PVS Formalization of Model Checking Intermediate Language

Natarajan Shankar
with Laura Gamboa (ISU), Chris Johansson (ISU),
Yi Lin (Rice), Karthik Nukula (CMU)

Computer Science Laboratory
SRI International
Menlo Park, CA

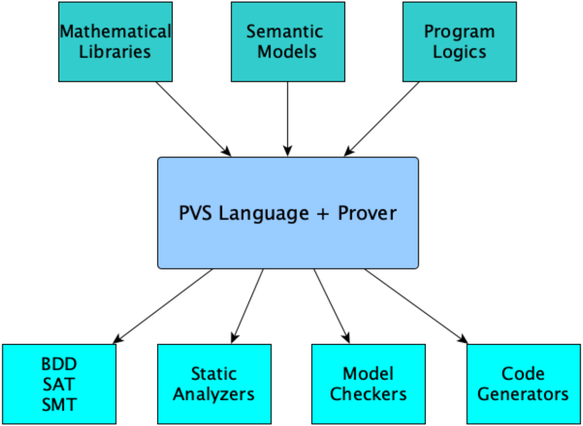
Oct 24, 2023

Brief Introduction to PVS

- PVS is an interactive proof assistant based on higher-order logic developed at SRI over the last three decades.
- PVS is also a *research prototype* for exploring ideas in formalization, automation, interaction, proof maintenance, and library construction.
- The interactive theorem prover combines automation (using SMT and other decision procedures) with interaction using powerful and robust proof commands that can be combined within proof strategies.
- *Almost all of the specification language is safely executable as a functional language, with code generators for Common Lisp, Clean, C, and Rust (with an ML generator in progress).*
- PVS is a *single language and proof platform* spanning formalization from mathematical modeling to practical system development.

PVS as an Interface Logic

- PVS blends an expressive specification language with an productive proof automation.
- It was essentially created as a semantic bridge between automated reasoning applications and automated tools.



The PVS Language in Brief

- A PVS specification is a collection of libraries.
  - Each library is a collection of files.
  - Each file is a sequence of theories.
  - Each theory is a sequence of declarations/definitions of types, constants, and formulas (Boolean expressions).
- Types include
  1. Booleans, number types
  2. Predicate subtypes: $\{x : T|p(x)\}$ for type T and predicate p.
  3. Dependent function $[x : D \to R(x)]$, tuple $[x : T_1, T_2(x)]$, and record $[\#a : T_1, b : T_2(x)\#]$ types.
  4. (Dependent) algebraic/coalgebraic datatypes: lists, ordinals.
- Expressions in PVS are
  1. Booleans, numbers
  2. Application : $f(a_1, \ldots, a_n)$
  3. Abstraction : $\lambda(x_1 : T_1, \ldots, x_n : T_n) : a$
  4. Tuples: $(a_1, \ldots, a_n)$, a‘3
  5. Records: $(\#l_1 := a_1, \ldots, l_n := a_n\#)$, a‘l_i
  6. Conditionals: IF a_1 THEN a_2 ELSE a_3 ENDIF
  7. Updates: a `WITH` $[(3)$‘1‘$age := 37]$.

PVS Example: Summation

```
hsummation: THEORY
 BEGIN
  i, m, n: VAR nat
  f: VAR [nat -> nat]

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

  id(n): nat = n
  hsum_id: LEMMA hsum(id)(n + 1) = (n * (n + 1)) / 2

  square(n): nat = n * n
  sum_of_squares: LEMMA 6 * hsum(square)(n + 1) = n * (n + 1) * (2 * n + 1)

  cube(n): nat = n * n * n
  sum_of_cubes: LEMMA 4 * hsum(cube)(n + 1) = n * n * (n + 1) * (n + 1)

  quart(n): nat = square(square(n))
  sum_of_quarts: LEMMA
    hsum(quart)(n + 1) =
     ((6 * (n ^ 5)) + (15 * (n ^ 4)) + (10 * (n ^ 3)) - n) / 30
 END hsummation
```

Transition Systems

- A transition system is a triple $\langle state, init, \texttt{trans} \rangle$ consisting of
  - A *state* type
  - An initialization predicate *init* on *state*
  - A transition relation *trans* on *state*
- This should be easily representable in PVS, so what's the problem?
- Transition systems/model checking was already integrated into PVS by 1994, with predicate abstraction added in 1997.
- However, writing transition systems directly in a specification logic can be painful.
- It is better to use the logic for semantic (deep or shallow) embedding of a transition system language.

Transition Systems

The complications in describing transition systems arise from

- Frame conditions
- Open versus Closed Systems
- Modules, module reuse, and module (multi-)composition
- Deadlock: Does every (reachable) state have a successor?
- Input Enabledness: Can any input be accepted in any state?
- Fairness: Which infinite behaviors are acceptable?
- Nondeterminism: Modelling needs nondeterminism, but implementations are deterministic (modulo inputs)
- Bounded vs. unbounded state
- Synchronous Interaction: Moore vs. Mealy machines.
- Refinement: When are the behaviors of one model subsumed by those of another?

While these can all be elegantly captured in a logic, the design choices are best captured in a bespoke language.

Transition System Languages

- UNITY
- TLA
- I/O Automata
- Reactive Modules
- SAL
- SMV, NuXmv
- MCMT, used by SALLY and Cubicle.
- Synchronous languages like Lustre, Signal, and Esterel.

Model Checking in PVS

CTL model checking was added to PVS in 1994/95.

```
ctlops[state : TYPE]: THEORY
 BEGIN
  u,v,w: VAR state
  f,g,Q,P,p1,p2: VAR pred[state]
  Z: VAR pred[[state, state]]
  N: VAR [state, state -> bool]
  CONVERSION+ K_conversion

  EX(N,f)(u):bool = (EXISTS v: (f(v) AND N(u, v)))
  EG(N,f):pred[state] = nu(LAMBDA Q: (f AND EX(N,Q)))
  EU(N,f,g):pred[state] = mu(LAMBDA Q: (g OR (f AND EX(N,Q))))

  EF(N,f):pred[state] = EU(N, TRUE, f)
  AX(N,f):pred[state] = NOT EX(N, NOT f)
  AF(N,f):pred[state] = NOT EG(N, NOT f)
  AG(N,f):pred[state] = NOT EF(N, NOT f)
  AU(N,f,g):pred[state]
    = NOT EU(N, NOT g, (NOT f AND NOT g)) AND AF(N, g)

  CONVERSION- K_conversion
 END ctlops
```

LTL in PVS

```
ltl  [state: TYPE+] : THEORY

  BEGIN

   ltlexpr: DATATYPE
   BEGIN
     atom(pred: [state -> bool]): atom?
     X(xarg: ltlexpr): X?
     LOR(orarg1, orarg2: ltlexpr): LOR?
     LNOT(notarg: ltlexpr): LNOT?
     G(garg: ltlexpr): G?
     U(uarg1, uarg2: ltlexpr): U?
   END ltlexpr

  ...
  END ltl
```

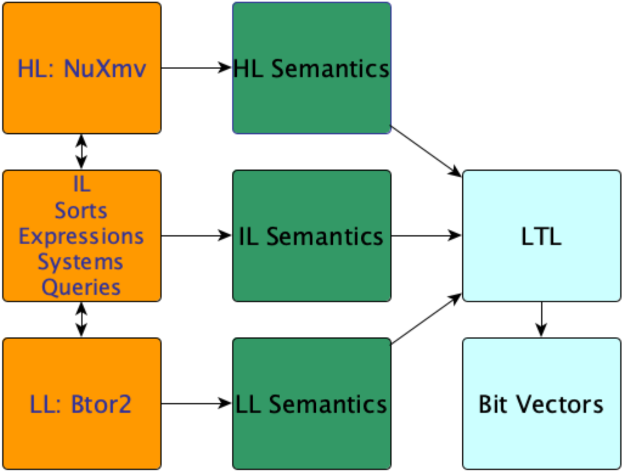
LTL in PVS

```
s, s1, s2: VAR sequence[state]
A, B, C: VAR ltlexpr
p, q, r: VAR [state -> bool]
i, j, k: VAR nat

models(s, A): RECURSIVE bool =
 (CASES A OF
  atom(p): p(s(0)),
  X(B): models(suffix(s, 1), B),
  LOR(B, C): models(s, B) OR models(s, C),
  LNOT(B): NOT models(s, B),
  G(B): FORALL i: models(suffix(s, i), B),
  U(B, C): EXISTS i: models(suffix(s, i), C) AND
                FORALL (j: below(i)): models(suffix(s, j), B)
  ENDCASES)
  MEASURE A BY <<
```

Roadmap

The goal with IL is to create a syntactic/semantic bridge between high-level and low-level languages mapping problems and witnesses.



Model Checking Intermediate Language

- The state of a system consists of its *input* $\bar{i}$, *local/state* $\bar{s}$, and *output* variables $\bar{o}$ (with SMT-LIB sorts).
- Additional constraints can be imposed by invariants $P(i, o, s)$ that must hold of every reachable state.
- A system is specified by an initialization predicate $I(i, o, s)$ and a transition relation $T(i, o, s, i', o', s')$.

```
(define-system S
  :input ((i1 isort1) ... (im isortm))
  :output ((o1 osort1) ... (on osortn))
  :local ((s1 ssort1) ... (sp ssortp))
  :init (I1 ... Ih)
  :trans (T1 ... Tt)
  :inv (P1 ... Pr)
)
```

- Other commands can be used to declare/define new sorts and functions, and pose queries.

Deep Embedding: The Sort Language

We define a *deep embedding* of the sort, expression, and system *syntax* and *semantics* in PVS.

```
ID: TYPE = nat

% An IL sort is a name, a Boolean, bit vector of length len,
% or an array from index to range.
ILSort: DATATYPE
BEGIN
  ILName(id: ID): ILName?
  ILBool: ILBool?
  ILBitVec(len: posnat): ILBitVec?
  ILArray(index: ILSort, range: ILSort): ILArray?
END ILSort
```

Examples: `ILBool`, `ILName(3)`, `ILBitVec(8)`, `ILArray(ILBitvec(8), ILBool)`.

Good Sort?

```
max_sort_id, m1, m2: VAR ID

% A good IL sort is one whose ids are all below the max sort id.
goodILSort?(max_sort_id)(S: ILSort): RECURSIVE bool =
  CASES S OF
    ILName(id): id < max_sort_id,
    ILArray(index, range): goodILSort?(max_sort_id)(index)
                           AND goodILSort?(max_sort_id)(range)
    ELSE TRUE
  ENDCASES
  MEASURE S BY <<
```

If `max_sort_id` is 8, then `ILName(8)` is not a good sort.

Good Sort Context

A sort context is finite, array-representable sequence of `ILSorts`.

```
ILSortCtx: TYPE = aseq[ILSort]

goodILSortCtx?(sort_ctx: ILSortCtx): bool =
 (FORALL (i: below(sort_ctx`length)):
      goodILSort?(i)(sort_ctx`seq(i)))

GoodILSortCtx: TYPE = (goodILSortCtx?)
```

A well-formed sort context is a finite sequence of sorts where each i'th sort is well-formed relative to i.

Syntax for a Bit-Vector Domain

The Bit-Vector domain admits types that are bit-vectors or spaces of maps between two types.

```
BitVecSyntax : THEORY

  BEGIN

  BitVecType: DATATYPE
  BEGIN
    BitVec(len: nat): BitVec?
    BVArray(dom, rng: BitVecType): BVArray?
  END BitVecType

END BitVecSyntax
```

Semantics

We need a domain in which to interpret ILSort.

```
BitVecSemantics : THEORY
BEGIN

  IMPORTING BitVecSyntax

  X, Y, Z: VAR BitVecType

  BitVecSem: DATATYPE
  BEGIN
    bv(len: nat, vector: bvec[len]): bv?
    barray(sequence: finseq[BitVecSem]): barray?
  END BitVecSem

  A, B, C: VAR BitVecSem
```

Sort Semantics

Easy to map each `ILSort` to its `BitVecType`, except for `ILName(id)` (mapped by `sort_map`) and `ILBool` (one-bit bit-vector).

```
ILSortSemMap(max_sort_id): TYPE = [below(max_sort_id)-> BitVecType]

ILSort2Sem(sort_ctx: GoodILSortCtx,
           max_sort_id : upto(sort_ctx`length),
           sort_map : ILSortSemMap(max_sort_id))
          (sort: GoodILSort(max_sort_id)): RECURSIVE BitVecType =
  CASES sort OF
      ILName(id): sort_map(id),
      ILBool: BitVec(1),
      ILBitVec(len): BitVec(len),
      ILArray(index, range):
          BVArray(ILSort2Sem(sort_ctx, max_sort_id, sort_map)(index),
                  ILSort2Sem(sort_ctx, max_sort_id, sort_map)(range))
  ENDCASES
  MEASURE sort BY <<
```

Sort Context Semantics

A good sort context ensures that *each* sort is defined only in terms of *prior* sorts.

```
goodSortCtxSem?(sort_ctx: GoodILSortCtx)
                 (sort_map : ILSortSemMap(sort_ctx`length))
    : bool =
  FORALL (i: below(sort_ctx`length)):
    ILSort2Sem(sort_ctx, i,
               restrict[below(sort_ctx`length), below(i), BitVecType]
                       (sort_map))(sort_ctx`seq(i))
      = sort_map(i)

SortCtxSem(sort_ctx: GoodILSortCtx): TYPE
       = (goodSortCtxSem?(sort_ctx))
```

Executing PVS Definitions

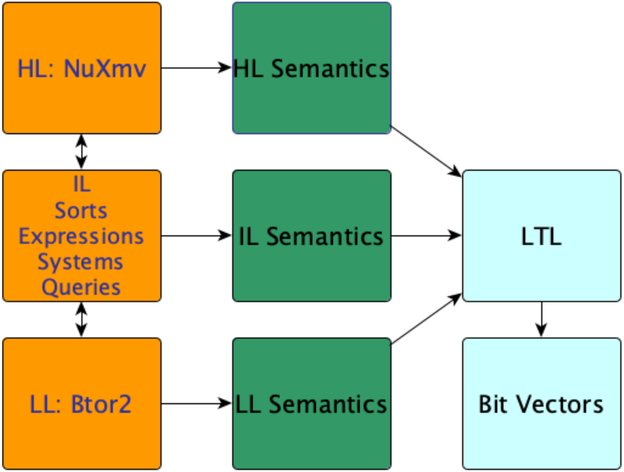
A few test cases:

```
test1_good_ILSort: bool =
   goodILSort?(7)(ILName(8));
test2_good_ILSort: bool =
   goodILSort?(9)(ILArray(ILBitVec(8), ILName(8)));
test3_good_ILSort: bool =
   goodILSort?(0)(ILArray(ILBitVec(8), ILBitVec(8)))

test1_goodILSortCtx: bool =
   (LET ctx: ILSortCtx =
        (# length := 3,
           seq := [: ILArray(ILBitVec(8), ILBitVec(8)),
                     ILArray(ILBitVec(8), ILName(0)),
     ILName(1) :] #)
     IN goodILSortCtx?(ctx))
```

Roadmap

The goal is to create a syntactic/semantic bridge between high-level and low-level languages mapping problems and witnesses.

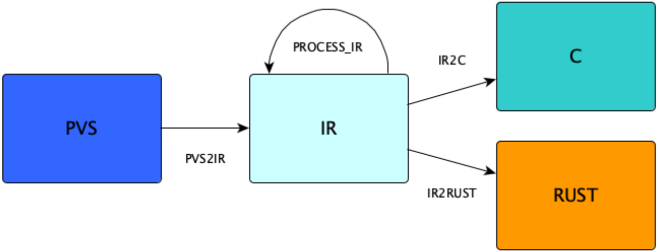


PVS Formalization of Roadmap

- `LTL2`: Syntax/semantics of linear-time temporal logic with two-state predicates.
- `ILExprSyntax`: Defines IL expressions with an executable typechecker.
- `ILExprSemantics`: Defines BitVec semantics for IL expressions — need to prove type soundness.
- `ILSystemSyntax`: Processes IL commands (define/declare sorts, functions, systems) to build context, and IL queries.
- `ILSystemSemantics` (under development): LTL2 semantics for IL Systems.
- `Btor2Syntax`: Syntax of Btor2 nodes (sorts, expressions, systems, properties).
- `Btor2Semantics`: LTL2 semantics of Btor2
- `IL2Btor2` (under development): Translation from IL to Btor2

The PVS2C Code Generator

- PVS2C generates safe, efficient, standalone C code for a full functional fragment of PVS.
- Each PVS theory `foo.pvs` generates a `foo.h` and `foo.c`.[1]
- The translation is factored through an intermediate language that represents PVS expressions in A-normal form and performs a light static analysis to identify the *release points* for references.



[1] Férey, G., Sh., N.: Code Generation using a formal model of reference counting, NFM 2016. See also Wolfram Schulte, Deriving reference count garbage collectors, *6th International Symposium on Programming Language Implementation and Logic Programming*, 1994.

PVS2C: From Theory to Practice

The full PVS2C implementation covers

1. Multi-precision rational numbers and integers, and floats
2. Fixed-size arithmetic: `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, with safe casting
3. Dependent (dynamically sized) and infinite arrays
4. Dependent records and tuples
5. Higher-order functions and closures (with updates)
6. Characters (ASCII and Unicode) and strings
7. Algebraic datatypes
8. Parametric theories with type parameters (unboxed polymorphism)
9. Memory-mapped File I/O
10. Semantic attachments
11. JSON representation for data

PVS2C captures a functional subset of PVS that is usable as a safe programming language - a well-typed program cannot fail (modulo resource limitations).

Experiments in Code Generation: HMAC from Wikipedia

```
function hmac is
    input:
        key:        Bytes    // Array of bytes
        message:    Bytes    // Array of bytes to be hashed
        hash:       Function // The hash function to use (e.g. SHA-1)
        blockSize:  Integer  // The block size of the hash function
                                           //(e.g. 64 bytes for SHA-1)
        outputSize: Integer  // The output size of the hash function
                                           //(e.g. 20 bytes for SHA-1)

    // Keys longer than blockSize are shortened by hashing them
    if (length(key) > blockSize) then
        key <- hash(key) // key is outputSize bytes long

    // Keys shorter than blockSize are padded to blockSize by padding
    //with zeros on the right
    if (length(key) < blockSize) then
        key <- Pad(key, blockSize) // Pad key with zeros to make it
                                   // blockSize  bytes long
    o_key_pad <- key xor [0x5c * blockSize]   // Outer padded key
    i_key_pad <- key xor [0x36 * blockSize]   // Inner padded key
    return hash(o_key_pad ‖ hash(i_key_pad ‖ message))
```

HMAC in PVS

```
hmac(blockSize: uint8,
     key : bytestring,
     (message : bytestring |
        message`length + blockSize < bytestring_bound),
     outputSize: upto(blockSize),
     hash: [bytestring->lbytes(outputSize)]): lbytes(outputSize)
= LET newkey = IF length(key) > blockSize THEN hash(key) ELSE key ENDIF,
      newerkey: lbytes(blockSize)
       = IF length(newkey) < blockSize
          THEN padright(blockSize)(newkey)
          ELSE newkey
         ENDIF,
      oKeyPad = lbytesXOR(blockSize)(newerkey, nbytes(0x5c, blockSize)),
      iKeyPad = lbytesXOR(blockSize)(newerkey, nbytes(0x36, blockSize))
   IN hash(oKeyPad ++ hash(iKeyPad ++ message))

hmac256((blockSize: uint8 | 32 <= blockSize),
        key : bytestring,
        (message : bytestring |
             message`length + blockSize < bytestring_bound))
      : lbytes(32)
= hmac(blockSize, key, message, 32, sha256message)
```

HMAC Generated by PVS2C

```
    bytestrings__bytestring_t
    HMAC__hmac256(uint8_t ivar_2, bytestrings__bytestring_t ivar_3,
                  bytestrings__bytestring_t ivar_4){
        bytestrings__bytestring_t  result;
        uint8_t ivar_18;
        ivar_18 = (uint8_t)32;
        HMAC_funtype_0_t ivar_19;
        HMAC_closure_3_t cl1230;
        cl1230 = new_HMAC_closure_3();
        ivar_19 = (HMAC_funtype_0_t)cl1230;
        bytestrings__bytestring_t ivar_14;
        ivar_14 = (bytestrings__bytestring_t)
        HMAC__hmac((uint8_t)ivar_2, (bytestrings__bytestring_t)ivar_3,
                   (bytestrings__bytestring_t)ivar_4, (uint8_t)ivar_18,
                   (HMAC_funtype_0_t)ivar_19);
        //copying to bytestrings__bytestring
        //from bytestrings__bytestring;
        result = (bytestrings__bytestring_t)ivar_14;
        if (result != NULL) result->count++;
        release_bytestrings__bytestring(ivar_14);
        return result;
}
```

Conclusions

- An expressive specification language like PVS is convenient for representing/automating the syntax and semantics of other formalisms
- Many embeddings have been done in this manner: TLA, B Method, Duration Calculus, Ag, CTL, LTL, I/O Automata, Differential Dynamic Logic, ...
- This offers some advantages compared to working directly with the semantics.
- PVS's proof automation can be applied to the embedded language at both the object level and the meta-level.
- Conversely, automation for the embedded language can be integrated with PVS.
- With safe/efficient code generation in the form of PVS2C, software for the embedded language can be written in PVS itself and proved correct.