Program Design in PVS

Jozef Hooman

Dept. of Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: wsinjh@win.tue.nl

Abstract. Hoare triples (precondition, program, postcondition) have been incorporated in the verification system PVS. Two approaches are presented: the conventional one, with a clear distinction between syntax and semantics, and another where programs are identified with their semantics. In the last approach specifications are embedded in the semantic framework, leading to a formalism where specifications and programming constructs can be mixed freely. This framework forms the basis of a formal method for the design of distributed real-time systems.

1 Introduction

General aim of our work is the formal specification and compositional verification of distributed real-time systems. To this end, a formalism based on Hoare triples (precondition, program, postcondition) has been devised and applied to a number of examples such as a distributed real-time arbitration protocol [Hoo94a], a chemical batch processing system [Hoo94c], and a mine pump system [Hoo96a]. These examples have been verified manually.

To investigate whether the method scales up to larger systems, clearly some form of mechanical support is indispensable. Usually there is a large number of simple verification conditions which should be discharged automatically. Since the design process is iterative, with frequently changing specifications, one would like to have a tool which keeps track of all dependencies and allows us to rerun and check proofs mechanically. Further, to avoid that one has to start from scratch, it is desirable that a number of basic theories are already present (e.g., for real-time applications some arithmetic on the real numbers should be available).

Based on these requirements, we decided to use the verification system PVS[1] (Prototype Verification System) [ORS92, ORSvH95]. An advantage is that the basics of PVS are rather easy to learn and one can quickly start experimenting with the intended application. The PVS specification language is a higher-order typed logic, with many built-in types including booleans, integers, rationals, sequences, lists, and sets. Specifications can be structured into a hierarchy of parameterized theories. The tool contains an interactive proof checker with powerful commands for, e.g., inductive proofs, automatic rewriting, and the use of decision procedures for propositional and arithmetical simplification.

[1] PVS is free available, see `http://www.csl.sri.com/sri-csl-pvs.html`

The focus of this paper is the formulation of Hoare logic, which forms the basis of our method, in the specification language of PVS. Using a simple sequential programming language, two approaches are presented.

- Section 2 shows the conventional approach, with a clear distinction between the syntax of programs and their semantics.
- In section 3 programs are identified with their semantics. By doing the same for specifications, a mixed framework is obtained where programming constructs and specifications are integrated (similar to e.g. [Old85]). This makes it easy to express the intermediate stages during top-down program design.

Although there is a possibility in PVS to generate LaTeX output, in this paper the plain text of the PVS specification language is shown, to give an impression of the user interface of PVS. Concluding remarks can be found in section 4.

2 Separate syntax and semantics of programs

In the first approach we define the syntax of programs using the powerful mechanism of PVS for abstract datatypes. To show the basic concepts we use a very simple sequential language with only one basic statement, assignment, and one compound construct, sequential composition. Hence datatype `program`, specified below, contains two *constructors* `assign` and `seq` and two corresponding *recognizers* `assgn?` and `sequent?` which can be used to construct subtypes of `program`. The construct `assign` has two *accessors*, `vari` representing the variable of the assignment and `expr` which maps a state (a function from variables to values, as shown below) to a value. Similarly, `seq` has two accessors `seq1` and `seq2`. Note that the datatype is parameterized by types for values and variables.

```
program [ Value : TYPE, Vars : TYPE ] : DATATYPE
BEGIN
assign (vari : Vars, expr : [[Vars -> Value] -> Value]): assgn?

seq (seq1 : program, seq2 : program ): sequent?
END program
```

Type-checking this datatype, the PVS system generates theory `program_adt`, with e.g. induction on the structure of programs, and `program_adt_reduce`, with functions for recursive definitions.

The semantics of programs is defined in theory `sem`. It is based on the notion of a state, a function from variables to values, and an action which is a relation on states (to allow non-determinism in extended languages with, for instance, parallelism). The semantics of programs, represented by the function `[||]`, is then defined as a recursive function, using `reduce` from imported theory `program_adt_reduce`. This function has two arguments, corresponding to the semantics of the two constructors. The notation `f WITH [ (x) := y ]` specifies the function which equals `f` except for the value of `x` which is overridden by `y`.

The lemmas with labels `sem_assign` and `sem_seq` give an alternative formulation. They can be proved in PVS by one command, namely `(grind)`. The special operator `[||]` allows us to write, e.g., `[| prog1 |]` instead of `[||](prog1)`.

```
sem [ Value : TYPE, Vars : TYPE ] : THEORY
BEGIN
State  : TYPE = [Vars -> Value]
Action : TYPE = [State, State -> bool]

IMPORTING program_adt_reduce [Value,Vars,Action]

s, s0, s1         : VAR State
prog1, prog2      : VAR program
act1, act2        : VAR Action
vvar              : VAR Vars
exp               : VAR [State -> Value]

[||] : [ program -> Action ] = reduce (

   (LAMBDA vvar, exp :
        (LAMBDA s0, s1 : s1 = s0 WITH [(vvar) := exp(s0)])),

   (LAMBDA act1, act2 :       % given semantics of seq components
        (LAMBDA s0, s1 : (EXISTS s : act1(s0,s) AND act2(s,s1)))))

sem_assign : LEMMA [| assign(vvar,exp) |](s0,s1)
                   IFF s1 = s0 WITH [(vvar) := exp(s0)]

sem_seq : LEMMA [| seq(prog1,prog2) |](s0,s1) IFF
            (EXISTS s : [| prog1 |](s0,s) AND [| prog2 |](s,s1))
END sem
```

Hoare triples are represented by a record with three fields: precondition, program, and postcondition. Validity of such triples is defined according to the conventional notion of partial correctness. Then it is easy to prove the classical rule of sequential composition (by the `(grind)` command).

```
htrip [ Value : TYPE, Vars : TYPE ] : THEORY
BEGIN
IMPORTING sem [Value,Vars]
HoareTrips : TYPE = [# pre  : pred[State],
                       prog : program,
                       post : pred[State]
                      #]
ht          : VAR HoareTrips
s0, s1      : VAR State
p, q, r     : VAR pred[State]
```

```
pr1, pr2   : VAR program

Valid(ht)  : bool = (FORALL s0, s1 :
       pre(ht)(s0) AND [| prog(ht) |](s0,s1) IMPLIES post(ht)(s1))

seq_comp_rule : THEOREM
      Valid( (# pre := p, prog := pr1, post := r #) ) AND
      Valid( (# pre := r, prog := pr2, post := q #) )
      IMPLIES
      Valid( (# pre := p, prog := seq(pr1,pr2), post := q #) )
END htrip
```

Note that we do not have the classical assignment axiom; this would require the definition of the syntax of assertions, syntactical substitution, the interpretation of assertions, etc. As illustrated by the next example it is far more easier to use the semantics of Hoare triples for simple programs.

In example `ex1` the predefined type `int` of integers is used as the domain of values. Variables are represented as an enumeration type `x,y`, which is a simple version of an abstract datatype. This, implicitly, implies that `x` and `y` are different, a fact which is used by the PVS decision procedures. Then theorem `cor1` can be proved easily by the command `(grind)` which expands all definitions and uses decision procedures. Note that it is essential that `x` and `y` are different, otherwise it is not possible to show that the value of `y` is not affected.

```
ex1 : THEORY
BEGIN
Vars : TYPE = {x,y}
IMPORTING  htrip[int,Vars]
s   : VAR State
p   : pred[State] = (LAMBDA s: s(x)=2 AND s(y)=3)
q   : pred[State] = (LAMBDA s: s(x)=7 AND s(y)=3)

cor1: THEOREM Valid((# pre  := p,
                        prog := assign(x, LAMBDA s: s(x)*s(y)+1),
                        post := q
                      #) )

cor2: THEOREM Valid((# pre  := p,
                       prog := seq(assign(x, LAMBDA s: s(x)+s(y)),
                                   assign(x, LAMBDA s: s(x)+2)),
                       post := q
                     #) )
END ex1
```

For theorem `cor2` we could follow the usual syntactic approach; use the sequential composition rule (`(use "seq_comp_rule")` in PVS), then instantiate assertion `r` in formula `-1` by `(inst -1 "(LAMBDA s : s(x) = 5 AND s(y) = 3)")`, and prove the two sub cases for the two assignments by `(grind)`. However, also here it much easier to prove the theorem by simply using `(grind)`. This command expands all definitions and hence proves the theorem using the semantic definitions. Advantage is that no intermediate assertion has to be found.

This small example already indicates that it is not always convenient to copy the syntactic, paper-and-pencil, style of proving in PVS. To allow a fast use of the semantics, it is desirable to avoid deep semantic encodings with many definitions that have to be expanded. This observation leads to the framework of the next section.

3 Identifying syntax and semantics of programs

To simplify the use of program semantics in proofs, we present here the extreme case where a program is identified with its semantics. This means that a program simply *is* a relation on states. Below we also define a while construct, using a finite sequence (a predefined type) of states.

```
prog [ Value : TYPE, Vars : TYPE ] : THEORY
BEGIN
State   : TYPE = [Vars -> Value]
Program : TYPE = [State, State -> bool]

s, s0, s1          : VAR State
vvar               : VAR Vars
exp                : VAR [State -> Value]
b                  : VAR [State -> bool]
prog, prog1, prog2 : VAR Program

assign(vvar,exp) : Program = (LAMBDA s0, s1 :
   s1 = s0 WITH [(vvar) := exp(s0)])

seq(prog1,prog2) : Program = (LAMBDA s0, s1 :
   (EXISTS s : prog1(s0,s) AND prog2(s,s1)) )

while(b,prog) : Program = (LAMBDA s0, s1 :
   (EXISTS (fs :finite_sequence[State]) : length(fs) > 0 AND
       LET fseq = seq(fs), k = length(fs)-1
       IN s0 = fseq(0) AND s1 = fseq(k) AND NOT b(fseq(k)) AND
          (FORALL (j : nat | j < k) : b(fseq(j)) AND
                                      prog(fseq(j),fseq(j+1)) )))
```

Assertions are state predicates and can be combined by the overloaded operators `NOT` and `AND`. An assertion is valid if it holds for all states. Next we define a specification, consisting of a pre- and a postcondition, as a program. Hence it can be used, for instance, as part of a sequential composition or while construct.

The infix operator `=>` is overloaded to represent program refinement. It is easy to show that this relation is reflexive and transitive (again using `(grind)`).

```
p, q : VAR pred[State]

NOT : [pred[State] -> pred[State]] =
                  (LAMBDA p : (LAMBDA s : NOT p(s) ) ) ;

AND : [pred[State], pred[State] -> pred[State]] =
                  (LAMBDA p, q : (LAMBDA s : p(s) AND q(s) ) ) ;

Valid : [pred[State] -> bool] = (LAMBDA p : (FORALL s : p(s)) )

spec(p, q) : Program = (LAMBDA s0, s1 : p(s0) IMPLIES q(s1)) ;

=> : [Program, Program -> bool] = (LAMBDA prog1, prog2 :
        (FORALL s0, s1 : prog1(s0,s1) IMPLIES prog2(s0,s1) )) ;

ref_refl  : THEOREM prog => prog

ref_trans : THEOREM (prog1 => prog2) IFF
            (EXISTS prog : (prog1 => prog) AND (prog => prog2))
END prog
```

Theory `ex2` contains the simple examples of `ex1`, now formulated in the current framework. Again the theorems can be proved by `(grind)`, but now the proof is significantly faster than in the previous section.

```
ex2 : THEORY
BEGIN
Vars : TYPE = {x,y}
IMPORTING prog[int,Vars]
s    : VAR State
p    : pred[State] = (LAMBDA s: s(x)=2 AND s(y)=3)
q    : pred[State] = (LAMBDA s: s(x)=7 AND s(y)=3)

cor1 : THEOREM assign(x, LAMBDA s : s(x)*s(y)+1)  => spec(p, q)

cor2 : THEOREM seq(assign(x, LAMBDA s : s(x)+s(y)),
                   assign(x, LAMBDA s : s(x)+2))  => spec(p, q)
END ex2
```

Finally, we present a few proof rules for the refinement relation. The while rule is a good example of a proof rule which is much easier to use than the semantic formulation. The correctness of the rule depends on lemma `while_lemma` which can be proved by induction (PVS has a powerful `(induct-and-simplify)` command).

```
rules [ Value : TYPE, Vars : TYPE ] : THEORY
BEGIN
IMPORTING prog[Value,Vars]
p, p0, q, q0, r, I              : VAR pred[State]
b                                : VAR [State -> bool]
prog, prog1,prog2, prog3, prog4 : VAR Program
fs                               : VAR finite_sequence[State]
j                                : VAR nat

cons_rule : THEOREM (Valid(p IMPLIES p0) AND Valid(q0 IMPLIES q))
                     IMPLIES
                    (spec(p0,q0) => spec(p,q))

seq_comp_rule : THEOREM seq( spec(p,r), spec(r,q) ) => spec(p,q)

while_lemma   : LEMMA
   length(fs) > 0 AND p(seq(fs)(0)) AND
   (FORALL (j | j < length(fs)-1) : b(seq(fs)(j)) AND
                     spec(p AND b, p)(seq(fs)(j),seq(fs)(j+1)) )
   IMPLIES
   (FORALL j : j <= length(fs)-1 IMPLIES p(seq(fs)(j)) )

while_rule : THEOREM while(b, spec(I AND b, I) )
                     => spec(I, I AND NOT b)

mono_seq   : THEOREM (prog3 => prog1) AND (prog4 => prog2)
                      IMPLIES
                     (seq(prog3,prog4) => seq(prog1,prog2))

mono_while : THEOREM (prog => prog1)
                      IMPLIES
                     (while(b, prog) => while(b, prog1))
END rules
```

4 Concluding Remarks

The framework of section 3 describes the main principles behind a formal method for the design of distributed real-time systems, as described in [Hoo94b]. Compared to that paper, notation and proofs could be improved due to the use of a more powerful version of PVS. For instance, it is no longer needed to define a special strategy for the verification of sequential programs without while constructs, since this can now be done directly by the `(grind)` command.

In [Hoo94b] the framework has been used for the top-down design of a chemical batch processing example, where all design steps have been proved correct by the interactive theorem prover of PVS. Based on this paradigm, also a steam boiler control system has been designed and verified in PVS [VH96].

Whereas the current paper, and parts of the examples mentioned above, deal with program verification, the designers of PVS emphasize the use of this tool during the early phases of system design, when the requirements are formulated and formalized [Rus95]. This is motivated by the fact that most errors are often due to mistakes in the requirements specification and high-level design.

Related own work in this field consists of the specification and verification of a part of the ACCESS.bus protocol [Hoo95], starting from an informal description. The translation from an informal to a formal specification was also addressed in the RPC-Memory specification problem [Hoo96b][2].

References

[Hoo94a] J. Hooman. Compositional verification of a distributed real-time arbitration protocol. *Real-Time Systems*, 6(2):173–205, 1994.

[Hoo94b] J. Hooman. Correctness of real time systems by construction. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40. LNCS 863, Springer-Verlag, 1994.

[Hoo94c] J. Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–825, 1994.

[Hoo95] J. Hooman. Verifying part of the ACCESS.bus protocol using PVS. In *Proc. 15th Conf. on the Foundations of Software Technology and Theoretical Computer Science*, pages 96–110. LNCS 1026, Springer-Verlag, 1995.

[Hoo96a] J. Hooman. Assertional specification and verification. In M. Joseph, editor, *Real-time Systems: Specification, Verification and Analysis*, chapter 5, pages 97–146. Prentice Hall, 1996.

[Hoo96b] J. Hooman. Using PVS for an assertional verification of the RPC-memory specification problem. In *Proc. Dagstuhl-seminar of the RPC-Memory Specification Problem*, to appear. LNCS, Springer-Verlag, 1996.

[Old85] E. R. Olderog. Process theory: semantics, specification and verification. In *ESPRIT/LPC Advanced School on Current Trends in Concurrency*, pages 509–519. LNCS 194, Springer-Verlag, 1985.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[Rus95] John Rushby. Mechanizing formal methods: Opportunities and challenges. In *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, pages 105–113. LNCS 967, Springer-Verlag, 1995.

[VH96] J. Vitt and J. Hooman. Assertional specification and verification using PVS of the steam boiler control system. In *Steam-Boiler Case Study Book*, to appear. LNCS, Springer-Verlag, 1996.

[2] Recent work is available on `http://www.win.tue.nl/win/cs/tt/hooman/JH.html`