A Brief Introduction to PVS

Tamarah Arons
tamarah@wisdom.weizmann.ac.il

6 June, 2002

What is PVS

- PVS is an extensive higher-order-logic deductive verification system based on sequent calculus.
- Specification language for writing theorems
- Interactive prover
- Full documentation at ~verify/PVS2-4.1/Doc/

Logic of PVS

The prover maintains a proof tree.

- Root is the premise to be proved
- Offspring follow from a node by a proof step
- Tree is complete when all leaves are true
- Each node is proof goal
- Each proof goal is a sequent

Sequents

A sequent is comprised of sequent formulas : antecedents followed by consequents.

Represented in the form

$p_1, p_2, \ldots, p_n \vdash q_1, q_2, \ldots, q_m$,

where p_i are the antecedents, q_i are the consequents.

Interpretation:

$$\forall free : (p_1 \wedge p_2 \wedge \ldots \wedge p_n) \rightarrow (q_1 \vee q_2 \vee \ldots \vee q_m)$$

where *free* denotes the free (unbound) variables.

Sequent Axioms

Used to prove that the leaf sequents are true

$$\begin{array}{llcl} \text{A1}: & \Gamma, p & \vdash & \Delta, p \\ \text{A2}: & \Gamma & \vdash & \Delta, \text{T} \\ \text{A3}: & \Gamma, \text{F} & \vdash & \Delta \end{array}$$

I.e. A1 : $\Gamma \wedge p \rightarrow \Delta \vee p$

A2 : $\Gamma \rightarrow \Delta \vee \text{T}$

$$\begin{array}{lll} \text{A3:} & \Gamma \wedge \text{F} \rightarrow \Delta & \equiv \\ & \neg(\Gamma \wedge \text{F}) \vee \Delta & \equiv \\ & \neg\Gamma \vee \neg\text{F} \vee \Delta & \equiv \\ & \neg\Gamma \vee \text{T} \vee \Delta & \equiv \\ & \text{T} & \end{array}$$

Corresponds to PVS `assert` command.

Propositional Rules

$$\frac{\Gamma \vdash \Delta, p \vee q}{\Gamma \vdash \Delta, p, q} \qquad \frac{\Gamma, p \wedge q \vdash \Delta}{\Gamma, p, q \vdash \Delta}$$

$$\frac{\Gamma, \neg p \vdash \Delta}{\Gamma \vdash \Delta, p} \qquad \frac{\Gamma \vdash \Delta, \neg p}{\Gamma, p \vdash \Delta}$$

Correspond to the PVS `flatten` command.

$$\frac{\Gamma, p \vee q \vdash \Delta}{\Gamma, p \vdash \Delta \qquad \Gamma, q \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, p \wedge q}{\Gamma \vdash \Delta, p \qquad \Gamma \vdash \Delta, q}$$

These expanding rules correspond to the PVS `split` command.

Quantifier Rules

Skolemization (`skolem!`, `skosimp*`)

Requires that t be a new constant that does not occur in the sequent

$$\frac{\Gamma, (\exists x : p) \vdash \Delta}{\Gamma, p\{x \leftarrow t\} \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, (\forall x : p)}{\Gamma \vdash \Delta, p\{x \leftarrow t\}}$$

Instantiation (`inst`, `inst-cp`)

$$\frac{\Gamma, (\forall x : p) \vdash \Delta}{\Gamma, (\forall x : p), p\{x \leftarrow t\} \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, (\exists x : p)}{\Gamma \vdash \Delta, (\exists x : p), p\{x \leftarrow t}$$

Strengthening Rules

Allow a stronger sequent to be derived from a weaker one by removing formulas

$$\frac{\Gamma, p \vdash \Delta}{\Gamma \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, p}{\Gamma \vdash \Delta}$$

Example

$$\varphi : (\forall x : P(x) \vee \neg Q(x)) \rightarrow (\exists y : P(y)) \vee (\forall z : \neg Q(z))$$

$\vdash \neg(\forall x : P(x) \vee \neg Q(x)) \vee (\exists y : P(y)) \vee (\forall z : \neg Q(z))$

`flatten`

$\vdash \neg(\forall x : P(x) \vee \neg Q(x)), (\exists y : P(y)), (\forall z : \neg Q(z))$

`flatten`

$(\forall x : P(x) \vee \neg Q(x)) \vdash (\exists y : P(y)), (\forall z : \neg Q(z))$

`skolemize`

$(\forall x : P(x) \vee \neg Q(x)) \vdash (\exists y : P(y)), \neg Q(a)$

`flatten`

$(\forall x : P(x) \vee \neg Q(x)), Q(a) \vdash (\exists y : P(y))$

`instantiate` x with a

$P(a) \vee \neg Q(a), Q(a) \vdash (\exists y : P(y))$

`split`

$P(a), Q(a) \vdash (\exists y : P(y))$ — `instantiate` y with a — $P(a), Q(a) \vdash P(a)$ — T - Axiom A1

$\neg Q(a), Q(a) \vdash (\exists y : P(y))$ — `flatten` — $Q(a) \vdash (\exists y : P(y)), Q(a)$ — T - Axiom A1

Basic Definitions in PVS

Specification files: text files containing theories. Include system definitions and lemmas. Extension `.pvs`.

Proof files save proofs that have been composed. Extension `.prf`.

Context: Set of specification and proof files in one directory.

Interface: Emacs editor.

Example - reservations

```
reservation:  THEORY
BEGIN
   room:  TYPE
   date:  TYPE
   name:  TYPE
   free:  name
   reservations:  TYPE = [room, date → name]
   reserve(r:room, d:date, n:name, reg:reservations):
      reservations = reg WITH [(r, d) := n]
   cancel(r:room, d:date, reg:reservations):
      reservations = reg WITH [(r, d) := free]
   reserved(r:room, d:date, reg:reservations):  bool=
      reg(r, d) ≠ free
END reservation
```

- `room`, `date`, `name` are uninterpreted types
- `free` is a constant
- `reservations` is a function type
- `reserve`, `reserved`, `cancel` are interpreted functions

Proving Lemmas

reserved(r, d, reg): bool $=$ reg(r, d) $\neq$ free
cancel(r, d, reg): reservations $=$
  reg WITH [(r, d) := free]

canceled_not_reserved: LEMMA
  $\forall$ r, d, reg: $\neg$ reserved(r, d, cancel(r, d, reg))

```
   |-------
{1}   FORALL r, d, reg:
          NOT reserved(r, d, cancel(r, d, reg))
Rule? (skosimp*)

{-1}  reserved(r!1, d!1, cancel(r!1, d!1, reg!1))
  |-------
Rule? (expand "reserved")

{-1}  cancel(r!1, d!1, reg!1)(r!1, d!1) /= free
  |-------
Rule? (expand "cancel")

{-1}  FALSE
  |-------
which is trivially true.
Q.E.D.
```


Proving Lemmas - ctd

Alternatively, the grind command would have proved this lemma.

grind is a strategy that expands definitions, skolemizes, instantiates, simplifies ...

It can often be used to complete a proof.

Another lemma

reserved(r, d, reg): bool $=$ reg(r, d) $\neq$ free
reserve(r, d, n, reg): reservations $=$
  reg WITH [(r, d) $:=$ n]

is_reserved: LEMMA
  $\forall$ r, d, n, reg:
    reserved(r, d, reserve(r, d, n, reg))

```
 is_reserved :
  |-------
{1}   FORALL r, d, n, reg:
        reserved(r, d, reserve(r, d, n, reg))
Rule? (skosimp*)
  |-------
{1}   reserved(r!1, d!1, reserve(r!1, d!1, n!1, reg!1)
Rule? (expand "reserved")
  |-------
{1}   reserve(r!1, d!1, n!1, reg!1)(r!1, d!1) /= free

Rule? (expand "reserve")
{-1}  n!1 = free
  |-------
Rule?
```


LTL framework

A set of PVS theories and strategies defining basic LTL constructs, and proof rules like BINV.

Example MUX-SEM

$$\begin{array}{|l|}\hline
\begin{array}{llcl}
\textbf{in} & N & : & \textbf{integer where } N > 1 \\
\textbf{local} & y & : & \{0,1\} \textbf{ where } y = 1
\end{array} \\[1em]
\overset{N}{\underset{p=1}{\|}} P[p] :: \left[\begin{array}{ll}
\ell_0 : & \textbf{loop forever do} \\
& \left[\begin{array}{ll}
\ell_1 : & \textbf{noncritical} \\
\ell_2 : & \textbf{request } y \\
\ell_3 : & \textbf{critical} \\
\ell_4 : & \textbf{release } y
\end{array}\right]
\end{array}\right] \\ \hline
\end{array}$$

Figure 1: Parameterized MUX-SEM

Proving Mutual Exclusion

```
reachable: ASSERTION =
  ...
  (∀ (i: PROC_ID): ∀ (j: PROC_ID):
     (loc(i) > 2 → y = 0) ∧
     (i = j ∨ loc(i) < 3 ∨ loc(j) < 3))
```

Hints :

- Existential quantification is expensive, often requiring manual instantiation : avoid when possible
- Try to take universal quantifiers to the top level
- Disjunction ($\vee$) is more difficult to work with than conjunction ($\wedge$), often requiring manual splitting.