Machine Assisted Specification and Verification[a]

N. Shankar
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/

Computer Science Laboratory
SRI International
Menlo Park, CA

[a] Funding sources include SRI, NSF, NASA, NRL, AFOSR, and ARPA.

Course Overview

- *Language influences deduction.*
- A specification is useless unless it can be deductively manipulated and probed.
- Proof construction can be simplified by careful attention to language features, powerful low-level automation, and flexible high-level strategies.
- PVS is a system that attempts to synergistically integrate an expressive language with powerful automated deduction.
- We review the specification language and proof checking capabilities of PVS.
- And present methodologies for formalizing and verifying examples from areas such as program transformation, fault tolerance, concurrent and reactive systems, and computer hardware.

Why Specify?

- A *specification* is a (possibly partial) characterization of "correct" system behavior.
- It describes *what* the system computes rather than *how* it is computed, e.g., a sorting operation returns an ordered permutation of its input.
- A specification explicates the input and environment assumptions that are needed to guarantee correct behavior.

What can one do with a specification?

- We can *test* whether the implementation violates the specification.
- Or *verify* that the implementation meets its specification.
- Or systematically *derive* the implementation from the specification.
- Or show that the specification is unimplementable or *inconsistent*.
- Or explore alternative assumptions and design choices.

Why Verify?

- Verification is the demonstration that a system implementation meets its specification.
- Errors can appear in either or both of the specification and the implementation.
- In discrete systems, small errors can have large consequences.
- "*Testing* can reveal the presence of errors but never their absence."

Why Verify?

Verification too typically only reveals the presence of errors, but:

- Testing is infeasible on large input or state spaces.
- Untested errors go undetected.

whereas

- Verification is less sensitive to the size of the input space.
- It can uncover unanticipated errors.
- And can handle partial implementations and nontestable specifications.

Why be Formal?

- A formal notation has a precise and unambiguous syntax, semantics, and proof theory.
- Verification can therefore be carried out by formal calculation.
- Calculation can reveal counterintuitive errors.
- Formalism provides the basis for establishing metatheoretic properties like soundness, completeness, and decidability.
- Formalism is a prerequisite for mechanization.

Why Mechanize?

- Mechanization makes it convenient to carry out difficult formal calculations.
- It can reveal flaws in a formal calculation.
- Mechanization provides *speed*, *accuracy*, *repeatability*, and *extensibility*.
- Unmechanized formal methods are neither formal nor methods.

Can Formal Methods be Mechanized?

Yes. Plenty of positive evidence:

- Model checking and state exploration have revealed numerous bugs in finite-state controllers [McMillan].
- Nqthm (the Boyer-Moore theorem prover) has been used to mechanically verify a *stack* consisting of processor, assembler, and compiler[Bevier & Hunt & Moore & Young].
- Ehdm verification of Lamport and Melliar-Smith's Interactive Convergence Clock Synchronization scheme revealed serious flaws in their informal proof [Rushby & von Henke].

Numerous Successful Mechanized Tools

Model Checkers: SMV, COSPAN, Concurrency Workbench, Unity Verifier, CESAR/ALDEBARAN.

State Explorers: SPIN, Murphi.

Hybrid Systems Checkers: HyTech, Kronos.

Proof Checkers: HOL, Isabelle, Coq, Nuprl, KIV, EVES, IMPS, Larch/LP, TLP.

Theorem Provers: Nqthm, Otter, RRL, SETHEO, SPIKE.

Course Overview

- **Goal:** To provide an introduction to formalization, formal specification, and mechanized verification.
- We will examine specific applications areas such as program transformation, hardware, and fault-tolerant, real time, and reactive systems.
- We will examine issues and challenges related to specification language design and effective theorem proving support.
- PVS is used for illustrating certain points and claims regarding mechanization. Similar or stronger claims can be made for many other mechanized tools.

What is PVS?

PVS (Prototype Verification System) is an integrated environment for logical specification and deductive verification.

It is intended for

1. Writing clear and logically meaningful specifications
2. Statically checking specifications to reveal common syntactic and semantic errors.
3. Constructing readable proofs of correctness that have been mechanically verified with minimum overhead human effort.
4. Deriving useful diagnostic information from failed proofs.
5. Presenting specifications and proofs in a form acceptable for human communication.

What is PVS? Language + Deduction

PVS is an experiment aimed at studying the interaction between language and deduction. Many language constructs only make sense in the context of a theorem prover:

- Parametric theories with assumptions
- Predicate subtyping and dependent types
- Typing Judgements
- Abstract datatypes

PVS typechecking is undecidable, but ...

The PVS typechecker checks simple correctness and generates proof obligations.

Most such proof obligations are automatically discharged using decision procedures.

Main challenge is in achieving a useful synergy between language and deduction.

Example of PVS Language

```
half: THEORY
  BEGIN

  even_nat : TYPE = {i : nat | even?(i)}

  i, j, k: VAR even_nat

  half(i): RECURSIVE nat =
      (IF i = 0 THEN 0
       ELSIF i = 1 THEN 0
       ELSE half(i - 2) + 1 ENDIF)
     MEASURE i

  JUDGEMENT 2 HAS_TYPE even_nat
  JUDGEMENT * HAS_TYPE [nat,      even_nat ->  even_nat]
  JUDGEMENT * HAS_TYPE [even_nat, nat      ->  even_nat]

  half_halves: THEOREM
    (FORALL (i: nat) : half(2 * i) = i)

  half_half: THEOREM
    (FORALL (i: nat): half(2 * half(2 * i)) = i)

  END half
```


Type Correctness Conditions

```
% Subtype TCC generated (line 9) for i - 2
half_TCC1: OBLIGATION
      (FORALL (i):
         NOT i = 0 AND NOT i = 1
       IMPLIES i - 2 >= 0 AND even?(i - 2));

% Termination TCC generated (line 9) for half
half_TCC2: OBLIGATION
      (FORALL (i): NOT i = 0 AND NOT i = 1
          IMPLIES i - 2 < i);

% Subtype TCC generated for 2
two_TCC1: OBLIGATION even?(2);

% Subtype TCC generated for x1 * x2
times_TCC1: OBLIGATION
      (FORALL (x1: nat, x2: even_nat): even?(x1 * x2));

% Subtype TCC generated for x1 * x2
times_TCC2: OBLIGATION
      (FORALL (x1: even_nat, x2: nat): even?(x1 * x2));
```


PVS: Higher-order Logic + Parametric Theories

```
functions [D, R: TYPE]: THEORY
 BEGIN
  f, g: VAR [D -> R]
  x, x1, x2: VAR D
  y: VAR R
  extensionality: POSTULATE
     (FORALL (x: D): f(x) = g(x)) IMPLIES f = g

  functionality: LEMMA
     f = g AND x1 = x2 IMPLIES f(x1) = g(x2)

  injective?(f): bool =
     (FORALL x1, x2: f(x1) = f(x2) => (x1 = x2))

  surjective?(f): bool = (FORALL y: (EXISTS x: f(x) = y))

  bijective?(f): bool = injective?(f) & surjective?(f)

  domain(f): TYPE = D

  range(f): TYPE = R

  graph(f)(x, y): bool = (f(x) = y)
 END functions
```

What is PVS? Decision Procedures + Strategies + Interaction

An attempt to build an interactive proof checker on top of powerful theorem proving technology.

PVS uses:

- Decision procedures for linear arithmetic and equality [Shostak][CADE'96].
- Rewriting (with caching) [TPCD'94].
- Datatype simplification.
- BDD-based propositional simplification [TPCD'94].
- BDD-based symbolic model-checking [CAV'95].

Decision procedures are heavily used in simplification and rewriting.

Main challenge is in coherently combining interaction and automation.

PVS: Strategies

A simple strategy language used to define strategies for induction/rewriting and brute-force theorem proving:

- `try` for backtracking
- `if` for conditional strategies
- `let` for invoking Lisp
- Recursion

```
(defstep prop ()
  (try (flatten) (prop$) (try (split)(prop$) (skip)))
  "A black-box rule for propositional simplification."
  "Applying propositional simplification")
```

Frequently used strategies include `GRIND`, `INDUCT-AND-SIMPLIFY`, and `INST?`.

Equality and Arithmetic Decision Procedures

```
decisions: THEORY
  BEGIN
   x,y,v: VAR real
   f: [real -> real]  %Illustrates congruence closure
   eq1: THEOREM x = f(x) IMPLIES f(f(f(x))) = x
   g : [real, real -> real]
   eq2: THEOREM  % Illustrates arithmetic/equality
       x = f(y)  % combination
       IMPLIES g(f(y + 2 - 2), x + 2) = g(x, f(y) + 2)

   arith: THEOREM  %Proved by decision procedures
     x < 2*y AND y < 3*v IMPLIES 3*x < 18*v

   badarith: CONJECTURE %Not proved; statement false.
     x < 2*y AND y < 3*v IMPLIES 3*x < 17*v
   badarith1: CONJECTURE %Not proved automatically;
     x<0 AND y<0 IMPLIES x*y>0   %statement nonlinear

   i, j, k: VAR int
   intarith: THEOREM %Proved by decision procedures
     2*i < 5 AND i > 1 IMPLIES i = 2
   badarith2: CONJECTURE  %Not proved; not true
     2*x < 5 AND x > 1 IMPLIES x = 2
  END decisions
```


Arithmetic Decision Procedures: Example

```
sum: THEORY
  BEGIN

   n: VAR nat
   f, g: VAR [nat -> nat]

   sum(f, n): RECURSIVE nat =
  (IF n = 0
      THEN 0
      ELSE f(n-1) + sum(f, n - 1)
   ENDIF)
  MEASURE n

  square(n): nat = n*n

  sum_of_squares: LEMMA
    6 * sum(square, n+1) = n * (n + 1) * (2*n + 1)

 END sum
```

PVS Proof of Sum of Squares

```
sum_of_squares :
  |-------
{1}   (FORALL (n: nat): 6 * sum(square, n + 1)
        = n * (n + 1) * (2 * n + 1))
Rule? (induct-and-simplify "n")
square rewrites square(0)
  to 0
sum rewrites sum(square, 0)
  to 0
sum rewrites sum(square, 1)
  to 0
square rewrites square(j!1)
  to j!1 * j!1
sum rewrites sum(square, 1 + j!1)
  to sum(square, j!1) + j!1 * j!1
square rewrites square(1 + j!1)
  to 1 + j!1 + (j!1 * j!1 + j!1)
sum rewrites sum(square, 2 + j!1)
  to 1 + sum(square, j!1) + 2 * (j!1 * j!1) + j!1 + j!1
By induction on n, and by repeatedly rewriting and
simplifying,
Q.E.D.

Run time  = 38.66 secs.
Real time = 55.26 secs.
```


An Interactive Proof

To show that any postage amount above 7 cents can be achieved using only some combination of 3 and 5 cent stamps.

The proof is by induction on the postage amount.

Base: 8 cents of postage requires one 3 and one 5 cent stamp.

Induction: We have 3 and 5 cent stamps adding up to n cents of postage for $n \geq 8$.

If there is a 5 cent stamp in the postage for n, then we discard it and add two 3 cent stamps.

If there isn't a 5 cent stamp, then there must be at least three 3 cent stamps. Discard these and add two 5 cent stamps.

Formalizing stamps in PVS

```
stamps: THEORY
  BEGIN

  i, threes, fives, j, k: VAR nat

  stamps: LEMMA
    (EXISTS threes, fives:
        i + 8 = 3 * threes + 5 * fives)

  END stamps
```


The Interactive Proof

```
stamps :

  |-------
{1}   (FORALL (i: nat):
         (EXISTS (threes: nat), (fives: nat):
              i + 8 = 3 * threes + 5 * fives))

Rule? (induct "i")
Inducting on i,
this yields  2 subgoals:
```

The Base Case

```
stamps.1 :

  |-------
{1}   (EXISTS (threes: nat), (fives: nat):
            0 + 8 = 3 * threes + 5 * fives)

Rule? (inst + 1 1)
Instantiating the top quantifier in + with the terms:
 1, 1,
this simplifies to:
stamps.1 :

  |-------
{1}   0 + 8 = 3 * 1 + 5 * 1

Rule? (assert)
Simplifying, rewriting, and recording with
decision procedures,

This completes the proof of stamps.1.
```


The Induction Step

```
stamps.2 :

  |-------
{1}   (FORALL (j: nat):
        (EXISTS (threes: nat), (fives: nat):
          j + 8 = 3 * threes + 5 * fives)
           IMPLIES
          (EXISTS (threes: nat), (fives: nat):
             j + 1 + 8 = 3 * threes + 5 * fives))

Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
stamps.2 :

{-1}   j!1 + 8 = 3 * threes!1 + 5 * fives!1
  |-------
{1}   (EXISTS (threes: nat), (fives: nat):
         j!1 + 1 + 8 = 3 * threes + 5 * fives)

Rule? (case "fives!1 = 0")
Case splitting on
   fives!1 = 0,
this yields  2 subgoals:
```


If there are no 5 cent stamps

```
stamps.2.1 :

{-1}   fives!1 = 0
[-2]   j!1 + 8 = 3 * threes!1 + 5 * fives!1
  |-------
[1]   (EXISTS (threes: nat), (fives: nat):
            j!1 + 1 + 8 = 3 * threes + 5 * fives)

Rule? (inst + "threes!1 - 3" 2)
Instantiating the top quantifier in + with the terms:
 threes!1 - 3, 2,
this yields  2 subgoals:
stamps.2.1.1 :

[-1]   fives!1 = 0
[-2]   j!1 + 8 = 3 * threes!1 + 5 * fives!1
  |-------
{1}   j!1 + 1 + 8 = 3 * (threes!1 - 3) + 5 * 2

Rule? (assert)
Simplifying, rewriting, and recording with
decision procedures,

This completes the proof of stamps.2.1.1.
```


Then there must be three 3 cent stamps

```
stamps.2.1.2 (TCC):

[-1]   fives!1 = 0
[-2]   j!1 + 8 = 3 * threes!1 + 5 * fives!1
  |-------
{1}   threes!1 - 3 >= 0

Rule? (assert)
Simplifying, rewriting, and recording with
decision procedures,

This completes the proof of stamps.2.1.2.


This completes the proof of stamps.2.1.
```

If there is a 5 cent stamp

```
stamps.2.2 :

[-1]   j!1 + 8 = 3 * threes!1 + 5 * fives!1
  |-------
{1}   fives!1 = 0
[2]   (EXISTS (threes: nat), (fives: nat):
          j!1 + 1 + 8 = 3 * threes + 5 * fives)

Rule? (inst + "threes!1 + 2" "fives!1 - 1")
Instantiating the top quantifier in + with the terms:
 threes!1 + 2, fives!1 - 1,
this yields  2 subgoals:
stamps.2.2.1 :

[-1]   j!1 + 8 = 3 * threes!1 + 5 * fives!1
  |-------
[1]   fives!1 = 0
{2}   j!1 + 1 + 8 =
         3 * (threes!1 + 2) + 5 * (fives!1 - 1)

Rule? (assert)
Simplifying, rewriting, and recording with
decision procedures,

This completes the proof of stamps.2.2.1.
```


Then it is okay to remove a 5 cent stamp

```
stamps.2.2.2 (TCC):

[-1]   j!1 + 8 = 3 * threes!1 + 5 * fives!1
  |-------
{1}   fives!1 - 1 >= 0
[2]   fives!1 = 0

Rule? (assert)
Simplifying, rewriting, and recording with
decision procedures,

This completes the proof of stamps.2.2.2.


This completes the proof of stamps.2.2.


This completes the proof of stamps.2.

Q.E.D.


Run time  = 8.46 secs.
Real time = 94.04 secs.
```


What has PVS been used for? Small Exercises

- Boyer-Moore Majority Algorithm
- Binomial theorem
- Ramsey theorem
- Arithmetic-Geometric Mean
- Schröder-Bernstein theorem
- Hardware: N-bit adder, N-bit ALU
- Rabin-Scott NFA-DFA equivalence
- Program transformation: Bird's Fusion Theorem, Wand's continuation-based transformation [SCP'96]

PVS: Larger Exercises

- Rockwell-Collins' AAMP5 processor verification [Miller&Srivas, CHDL'95]
- Hybrid Byzantine Agreement algorithms [Lincoln&Rushby, FTCS'93]
- Real-time/concurrency examples [CAV'93]
- Real-time Hoare logic [Hooman, FTRTFT'94]
- Embedding Duration Calculus [Skakkebæk, FTRTFT'94]
- Invariant Discovery [Bensalem&Lakhnech&Saidi, CAV'96]
- Libraries for bit-vectors, finite sets, cardinalities [Butler, Jamsek, *et al*], analysis [Dutertre]
- IEEE-854 floating point standard specification [Miner]
- SRT Division [CAV'96]

Conclusions

- Language influences deduction.
- The goal of mechanization should be to combine an expressive specification language with powerful interactive and automated deductive support.
- PVS is a collaborative effort principally involving Sam Owre, and including John Rushby, David Cyrluk, Pat Lincoln, Mandayam Srivas, and Judy Crow, at SRI, with contributions from Sree Rajan (Fujitsu), Carl Witty (Newton Labs), Friedrich von Henke and Harald Ruess (Ulm), and Steven Phillips (AT&T), Ricky Butler and Paul Miner (NASA), Myla Archer and Ralph Jeffords (NRL), Damir Jamsek and Doug Hoover (ORA), Steve Miller and Dave Greve (Rockwell-Collins), Bruno Dutertre (RHNBC), Jozef Hooman (Eindhoven), Klaus Havelund (LITP), and Geert Janssen (Eindhoven), among many others.