names, which are now bindings of the following schema type.

$$\mathit{VAR} \mathrel{\widehat{=}} [\mathit{name} : \mathit{STRING};\ \mathit{dashes} : \mathbb{N};\ \mathit{subscript} : \mathit{SUBSCRIPT};\ \mathit{type} : \mathit{TYPE}]$$

The type STRING represents character sequences, and $\mathit{SUBSCRIPT}$ is a free type for representing a possible subscript. Importantly, the *type* component records the type of the variable, and TYPE is equated with the non-empty subsets of VALUE, that is, $\mathit{TYPE} \mathrel{\widehat{=}} \mathbb{P}_1\ \mathit{VALUE}$. In comparison, in the previous model, VAR only recorded a unique identifier (name), dashes, and a subscript.

This implies that the previous notion of 'universe' is subsumed by the conventional notion of an alphabet, which now implicitly carry type information. The encoding we obtain is in fact very similar to that originally developed by Oliveira [12], but with the added benefit of concisely capturing type information. The constraint $bs \subseteq \mathit{Bindings}_U\ u$ is notably redundant now in the definition of $\mathit{ALPHA\_PREDICATE}$. We can also drop additional constraints in definitions that require compatibility of types. Thus, the definition of $\wedge_P$ becomes

$$\begin{array}{|l} (\_ \wedge_P \_) : \mathit{ALPHA\_PREDICATE} \times \mathit{ALPHA\_PREDICATE} \rightarrow \ldots \\ \hline \forall p_1, p_2 : \mathit{ALPHA\_PREDICATE} \bullet p_1 \wedge_P p_2 = (t_B, t_A) \end{array}$$

Most other definitions of our encoding can be simplified in a similar manner. The notion of compatibility becomes obsolete since the common variables of alphabetised predicates necessarily have the same type.

This enhancement also displayed benefits in terms of modularising proofs. First, reasoning about alphabets (which are sets) is logically simpler than reasoning about universes (which are functions). Secondly, it is now possible to introduce a notion of well-typed expressions independently of the universe context; this makes provisos related to evaluation of expressions simpler.

We, however, still have many provisos like $p \in \mathit{ALPHA\_PREDICATE}$ or $p \in \mathit{CIRCUS\_ACTIONS}$. The following section explains how we deal with them.

3.2 A semantic formalisation of well-formedness

Our approach to capture well-formedness at the semantic level gives us the same benefits as a syntactic characterisation of well-formedness in a deep embedding, and importantly does not compromise soundness. To formalise well-formedness, we introduce a generic HOL function wd of type $'a \rightarrow \mathit{BOOL}$ where $'a$ is a type variable. (Hereafter we use the term 'well-defined' in favour of 'well-formed'.) Initially we do not specify any properties of wd, but for each semantic operator op we add an axiomatic constraint of the following form.

$$\vdash wd(op(x_1, \ldots, x_n)) \Leftrightarrow wd\ x_1 \wedge \ldots \wedge wd\ x_n \wedge (x_1, \ldots, x_n) \in \operatorname{dom} op$$

The proposition $wd(op(x_1, \ldots, x_n))$ entails that op is applied in its domain, and that all arguments are well-defined. Domain membership enables us to extract properties of the arguments; for instance $wd(p_1 \wedge_P p_2)$ implies that $p_1 \in \mathit{ALPHA\_PREDICATE}$. Moreover, for complex program terms, the property of well-defined arguments allows us to extract well-definedness of any subterm. For example, $wd(p_1 \wedge_P (p_2 \wedge_P p_3))$ implies $wd\ p_1$, $wd\ p_2$ and $wd\ p3$.

A potential risk with this approach is due to the definition of wd not being obviously conservative. Generally in logic, conservative extensions maintain consistency of the extended theory, and in many case this can be established by the mere shape of the defining axiom. What we are doing, however, in defining wd is in fact treating some Z function application $op(x_1, \ldots, x_n)$ *as if* it revealed information whether the function was applied in its domain. It is not immediately evident if and under what conditions such a treatment is sound.

To prove consistency, we provide a model which fulfils the defining axioms. The essence of our model is the identification of values outside the domain and range of each operator with undefinedness, and axioms that constrain applications of operators outside their domains to be closed under this set. This is possible because the underlying HOL logic is based on total functions, and therefore any term denotes a value. Specifically, $f(x)$ denotes a value even when $x \notin \operatorname{dom} f$, and constraining this value by an axiom such as $\vdash f(c) = v$ for particular c and v neither impinges on f's domain nor produces unsoundness if $c \notin \operatorname{dom} f$. This is a property of the embedding of Z partial functions into HOL.

To illustrate the conceptual idea of the model, we consider, for example, the set $ALPHA\_PREDICATE$. To capture undefinedness for functions encoding alphabetised predicate operators, we introduce the following set.

$$\perp_{ALPHA\_PREDICATE} \;\widehat{=}\; \mathbb{U} \setminus ALPHA\_PREDICATE$$

It is simply the complement of $ALPHA\_PREDICATE$ with respect to its corresponding maximal type. In ProofPower-Z, $\mathbb{U}$ acts as the carrier set of a generic type which is inferred by the type checker, and which is always maximal. Here, it would be the set $\mathbb{P}((\mathbb{P}\,VAR) \times \mathbb{P}(VAR \leftrightarrow VALUE))$.

For $\wedge_P$, for example, we have the supplementary axiom below.

$$\vdash \forall\, p_1, p_2 : \mathbb{U} \mid (p_1, p_2) \notin \operatorname{dom}(\_ \wedge_P \_) \bullet p_1 \wedge_P p_2 \in \perp_{ALPHA\_PREDICATE}$$

It does not affect (relative) consistency because none of the original definitions impose any constraints on function applications outside their domains.

Finally, we can give a conservative definition for wd if applied to elements of type $ALPHA\_PREDICATE$: $wd\ p \Leftrightarrow p \notin \perp_{ALPHA\_PREDICATE}$. The definition provably satisfies the axiom for $wd(p_1 \wedge_P p_2)$ as it has been specified before, and thereby establishes the correctness of the model, which itself is sound.

This model in a way simulates a treatment of undefinedness. It is correct if the types are 'large enough' so that we can always find a witness that serves to distinguish defined from undefined function applications.

There are cases where a collection of semantic types T_1, $T_2, \ldots$ have the same maximal type T_{max}. For example, the semantic domain $CIRCUS\_ACTION$ for actions is a subset of $ALPHA\_PREDICATE$. In those cases, a single set $\perp_T$ is defined as $T_{max} \setminus \bigcup T_i$ to ensure that $\perp_T$ is disjoint from all sets T_i. In such a situation, the union $\bigcup T_i$ needs to be a proper subset of T_{max}, so that there is some $x \in T_{max}$ for which $\forall\, i \bullet x \notin T_i$. This implies that $\perp_T \neq \varnothing$, which is crucial to prove that the model satisfies the axioms for wd.

Accordingly, there are functions in our encoding to which we cannot apply wd. These are first the operators involving the ProofPower-Z $\mathbb{B}$ type, which is maximal. Since refinement is defined by a function with range $\mathbb{B}$, we cannot specify $wd(p_1 \sqsubseteq p_2) \Leftrightarrow wd\ p_1 \wedge wd\ p_2 \wedge (p_1, p_2) \in \operatorname{dom}(\_ \sqsubseteq \_)$. The impact of this restriction is on provisos that involve refinements themselves. The extension of our technique to handle such cases is left as future work.

Additionally, the domains of the functions that encode the various operators on values is $VALUE$, which is also maximal. This prevents us from specifying a well-definedness axiom, for example, for $Eval(b, e)$, which evaluates an expression under a binding and yields an element of $VALUE$. To handle expressions, we axiomatise wd slightly differently; we define wd inductively over the free type $EXPRESSION$ that encodes the syntax of expressions. (Unlike alphabetised predicates, they are embedded deeply.) We introduce a set $WT\_EXPRESSION$ containing the expressions that are well-defined, that is $\{e : EXPRESSION \mid wd\ e\}$. Since, the functions that encode *Circus* operators involving expressions are parameterised in terms of $WT\_EXPRESSION$, their domains are not maximal, and so we can give wd axioms for them.

We do not actually define our model for wd in ProofPower-Z, as it would unnecessarily complicate the various definitions of operators. The main point is that we can introduce wd capturing the well-formedness of terms purely in semantic terms, and given the above caveats this definition is sound.

It is now possible to specify laws that either exploit or prove wd theorems. In particular, the associativity law for conjunction is now expressible as below.

$$\begin{array}{l}\forall\, p_1, p_2, p_3 : \mathbb{U} \mid wd((p_1 \wedge_P p_2) \wedge_P p_3) \bullet \\ \quad (p_1 \wedge_P p_2) \wedge_P p_3 \equiv p_1 \wedge_P (p_2 \wedge_P p_3) \;\wedge\; wd(p_1 \wedge_P (p_2 \wedge_P p_3))\end{array}$$

This directly mirrors the intuition that if the left-hand side $p_1 \wedge_P (p_2 \wedge_P p_3)$ is well-defined, the equivalence holds and also the right-hand side $(p_1 \wedge_P p_2) \wedge_P p_3$ is well-defined. Its only non-trivial proviso is the assumption of well-definedness of the initial program. We mechanically proved the above law without much effort by utilising its original version and rewriting applications of wd.

This illustrates how wd can provide a framework in which we can handle the emerging proof obligations for typing with theorems that establish that well-formedness is preserved. We work in a setting similar to that of pen-and-paper refinement proofs, where we normally assume that the initial program is well-formed, as are the programs of each law. If additionally the arguments of parameterised laws are well-formed, we conclude that all programs in the derivation chain must be well-formed. In summary, we work under assumptions that mean that we do not need to worry about issues of well-formedness.

In the following section we explain how the implementation of ArcAngel*C* has been amended to make use of theorems that possess this shape.

4 Extensions to the ArcAngelC implementation

We present here how we have extended the ArcAngel implementation to take advantage of the wd function.

4.1 Extended refinement theorems

Refinement theorems in our implementation of ArcAngel*C* had to be of the form $\Gamma \vdash A \sqsubseteq B$, where Γ is a list of proof obligations, and A and B are program expressions. To integrate well-definedness constraints, we have generalised the permissible shape of such theorems to $\Gamma, wd\ A \vdash A \sqsubseteq B \wedge wd\ B$. We call them *extended refinement theorems*. Their conclusions are conjunctions in which the first conjunct provides the actual refinement, and the second conjunct establishes well-definedness of the result B of the program transformation. We also have an assumption that asserts well-definedness of the initial program A.

The implementation has been adapted to handle these kinds of theorems. Importantly, the ArcAngel*C* mechanics has been adjusted as to only retain the wd

theorem of the initial program, and, as much as possible, discard intermediate wd provisos via incremental proofs. We also importantly preserve the general shape of an extended refinement theorem during tactic applications. To illustrate this, we consider the application of a refinement law. In the previous encoding these laws were typically of the form

$$\vdash \forall v_1 : T_1;\ \ldots;\ v_n : T_n \mid P[v_1, \ldots, v_n] \bullet A[v_1, \ldots, v_n] \sqsubseteq B[v_1, \ldots, v_n]$$

with type provisos $v_1 : T_1$, $v_2 : T_2$, and so on. The notation $A[v_1, \ldots, v_n]$ is used to emphasise that the variables v_i are free in A. We now rephrase such laws as

$$\begin{aligned}&\vdash \forall v_1 : \mathbb{U};\ \ldots;\ v_n : \mathbb{U} \mid wd\ A[v_1, \ldots, v_n] \wedge \\ &\qquad P[v_1, \ldots, v_n] \bullet A[v_1, \ldots, v_n] \sqsubseteq B[v_1, \ldots, v_n] \wedge wd\ B[v_1, \ldots, v_n]\ .\end{aligned}$$

Here, $\mathbb{U}$ is the carrier set of a type that is dynamically inferred by the type checker. It is maximal, and so only incurs trivial proof obligations. This new version of the law can be proved from the former by rewriting the wd function on both sides and extracting the type provisos. For example, if A is $v_1 \wedge_P v_2$ we have that $wd(v_1 \wedge_P v_2)$ implies that both v_1 and v_2 belong to $ALPHA\_PREDICATE$.

We observe that all type provisos have been absorbed into just one assertion $wd\ A[v_1, \ldots, v_n]$ that establishes well-definedness of the left-hand program. If we apply the new law to an extended refinement theorem $\Gamma, wd\ X \vdash X \sqsubseteq Y \wedge wd\ Y$, the right-hand program Y is matched against the left-hand program A of the law to instantiate the quantified variables. We then obtain an instantiation $P', wd\ Y \vdash Y \sqsubseteq Y' \wedge wd\ Y'$ after moving antecedents to the hypotheses.

The original extended refinement theorem and the instantiated law give rise to the following four theorems (after eliminating the conjunction in the conclusions): (1) $\Gamma, wd\ X \vdash X \sqsubseteq Y$; (2) $\Gamma, wd\ X \vdash wd\ Y$; (3) $P', wd\ Y \vdash Y \sqsubseteq Y'$; (4) $P', wd\ Y \vdash wd\ Y'$. We use (1), (3) and (4) to obtain

$$(5)\ \ \Gamma, P', wd\ X, wd\ Y \vdash X \sqsubseteq Y' \wedge wd\ Y'$$

by transitivity of refinement using (1) and (3), and conjunction using (4). The transitivity model theorem is now recast as

$$\vdash wd\ p_1 \wedge wd\ p_2 \wedge wd\ p_3 \bullet p_1 \sqsubseteq p_2 \wedge p_2 \sqsubseteq p_3 \Rightarrow p_1 \sqsubseteq p_3\ .$$

It is formulated in terms of wd theorems rather than type provisos.

The intermediate theorem (5) contains the surplus assumption $wd\ Y$, and our goal is to eliminate it. This can be achieved by the cut rule together with (2). The cut rule states that from $\Gamma_1 \vdash P$ and $\Gamma_2, P \vdash Q$ we can derive $\Gamma_1, \Gamma_2 \vdash Q$. All this requires no real proof effort and can be done with the application of rules. (Rules are theorem-generating functions in ProofPower, and their evaluation usually requires less effort.) We then obtain the extended refinement theorem

$$(6)\ \ \Gamma, P', wd\ X \vdash X \sqsubseteq Y' \wedge wd\ Y'$$

with the desired shape. Apart from the wd assertion for the initial program and provisos Γ, it only introduces the relevant proof obligations P' of the law.

All this takes place as part of the mechanics of the implementation of law applications. It relies, however, on the model and law theorems having the correct shape. For instance, if the law does not have the *wd* term of the transformed program in its consequent, the elimination of the proviso in (5) fails. The implementation is, nonetheless, sufficiently robust to handle such cases; this merely shows in the accumulation of provisos that could not be removed.

The examples used so far do not illustrate how provisos in the recast laws reduce those of the former laws beyond typing conditions. For example, assignment is encoded by the semantic function $\mathit{Assign}_C(a, ns, es)$ where a is the alphabet, and ns and es are sequences of variables and expressions. Here, $wd\ \mathit{Assign}_C(a, ns, es)$ does not merely constrain a to be of type $\mathit{ALPHABET}$, ns to be of type $\operatorname{seq}\ \mathit{VAR}$, and es to be of type $\mathit{WT\_EXPRESSION}$, but states the stronger $(a, ns, es) \in \operatorname{dom} \mathit{Assign}_C$, which, by definition of Assign_C, moreover implies that the sequences ns and es have the same length. Conditions like these would formerly have to be explicitly specified in the antecedent of the law. The simplification becomes even more apparent when the left and right-hand programs of the law are more complex. Despite, experience so far shows that in certain cases we still need to specify non-trivial type provisos, namely when in a law $A \sqsubseteq B$ we cannot prove $wd\ B$ from $wd\ A$, but this is not very often the case.

The next section discusses an approach to the treatment of provisos similar to the one above, but in the context of monotonicity theorems. It ensures that the application of structural combinators does not introduce further assumptions.

4.2 Structural combinators

The ArcAngel*C* implementation requires, for each combinator, two monotonicity theorems: one for equivalence and one for refinement. These theorems are used when applying the respective combinator tactic. They also give rise to provisos that accumulate. In general, the monotonicity theorems are of the form

$$\begin{array}{l}\vdash \forall\, a_1 : T_1;\ \ldots;\ a_k : T_k;\ p_1, \ldots, p_n : T;\ p'_1, \ldots, p'_n : T \mid \\ \quad P[a_1, \ldots, a_k, p_1, \ldots, p_n] \wedge P'[a_1, \ldots, a_k, p'_1, \ldots, p'_n] \bullet \\ \qquad p_1 \sqsubseteq p'_1 \wedge p_2 \sqsubseteq p'_2 \wedge \ldots \wedge p_n \sqsubseteq p'_n \Rightarrow \\ \qquad\quad \mathbf{op}(a_1, \ldots, a_k, p_1, \ldots, p_n) \sqsubseteq \mathbf{op}(a_1, \ldots, a_k, p'_1, \ldots, p'_n)\end{array}$$

where $\mathbf{op}(a_1, \ldots, a_k, p_1, \ldots, p_n)$ is an $(n{+}k)$-ary operator with fixed arguments $a_1, \ldots, a_k$ and monotonic arguments $p_1, \ldots, p_n$. For example, a conditional is a programming operator whose condition is a fixed argument, and whose programs are monotonic arguments. For action operators, T is $\mathit{CIRCUS\_ACTION}$. The provisos P and P' establish that the application of $\mathbf{op}$ is well-defined on both sides of the concluding refinement. They imply that $(a_1, \ldots, a_k, p_1, \ldots, p_n)$ and $(a_1, \ldots, a_k, p'_1, \ldots, p'_n)$ both belong to $\operatorname{dom} \mathbf{op}$.

Using the *wd* function, we generally express these theorems now as

$$\begin{array}{l}\vdash \forall\, a_1 : \mathbb{U};\ \ldots;\ a_k : \mathbb{U};\ p_1, \ldots, p_n : \mathbb{U};\ p'_1, \ldots, p'_n : \mathbb{U} \mid \\ \quad wd(\mathbf{op}(a_1, \ldots, a_k, p_1, \ldots, p_n)) \wedge p'_1 \in T \wedge \ldots \wedge p'_n \in T \bullet \\ \qquad p_1 \sqsubseteq p'_1 \wedge p_2 \sqsubseteq p'_2 \wedge \ldots \wedge p_n \sqsubseteq p'_n \Rightarrow \\ \qquad\quad \mathbf{op}(a_1, \ldots, a_k, p_1, \ldots, p_n) \sqsubseteq \mathbf{op}(a_1, \ldots, a_k, p'_1, \ldots, p'_n) \wedge \\ \qquad\quad wd(\mathbf{op}(a_1, \ldots, a_k, p'_1, \ldots, p'_n))\ .\end{array}$$

This shape is similar to that of rephrased laws, but additionally includes the provisos $p'_i \in T$ for all $i \in \{1, \ldots, n\}$. We explain below how they are discarded during the application of the tactic. We use $\boxed{\sqcap}$, the structural combinator for internal choice, as an example. The approach applies to all unary, binary and n-ary combinators of the ArcAngel*C* program model. An exception is the combinator $\boxed{\mu}$ for recursion, which we address separately.

The combinator $\boxed{\sqcap}$ for *Circus* internal choice has the following monotonicity theorem in our original implementation. (There are no fixed arguments.)

$$\begin{array}{l}\vdash \forall p_1, p_2, p'_1, p'_2 : \mathit{CIRCUS\_ACTION} \mid \alpha\ p_1 = \alpha\ p_2 \bullet \\ \quad p_1 \sqsubseteq p'_1 \land p_2 \sqsubseteq p'_2 \Rightarrow p_1 \sqcap_C p_2 \sqsubseteq p'_1 \sqcap_C p'_2\end{array}$$

Besides the type provisos, we require with $\alpha\ p_1 = \alpha\ p_2$ that p_1 and p_2 have the same alphabet. This is crucial for the well-definedness of $p_1 \sqcap_C p_2$. The refinements in the antecedent imply that $\alpha\ p_1 = \alpha\ p'_1$ and $\alpha\ p_2 = \alpha\ p'_2$, hence we can conclude that $\alpha\ p'_1 = \alpha\ p'_2$ holds too. This ensures well-definedness of $p'_1 \sqcap_C p'_2$. Accordingly, the new monotonicity theorem is as follows.

$$\begin{array}{l}\vdash \forall p_1, p_2 : \mathbb{U};\ p'_1, p'_2 : \mathbb{U} \mid wd(p_1 \sqcap_C p_2) \land \\ \quad p'_1 \in \mathit{CIRCUS\_ACTION} \land \\ \quad p'_2 \in \mathit{CIRCUS\_ACTION} \bullet \\ \qquad p_1 \sqsubseteq p'_1 \land p_2 \sqsubseteq p'_2 \Rightarrow p_1 \sqcap_C p_2 \sqsubseteq p'_1 \sqcap_C p'_2 \land wd(p'_1 \sqcap_C p'_2)\end{array}$$

We observe that we cannot rid ourselves of all type provisos: membership of p'_1 and p'_2 to $\mathit{CIRCUS\_ACTION}$ stays, because refinement of p_1 and p_2 does not necessarily preserve membership to $\mathit{CIRCUS\_ACTION}$. We can, however, use a proof tactic to remove these provisos in the resulting theorem by proving them from the residual assumptions and again using the cut rule as follows.

When a structural combinator is applied, the first step is to dissect the operator application and apply the combinator tactics to the program operands. Here, for example, applying $t_1\ \boxed{\sqcap}\ t_2$ to $A \sqcap_C B$ applies t_1 to A and t_2 to B. This results in the refinement theorems $\Gamma_1, wd\ A \vdash A \sqsubseteq A' \land wd\ A'$ and $\Gamma_2, wd\ B \vdash B \sqsubseteq B' \land wd\ B'$. We then conjoin these theorems to obtain $\Gamma_1, \Gamma_2, wd\ A, wd\ B \vdash A \sqsubseteq A' \land B \sqsubseteq B'$ whose conclusion has now the right shape to apply the monotonicity theorem in a forward-chaining manner, that is, using modus ponens to obtain the conclusion of the theorem. This yields

$$\begin{array}{l}\Gamma_1, \Gamma_2, A' \in \mathit{CIRCUS\_ACTION}, B' \in \mathit{CIRCUS\_ACTION}, \\ \quad wd\ A, wd\ B, wd(A \sqcap_C B) \vdash A \sqcap_C B \sqsubseteq A' \sqcap_C B' \land wd(A' \sqcap_C B')\ \ .\end{array}$$

From the application of wd to the original program, in our example $A \sqcap_C B$, it follows, by the definition of wd, that the operands are well-defined; in our example, $wd\ A$ and $wd\ B$ hold, so both can be easily eliminated as before. For the elimination of the assumptions related to the restrictions on the components of the new program, that is, $A' \in \mathit{CIRCUS\_ACTION}$ and $B' \in \mathit{CIRCUS\_ACTION}$ in the example above, we exploit the wd theorems of the individual program refinements. They are in the case above $\Gamma_1, wd\ A \vdash wd\ A'$ and $\Gamma_2, wd\ B \vdash wd\ B'$.

A proof tactic uses these theorems by rewriting $wd\ A'$. In the example above, for a binary structural combinator, there are two terms $wd\ A'$ and $wd\ B'$. In the case of a unary combinator, there is only one, and so on. If A' is an application

$op_C(A'')$ of a *Circus* operator op_C, then $wd\ A'$ yields that $A'' \in \operatorname{dom} op_C$. A simple law can then be used to infer that $op_C(A'') \in \operatorname{ran} op$. Since, the range of *Circus* operators is usually $CIRCUS\_ACTION$, we obtain a proof for $A' \in CIRCUS\_ACTION$. Considering such results for all terms $wd\ A'$, $wd\ B'$, and so on, is usually enough to establish well-definedness of the new program.

We observe that this reasoning does not depend on the structural combinator *per se*. The assumptions are that the provisos are exclusively of the form $p_i' \in CIRCUS\_ACTION$, that there exists a wd axiom for every *Circus* operator, and that all *Circus* operators have domains $CIRCUS\_ACTION$.

In our example, this permits us to eliminate the two other provisos too.

$$\Gamma_1, \Gamma_2, wd(A \sqcap_C B) \vdash A \sqcap_C B \sqsubseteq A' \sqcap_C B' \land wd(A' \sqcap_C B')$$

The result contains only the genuine proof obligations, so no overhead is incurred by using the monotonicity rule in the mechanisation of the combinator.

The crucial point here is that in the case of structural combinators some real proof effort is needed to eliminate the surplus provisos. Our implementation of ArcAngel*C* is in fact an instantiation of a framework that we have developed for angelic languages for refinement tactics. It supports ArcAngel*C* and its predecessor ArcAngel [11] for Morgan's refinement calculus, and can be extended for other languages of the same nature. To manage structural combinators, we have extended the framework to support the dynamic configuration of tactics used to discharge provisos emerging from their application. For all ArcAngel*C* combinators, the tactics we define are sufficient. The provided flexibility, however, permits the inclusion of custom tactics that may be needed for other languages.

There are cases in which elimination of the assumptions fails, namely, if the arguments of the operator are not of the form $op_C(\ldots)$ where op_C is some *Circus* operator. Experience, however, shows that this does not happen very often.

More specific extensions have been necessary to handle the structural combinator $\boxed{\mu}$ for recursion, which, unlike the other combinators, is supplied with a function on alphabetised predicates, usually given as a λ-expression. The proof steps for removing provisos are more involved, and in fact it was not possible to remove all of them due to the fact that pointwise refinement of a function does not preserve monotonicity of the function. Precisely, from monotonicity of f $(\forall x, y \in \operatorname{dom} f \bullet x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y))$, and the pointwise property $\forall x \in \operatorname{dom} f \bullet f(x) \sqsubseteq f'(x)$ we cannot conclude that f' is monotonic too. In pen-and-paper proofs, we justify monotonicity assumptions by the particular shape of actions only using monotonic constructs. In a shallow embedding, capturing this is more difficult because we cannot give a general theorem stating that *Circus* actions are monotonic. Solving this problem is part of ongoing work.

The next section comments on experiences in the context of a case study.

5 Practical experiences

The primary motivation for the treatment of well-definedness in the last two sections is the automation of complex refinement strategies using our tools and the semantic embedding of *Circus*. As a case study, we have automated part of the refinement strategy for control laws described in [2,3].

The work in [2,3] describes first how a formal account of (a subset of) the Simulink notation can be given by translating the diagrams into *Circus* specifications. (Simulink is a *de facto* industry standard for the design of control systems.) It then presents a strategy for refinement that gradually transforms the diagram specification into a *Circus* model of a given Ada program, and thereby verifies the correctness of the Ada implementation. For the encoding of the ArcAngel*C* tactics, we refer to the work in [13]. The tactics we mechanise are mostly literal translations of those in [13], including the various *Circus* laws they rely upon.

The refinement strategy is structured into four phases, and so far we only automated the first phase NB. This is itself subdivided into 7 steps, NBStep1 to NBStep7, which are assembled into one compound tactic NBMain. The tactic deals with the normalisation of *Circus* processes representing blocks (or subsystems) that are realised sequentially in code. Their model is given by a single centralised process with two parallel actions. The tactic attempts to remove the parallelism between these actions. A more detailed account of the refinement strategy is omitted as it would take us too far from the agenda of the paper.

Our example specification is part of the model for the PID controller in [3]. In particular, we have applied the tactics to the model of a differentiator. It is a simple example, which, however, as we discuss below, already reveals the importance and effectiveness of our treatment of well-formedness.

Having encoded the tactics and laws, we have verified by inspection, that all assumptions that remain after the application of the tactics are genuine proof obligations. As explained in Section 4.2, particular kinds of provisos, namely those resulting from $\boxed{\mu}$, at present cannot be discharged automatically. We verified, however, that these are the *only* residual provisos, apart from the well-definedness assumption of the initial program of the refinement.

This initial analysis has uncovered some glitches in the recast theories and implementation that have been subsequently fixed. One of them was a missing wd axiom for one of the *Circus* operators which resulted in a simplification tactic failing. As already mentioned, the implementation has been designed to robustly deal with such cases, hence no error is raised when individual simplifications of provisos do not succeed. (In some cases this leads to a partial proof encoded as a 'reduced' proviso, and in other cases the proviso remains unchanged.) We have increased the efficiency of simplification tactics by incorporating guarding conditions that ensure they are only applied to assumptions of the correct shape, and furthermore only involve a predictably small number of proof steps.

We have also compared the efficiency of our technique to that of the standard explicit treatment of provisos. Since our extensions to the ArcAngel*C* implementation are fully backward compatible, it is possible to revert to the old shape of laws and model theorems and execute the same tactics under the same conditions. Table 1 summarises the results obtained for the tactics NBStep1 to NBStep7 of the refinement strategy. (NBStep5 and NBStep6 have been merged into one tactic). Each row entails the invocation of the preceding tactics too; for instance, the second row refers to the execution of both NBStep1 and NBStep2.

Phase	POs Before	POs Current	Run-time Before	Run-time Current
NBStep1	1	1	0.0 sec	0.1 sec
NBStep2	44	4	2.3 sec	1.1 sec
NBStep3	101	8	9.1 sec	2.4 sec
NBStep4	163	15	17.9 sec	3.9 sec
NBStep5_6	198	18	19.9 sec	4.7 sec
NBStep7	243	24	21.1 sec	5.9 sec

Fig. 1. Comparison of the number of generated provisos for NB.

We report on the number of remaining provisos as well as the execution time of the tactics measured by the timing facilities of Poly/ML.

The results show that assumptions are reduced by approximately 90%. This remains fairly constant across tactics, and invariant with the growth in size. We may expect a similar reduction in runtime, but it is offset by the effort required to discharge the provisos. Despite, we see that overall this effort is recovered, and the gain increases with the complexity of the generated refinement theorems. For example, after NBStep2 we only have a small speed-up, and for NBStep1 even a slight loss. The speed-up, however, becomes larger in the next two phases.

To check if this trend continues with more complex examples, we have slightly amended the Diff process. This has resulted in larger ProofPower-Z terms and the need for more elementary steps to transform the program. In this case, the original implementation takes 61 sec to execute phases NBStep1 to NBStep5_6 and generates a theorem with 259 assumptions. In comparison, the new technique requires 7.2 sec producing 24 assumptions, giving a further speed-up increase to 8.5. After once more elaborating the complexity of the example specification, the former approach requires 154.4 sec and produces 301 assumptions whereas the new ArcAngel*C* implementation only takes 10.9 sec. Although we still have a similar 90% reduction in assumptions, the speed-up has now increased to 14.1.

The above suggests that with growing complexity of the ArcAngel*C* tactics and *Circus* specifications, the speed-up may further increase, even so non-linearly. Reasons for this may be that certain operations on theorems are slowed down in a non-proportional way when theorems become larger. It should be pointed out that the system we compare with already includes the simplification to the mechanisation reducing type provisos that we discussed in Section 3.1.

Our results increase confidence that, with the new technique and tool support, it is possible now to apply the entire refinement strategy and obtain a result within a reasonable amount of time.

6 Conclusions

We have presented a treatment of well-formedness in a shallow embedding of *Circus* that considerably simplifies the provisos generated in the application of refinement (and other) laws. Although it was described in the context of our *Circus* mechanisation, the techniques and results obtained generalise to other language embeddings in other tools that raise similar issues.

We have introduced the *wd* function to capture well-formedness of terms *as if* they were syntactic rather than semantic entities. We have then identified constraints to establish that this axiomatisation, though non-conservative, is sound. We have thus avoided the added complexity of a deep embedding while reaping some of its benefits in relation to properties that are inherently syntactic.

The underlying model for *wd* is actually a strict treatment of undefined values. We, however, do not make its model explicit and content ourselves with its mere existence. Since the logic of HOL does not accommodate undefinedness, limitations arise in that we cannot, for instance, utilise *wd* on boolean functions because the type $\mathbb{B}$ is not 'big enough' to provide enough values to represent the undefined case. This is a trade off that we have to accept.

The *wd* function crucially paves the way for the efficient application of more complex refinement tactics as it enabled us to recast the implementation of ArcAngel*C* in such a way that, apart from genuine proof obligations, only one additional proviso is generated — to establish the well-definedness of the initial program. This keeps the complexity of emerging refinement theorems at bay, and thereby creates opportunities for the use of our mechanisation of *Circus* and tools for automatic refinement in the context of real industrial systems.

In [17], von Wright presents a tool for stepwise refinement in a simple sequential command language. Its semantics is characterised in terms of wp predicate transformers, which are shallowly embedded into HOL. Well-definedness conditions loosely correspond to establishing the monotonicity of predicate transformers, however the semantic domain is not *a priori* restricted to those.

The implication of von Wright's and similar approaches are that we required more relaxed definitions of operators, and also accept limitations on what laws can be *generally* established for the semantic entities. In the UTP, we often rely on proving properties from healthiness conditions rather than by induction over some syntax, which makes our approach more suitable here.

Other related work is the refinement editor Refine and Gabriel [14] which supports the specification and interactive application of ArcAngel tactics to derive programs in Morgan's calculus. These tools notably offer facilities to interactively apply tactics and laws. Otherwise, Refine and Gabriel were developed in view of a specific language, and are essentially rewrite systems.

More recent work has been done on developing Saoithín [1], a proof assistant specifically designed for the UTP. Its advantages are that it essentially operates at the level of syntax, and naturally some of the semantic issues we reported on do not arise. It also provides proof strategies that are optimised for proofs in the UTP. On the other hand, it does not specify a model for its deductive system and calculus, and this imposes limitations on proving consistency of theory extensions. An interesting line of work could be to try and combine Saoithín with our tools to see if we can reap individual benefits of both of them.

Future work will first consist of mechanising the entire refinement strategy for control laws in [13]. Secondly, further work is required to provide proof automation in relation to the well-definedness of the initial program. To obtain unqualified theorems, those assumptions need to be discharged. It is still an open issue how provisos in such theorems are best proved, and whether some of the *wd* theorems that we discard as we go along should be cached for later use.

Acknowledgements This work has been funded by EPSRC as part of the Programming from Control Laws research grant EP/E025366/1.

References

1. A. Butterfield. *Saoithín Proof Assistant*. Available for download at http://www.scss.tcd.ie/Andrew.Butterfield/Saoithin/.
2. A. Cavalcanti, P. Clayton, and C. O'Halloran. Control Law Diagrams in *Circus*. In *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 253–268. Springer, July 2005.
3. A. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via *Circus*. Technical report, University of York, York, U.K., April 2008.
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. E. Dijkstra. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall, 1976.
6. C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall, February 1998.
7. A. Martin, P. Gardiner, and J. Woodcock. A Tactic Calculus - Abridged Version. *Formal Aspects of Computing*, 8(4):479–489, July 1996.
8. C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, 1998.
9. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science, University of York, 2005.
10. M. Oliveira and A. Cavalcanti. ArcAngel*C*: a refinement tactic language for *Circus*. *Electronic Notes in Theoretical Computer Science*, 214:203–229, June 2008.
11. M. Oliveira, A. Cavalcanti, and J. Woodcock. ArcAngel: a Tactic Language for Refinement. *Formal Aspects of Computing*, 15(1):28–47, July 2003.
12. M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for *Circus*. *Formal Aspects of Computing*, Online First, December 2007.
13. M. Oliveira, A. Cavalcanti, and F. Zeyda. A Tactic Language for Refinement of State-Rich Concurrent Specifications. To appear.
14. M. Oliveira, M. Xavier, and A. Cavalcanti. Refine and Gabriel: Support for Refinement and Tactics. In *Proceedings of the Second Int. Conference on Software Engineering and Formal Methods*, pages 310–319. IEEE Computer Society, 2004.
15. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science. Prentice Hall, November 1997.
16. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International Series In Computer Science. Prentice Hall PTR, June 1992.
17. J. von Wright. Program Refinement by Theorem Prover. In *BCS FACS Sixth Refinement Workshop – Theory and Practise of Formal Software Development, London, U.K.* Springer, January 1994.
18. F. Zeyda and A. Cavalcanti. Supporting ArcAngel in ProofPower. *Electronic Notes in Theoretical Computer Science*, 259:225–243, December 2009.
19. F. Zeyda and A. Cavalcanti. Mechanical Reasoning about Families of UTP Theories. *Science of Computer Programming*, March 2010. DOI dx.doi.org/10.1016/j.scico.2010.02.010.