

Using PVS to Support a Real-Time Refinement Calculus (Work-in-Progress and Extended Abstract)

Richard F. Paige¹ and Phillip J. Brooke²

¹ Department of Computer Science, University of York,
Heslington, York YO10 5DD, United Kingdom. paige@cs.york.ac.uk

² School of Computing, University of Plymouth,
Drake Circus, Plymouth, Devon, PL4 8AA, United Kingdom. philb@soc.plym.ac.uk

Abstract. Hehner's theory of predicative programming is a general-purpose refinement calculus for producing correct sequential, concurrent, real-time, and communicating programs from specifications. However, only limited tool support exists for this theory. We give an overview of an ongoing project on how we are providing support for predicative programming via PVS, particularly for discharging the proof obligations that arise during algorithm refinement, and specifically, in the domain of real-time applications.

1 Introduction and Motivation

The domain of real-time systems has benefited substantially from the development and application of formal methods and theories of programming. In particular, a number of real-time refinement calculi, e.g., [5, 6] have been proposed. Extensions to the Z specification language for supporting real-time refinement have also been developed, e.g., [4]. These approaches typically support both algorithm and data refinement, and can be used for developing provably correct real-time programs from specifications that include timing constraints. Limited tool support exists for such calculi, e.g., the window inference tool extension of PRT built atop the Ergo theorem prover [12]. This tool is limited to proof assistance for sequential real-time programs, in part because the underlying theory of refinement is limited in this regard.

Hehner's calculus of predicative programming [6] is a simple and general-purpose refinement calculus. It supports sequential, concurrent, real-time, communicating, and object-oriented specification, as well as the refinement of these specifications to programs. It provides only very limited tool support for editing specifications and proofs, and for checking (simple) proofs via HOL [7]. There is currently no full-featured tool support for this calculus for discharging all of the proof obligations that arise during refinement.

We are currently experimenting with the use of the PVS system for supporting real-time refinement in predicative programming. This extended abstract

gives an overview of the calculus, explains how we plan to support it with PVS, provides an initial implementation of specifications and programs in PVS, and discusses the meaning of refinement in PVS. It then details ongoing and future work, and key problems that we anticipate encountering.

2 Overview of Predicative Programming

Predicative programming is a refinement calculus in which programs are specifications. In this approach, programs and specifications are predicates on pre- and poststate (as in Z, final values of variables are annotated with a prime; initial values of variables are undecorated). The weakest predicative specification is \top (“true”), and the strongest specification is \perp (“false”). Here is an example of a specification stating that the final value of natural number variable x is the element n in the Fibonacci sequence fib , while the final value of y is element $n + 1$ of this sequence.

$$x' = fib\ n \wedge y' = fib\ (n + 1)$$

(Juxtaposition is function application.)

The specification language in Hehner’s calculus includes both programs and specifications. Programs are given a predicate semantics. Here is a list of a standard set of program constructs and their predicative meanings (ignoring time, which is considered in the sequel). P and Q represent specifications, while e_1, \dots, e_n represent variables. The notation $P[x := y]$ represents (free) textual substitution of y for x in predicate P .

$$\mathbf{skip} \hat{=} e_1' = e_1 \wedge e_2' = e_2 \wedge \dots \wedge e_n' = e_n \quad (1)$$

$$e_1 := exp \hat{=} e_1' = exp \wedge e_2' = e_2 \wedge \dots \quad (2)$$

$$\mathbf{if\ } b \mathbf{\ then\ } P \mathbf{\ else\ } Q \hat{=} (b \rightarrow P) \wedge (\neg b \rightarrow Q) \quad (3)$$

$$P; Q \hat{=} \exists \hat{\sigma} \bullet P[\sigma' := \hat{\sigma}] \wedge Q[\sigma := \hat{\sigma}] \quad (4)$$

$$\mathbf{(local\ } e : T; P) \hat{=} (\exists e, e' : T \bullet P) \quad (5)$$

The programming language subset does not include a loop construct, e.g., a **while**. Looping programs are produced, in Hehner’s calculus, by carrying out a *recursive refinement* [6], where a recursive program is generated. The recursive program can then be transformed directly to a looping program, by a simple transformation rule [6]. Alternatively, loops can be introduced via least fix-points, or via the variant/invariant rules given in [10]. The process of recursive refinement is discussed in Section 3.2; supporting this process via PVS is a key element of our current research.

2.1 Refinement in predicative programming

Algorithm refinement in Hehner's calculus is just boolean implication. Since programs and specifications are both predicates, all refinement steps – whether they involve refining a specification to another specification, or a specification to a program – involve proving a boolean implication.

Definition 1. A specification P on prestate σ and poststate σ' is refined by a specification Q if $\forall \sigma, \sigma' \cdot (P \Leftarrow Q)$.

The refinement relation enjoys various properties that allow specifications to be refined by parts, steps, and cases. As well, specifications can be combined using the familiar operators of boolean theory, along with all the usual program combinators, as well as combinators for parallelism and communication through channels.

2.2 Real-time and concurrency

Predicative programming is well-suited to specifying and reasoning about real-time, concurrent, and communicating systems. To talk about time, a time variable t is used; the theory does not need to be changed at all. The interpretation of t as time is justified by how the variable is used. t is used as the initial time (when the execution of a computation starts), and t' for final time (when execution of a computation ends). To allow for nontermination, the domain of time is a number system extended with an infinite number, ∞ . The number system can be naturals, reals, et cetera.

The following example says that the final value of variable h should be the index of the first occurrence of x in list L , and that any computation satisfying the specification must provide an execution time that is linear in the length of L .

$$(\neg x : L(0, ..h') \wedge (Lh' = x \vee h' = \#L)) \wedge t' \leq t + \#L$$

No new rules need to be introduced in order to refine specifications that include time: time is just another variable. To prove that a program satisfies a specification with time constraints, a *refinement by parts* approach is typically used. In this approach, the specification is first refined to a program and time is ignored. Afterwards, the refinement tree that is produced is reused and time is added, thus generating a set of proof obligations involving, primarily, the time variable. The resulting second refinement is usually simpler to carry out than the first, because the refinement tree is reused, and because many elements from the initial refinement, e.g., state changes that do not effect timing, can be discarded.

Predicative programming includes notations for concurrent specification and for communication. Concurrency is provided via a parallelism operator, \parallel . Communication is via shared channels. We direct the reader to [6] for details on communication and concurrency.

We now outline how we plan to support refinement in this calculus using PVS.

3 Predicative Specification in PVS

PVS [8] is a suitable tool to support refinement in predicative programming: it provides an expressive higher-order specification language and an extensive library of theories that can be used to represent data types and programs; it provides an industrial-strength theorem prover and model checker; and predicates are directly supported in PVS.

There are several key problems in using PVS to support refinement in predicative programming:

- representing specifications (and, therefore, programs), their state, and frames.
- representing refinement trees and the requisite proof obligations that arise.
- utilizing feedback provided by the theorem prover when attempting to carry out a refinement proof.

We now discuss our ongoing work on these problems, and consider those elements of the mapping to PVS that remain under development.

3.1 Representing specifications, state, and programs

Since a predicative specification and program is just a predicate, it seems plausible that we could represent a predicative specification directly in as a boolean-valued function. It turns out that this is not possible. A critical problem arises when we desire to represent sequential composition (equation (4), above): sequential composition is defined as an existential quantification over the *state* of the specification. We thus need an explicit representation of a specification's state. This is also needed for representing a specification's frame, the set¹ of variables whose value may be changed by the computation being specified.

When using Hehner's calculus, an initial specification will be written and refined. At each refinement step, a new specification will be produced, and a proof obligation will have to be discharged, showing that the new specification

¹ In [6], frames are actually *bunches* of variables; a bunch is an unpackaged, unindexed data structure. We are currently investigating how to implement bunch theory in PVS, and an initial prototype of bunch theory in PVS has been generated.

refines the original. The set of refinement steps, and corresponding specifications, will be translated into a single PVS theory. This theory will contain PVS translations of the state of the specifications and the specifications themselves. The state, consisting of a set of entities, will be translated into a PVS record type. A new record type must be produced for each specification (since the state of each specification may differ from any other).

We illustrate the translation process – both for state and specifications – by example. Consider the predicative specification P , defined as

$$P \hat{=} x' \geq x \wedge y' = 2y \wedge z' = z \wedge t' \leq t + 5$$

where all variables are assumed to be integer, for simplicity. Its state consists of the entities x , y , and z .

The PVS specification of P includes a record consisting of the three entities (as well as the time variable t), plus functions that map entities into values. These functions effectively bind an entity to its value. One evaluator will be required for each variable type, e.g., *int*, *nat*, *real*, *list*. Since in the example we assume all variables have the same type, *int*, then one evaluator is needed.

```

ENTITY: TYPE+          % general type of variables
EVAL: TYPE+ = [ ENTITY -> int ] % evaluators

% The state of specification P
P_STATE: TYPE+ = [# x:ENTITY, y:ENTITY, z:ENTITY, t:ENTITY #]

```

We can now translate specifications that make use of this state. A predicative specification is translated to a record, consisting of: the old and new state, an encoding of the specification as a function from pre- and poststate to a boolean, and the frame of the specification. The frame specifies the variables that can be changed by the specification; all variables in the state of the specification but not in the frame must remain unchanged.

```

SPECTYPE: TYPE+ =
  [# old: P_STATE,          % prestate
   new: P_STATE,          % poststate
   frame: set[ENTITY],    % frame of variables
   eval: [ o:{ol:P_STATE|ol=old}, n:{nl:P_STATE|pl=new} -> bool ] #]

```

A predicative specification can now be translated into a PVS specification in terms of SPECTYPE. The general rules are thus:

- occurrences of unprimed variables, e.g., x , y , are translated to record accessors in the old state of the specification, e.g., $x(\text{old})$ (note that `old` is itself a field of a record that must be applied to an instance of `P_STATE`).

- occurrences of primed variables, e.g., x' , y' , are translated to record accesses in the new state of the specification, e.g., $y(\text{new})$.
- predicative connectives such as \vee , \wedge , and \Rightarrow are translated to their PVS equivalents, i.e., OR, AND, and IMPLIES.
- the predicative specification as a whole is translated to a lambda binding which is assigned to the `eval` field of the `SPECTYPE` record.
- the frame of a predicative specification will be mapped to a set of evaluations. These evaluations will then be used in generating a frame axiom, following the approach of Borgida et al [3].

In general, the predicative specification will be mapped to the fields of a constant of type `SPECTYPE`, and two constants will also be declared, representing the pre- and poststate. Three axioms will be generated which define the values of the fields of the `SPECTYPE` instance: an axiom defining the state, one defining the frame, and one defining the evaluation.

Here is an example of the axioms to be generated for specification P , above.

```

p_spec: SPECTYPE          % the specification
old_p, new_p: P_STATE    % pre- and poststate of P

% Define the state of p_spec
p_state_ax: AXIOM (old(p_spec)=old_p AND new(p_spec)=new_p)

% Define the frame of p_spec
p_frame_ax: AXIOM
(member(x(old_p),frame(p_spec)) AND
 member(y(old_p),frame(p_spec)) AND
 member(z(old_p),frame(p_spec)) AND
 member(t(old_p),frame(p_spec)))

% Define the specification as a lambda binding.
p_eval_ax: AXIOM
(eval(p_spec) :=
  (LAMBDA (o:{p1:P_STATE | p1=old(p_spec)}),
   (n:{p2:P_STATE | p2=new(p_spec)}):
    x(n)>=x(o) AND y(n)=2*y(o) AND
    z(n)=z(o) AND t(n)=t(o)+5))

```

The `p_eval_ax` axiom states that for specification `p_spec`, the value of the specification is a function from pre- and poststate to a boolean, where the boolean is *true* if and only if the translation of `p_spec` into PVS's specification language is *true*.

In `p_eval_ax`, a specific translation, of predicate P , is given. In general, the body of the lambda binding will be a translation of a predicative specification s into a PVS specification `trans(s)`. This is a syntax-directed translation on the structure of the predicative specification. A full definition of `trans` is beyond the scope of this abstract. Many of the details are straightforward, e.g.,

translations of expressions and basic data structures, though full details remain to be worked out. Some examples illustrating the translation are as follows.

- The predicative specification $x' = e \wedge y' = y (x := e)$ is translated to the PVS specification

```
x(new(p_spec))=trans(e) AND y(new(p_spec))=y(old(p_spec))
```

- The predicative specification **if** b **then** P **else** Q is mapped to

```
(trans(b) AND trans(P)) OR (NOT trans(b) AND trans(Q))
```

- The predicative variable x is mapped to the PVS record access

```
x(old(p_spec))
```

- The predicative variable x' is mapped to the PVS record access

```
x(new(p_spec))
```

We point out the recursive nature of the translation, particularly as applied to the **if**. We explain how to translate sequential composition shortly.

The translation of specification state, as a record of entities, is not without its awkwardness. Also, because we represent state as a record of entities, it is left to the translator (or a tool) to check that specifications use only those entities declared within the state.

Sequential composition must be translated in a different way. Sequencing is complicated by its use of an existential quantifier and an intermediate state (the state that a sequential computation is in after executing its first part). To translate a sequential composition $P; Q$, we assume that we can translate predicative specifications P and Q (thus producing two SPECTYPES) and then apply PVS function `seqspecs`, as defined below.

```
seqspecs: [ SPECTYPE,SPECTYPE -> SPECTYPE ]

seqspecs_ax: AXIOM
(FORALL (s1,s2: SPECTYPE):
  seqspecs(s1,s2) =
    (# old := old(s1),
     new := new(s2),
     frame := union(frame(s1),frame(s2)),
     eval := (LAMBDA (o:{o1:P_STATE | o1=old(s1)}),
              (n:{n1:P_STATE | p2=new(s2)}):
                (EXISTS (i:P_STATE): eval(s1)(o,i) AND eval(s2)(i,n)))
    #) )
```

`seqspecs` is a PVS formalization of the sequencing operator ; from predicative programming. It produces a new specification from two existing specifications. The prestate of the new specification is the prestate of the function's

first argument; the poststate of the new specification is the poststate of the function's second argument. The evaluator of the new specification is the relational composition of the two argument specifications.

Thus, the predicative specification $P; Q$ (for predicates P and Q) will be translated to the PVS specification $\text{seqspecs}(\text{trans}(P), \text{trans}(Q))$.

3.2 Looping structures

We are currently investigating how to represent predicative looping structures in PVS. As it stands, predicative programming recommends against using a loop construct, e.g., **while** or **repeat**, in refinement. The recommended approach to introduce looping computations is via so-called recursive refinement: using a specification as a procedure call within a refinement tree.

Here is a short, simple example to illustrate the concept. Suppose we desire to write a program to find the first occurrence of a given item x in a list L , and to do so in linear execution time. Let h be a variable holding the index of the first occurrence of x if it is in the list, otherwise h will be set to the length of the list, $\#L$. Here is the specification R .

$$R = \neg x : L(0, ..h') \wedge (Lh' = x \vee h' = \#L)$$

This ignores time; the full specification is thus $R \wedge t' \leq t + \#L$. To refine the specification to the obvious looping program, the process is to first ignore time and refine R as follows. First, define A as

$$A = \neg x : L(0, ..h)$$

Then the steps to refine R to code are as follows.

$$R \Leftarrow h := 0; h \leq \#L \wedge A \Rightarrow R \tag{6}$$

$$h \leq \#L \wedge A \Rightarrow R \Leftarrow \mathbf{if} \ h = \#L \ \mathbf{then} \ \mathbf{ok} \tag{7}$$

$$\mathbf{else} \ h < \#L \wedge A \Rightarrow R$$

$$h < \#L \wedge A \Rightarrow R \Leftarrow \mathbf{if} \ Lh = x \ \mathbf{then} \ \mathbf{ok} \tag{8}$$

$$\mathbf{else} \ (h := h + 1; h \leq \#L \wedge A \Rightarrow R)$$

In the last step, (8), the specification $h \leq \#L \wedge A \Rightarrow R$ appears once again; it is being refined in step (7). This is a recursive call. In implementing the refinement in a real programming language, the recursive call would be replaced by a loop.

To support recursive refinement in PVS we need to be able to (a) sequentially compose specifications (this is supported by `seqspecs`, above) and (b) provide an evaluator for each specification. Thus, we expect that the PVS infrastructure that we have provided so far is sufficient for supporting the production of looping programs.

In [10], a set of refinement rules for introducing programs that use standard **while** loops (based on variants and invariants) was specified for the predicative calculus. The approach to developing looping programs offered by these rules is more complex than recursive refinement; however, some developers may prefer to use variants and invariants to develop loops. Thus, we are exploring how to represent the seven proof obligations discussed in [10] for loops in PVS, and will thus allow developers to choose between use of recursive refinement and the variant/invariant approach in producing looping programs. We note that, in general, recursive refinement leads to simpler refinement trees and proof obligations than with use of explicit variants and invariants.

3.3 Refinement

Refinement is straightforward to support with PVS. A specification P is refined by a specification Q if everywhere $P \Leftarrow Q$. In PVS, we define a function `ref` that takes two `SPECTYPE`s and returns true if and only if its first argument is refined by its second.

```

ref: [SPECTYPE, SPECTYPE -> bool]

ref_ax: AXIOM
(FORALL (s1,s2:SPECTYPE):
  (ref(s1,s2) =
    old(s1)=old(s2) AND new(s1)=new(s2) AND eval(s2)(old(s2))(new(s2))
    IMPLIES eval(s1)(old(s1))(new(s1))))

```

Effectively, the refinement axiom states that $s1$ is refined by $s2$ if evaluating $s2$ on its pre- and poststate implies $s1$ evaluated on its pre- and poststate. We envision that it will be helpful to define PVS functions that can be used to extend the state of one specification to be identical to the state of a second specification (by addition of extra conjuncts of the form $x' = x$). This infrastructure will also be useful in formalizing frames.

A predicative refinement consists of an initial specification, and one or more refinement steps, where a proof must be discharged. At each refinement step, a new specification will be produced. Each refinement step will be translated into a conjecture that applies the PVS function `ref` to the two translated predicative specifications.

4 Discussion and Ongoing Work

The work we have outlined is very much ongoing. Our plan is to focus on two specific aspects of refinement within PVS:

1. Better support within PVS for refinement, via improved (simpler) translations of predicative constructs. We also desire to construct suitable PVS strategies for automating the discharge of proof obligations. As well, we want to make it easier to use the feedback provided by PVS during proof in restructuring and revising a refinement step or specification.
2. Applications, wherein PVS will be used to support real-time refinement of a number of case studies.

In terms of (1), particular questions that we desire to ask include the following.

- *Representations of state and frames.* Here we have proposed to represent specification state in PVS using records, but it may be easier to use tuples or functions. Experiments will be carried out to decide which approach leads to simpler proof obligations and less need for user intervention in proof. In this, we are examining the work of [2], which presents a mapping from B machines to PVS. The mapping from B to PVS is different than what we require, since B provides machines for encapsulation of operations, but we may be able to leverage some of the ideas of [2] in translating predicative specifications.
- *Fewer axioms.* To represent sequential composition, a new axiom is needed to define `seqspecs` on each state. A generic specification would be preferable. We are exploring parameterized theories in PVS for this, and an initial investigation suggests that they are suitable.
- *Tools.* A tool needs to be written to automatically generate the PVS theories from predicative specifications. We envisage this as a GUI-based editor with a suitable code generator. We envisage a proof editor akin to the one developed in [7], or Verhoeven and Backhouse’s Mathpad [13].
- *Feedback.* PVS may be able to automatically discharge the conjectures that are needed to prove a sequence of refinement steps, though oftentimes it will be necessary for the developer to intervene in order to provide proof hints. However, there will be occasions when a conjecture cannot be proved: because the refinement step is incorrect, or the initial specification did not correctly capture the requirements, etc. If a conjecture cannot be proved, PVS will provide feedback in the form of simplified proof obligations that cannot be discharged. We desire to be able to use this feedback to restructure and rewrite both the original predicative specifications, and the attempted

refinement steps. This will require a backwards translation – of PVS’s specification and conjecture language in to predicative notation – and a feedback loop between the theorem prover and the proof editor mentioned in the previous point.

- *Strategies for recursive refinement.* The recursive refinement steps, for introducing loops, will likely require a measure to be provided, since the PVS proof obligations for these steps will require induction. We will experiment with automatically extracting measures from time bounds and time variables. This idea is akin to the work on automatic invariant generation [1].

In terms of (2), we will carry out a number of case studies on real-time refinement, ranging from small experiments where exact bounds will be proven, to experiments where non-deterministic time constraints are provided, to larger case studies, e.g., involving the profiling of garbage collection algorithms. These case studies all aim at validating our PVS translation and the utility of PVS by experiment.

References

1. S. Bensalem. Powerful Techniques for the Automatic Generation of Invariants, in *Proc. CAV’96*, LNCS, Springer-Verlag, 1996.
2. J.-P. Bodeveix and M. Filali. Type Synthesis in B and the Translation of B to PVS. In *Proc. ZB-2002*, LNCS 2272, Grenoble, France, Springer-Verlag, 2002.
3. A. Borgida, J. Mylopoulos, and R. Reiter. And Nothing Else Changes: the frame problem in procedure specifications, *Proc. International Conference on Software Engineering 1993*, ACM Press, May 1993.
4. C. Fidge. Real-time Refinement. In *Proc. Formal Methods Europe 1993*, LNCS 670, Springer-Verlag, 1993.
5. I. Hayes and M. Utting. A Sequential Real-Time Refinement Calculus. *Acta Informatica* 37(6):385-448, 2001.
6. E.C.R. Hehner. *A Practical Theory of Programming*, Springer-Verlag, 1993.
7. A.Y.C. Lai. *A Tool for a Formal Refinement Method*, M.Sc. Thesis, Department of Computer Science, University of Toronto, 2000.
8. S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS System Guide 2.4*, Computing Science Laboratory, SRI International, November 2001.
9. R.F. Paige and E.C.R. Hehner. Bunches for Object-Oriented, Real-Time, and Concurrent Specification, *Proc. World Congress on Formal Methods 1999*, LNCS 1708, Toulouse, France, Springer-Verlag, 1999.
10. R.F. Paige and J.S. Ostroff. ERC - An Object-Oriented Refinement Calculus for Eiffel. CS-TR-2001-05, York University, Toronto, Canada, August 2001.
11. R.F. Paige, J.S. Ostroff, and P.J. Brooke. Checking the Consistency of Collaboration and Class Diagrams using PVS, in *Proc. Fourth Workshop on Rigorous Object-Oriented Methods (ROOM4)*, British Computer Society, London, U.K., March 2002.
12. L. Wildman, C. Fidge, and D. Carrington. Computer-Aided Development of a Real-Time Program. *Software - Concepts and Tools* 19(4), August 2000.
13. R. Verhoeven and R. Backhouse. Interfacing program construction and verification, in *Proc. World Congress on Formal Methods 1999*, LNCS 1709, Springer-Verlag, 1999.