A Tutorial Introduction to PVS

Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995

Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, Mandayam Srivas*
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

WWW: `http://www.csl.sri.com/sri-csl-fm.html`

Updated June 1995

Abstract

This document provides an introductory example, a tutorial, and a compact reference to the PVS verification system. It is intended to provide enough information to get you started using PVS, and to help you appreciate the capabilities of the system and the purposes for which it is suitable.

*Dave Stringer-Calvert provided valuable comments on earlier versions of this tutorial, and also checked the specifications and proofs appearing here. Preparation of this tutorial was partially funded by NASA Langley Research Center under Contract NAS1-18969, and by the Advanced Research Projects Agency through NASA Ames Research Center NASA-NAG-2-891 (Arpa order A721) to Stanford Unversity.

since it does simplification, automatic rewriting, and records the sequent formulas in the decision procedure database. The `inst?` command is the most effective way to automatically instantiate quantifiers of existential strength. The `induct-and-simplify` command is a powerful way to construct proofs by induction. The commands `induct-and-rewrite` and `induct-and-rewrite!` are variants of `induct-and-simplify`. These induction commands are able to automatically complete a fairly large class of induction proofs.

It is not necessary to master all the proof commands in order to use the PVS proof checker effectively. In general, it is advisable to learn the most powerful commands first and only rely on the simpler commands when the powerful ones fail. For example, the initial step in a proof is usually skolemization, and the preferred and most powerful form here is `skosimp*`. Similarly, `induct-and-simplify` or one of its variants should be used to initiate induction proofs.

Typically, the creative choices in a proof are:

1. The induction scheme: One of the above induction commands should be employed here.

2. The case analysis: If the case analysis is not explicit in the propositional structure, then it might be implicit in an embedded `IF-THEN-ELSE` or `CASES` structure in which case the `lift-if` command should be used to bring the case analyses to the surface of the sequent where they can be propositionally simplified. Otherwise, the case analysis has to be supplied explicitly using the `case` command.

3. The quantifier instantiations: The instantiation of antecedent universal and succedent existential quantifiers is done automatically by the `inst?` command. When this fails, the more manual `inst` and `inst-cp` commands should be used.

The `bddsimp` command is the most efficient way to do propositional simplification, but `prop` will do when efficiency is not important. Propositional simplification has to be used with care because it can generate many subgoals that share the same proof. The `flatten` and `split` commands are used to do the propositional simplification more delicately.

User-defined *proof strategies*, similar to the tactics and tacticals of LCF, can be employed by more advanced PVS users. A file containing definitions of basic strategies is distributed with PVS and provides a good introduction to this topic. The PVS Prover Checker Reference Manual [SOR93] can be consulted for additional information on user-defined proof strategies.

Finally, it is helpful to be familiar with the PVS prelude theories, which provide very useful background mathematics, as well as a rich source of examples.

5 Two Hardware Examples

In this final section, we develop two hardware examples that illustrate a more sophisticated use of PVS and suggest the intellectual discipline involved in specifying and proving industrial-strength applications. The pipelined microprocessor and n-bit ripple-carry adder examples provide an opportunity to touch on modeling issues, specification styles, and hardware proof strategies, as well as a chance to review many of the PVS language and prover features described in earlier sections of this tutorial.[41]

5.1 A Pipelined Microprocessor

We first develop a complete proof of a correctness property of the controller logic of a simple pipelined processor design described at a register-transfer level. The design and the property verified are both based on the processor example given in [BCM+90]. The example has been used as a benchmark for evaluating how well finite state-enumeration based tools, such as model checkers, can handle datapath-oriented circuits with a large number of states by varying the size of the datapath. From the perspective of a theorem prover, the size of the datapath is irrelevant because the specification and proof are independent of the datapath size. As a theorem proving exercise, the challenge is to see if the proof can be done as automatically as a model checker proof.

Informal Description

Figure 11 shows a block diagram of the pipeline design. The processor executes instructions of the form `(opcode src1 src2 dstn)`, i.e., "destination register `dstn` in the register file `REGFILE` becomes some `ALU` function determined by `opcode` of the contents of source registers `src1` and `src2`. Every instruction is executed in three stages (cycles) by the processor:

[41] One point worth noting that may not be apparent in reading these examples is that the process of specification and verification is an iterative one in which proof is used not to certify a completed specification, but as an aid to developing the specification.

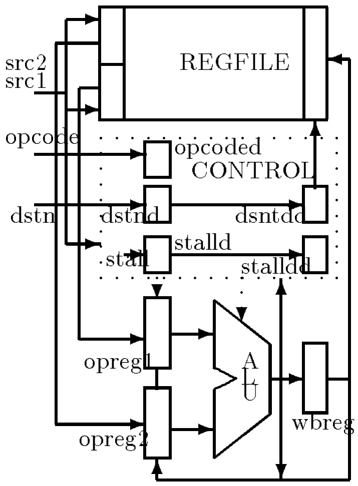

Figure 11: A Pipelined Microprocessor

1. *Read*: Obtain the proper contents of the register file at `src1` and `src2` and clock them into `opreg1` and `opreg2`, respectively.
2. *Compute*: Perform the ALU operation corresponding to the opcode (remembered in `opcoded`) of the instruction and clock the result into `wbreg`.
3. *Write*: Update the register file at the destination register (remembered in `dstndd`) of the instruction with the value in `wbreg`.

The processor uses a three-stage pipeline to simultaneously execute distinct stages of three successive instructions. That is, the read stage of the current instruction is executed along with the compute stage of the previous instruction and the write stage of the previous-to-previous instruction. Since the `REGFILE` is not updated with the results of the previous and previous-to-previous instructions while a read is being performed for the current instruction, the controller "bypasses" `REGFILE`, if necessary, to get the correct values for the read. The processor can abort, i.e., treat as `NOP`, the instruction in the read stage by asserting the `stall` signal true. An instruction is aborted by inhibiting its write stage by remembering the `stall` signal until the write stage via the registers `stalld` and `stalldd`. We verify that an instruction entering the pipeline at any time gets completed correctly, i.e., will write the correct result into the register file, three cycles later, provided the instruction is not aborted.

Formal Specification

PVS specifications consist of a number of files, each of which contains one or more theories. The microprocessor specification is organized into three theories, selected parts of which are shown in Figures 12 and 13. (The complete specification can be found in [SSR95].) The theory `pipe` (Figure 12) contains a specification of the design and a statement of the correctness property to be proved. The theories `signal` and `time`, (Figure 13) imported by `pipe`, declares the types `signal` and `time` used in `pipe`.

The theory `pipe` is parameterized with respect to the types of the register address, data, and the opcode field of the instructions. A theory parameter in PVS can be either a type parameter or a parameter belonging to a particular type, such as `nat`. Since `pipe` does not impose any restriction on its parameters, other than the requirement that they be nonempty, which is stated in the `ASSUMING` part of the theory, one can instantiate them with any type. Every entity declared in a parameterized theory is implicitly parameterized with respect to the parameters of the theory. For example, the type `signal` declared in the parameterized theory `signal` is a parametric type denoting a function that maps `time` (a synonym for `nat`) to the type parameter `T`. (The type `signal` is used to model the wires in our design.) By importing the theory `signal` uninstantiated in `pipe`, we have the freedom to create any desired instances of the type `signal`.

In this example, we use a *functional* style of specification to model register-transfer-level digital hardware in logic. In this style, the inputs to the design and the outputs of every component in the design are modeled as signals. Every signal that is an output of a component is specified as a function of the signals appearing at the inputs to the component.

```
pipe[addr: TYPE, data: TYPE, opcodes: TYPE]: THEORY
  BEGIN
   IMPORTING signal, time

   ASSUMING
    addr_nonempty: ASSUMPTION (EXISTS (a: addr): TRUE)
    data_nonempty: ASSUMPTION (EXISTS (d: data): TRUE)
    opcodes_nonempty: ASSUMPTION (EXISTS (o: opcodes): TRUE)
   ENDASSUMING

   t: VAR time

   %% Signal declarations
      opcode: signal[opcodes]
      src1, src2, dstn: signal[addr]
      stall: signal[bool]
      aluout: signal[data]
      regfile: signal[[addr -> data]]
       ...

    %% Specification of constraints on the signals
      dstnd_ax: AXIOM dstnd(t+1) = dstn(t)
      dstndd_ax: AXIOM dstndd(t+1)= dstnd(t)
      .....

      regfile_ax: AXIOM regfile(t+1) =
                         IF stalldd(t) THEN regfile(t)
                         ELSE regfile(t)
                              WITH [(dstndd(t)) := wbreg(t)]
                         ENDIF
      opreg1_ax: AXIOM opreg1(t+1) =
                    IF src1(t) = dstnd(t) & NOT stalld(t)
                       THEN aluout(t)
                    ELSIF src1(t) = dstndd(t) & NOT stalldd(t)
                          THEN wbreg(t)
                    ELSE regfile(t)(src1(t)) ENDIF
       opreg2_ax: AXIOM ...

       aluop: [opcodes, data, data -> data]
       ALU_ax: AXIOM aluout(t) = aluop(opcoded(t), opreg1(t),
                                        opreg2(t))
      correctness: THEOREM (FORALL t:
        NOT(stall(t)) IMPLIES regfile(t+3)(dstn(t)) =
                     aluop(opcode(t), regfile(t+2)(src1(t)),
                                     regfile(t+2)(src2(t))) )
  END pipe
```

Figure 12: Microprocessor Specification

This style should be contrasted with a *predicative* style, which is commonly used in most HOL applications. In the predicative style every hardware component is specified as a predicate relating the input and output signals of the component and a design is specified as a conjunction of the component predicates, with all the internal signals used to connect the components hidden by existential quantification. A proof of correctness for a predicative style specification usually involves executing a few additional steps at the start of the proof to essentially transform the predictative specification into an equivalent functional style. After that, the proof proceeds similar to that of a proof in a functional specification. The additional proof steps required for a predicative specification essentially

```
signal[val: TYPE]: THEORY
  BEGIN
   signal: TYPE = [time -> val]
  END signal

time: THEORY
  BEGIN
   time: TYPE nat
  END signal
```

Figure 13: Signal Specification

unwind the component predicates using their definitions and then appropriately instantiate the existentially quantified variables. An automatic way of performing this translation is discussed in [SSR95], which illustrates more examples of hardware design verification using PVS.

Returning to our example, the microprocessor specification in `pipe` consists of two parts. The first part declares all the signals used in the design—the inputs to the design and the internal wires that denote the outputs of components. The composite state of `REGFILE`, which is represented as a function from `addr` to `data`, is modeled by the signal `regfile`. The signals are declared as uninterpreted constants of appropriate types. The second part consists of a set of AXIOMs that specify the the values of the signals over time. (To conserve space, we have only shown the specification of a subset of the signals in the design.) For example, the signal value at the output of the register `dstnd` at time `t+1` is defined to be that of its input a cycle earlier. The output of the ALU, which is a combinational component, is defined in terms of the inputs at the same time instant.

In PVS, we can use a descriptive style of definition, as illustrated in this example, by selectively introducing properties of the constants declared in a theory as AXIOMs. Alternatively, we can use the definitional forms provided by the language to define the constants. An advantage of using the definitions is that a specification is guaranteed to be consistent. A disadvantage is that the resulting specification may be overly specific (i.e., overspecified). An advantage of the descriptive style is that it gives better control over the degree to which an entity is defined For example, we could have specified `dstnd` prescriptively, using the conventional function definition mechanism of PVS, which would have forced us to specify the value of the signal at time `t = 0` to ensure that the function is total. In the descriptive style used, we have left the value of the signal at `0` unspecified.

In the present example, the specifications of the signals `opreg1` and `opreg2` are the most interesting of all. They have to check for any register collisions that might exist between the instruction in the read stage and the instructions in the later stages and bypass reading from the register file in case of collisions. The `regfile` signal specification is recursive since the register file state remains the same as its previous state except, possibly, at a single register location. The `WITH` expression is an abbreviation for the result of updating a function at a given point in the domain value with a new value. Note that the function `aluop` that denotes the operation ALU performs for a given `opcode` is left completely unspecified since it is irrelevant to the controller logic.

The theorem (`correctness`) to be proved states a correctness property about the execution of the instruction that enters the pipeline at `t`, provided the instruction is not aborted, i.e., `stall(t)` is not true. The equation in the conclusion of the implication compares the actual value (left hand side) in the destination register three cycles later, when the result of the instruction would be in place, with the expected value. The expected value is the result of applying the `aluop` corresponding to the opcode of the instruction to the values at the source field registers in the register file at `t+2`. We use the state of the register file at `t+2` rather than `t` to allow for the results of the two previous instructions in the pipeline to be completed.

Proof of Correctness

Once the specification is complete, the next step is to typecheck the file, which parses and checks for semantic errors, such as undeclared names and ambiguous types. As we have already seen, typechecking may build new files or internal structures such as *type correctness conditions* (*TCCs*) that represent *proof obligations* that must be discharged before the `pipe` theory can be considered typechecked. The typechecker does not generate any TCCs in the present example. If, for example, one of the assumptions, say for `addr`, in the `ASSUMING` part of the theory was missing, the typechecker would generate the following TCC to show that the `addr` type is nonempty. The declaration of the signal `src1` forces generation of this TCC because a function is nonexistent if its range is empty.

```
% Existence TCC generated (line 17) for src1: signal[addr]
% May need to add an assuming clause to prove this.
  % unproved
src1_TCC1: OBLIGATION (EXISTS (x1: signal[addr]): TRUE);
```

By way of review, the basic objective of developing a proof in PVS as in other subgoal-directed proof checkers (e.g., HOL), is to generate a *proof tree* in which all of the leaves are trivially true. The nodes of the proof tree are sequents, and in the prover you are always looking at an unproved leaf of the tree. The *current* branch of a proof is the branch leading back to the root from the current sequent. When a given branch is complete (i.e., ends in a true leaf), the prover automatically moves on to the next unproved branch, or, if there are no more unproven branches, notifies you that the proof is complete.

The primitive inference steps in PVS are a lot more powerful than in HOL; it is not necessary to build complex tactics to handle tedious lower level proofs in PVS. A knowledgeable PVS user can typically get proofs to go through mostly automatically by making a few critical decisions at the start of the proof. However, as noted previously, PVS does provide the user with the equivalent of HOL's tacticals, called *strategies*, and other features to control the desired level of automation in a proof.

The proof of the microprocessor property shown below follows a certain general pattern that works successfully for most hardware proofs. This general proof pattern, variants of which have been used in other verification exercises [KSK93, ALW93], consists of the following sequence of general proof tasks.

Quantifier elimination: Since the decision procedures work on ground formulas, the user must eliminate the relevant universal quantifiers by skolemization or selecting variables on which to induct and existential quantifiers by suitable instantiation.

Unfolding definitions: The user may have to simplify selected expressions and defined function symbols in the goal by rewriting using definitions, axioms or lemmas. The user may also have to decide the level to which the function symbols have to rewritten.

Case analysis: The user may have to split the proof based on selected boolean expressions in the current goal and simplify the resulting goals further.

Each of the above tasks can be accomplished automatically using a short sequence of primitive PVS proof commands. The complete proof of the theorem is shown below. Selected parts of the proof session are reproduced below as we describe the proof.

```
1: ( then* (skosimp)
2:         (auto-rewrite-theory ''pipe'' :always? t)
3:         (repeat (do-rewrite))
4:         (apply (then* (repeat (lift-if))
5:                       (bddsimp)
6:                       (assert))))
```

In the proof, the names of strategies are shown in *italics* and the primitive inference steps in `type-writer font`. (We have numbered the lines in the proof for reference.) `Then*` applies the first command in the list that follows to the current goal; the rest of the commands in the list are then applied to each of the subgoals generated by the first command application. The `apply` command used in line 5 makes the application of a compound proof step implemented by a strategy behave as an atomic step.

The first goal in the proof session is shown below. It consists of a single formula (labeled {1}) under a dashed line. This is a *sequent*; formulas above the dashed lines are called *antecedents* and those below are called *succedents*. The interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the succedents.

```
correctness :

  |-------
{1}   (FORALL t: NOT (stall(t))
                IMPLIES regfile(t + 3)(dstn(t)) =
                   aluop(opcode(t), regfile(t + 2)(src1(t)),
                         regfile(t + 2)(src2(t))))
```

The quantifier elimination task of the proof is accomplished by the command `skosimp`, which skolemizes all the universally quantified variables in a formula and flattens the sequent resulting in the following goal. Note that `stall(t!1)` has been moved to the succedent in the sequent because PVS displays every atomic formula in its positive form.

```
Rule? (skosimp)
Skolemizing and flattening, this simplifies to:
correctness :

  |-------
{1}   (stall(t!1))
{2}   regfile(t!1 + 3)(dstn(t!1))
        =
       aluop(opcode(t!1), regfile(t!1 + 2)(src1(t!1)),
             regfile(t!1 + 2)(src2(t!1)))
```

The next task—unfolding definitions—is performed by the commands in lines 2 through 3. PVS provides a number of ways of unfolding definitions ranging from unfolding one step at a time to automatic rewriting that performs unfolding in a brute-force fashion. Brute-force rewriting usually results in larger expressions than controlled unfolding and, hence, potentially larger number of cases to consider. If a system provides automatic and efficient rewriting and case analysis facilities, then the automatic approach is viable, as illustrated here. In PVS automatic rewriting is performed by first entering the definitions and AXIOMs to be used for unfolding as rewrite rules. Once entered, the commands that perform rewriting as part of their repertoire, such as `do-rewrite` and `assert`, repeatedly apply the rewrite rules until none of the rules is applicable. To control the size of the expression resulting from rewriting and the potential for looping, the rewriter uses the following restriction for stopping a rewrite: If the right-hand-side of a rewrite is a conditional expression, then the rule is applied only if the condition simplifies to true or false.

Here our aim is to unfold every signal in the sequent so that every signal expression contains only the start time `t!1`. So, we make a rewrite rule out of every AXIOM in the theory `pipe` by means of the command `auto-rewrite-theory` on line 2. We also force an over-ride of the default restriction for stopping rewriting by setting the tag[42] `always?` to true in the `auto-rewrite-theory` command and embed `do-rewrite` inside a `repeat` loop to force maximum rewriting. In the present example, the rewriting is guaranteed to terminate because every feedback loop is cut by a sequential component.

At the end of automatic rewriting, the succedent we are trying to prove is in the form of an equation on two deeply nested conditional expressions as shown below in an abbreviated fashion. The various cases in conditional expression shown above arise as a result of the different possible conflicts between instructions in the pipeline. The equation we are trying to prove contains two distinct, but equivalent conditional expressions, as in `IF a THEN b ELSE c ENDIF = IF NOT a THEN c ELSE b ENDIF`, that can only be proved equal by performing a case-split on one or more of the conditions. While `assert` simplifies the leaves of a conditional expression assuming every condition along the path to the leaves holds, it does not split propositions. One way to perform the case-splitting task automatically is to "lift" all the `IF-THEN-ELSE`s to the top so that the equation is transformed into a propositional formula with unconditional equalities as atomic predicates. After performing such a lifting, we can try to reduce the resulting proposition to true using the propositional

[42] Tags are one of the ways in which PVS permits the user to modify the functionality of proof commands.

simplification command `bddsimp`. If `bddsimp` does not simplify the proposition to true, then it is most likely the case that equations at one or more of the leaves of the proposition need to be further simplified, e.g., by `assert`, using the conditions along the path. If the propositional formula does not reduce to true or false, `bddsimp` produces a set of subgoals to be proved. In the present case, each of these goals can be discharged by `assert`. The compound proof step appearing on lines 4 through 6 of the proof accomplishes the case-splitting task.

```
correctness :

  |-------
[1]   (stall(t!1))
{2}   aluop(opcode(t!1),
            IF src1(t!1) = dstnd(t!1) & NOT stalld(t!1)
              THEN aluop(opcoded(t!1), opreg1(t!1), opreg2(t!1))
            ELSIF src1(t!1) = dstndd(t!1) & NOT stalldd(t!1)
            THEN wbreg(t!1)
            ELSE regfile(t!1)(src1(t!1)) ENDIF,
            ....
            ENDIF)
        = aluop(opcode(t!1),
              IF stalld(t!1) THEN IF stalldd(t!1) THEN regfile(t!1)
                ELSE regfile(t!1) WITH [(dstndd(t!1)) := wbreg(t!1)]
                ENDIF
              ELSE ...
              ENDIF(src1(t!1)),
              IF stalld(t!1) THEN IF stalldd(t!1) THEN regfile(t!1)
                ELSE ... ENDIF
              ELSE ...
              ENDIF(src2(t!1)))
```

We have found that the sequence of steps shown above works successfully for proving safety properties of finite state machines that relate states of the machine that are finite distance apart. If the strategy does not succeed, the most likely cause is that either the property is not true or that a certain property about some of the functions in the specification unknown to the prover needs to be proved as a lemma. In either case, the unproven goals remaining at the end of the proof provide information about the probable cause.

5.2 An N-bit Ripple-Carry Adder

The second example we consider is the verification of a parametrized N-bit ripple-carry adder circuit. The theory `adder`, shown in Figure 14, specifies a ripple-carry adder circuit and a statement of correctness for the circuit.

The theory is parameterized with respect to the length of the bit-vectors. It imports the theories (not shown here) `full_adder`, which contains a specification of a full adder circuit (`fa_cout` and `fa_sum`), and `bv`, which specifies the bit-vector type (`bvec[N]`) and functions. An N-bit bit-vector is represented as an array, i.e., a function, from the the type

```
adder[N: posnat] : THEORY
BEGIN
  IMPORTING bv[N], full_adder

  n: VAR below[N]
  bv, bv1, bv2: VAR bvec
  cin: VAR bool

  nth_cin(n, cin, bv1, bv2): RECURSIVE bool =
      IF n = 0 THEN cin
      ELSE fa_cout(nth_cin(n - 1, cin, bv1, bv2), bv1(n - 1), bv2(n - 1))
      ENDIF
    MEASURE n

  bv_sum(cin, bv1, bv2): bvec =
    (LAMBDA n: fa_sum(bv1(n), bv2(n), nth_cin(n, cin, bv1, bv2)))

  bv_cout(n, cin, bv1, bv2): bool =
    fa_cout(nth_cin(n, cin, bv1, bv2), bv1(n), bv2(n))

  adder_correct_n: LEMMA
      bvec2nat_rec(n, bv1) + bvec2nat_rec(n, bv2) + bool2bit(cin)
        = exp2(n + 1) * bool2bit(bv_cout(n, cin, bv1, bv2))
          + bvec2nat_rec(n, bv_sum(cin, bv1, bv2))

  adder_correct: THEOREM
      bvec2nat(bv1) + bvec2nat(bv2) + bool2bit(cin)
        = exp2(N) * bool2bit(bv_cout(N - 1, cin, bv1, bv2))
          + bvec2nat(bv_sum(cin, bv1, bv2))
END adder
```

Figure 14: Adder Specification

`below[N]`, a subtype of `nat` ranging from `0` through `N-1`, to `bool`; the index `0` denotes the least significant bit. Note that the parameter `N` is constrained to be a `posnat` since we do not permit bit vectors of length `0`. The `adder` theory contains several declarations including a set of initial variable declarations.

The carry bit that ripples through the full adder is specified recursively by means of the function `nth_cin`. The function `bv_cout` and `bv_sum` define the carry output and the bit-vector sum of the adder, respectively. The theorem `adder_correct` expresses the conventional correctness statement of an adder circuit using `bvec2nat`, which returns the natural number equivalent of an N-bit bit-vector. Note that variables that are left free in a formula are assumed to be universally quantified. We state and prove a more general lemma `adder_correct_rec` of which `adder_correct` is an instance. For a given `n < N`, `bvec2nat_rec` returns the natural number equivalent of the least significant n-bits of a given bit-vector and `bool2bit` converts the boolean constants `TRUE` and `FALSE` into the natural numbers `1` and `0`, respectively.

Typechecking

The typechecker generates several TCCs (shown in Figure 15 below) for `adder`.

```
% Subtype TCC generated (line 13) for n - 1
  % unproved
nth_cin_TCC1: OBLIGATION (FORALL n: NOT n = 0 IMPLIES n - 1 >= 0 AND n - 1 < N)

% Subtype TCC generated (line 31) for N - 1
  % unproved
adder_correct_TCC1: OBLIGATION N - 1 >= 0
```

Figure 15: TCCs for Theory `adder`

The first TCC is due to the fact that the first argument to `nth_cin` is of type `below[N]`, but the type of the argument (`n-1`) in the recursive call to `nth_cin` is integer, since `below[N]` is not closed under subtraction. Note that the TCC includes the condition `NOT n = 0`, which holds in the branch of the `IF-THEN-ELSE` in which the expression `n - 1` occurs. A TCC identical to this one is generated for each of the two other occurrences of the expression `n-1` because `bv1` and `bv2` also expect arguments of type `below[N]`. These TCCs are not retained because they are subsumed by the first one.

The second TCC is generated by the expression `N-1` in the definition of the theorem `adder_correct` because the first argument to `bv_cout` is expected to be the subtype `below[N]`.

There is yet another TCC that is internally generated by PVS but is not even included in the TCCs file because it can be discharged trivially by the typechecker, which calls the prover to perform simple normalizations of expressions. This TCC is generated to ensure that the recursive definition of `nth_cin` terminates. PVS does not directly support partial functions, although its powerful subtyping mechanism allows PVS to express many operations that are traditionally regarded as partial. As discussed earlier, the measure function is used to show that recursive definitions are total by requiring the measure to decrease with each recursive call. For the definition of `nth_cin`, this entails showing `n-1 < n`, which the typechecker trivially deduces.

In the present case, all the remaining TCCs are simple, and in fact can be discharged automatically by using the `typecheck-prove` command, which attempts to prove all TCCs that have been generated using a predefined proof strategy called `tcc`.

Proof of Adder_correct_n

The proof of the lemma uses the same core strategy as in the microprocessor proof except for the quantifier elimination step. Since the specification is recursive in the length of the bit-vector, we need to perform induction on the variable `n`. As we've seen in earlier proofs, the user invokes an inductive proof in PVS by means of the command `induct` with the variable to induct on (`n`) and the induction scheme to be used (`below_induction[N]`) as arguments. The induction used in this case is defined in the PVS prelude and is parameterized, as is the type `below[N]`, with respect to the upper limit of the subrange.

This command generates two subgoals: the subgoal corresponding to the base case, which is the first goal presented to prove, is shown in Figure 16.

The goal corresponding to the inductive case is shown below.