Advanced Theorem Proving Techniques in PVS and Applications

César A. Muñoz[1]* and Ramiro A. Demasi[2]

[1] MS 130, NASA Langley Research Center
Hampton VA 23681, USA
`cesar.a.munoz@nasa.gov`

[2] Department of Computing and Software
McMaster University
Hamilton, ON, Canada L8S 4K1
`demasira@mcmaster.ca`

Abstract. The Prototype Verification System (PVS) is an interactive verification environment that combines a strongly typed specification language with a classical higher-order logic theorem prover. The PVS type system supports: predicate subtypes, dependent types, abstract data types, compound types such as records, unions, and tuples, and basic types such as numbers, Boolean values, and strings. The PVS theorem prover includes decision procedures for a variety of theories such as linear arithmetic, propositional logic, and temporal logic. This paper surveys advanced PVS features, including: types for specifications, implicit induction, iterations, rapid prototyping, strategy writing, and computational reflection. These features are illustrated with simple examples taken from NASA PVS developments.

Keywords: Formal Methods, Theorem Proving, PVS.

1 Introduction

Tool customization and extensibility are important issues in the technology transfer of Formal Methods research. In the context of theorem provers, Lüttgen et al. argue in [7] that only tools with a high degree of flexibility and automation can cope with the complexity of digital systems and with the usability requirements imposed by non-expert users of the formal verification technology. In that paper, the authors suggest several improvements to the Program Verification System (PVS) [14] based on their experience with academic and real-world uses of the system. Since the publication of that paper in 2000, PVS has seen major enhancements to its specification language and theorem prover capabilities. These enhancements implement customization and extensibility features suggested in [7] and by other members of the PVS community. Some of these

* The first author is the author of the material presented here and he is responsible for its technical content. The second author provided assistance on the preparation of the document.

features have been implemented by PVS users. In particular, the Formal Methods group at NASA Langley[3] has extensive experience developing and applying PVS theorem proving technology to the verification of safety-critical aerospace systems.

This paper gives an overview of recent and advanced features in PVS through the analysis of a simple formal model of a National Aerospace System (NAS). The idea is to explore this problem from the system requirements to a typical verification task. The main activities involve writing the logical and operational requirements using the PVS specification language, proving properties of these specifications via the PVS theorem prover, and simulating and testing the functional specifications through the PVS ground evaluator.

An overview of PVS is given in Section 2. Section 3 presents a formal model of the NAS, which serves as a running example along this paper. Initially a basic model is specified. This model is then improved by using advanced typing features provided by PVS such as dependent types and predicate subtyping. Recursion is one of the main algorithmic techniques in functional languages. However, proofs of properties on recursive functions are often tedius and difficult. Section 4 provides a non-tradional presentation of recursion in PVS and shows a proof technique, called *implicit induction*, to construct inductive proofs by using the type checker. Finally, section 5 shows how functional specifications can be animated through the utility PVSio [9], which provides a user friendly interface to the PVS ground evaluator [19].

All the PVS features presented here are either part of the current version of the system[4] or available as part of the NASA PVS Libraries.[5] The material presented in this paper is based on the lecture on "Advanced Theorem Proving Techniques in PVS with Applications" presented at the 8th LASER Summer School on Software Engineering, organized by the ETH Chair of Software Engineering on September 4-10, 2011 in Elba Island, Italy. The lectures and the examples presented in this paper are available from `http://shemesh.larc.nasa.gov/people/cam/PVS`.

2 Overview of PVS

The Prototype Verification System (PVS) is a *formal methods* environment that consists of a specification language, based a on a classical higher-order logic enriched with an expressive type system [16], and an interactive theorem prover for this logic.

The PVS specification language is strongly typed. Types in PVS are built from basic types such as `bool` (Boolean values), `number` (numbers), `char` (characters), etc. The basic type `bool` consists of the elements **TRUE** and **FALSE**. The type `char` is defined in the PVS prelude and consists of the extended 256 ASCII characters. There are not literals of type `char`. However, there are literals of type

[3] `http://shemesh.larc.nasa.gov/fm/fm-pvs.html`.
[4] `http://pvs.csl.sri.com`.
[5] `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library`.

`string`, which in PVS is defined as the type of finite sequences of `char`. PVS automatically converts a string literal of one character, e.g., "`a`", into a character. PVS supports subtyping. For example, the types `int` (integer numbers), `rat` (rational numbers), and `real` (real numbers) are axiomatically defined such that `int` is a subtype of `rat`, `rat` is a subtype of `real`, and `real` is a subtype of `number`. Therefore, all numerical constants, including fractions and numbers in decimal notation, are members of `number`. Decimal notation is supported as a syntactic sugar for rational numbers, e.g., the decimal number 0.52 represents the fraction 52/100. In PVS, rational arithmetic is built-in, e.g., 1/3+1/3+1/3 is exactly equal to 1 and this fact *does not* require a proof.

The type $[A\text{->}B]$ represents the type of functions with domain type A and range type B. For example, $[A\text{->}$`bool`$]$ represents the type of predicates over the type A. Predicates can be also be written as **PRED**$[A]$ or as `set`$[A]$. Literals of type $[A\text{->}B]$ are written **LAMBDA**$(x\!:\!A)\!:\!e$, where e is an expression of type B. Literals of type $A\text{->}$`bool`, can also be written $\{x\!:\!A \mid e\}$. PVS supports predicate subtyping. In particular, any predicate can be used as a type, e.g., `FLT :` **TYPE** `= {n:nat |` **EXISTS** `(a,b,c:posnat): a^n + b^n = c^n}` is a type declaration of all naturals numbers that satisfy Fermat's Last Theorem. Predicate subtyping yields a powerful type system. Consequently, type-checking in PVS is undecidable. In particular, PVS cannot automatically check if, for example, the number 2 has the type `FLT` or not. The type-checker will generate Type Correctness Conditions (TCCs) that the user can discharge using the theorem prover. Usually, TCCs are not as hard to prove as `FLT` and PVS will automatically discharge most of them.

The type $[A_1, \ldots, A_n]$ represents the type of n-tuples $(e_1, \ldots, e_n)$ where e_i, with $1 \leq i \leq n$, has the type A_i. If e is the tuple $(e_1, \ldots, e_n)$, for $1 \leq i \leq n$, $e`i$ is exactly equal to e_i. Record types are written $[\#a_1\!:\!A_1, \ldots, a_n\!:\!A_n\#]$, where a_i, with $1 \leq i \leq n$, is the identifier of the i-th field and has the type A_i. If $e_1,\ldots e_n$ are expressions of type $A_1,\ldots,A_n$, respectively, then the literal $(\#a_1\!:=\!e_1, \ldots, a_n\!:=\!e_n\#)$ is a record of type $[\#a\!:\!A_1, \ldots, a\!:\!A_n\#]$. Furthermore, if e is the record $(\#a_1\!:=\!e_1, \ldots, a_n\!:=\!e_n\#)$, for $1 \leq i \leq n$, $e`a_i$ is exactly equal to $e_i$. An alternative notation for $e`a_i$ is $a_i(e)$. PVS provides an overwriting operator for records and functions. In particular, if b_1, b_n are expressions of type A_1, A_n, respectively, then e **WITH** $[\,`a_1\!:=\!b_1\,,`a_n\!:=\!b_n]$ is a record that is equal to e in all fields but a_1, a_n where it has the values b_1, b_n, respectively. In the case of functions, if f is a function of type $[A\text{->}B]$ and a, b are expressions of type A, B, respectively, the expression f **WITH** $[(a)\!:=\!b]$ is a function that is equal to f in all points but a where it has the value b.

PVS supports dependent types, e.g., in a record type $[\#a_1\!:\!A_1, \ldots, a_n\!:\!A_n\#]$, the type A_i, for $1 \leq i \leq n$, can depend on the fields a_j with $1 \leq j \leq i$. This feature extends to the types of tuples and functions. In particular, in a function declaration $f(x\!:\!A) : B = \ldots$ the type B can depend on the parameter x.

In addition to basic types, tuples, records, and functions, PVS also includes abstract data types [15] and co-inductive types. Enumerations and disjoint types are supported as special cases of abstract data types. In order to support modularity and reuse, specifications are logically organized into parametric *theories*. Theories are linked by **IMPORTING** clauses. Theory parameters can be instantiated at the importing clause or at each particular use of a definition coming from an imported parametric theory. Furthermore, PVS supports name overloading. Names are often automatically disambiguated by the system using the type information. If this is not possible, the user needs to provide a qualified name that may include the name of theory and its parameters.

In addition to declarations and definitions, theories may have logical formulas such as type judgements, lemmas, theorems, and axioms. These formulas are discharged by the user via an interactive proof assistant. For each formula, PVS maintains a proof tree. Each node of the proof tree is a *sequent* of the form

```
{-1} P_1
     ...
{-n} P_n
  |-------
{1}  Q_1
     ...
{m}  Q_m
```

where the set of formulas $P_1, \ldots, P_n$, with $n \geq 0$, are called the *antecedent* and the set of formulas $Q_1, \ldots, Q_m$, with $m \geq 0$, are called the *consequent*. The logical interpretation of a sequent is that disjunction of formulas in the consequent can be derived from the conjunction of formulas in the antecedent, i.e., $P_1 \wedge \ldots P_n \vdash Q_1 \vee \ldots \vee Q_m$. An empty antecedent represents the formula **TRUE** and an empty consequent represents the formula **FALSE**.

The proof tree starts with a root node of the form A, where A is the formula, e.g., theorem or lemma, to be proved. The proof tree is interactively constructed by adding subtrees to leaf nodes as directed by proof commands, which are prompted by the user. A proof command has the form $(r\ a_1\ \ldots\ a_n)$, where r is the name of a proof rule and $a_1, \ldots, a_n$ are its arguments. The arguments can be either proof commands or Lisp objects. The parameters of a proof rule are named so the corresponding arguments can be specified in a different order to which the parameters were defined. Named arguments in a proof command are specified with the syntax $:n\ a$, where a is the argument of the parameter named n. For reference, the proof command (`help` r) prints a help message for the proof rule r.

Proof commands can either generate further branches, or complete a branch and move the control over to the next branch in the proof tree. These commands can be used to introduce lemmas, expand definitions, apply decision procedures, eliminate quantifiers, and so on. For example, the primitive proof rule `flatten` simplifies conjunctive antecedents and disjunctive consequents. The proof rule `assert` performs several simplifications using decision procedures for equality and linear arithmetic. PVS provides a large set of proof rules that implement automated procedures for various domains such as binary decision diagrams, satisfiability modulo theories, rewriting, and, most recently, non-linear arithmetic.

When a proof command reduces either the antecedent of a sequent to **FALSE** or the consequent to **TRUE**, the current branch of the proof tree is terminated and the sequent is said to be *discharged*. When all the branches of the proof tree have been discharged the original formula is successfully proved.

Proof commands are sound, but not necessarily complete, i.e., discharging all the sequents generated by a given proof command is sufficient, but not necessary to discharge the original sequent. Some proof commands can generate an empty sequent. This does not mean that the original sequent is false, but rather that the current proof tree does not lead to a proof of the original sequent. In these cases, the proof command (`undo`) can be used to backtrack to the previous sequent before the last proof command. PVS also provides a strategy language and proof command combinator that can be used to define *strategies*, i.e., user-defined proof commands [1]. Strategies conservatively extend the theorem prover. Therefore, the soundness of the theorem prover is not compromised by defining new strategies.

PVS *packages*, which are also called *prelude extensions*, are the mechanism offered by PVS to modularly and conservatively extend the system with user-defined Emacs Lisp code, Common Lisp code, proof strategies, and prelude PVS theories. The formal methods group at NASA Langley has developed several PVS packages that extends the functionality provided by the system with batch proving and proof-scripting capabilities (ProofLite), animation of specifications (PVSio), strategies for manipulation of algebraic expressions (Manip, Field), and automated procedures for non-linear arithmetic (Interval, Bernstein). These extensions either have been integrated into the most recent version of PVS (5.0) or they are available as part of the PVS NASA Libraries (5.8). Examples of some of these features are given in this paper.

3 Simple Model of a National Aerospace System in PVS

In a typical verification task, a user has a set of requirements for a system and uses PVS to

– write logical requirements as formal specifications,
– write operational requirements as functional algorithms, and
– prove that the algorithms satisfy the specifications.

Given the different nature of real and formal worlds, the formal world will never be able to perfectly capture all the aspects of the real world. Hence, the formal specifications and the algorithms are intrinsically imprecise models of the actual requirements and the computer program implementation. For this reason, the verification of a real system will always require simulation and testing to check the validity of the assumptions on which the formal specifications rely. This is illustrated in Figure 1.

This paper uses a simple formal model of a National Aerospace System (NAS). From a very abstract point of view, a NAS consists of a collection of

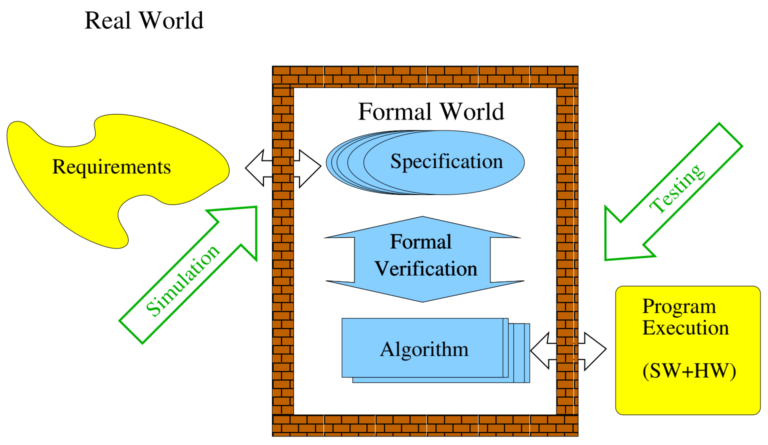

Fig. 1. Real World vs. Formal World

aircraft along with some basic functionality such as adding and removing aircraft, and more advanced functionality such as detecting and resolving conflicts between aircraft.

3.1 Basic Model

The basic unit of a PVS specification is a *theory*, which is a collection of declarations and definitions of mathematical objects. Theories in PVS can be parametric. Hence, a theory can be used to specify a family of systems all of which share the same structure. A National Aerospace System can be defined as a parametric theory with respect to the type of the aircraft identifiers, i.e., `Identifier`, and the type of the aircraft state information, i.e., `State`. The parameters `Identifier` are `State` are declared as an uninterpreted non-empty types. The theory `NAS` can be imported in another theory with concrete types that instantiate `Identifier` and `State`.

```
NAS[Identifier:TYPE+,State:TYPE+] : THEORY
BEGIN
  % ...
END NAS
```

Within this theory, an aircraft will be defined as a record with a field `id` of type `Identifier` and a field `state` of type `State`:

```
  Aircraft : TYPE = [#
    id : Identifier,
    state : State
  #]
```

A NAS can be modeled as a collection of elements of type `Aircraft`. There are several ways to define such as collection in PVS, e.g., lists, sets, sequences, and arrays. All these types are defined in the PVS prelude library as parametric types. The type `ARRAY[Identifier->Aircraft]` specifies a type of elements of type `Aircraft` indexed by elements of type `Identifier`. PVS does not assume anything about the range type of an array. In fact, there is no difference between the array type `ARRAY[Identifier->Aircraft]` and the function type `[Identifier->Aircraft]`. It should be noted that not all arrays of type `ARRAY[Identifier->Aircraft]` represent a NAS. It is required that the aircraft indexed by a given identifier has this value in the field `id`. A type `NAS` of arrays that satisfy this contract can be defined in PVS using the subtyping mechanism as follows.

```
NAS : TYPE = {acs:ARRAY[Identifier->Aircraft] |
              FORALL (id:Identifier): acs(id)`id = id}
```

PVS is a pure functional language and so it does not have a built-in notion of memory or state. Thus, in PVS, the concept of variable corresponds to the mathematical concept of unspecified arbitrary value a opposed to the concept of memory cell used in imperative programming languages. Variables of type `NAS`, `Aircraft`, and `Identifier` can be declared as follows.

```
nas : VAR NAS
ac  : VAR Aircraft
id  : VAR Identifier
```

A function that given a NAS `nas` and an identifier `id` and returns the aircraft indexed by that identifier in `nas` can be defined as follows.

```
find(nas,id): Aircraft = nas(id)
```

From this definition, the following soundness lemma is automatically discharged by the proof command `(grind)`.

```
find_sound : LEMMA
  LET ac = find(nas,id) IN ac`id=id
```

In the declaration of the function `find` and in the statement of the lemma `find_sound`, the names `nas` and `id` corresponds to the variables previously declared. By default, if variables are not quantified in the statement of a logical formula, they are assumed to be universally quantified.

A function that adds an aircraft to a given NAS array and its soundness lemma can be defined as follows.

```
add(nas,ac): NAS =
 LET id = ac`id IN
 nas WITH [(id) := ac]

add_sound : LEMMA
```

```
    find(add(nas,ac),ac`id) = ac
```

As in the previous case, the lemma `add_sound` is proved by the proof command (`grind`). Since the function `add` returns an array of type `NAS`, the type checker generates the following Type Correctness Condition (TCC), which is automatically discharged by the theorem prover.

```
add_TCC1: OBLIGATION
  FORALL (nas: NAS, ac: Aircraft, id: Identifier):
    id = ac`id IMPLIES
     (FORALL (id_1: Identifier):
        (nas WITH [(id) := ac])(id_1)`id = id_1);
```

The representation of a NAS as an array provides a direct way to access an aircraft through its identifier and a convenient way to modify the content of a NAS using the operator WITH. However, this basic model has several limitations. In particular, since the type `Identifier` is uninterpreted, there is not a simple mechanism to define a function that iterates over all aircraft in a NAS or that computes the number of aircraft in a NAS.

3.2 NAS Model Based on Finite Sequences

In order to solve the limitations of the basic model presented in the previous section, a more sophisticated model based on finite sequences is specified. More precisely, instead of an array over an uninterpreted type, the collection of aircraft in a NAS is modeled using a finite sequence.

Finite sequences are defined in the PVS prelude library as a dependent record consisting of a length and an array of exactly length elements, i.e., the range of the array is restricted to natural numbers less than the length:

```
  finseq: TYPE = [# length: nat, seq: ARRAY[below[length] -> T] #]
```

In this definition, the type `T` is a type parameter. The PVS prelude library also defines the following conversion, which automatically casts finite sequences of type `T` into arrays of type `T`.

```
  finseq_appl(fs): [below[length(fs)] -> T] = fs`seq;
  CONVERSION finseq_appl;
```

Because of the conversion `finseq_appl`, the element of type `T` located at position `i` of a finite sequence `fs`, assuming that `i` has the type `below(length(fs))`, can be simply written `fs(i)` as opposed to `fs`seq(i)`.

A finite sequence of `Aircraft` is specified as `finseq[Aircraft]`. Since the array in the field `seq` is indexed by a finite range of natural numbers, it is simple to specify functions that iterate on all the elements of a finite sequence. In order to have direct access to aircraft through their identifiers, a hash table represented by an array of type `[Identifier->Maybe[nat]]` is used. The keys of this type of hash tables are aircraft identifiers. The values are positions in a finite sequence where the aircraft are located. Since not all identifiers appear in a finite sequence and PVS functions are total, the values of the hash table are specified using the parametric abstract data type `Maybe`. The type `Maybe`, which models invalid and valid values of a given type, is defined in the PVS NASA library `structures` as follows.

```
Maybe[T:TYPE] : DATATYPE
BEGIN
  None : none?
  Some(some:T): some?
END Maybe
```

The constructors of this inductive data type are `None` and `Some`. They represent, respectively, an invalid and a valid element of type `T`. These constructors are recognized by the functions `none?` and `some?`, respectively. The selector for the constructor `Some` is `some`.

A new type that represents a NAS can be modeled by a record with the fields `acs` of type `finseq[Aircraft]` and `hash` of type `ARRAY[Identifier->Maybe[nat]]`. As in the previous case, not all elements of this record type represent a NAS. In PVS, it is convenient to first define a base record type, e.g.,

```
  PreNAS : TYPE = [# acs  : finseq[Aircraft],
                      hash : ARRAY[Identifier->Maybe[nat]]
                   #]
```

Next, a predicate on this type that characterizes the valid elements of the base type is defined, e.g.,

```
  nas?(nas:PreNAS): bool =
     (FORALL(i:below(length(nas`acs))):
       LET ac = nas`acs(i) IN
         nas`hash(ac`id) = Some(i))
     AND
     (FORALL(id:Identifier):
        LET hi = nas`hash(id) IN
          (some?(hi) IMPLIES
            some(hi) < length(nas`acs) AND
            nas`acs(some(hi))`id = id))
```

The predicate `nas?` on elements `nas` of type `PreNAS` holds when (a) all the identifiers of aircraft in the finite sequence appear in hash table and (b) all the valid values in the hash table point to a valid position in the finite sequence and the aircraft in this position has as identifier the key in the hash table.

Finally, the type `NAS` is a defined as the subtype of `PreNAS` that satisfies the predicate `nas?`. This subtype can be specified as `{x:PreNAS | nas?(x)}`. However, these kinds of predicate subtype definitions are so common that PVS provides an alternative and less verbose notation. If p is a predicate over a type T, the type (p) specifies the type of elements of type T that satisfy p, e.g.,

```
  NAS : TYPE = (nas?)
```

Given an element `nas` of type `NAS` and an identifier `id` of type `Identifier`, the dependent type `IdNAS(nas)` is declared as the subtype of `Identifier` that consists of all the identifiers of aircraft in `nas`. The function `find` has as inputs a `nas` of type `NAS` and an identifier `id` of type `IdNAS(nas)`. It returns the aircraft associated to this identifier in `nas`.

```
  id_nas?(nas:NAS)(id:Identifier): bool = some?(nas‘hash(id))
  IdNAS(nas): TYPE = (id_nas?(nas))

  find(nas:NAS,id:IdNAS(nas)): Aircraft =
    LET hi = nas‘hash(id) IN
      nas‘acs(some(hi))
```

3.3 Proving Properties

Given the previous definition of the type `NAS`, it can be proved that if two aircraft in a NAS have the same identifier, then they are equal. This property can be specified in PVS as follows.

```
  nas_sound : LEMMA
    FORALL (i,j:below(length(nas‘acs))):
      nas‘acs(i)‘id = nas‘acs(j)‘id IMPLIES
        i = j
```

In the theorem prover, the lemma `nas_sound` yields the following sequent.

```
 |-------
{1}   FORALL (nas: NAS, i, j: below(length(nas‘acs))):
        finseq_appl(nas‘acs)(i)‘id = finseq_appl(nas‘acs)(j)‘id
        IMPLIES i = j
```

The proof command (`skeep :preds? t`) reduces the universal quantification by introducing new arbitrary constants. These constants are called *Skolem* constants. The symbol `:preds?` is the name of a parameter of the proof rule `skeep` and the argument `t` is the Lisp symbol that represents *true.* This parameter, which by default has the value `nil`, i.e., *false*, tells the proof command whether or not to introduce in the sequent the types of the Skolem constants.

```
{-1}  nas?(nas)
{-2}  i < length(nas‘acs)
{-3}  j < length(nas‘acs)
{-4}  finseq_appl(nas‘acs)(i)‘id = finseq_appl(nas‘acs)(j)‘id
  |-------
{1}   i = j
```

The proof rule `skeep` uses as names of the Skolem constants those of the quantified variables. In this case, the Skolem constants are named `nas`, `i`, and `j`. All the formulas in the antecedent of the current sequent, but formula –4, come from the type information of the Skolem constants.

The proof command (`expand*` "`nas?`" "`finseq_appl`") expands the definition of the predicate `nas?` and the conversion `finseq_appl`.

```
{-1}  (FORALL (i: below(length(nas`acs))):
        nas`hash(nas`acs`seq(i)`id) = Some(i)) AND
        (FORALL (id: Identifier):
           (some?(nas`hash(id)) IMPLIES some(nas`hash(id)) <
           length(nas`acs) AND
           nas`acs`seq(some(nas`hash(id)))`id = id))
[-2]  i < length(nas`acs)
[-3]  j < length(nas`acs)
{-4}  nas`acs`seq(i)`id = nas`acs`seq(j)`id
  |-------
[1]   i = j
```

The conjunction in formula –1 is simplified with the proof command (`flatten`).

```
{-1}  FORALL (i: below(length(nas`acs))):
        nas`hash(nas`acs`seq(i)`id) = Some(i)
{-2}  FORALL (id: Identifier):
        (some?(nas`hash(id)) IMPLIES some(nas`hash(id)) <
        length(nas`acs) AND
        nas`acs`seq(some(nas`hash(id)))`id = id)
[-3]  i < length(nas`acs)
[-4]  j < length(nas`acs)
[-5]  nas`acs`seq(i)`id = nas`acs`seq(j)`id
  |-------
[1]   i = j
```

Formula –2 is not required in the proof so it can be hidden. However, two instances of formula –1 are needed, so it has to be copied. This can be achieve with the proof commands (`hide` –2) and then (`copy` –1).

```
{-1}  FORALL (i: below(length(nas`acs))):
        nas`hash(finseq_appl(nas`acs)(i)`id) = Some(i)
[-2]  FORALL (i: below(length(nas`acs))):
        nas`hash(finseq_appl(nas`acs)(i)`id) = Some(i)
[-3]  i < length(nas`acs)
[-4]  j < length(nas`acs)
[-5]  finseq_appl(nas`acs)(i)`id = finseq_appl(nas`acs)(j)`id
  |-------
[1]   i = j
```

Formulas –1 and –2 are instantiated with the variables `i` and `j`, respectively. This is achieved with the proof commands (`inst` –1 "`i`") and (`inst` –2 "`j`").

```
[-1]  nas`hash(nas`acs`seq(i)`id) = Some(i)
{-2}  nas`hash(nas`acs`seq(j)`id) = Some(j)
[-3]  i < length(nas`acs)
[-4]  j < length(nas`acs)
[-5]  nas`acs`seq(i)`id = nas`acs`seq(j)`id
  |-------
[1]   i = j
```

Formula –5 and –1 are replaced in the sequent with the proof command (`replaces` (–5 –1)).

```
{-1}  Some(i) = Some(j)
{-2}  i < length(nas`acs)
{-3}  j < length(nas`acs)
  |-------
{1}   i = j
```

This sequent is discharged by decomposing the equality in formula –1 with the proof command (`decompose-equality` –1).

3.4 Other Operations

Functions that return an empty NAS and that update a NAS by either adding or modifying the information of a given aircraft, and their soundness theorems can be defined as follows.

```
  empty : NAS = (# acs  := empty_seq,
                   hash := LAMBDA(id):None
                #)

  empty_sound : LEMMA
    LET hi = empty`hash(ac`id) IN
    none?(hi)

  update(nas,ac): NAS =
    LET hi = nas`hash(ac`id) IN
    IF none?(hi) THEN
      nas WITH [`acs := add(ac,nas`acs),
                `hash(ac`id) := Some(length(nas`acs))]
    ELSE
      nas WITH [`acs`seq(some(hi)) := ac]
    ENDIF

  update_sound : LEMMA
```

```
    LET unas = update(nas,ac),
        hi   = unas`hash(ac`id) IN
    some?(hi) AND unas`acs(some(hi)) = ac
```

The lemmas `empty_sound` and `update_sound` are automatically discharged by the proof command `(grind)`. The definition of the functions `empty` and `update` generate TCCs that guarantee that they return records that satisfy the predicate `nas?`. The function `empty` generates one TCC that is automatically discharged by the type checker. The later definition generates 4 TCCs, two of which are automatically discharged by the type checker. The other two TCCs require a few simple manual steps similar to those in the proof of lemma `nas_sound` given in Section 3.3.

4 Recursion, Induction, and Iteration

In functional specification languages such as PVS, repetitive processes are typically modeled using recursion. In PVS, the recursive definition of a function f of type `[[`$T_1, \ldots, T_n$`]->`T`]` is jointly specified with a measure function M of type `[[`$T_1, \ldots, T_n$`]->`A`]` and a order relation $\prec$ over the type A:

```
f(x1:T1,...,xn:Tn): RECURSIVE T =
  ... f(e1,...,en) ...
MEASURE M BY ≺
```

The type checker ensures that the definition of f is total by generating TCCs that asserts the following termination conditions.

– The order relation $\prec$ is well-founded, i.e., it does not admit infinite descending sequences $\ldots \prec a_m \prec \ldots \prec a_o$ of elements of type A.
– The measure function M strictly decreases at each recursive call, e.g.,

$$M(e_1, \ldots, e_n) \prec M(x_1, \ldots, x_n).$$

By default, if a order relation is not provided in the definition, PVS assumes that the order relation is the strict order < over the type `nat` of natural numbers. The PVS prelude includes the following axiom that asserts that the order < is well-founded.

```
wf_nat: AXIOM well_founded?(LAMBDA(i,j:nat): i < j)
```

Other well-founded orders known to PVS are the order < over the type `ord` of ordinal numbers, the lexicographical order `lex2` for pairs of natural numbers, and a generic structural order << for inductively defined abstract data types. If instead of a function, M is specified as an expression of type `A`, PVS assumes the measure function `LAMBDA(`$x_1 : T_1, \ldots, x_n : T_n$`)`$: M$.

4.1 Tail Recursion

Some operations on a National Airspace System require computing upper (and lower) bounds of values such as altitude, ground speed, vertical speed, etc. for all aircraft in the airspace. Using the formal theory presented in Section 3.2, this section defines a higher-order function `maxf(nas:NAS,f:[Aircraft->real]): real`, where the parameter `f` is a function that computes the value of interest for a given aircraft. The requirement that `maxf` computes an upper bound of `f`(a), for all aircraft a in `nas`, is specified by the following lemma.

```
maxf_sound : LEMMA
  FORALL (nas:NAS,i:below(length(nas`acs)),f:[Aircraft->real]):
    f(nas`acs(i)) <= maxf(nas,f)
```

In order to satisfy its requirement, the definition of `maxf` must iterate `f` on all aircraft in `nas`. This repetitive process is encoded using a recursive definition. In a typical recursive definition of a function over natural numbers or terms of an abstract data type, the function is first defined on the base case and then the inductive case is defined by recursively calling the function on terms that are smaller than the original term. For data structures such as arrays and finite sequences, *tail recursion* is often more convenient. A tail recursive definition is a recursive definition where recursive calls are always the last statements in the definition. In a typical tail recursive definition of a function, a counter over a finite range and an accumulator are used as parameters. The accumulator is initiated with the value of the function that corresponds to the initial value of the counter. At every recursive step, the accumulator, as its name indicates, has the accumulated value of the function up to the counter. When the counter reaches its limits, the value of the accumulator is returned.

The function `maxf` is defined using the tail recursive function `maxf_it` as follows.

```
1   maxf_it(nas:NAS,f:[Aircraft->real],
2           i:upto(length(nas`acs)),max:real): RECURSIVE real =
3   IF i = length(nas`acs) THEN max
4   ELSIF i=0 OR max < f(nas`acs(i)) THEN
5     maxf_it(nas,f,i+1,f(nas`acs(i)))
6   ELSE
7     maxf_it(nas,f,i+1,max)
8   ENDIF
9   MEASURE length(nas`acs)-i
10
11  maxf(nas:NAS,f:[Aircraft->real]): real =
12    maxf_it(nas,f,0,0)
```

The parameter `i` of the tail recursive function `maxf_it` is a counter that goes from 0 up to `length(nas`acs)`, while the parameter `max` is an accumulator for the function. At line 3, if the counter has reached its limit, the accumulator is returned and the tail recursive function terminates. The recursive calls are made at lines 5 and 7. In both cases, they are the last function calls in the definition of the function `maxf_it`. The recursive call at line 5 corresponds to the case when the value of the accumulator is updated. The measure function is given at line 9 through the expression `length(nas‘acs)-i`. Since the counter is incremented by one at each recursive call, the measure strictly decreases at each call. The type checker generates 7 TCCs for the definition of the function `maxf_it`, including two TCCs that correspond to termination conditions. All of them are automatically discharged by the type checker. The function `maxf` is defined at lines 11 and 12. It calls the function `maxf_it`, where the counter and the accumulator are both initiated with the value 0.

4.2 Induction

Generally, proofs of properties involving recursive functions are done by induction. PVS provides several proof rules to construct inductive proofs. Simple inductions on natural numbers and abstract data types are automatically handled by the proof rule `induct-and-simplify` and its variants `induct-and-rewrite` and `induct-and-rewrite!`. The most general proof rule `induct` is used to manually prove an inductive property, when the more automated proof rules fail.

Given a variable in a universally quantified formula in the consequent, the proof rule `induct` applies an induction schema on this variable. The induction schema is inferred by the theorem prover from the type of the variable, but it can also be specified by the user through the proof rule's parameter `name`. The PVS prelude library provides the following weak and strong induction schemas for natural numbers.

```
nat_induction: LEMMA
        (p(0) AND (FORALL j: p(j) IMPLIES p(j+1)))
               IMPLIES (FORALL i: p(i))

NAT_induction: LEMMA
        (FORALL j: (FORALL k: k < j IMPLIES p(k)) IMPLIES p(j))
               IMPLIES (FORALL i: p(i))
```

Similar lemmas are provided for many subtypes of natural numbers. For instance, weak and strong induction schemas for ranges of natural numbers are specified as follows.

```
subrange_inductions[i: int, j: upfrom(i)]: THEORY
 BEGIN
  k, m: VAR subrange(i, j)
  p: VAR pred[subrange(i, j)]

  subrange_induction: LEMMA
    (p(i) AND (FORALL k: k < j AND p(k) IMPLIES p(k + 1)))
        IMPLIES (FORALL k: p(k))
  SUBRANGE_induction: LEMMA
    (FORALL k: (FORALL m: m < k IMPLIES p(m)) IMPLIES p(k))
      IMPLIES (FORALL k: p(k))

END subrange_inductions
```

The fact that the measure function of a recursive function guarantees termination hints that an inductive proof on a recursive function follows the same direction as its measure function, i.e., the base case of the induction is when the measure has the value 0 and from there the inductive step is constructed. Therefore, properties on a recursive function are usually proved by induction on a variable that keeps the value of the function's measure function at each recursive call. When such as variable does not exist in the statement of the property, the statement has to be generalized to introduce this new variable. Finding this generalization is not always obvious. This is particularly true for statements on tail recursive functions since, in this case, the measure function and the counter used in the tail recursion go in opposite directions.

To prove lemma `maxf_sound`, the following parametric predicate on real numbers and lemma are defined.

```
maxf_below?(nas:NAS,f:[Aircraft->real],i:upto(length(nas`acs)))
           (max:real): bool =
  FORALL(j:below(i)): f(nas`acs(j)) <= max

maxf_sound_ind : LEMMA
 FORALL (m:nat,nas:NAS,i:upto(length(nas`acs)),f:[Aircraft->real],
        max:real):
   i = length(nas`acs)-m AND
   maxf_below?(nas,f,i)(max) IMPLIES
   maxf_below?(nas,f,length(nas`acs))(maxf_it(nas,f,i,max))
```

Given parameters `nas`, `f`, and `i` of the specified types, `maxf_below?(nas,f,i)` holds for a real number `max` if `max` is an upper bound of `f(nas`acs(j))`, for all `j < i`. Lemma `maxf_sound_ind`, which generalizes `maxf_sound` with a variable `m` that is equal to `length(nas`acs)-i`, is automatically discharged by the proof command `(induct-and-simplify "m")`.

The proof of lemma `maxf_sound` consists of `(skeep)`, which introduces Skolem constants, `(lemma "maxf_sound_ind")`, which introduces lemma `maxf_sound_ind` to the sequent as formula -1, `(inst -1 "length(nas`acs)" "nas" "0" "f" "0")`, which instantiates formula -1 with the given values, and `(grind)`, which applies several decision procedures and discharges the lemma.

4.3 Implicit Induction

Finding an inductive property for a nontrivial measure function is often a tedious exercise even for PVS expert users. Fortunately, the PVS type system provides an elegant and relatively simple mechanism to make inductive proofs on recursive definitions without explicitly using an induction schema. The key idea is to encode the property to be proved in the domain and range types of the function. The domain type, i.e., the type of the parameters, becomes an invariant of the function and the range type, i.e., the return type, becomes a postcondition of the function. The type checker generates as TCCs the fact that the invariant is satisfied at each recursive call and that the postcondition is satisfied after the execution of the last recursive call. These TCCs correspond to the base and inductive cases of an inductive proof of the property encoded by the types where the induction schema is based on the syntactical structure of the recursive definition rather than on the measure of the function.

The idea of using PVS types for specifications is well-known [18]. However, surprisingly, most users are unaware of the *implicit induction* approach it provides when used on recursive definitions. For instance, a simpler proof of lemma `maxf_sound` can be found by restricting the type of the parameter `max` and the return type of the function `maxf_it` as follows.

```
  maxf_it(nas:NAS,f:[Aircraft->real],i:upto(length(nas`acs)),
         max:(maxf_below?(nas,f,i))):
        RECURSIVE (maxf_below?(nas,f,length(nas`acs))) = ...
```

The modified type of the accumulator `max` states that it is an upper bound of `f(nas`acs(j))` for `j < i`. Similarly, the modified return type of the function states that the computed value is an upper bound of `f(nas`acs(j))` for `j < length(nas`acs)`. The body of the function remains unchanged.

The modified definition of `maxf_it` generates 10 TCCs. All the TCCs but `maxf_it_TCC6` are automatically discharged by the type checker.

```
maxf_it_TCC6: OBLIGATION
  FORALL (nas: NAS, f: [Aircraft -> real],
          i: upto(nas`acs`length), max: (maxf_below?(nas, f, i))):
    (i = 0 OR max < f(finseq_appl(nas`acs)(i))) AND
    NOT i = nas`acs`length
     IMPLIES
     maxf_below?(nas, f, 1 + i)(f(finseq_appl(nas`acs)(i)));
```

Since TCCs are mechanically generated by the type checker, they are verbose and seem complicated. However, once this initial impression is overcome, their intuitive meaning becomes clear. In this case, `maxf_it_TCC6` states that the type condition on the parameter `max` is satisfied by the first recursive call of the function `maxf_it`. The proof of `maxf_it_TCC6` proceeds as follows. The proof command (`grind`) yields the following sequent.

```
{-1}  j!1 < 1 + i!1
{-2}  nas!1`hash(nas!1`acs`seq(i!1)`id) = Some(i!1)
{-3}  i!1 <= length(nas!1`acs)
{-4}  FORALL (j: below(i!1)): f!1(nas!1`acs`seq(j)) <= max!1
{-5}  max!1 < f!1(nas!1`acs`seq(i!1))
```

```
  |-------
{1}   i!1 = length(nas!1`acs)
{2}   f!1(nas!1`acs`seq(j!1)) <= f!1(nas!1`acs`seq(i!1))
```

This sequent is discharged by the proof command (`inst -4 "j!1"`), followed by (`assert`).

Type specifications are not only useful for recursive functions. Indeed, the return type of the function `maxf` can be specified as follows.

```
  maxf(nas:NAS,f:[Aircraft->real]):
    (maxf_below?(nas,f,length(nas`acs))) = ...
```

Using this definition, lemma `maxf_sound` is discharged by the proof commands (`skeep`), (`typepred "maxf(nas,f)"`), which introduces the most restricted type information of the expression `maxf(nas,f)`, and, finally, (`grind`).

4.4 Recursive Judgements

A disadvantage of the implicit technique described in Section 4.3 is that in order to prove a property on a recursive function, the type declaration of the function needs to be modified to encode the property of interest. This may be a nuisance if the function has been already defined, for example in another theory, or if the property to be proved is just a special case that significantly restricts the original type of the function. To overcome this problem, PVS provides *judgements*.

In PVS, a *judgement* declares an additional type constraint to an existing definition. Judgements are automatically used by the theorem prover, but they can also be used as lemmas in manual proofs. Judgments generate TCCs that must be discharged either manually or automatically. These judgements conditions guarantee the correctness of the additional type declarations. Two kinds of judgments are supported by PVS: regular judgments, which are declared with the clause **JUDGEMENT**, and recursive judgements, which are declared with the clause **RECURSIVE JUDGEMENT**. Recursive judgments can only be used on recursive definitions. Although regular judgements can also be declared on recursive definitions, recursive judgements are preferable since they generate TCCs that follow the recursive structure of the definitions. In other words, recursive judgments enable the use of the implicit induction approach on existing recursive definitions without modifying the original declarations.

The type constraints specified in Section 4.3 to the functions `maxf_it` and `maxf` can also be specified on the original declarations of the functions given in Section 4.1 as follows.

```
  maxf_it(nas:NAS,f:[Aircraft->real],
          i:upto(length(nas`acs)),max:real): RECURSIVE real = ...

  maxf(nas:NAS,f:[Aircraft->real]): real = ...

  maxf_it_rj : RECURSIVE JUDGEMENT
```

```
    maxf_it(nas:NAS,f:[Aircraft->real],i:upto(length(nas`acs)),
            max:(maxf_below?(nas,f,i)))
      HAS_TYPE (maxf_below?(nas,f,length(nas`acs)))

  maxf_j : JUDGEMENT
    maxf(nas:NAS,f:[Aircraft->real])
      HAS_TYPE (maxf_below?(nas,f,length(nas`acs)))
```

The recursive judgment maxf_it_rj generates 8 TCCs. All the TCCs but maxf_it_rj_TCC8 are automatically discharged by the type checker. The statement of maxf_it_rj_TCC8 is exactly the same as the statement of maxf_it_TCC6 in Section 4.3 and it is proved in the same way. The judgement maxf_j generates the following proof obligation.

```
maxf_j: OBLIGATION
  FORALL (f: [Aircraft -> real], nas: NAS):
    maxf_below?(nas, f, nas`acs`length)(maxf(nas, f));
```

This obligation is easily discharged using maxf_it_rj as a lemma.

Implicit induction usually produces simpler proofs than those proofs based on explicit induction schemas. However, users should be cautious of TCC reordering when using types for specifications. Since TCCs are mechanically generated by the type checker, the user cannot control the order in which they are generated. Therefore, a simple modification to the definition of a function may change the order in which TCCs in a theory are generated. Although most TCCs are automatically discharged by the type checker, manual proofs of TCCs can be lost after type checking a modified version of a theory. PVS provides mechanisms to recover *orphaned proofs*, i.e., proofs that get lost due to changes in a theory. A good practice when using types for specifications is to create explicit lemmas with the statements of important TCCs. Once these lemmas are proven, the important TCCs can be easily proven by using the lemmas . The proof of these TCCs may still get lost due to TCC reordering, but those proofs only involve one proof command.

4.5 Iterations

Tail recursions naturally represent iterations and loops of imperative programming languages. The definitions of functions maxf_it and maxf in Section 4.1 correspond to the following pseudo-code of an imperative procedure that computes the maximum value f(nas`acs(i))) for all aircraft in nas`acs.

```
  local max : real := 0;
  local i : int;
  for (i := 0; i < length(nas`acs); i++) {
    if (i = 0 OR max < f(nas`acs(i))) {
      max := f(nas`acs(i));
    }
  }
  return max;
```

For this type of iterations, the theory `for_iterate` in the PVS NASA Library `structures` provides the construct `for(m,n:int,init:T,f:ForBody(m,n)):T`. This construct has the intended semantics of the following imperative loop.

```
  local a : T := init;
  local i : int;
  for (i := m; i <= n; i++) {
    a := f(i,a);
  }
  return a;
```

The values `m` and `n` are the first and last indices of the iteration, respectively. The value `init` is the initial value of the accumulator. The function `f` is a function that has as parameter the current iteration index and accumulated value, and computes the next value of the accumulator. The return type of `f` and the type of the accumulator is a type parameter `T`.

Using the construct `for`, the function `maxf` can be defined as follows.

```
IMPORTING structures@for_iterate

  maxf(nas:NAS,f:[Aircraft->real]): real =
    for[real](0,length(nas`acs)-1,0,
      LAMBDA(i:below(length(nas`acs)),max:real):
        IF i=0 OR max < f(nas`acs(i)) THEN
          f(nas`acs(i))
        ELSE
          max
        ENDIF)
```

The theory `for_iterate` also provides the following induction schema on the number of iterations for proving than an invariant predicate is satisfied in all iterations.

```
  for_induction : THEOREM
    FORALL(m:int,(n:int| n >= m-1),init:T,f:ForBody(m,n),
           inv:PRED[[upto(n-m+1),T]]):
      (inv(0,init) AND
       FORALL (k:subrange(0,n-m),ak:T) : inv(k,ak) IMPLIES
                                         inv(k+1,f(m+k,ak)))
      IMPLIES
      inv(n-m+1,for(m,n,init,f))
```

In this theorem, the predicate `inv` has as parameters an iteration number and the value of the accumulator up to that iteration.

Given the definition of the function maxf above, the proof of maxf_sound proceeds as follows. The proof command (skeep) introduces Skolem constants. The proof command (expand "maxf") expands the definition of maxf.

```
  |-------
{1}   f(finseq_appl(nas`acs)(i)) <=
       for[real]
           (0, length(nas`acs) - 1, 0,
            LAMBDA (i: below(length(nas`acs)), max: real):
              IF i = 0 OR max < f(finseq_appl(nas`acs)(i))
                THEN f(finseq_appl(nas`acs)(i))
              ELSE max
              ENDIF)
```

The induction schema for the construct for is introduced with the proof command (lemma "for_induction[real]"). The command (inst? -) finds an appropriate instantiation for most of the universally quantified variables in the statement of the lemma. The only variable that is not automatically instantiated is inv, which corresponds to the invariant predicate. The proof command

```
(inst -1 "LAMBDA(n:upto(length(nas`acs)),max:real):
           maxf_below?(nas,f,n)(max)")
```

instantiates this variable. The core of the rest of the proof is the following sequent.

```
  |-------
[1]   FORALL (k: subrange(0, length(nas`acs) - 1), ak: real):
        maxf_below?(nas, f, k)(ak) IMPLIES
         maxf_below?(nas, f, k + 1)
                    (IF k = 0 OR ak < f(finseq_appl(nas`acs)(k))
                       THEN f(finseq_appl(nas`acs)(k))
                     ELSE ak
                     ENDIF)
```

This sequent is discharged by the proof command (grind), which simplifies the sequent and introduces the Skolem constant j!1, followed by (inst -5 "j!1"), and (assert).

In addition to the construct for, the theory for_iterate provides the constructs for_down, iterate_left, iterate_right, and their respective induction schemas.

```
 %% local a : T := init;
 %% local i : int;
 %% for (i := m; i >= m; i--) {
 %%   a := f(i,a);
 %% }
 %% return a;
 for_down(n,m:int,init:T,f:ForBody(m,n)) = ...

 for_down_induction : THEOREM
   FORALL(n:int,(m:int| m <= n+1),init:T,f:ForBody(m,n),
          inv:PRED[[upto(n-m+1),T]]):
     (inv(0,init) AND
      FORALL (k:subrange(0,n-m),ak:T) : inv(k,ak) IMPLIES
                                        inv(k+1,f(n-k,ak)))
     IMPLIES
     inv(n-m+1,for_down(n,m,init,f))

 %% local a : T = f(m);
 %% local i : int;
 %% for (i := m+1; i <= n; i++) {
 %%   a := a o f(i)
 %% }
 %% return a;
 iterate_left(m:int,n:upfrom(m),f:IterateBody(m,n),
              o:[[T,T]->T]) : T = ...

 iterate_left_induction : THEOREM
   FORALL(m:int,n:upfrom(m),f:IterateBody(m,n),o:[[T,T]->T],
          inv:PRED[[upto(n-m),T]]):
     (inv(0,f(m)) AND
      FORALL (k:below(n-m),ak:T) : inv(k,ak) IMPLIES
                                   inv(k+1,ak o f(m+k+1)))
     IMPLIES
     inv(n-m,iterate_left(m,n,f,o))

 %% local a : T = f(n);
 %% local i : int;
 %% for (i := n-1; i >= m; i--) {
 %%   a := f(i) o a
 %% }
 %% return a;
iterate_right(m:int,n:upfrom(m),f:IterateBody(m,n),
              o:[[T,T]->T]) : T = ...

 iterate_right_induction : THEOREM
   FORALL(m:int,n:upfrom(m),f:IterateBody(m,n),o:[[T,T]->T],
          inv:PRED[[upto(n-m),T]]):
     (inv(0,f(n)) AND
      FORALL (k:below(n-m),ak:T) : inv(k,ak) IMPLIES
                                   inv(k+1,f(n-k-1) o ak))
     IMPLIES
```

```
      inv(n-m,iterate_right(m,n,f,o))
```

5 Animation and Ground Evaluation

In addition to specifying formal models and proving properties about them, PVS also allows for the animation of functional specifications. Animation of specifications is supported for concrete types and expressions through the utility PVSio [9], which provides a user friendly interface to the PVS ground evaluator [19].[6]

5.1 Animations

In order to animate the theory `NAS` defined in Section 3, its type parameters `Identifier` and `State` must be instantiated. Generally, aircraft identifiers are represented by strings. An aircraft state may contain different information depending on the application. Here, it will be considered that the state of an aircraft consists of its current position and velocity vector. Positions and velocities can be specified in several ways, e.g., positions can be given in either Euclidean or Geodesic coordinates, velocity vectors can be given in either rectangular or polar coordinates. An instantiation of the theory `NAS`, where aircraft identifiers are strings and aircraft states are composed of a 3-dimensional position `s` and a 3-dimensional velocity vector `v` can be specified as follows.

```
NAS3D : THEORY
BEGIN
 IMPORTING vectors@vectors_3D
 State3D : TYPE = [# s,v : Vect3 #]
 IMPORTING NAS[string,State3D]
 ...
END NAS3D
```

The type `Vect3`, defined in the theory `vectors_3D` of the PVS NASA library `vectors`, is a record with fields `x`, `y`, and `z` of type `real`.

After the clause IMPORTING `NAS[string,State3D]`, the definitions and declarations in the theory `NAS` are instantiated with the corresponding parameters. For example, functions that compute upper bounds for the altitude, vertical speed, and ground speed of all aircraft in a NAS are defined as follows.

```
  max_z(nas:NAS): real =
    maxf(nas,LAMBDA(ac:Aircraft):ac`state`s`z)

  max_vz(nas:NAS): real =
    maxf(nas,LAMBDA(ac:Aircraft):ac`state`v`z)
```

[6] PVSio is a standard feature of PVS since PVS 5.0.

```
  max_gs(nas:NAS): real =
    maxf(nas,LAMBDA(ac:Aircraft):sqrt(sq(ac`state`v`x) +
                                     sq(ac`state`v`y)))
```

The function `nas_at`, with parameters `nas` of type `NAS` and `t` of type `postnat`, computes a new NAS where the states of aircraft in `nas` are projected `t` units of time, assuming constant velocity.

```
 ac_at(ac:Aircraft,t:real) : Aircraft =
    ac WITH [`state`s := ac`state`s + t*ac`state`v]

  nas_at(nas:NAS,t:real) : NAS =
    nas WITH [`acs`seq := LAMBDA(i:below(length(nas`acs))):
                            ac_at(nas`acs`seq(i),t)]
```

Particular instances of aircraft and NAS can be defined as follows. In these examples, distances are given in meters and times in seconds.

```
  ac0 : Aircraft = (#
    id := "AC0",
    state := (# s := (-90000,0,6000), % [m,m,m]
                v := (400,0,0) #)     % [m/s,m/s,m/s]
  #)

  ac1 : Aircraft = (#
    id := "AC1",
    state := (# s := (0,-78000,6000), % [m,m,m]
                v := (0,300,0) #)     % [m/s,m/s,m/s]
  #)

  ac2 : Aircraft = (#
    id := "AC2",
    state := (# s := (-110000,0,5500), % [m,m,m]
                v := (450,0,1) #)      % [m/s,m/s,m/s]
  #)

  nas0 : NAS = update(update(update(empty,ac0),ac1),ac2)
```

PVSio can be called from the PVS editor via the command `M-x PVSio` or from the command line via the utility `pvsio`. PVSio provides a simple read-and-eval loop where users enter PVS expressions and the tool outputs their evaluations. The following examples show simple animations of functions defined or imported in `NAS3D`.

```
<PVSio> find(nas0,"AC0");
==>
(# id := "AC0",
   state
```

```
    := (# s := (# x := -90000, y := 0, z := 6000 #),
          v := (# x := 400, y := 0, z := 0 #) #) #)

<PVSio> find(nas_at(nas0,600),"AC2");
==>
(# id := "AC2",
   state
     := (# s := (# x := 160000, y := 0, z := 6100 #),
           v := (# x := 450, y := 0, z := 1 #) #) #)

<PVSio> max_z(nas0);
==>
6000

<PVSio> max_z(nas_at(nas0,600));
==>
6100

<PVSio> max_vz(nas0);
==>
1

<PVSio> max_gs(nas0);
Hit uninterpreted term sqrt.sqrt during evaluation
```

The reason for the error message in the animation of max_gs is that it uses the function sqrt, which is defined in the theory sqrt of the PVS NASA Library reals as an uninterpreted function that satisfies sqrt(x)*sqrt(x)=x for any non-negative real x. A proven TCC in the theory sqrt states that such as function exists. However, the proof is not constructive. Therefore, the expression max_gs(nas0) is not a ground term. Similarly, the expression nas0‘acs cannot be evaluated, even though it is a finite sequence of ground elements of type Aircraft. However, finite sequences are functional closures, which can only be ground evaluated when applied to ground arguments. The PVS prelude library provides the function finseq2list that translates finite sequences into lists, which can be evaluated.

```
<PVSio> nas0‘acs;
==>
Result not ground. Cannot convert back to PVS.

<PVSio> finseq2list(nas0‘acs);
==>
(: (# id := "AC0",
      state
        := (# s := (# x := -90000, y := 0, z := 6000 #),
             v := (# x := 400, y := 0, z := 0 #) #) #),
   (# id := "AC1",
      state
        := (# s := (# x := 0, y := -78000, z := 6000 #),
             v := (# x := 0, y := 300, z := 0 #) #) #),
   (# id := "AC2",
      state
        := (# s := (# x := -110000, y := 0, z := 5500 #),
             v := (# x := 450, y := 0, z := 1 #) #) #) :)
```

By default, PVSio ignores the TCCs of the expressions that are being evaluated. Therefore, some ground expressions are not safely evaluated in PVSio. Indeed, the evaluation of expressions whose type conditions are not satisfied may, and most probably will, break the execution of PVSio. For safer executions, the user can turn on the generation of TCCs. In this case, PVSio will request confirmation from the user that TCCs are satisfied, before proceeding with any evaluation. The commands `tccs` and `notccs` turn on and off, respectively, the generation of TCCs.

```
<PVSio> tccs;
Enabled TCCs generation

<PVSio> find(nas0,"AC3");
Typechecking "find(nas0,\"AC3\")" produced the following TCCs:

subtype TCC for "AC3": id_nas?[string, State3D](nas0)("AC3")

Evaluating in the presence of unproven TCCs may be unsound
Do you wish to proceed with evaluation?(Y or N): N
```

PVSio supports evaluations of bounded quantification over integers. For example, the function `cdnas3d?`, which checks if there is a conflict between two aircraft in a NAS, can be animated in PVSio. A conflict between two aircraft is specified as a predicted horizontal separation less than `D` and vertical separation less than `H`, within a lookahead time `T`, where `D`, `H`, and `T` are given positive constants.

```
  D : posreal = 10000 % [m]
  H : posreal =   250 % [m]
  T : posreal =   180 % [s]

  IMPORTING ACCoRD@cd3d[D,H,0,T]

  cdnas3d?(nas:NAS) : bool =
    EXISTS(k:below(length(nas`acs)),
           i:subrange(k+1,length(nas`acs)-1)):
        LET aco = nas`acs(k)`state,
          aci = nas`acs(i)`state IN
        cd3d?(aco`s-aci`s,aco`v-aci`v)
```

The PVS NASA Library ACCoRD provides several Air Traffic Management algorithms, including `cd3d?`, a formally verified algorithm for checking conflicts between two aircraft. The parameters of `cd3d?` are the relative position and velocity vector of the aircraft. The function `cdnas3d?` checks if there exist indices `k` and `i` in `nas`acs`, with `k < i`, where the function `cd3d?` returns TRUE for the aircraft `nas`acs(k)` and `nas`acs(i)`.

```
<PVSio> cdnas3d?(nas0);
==>
FALSE

<PVSio> cdnas3d?(nas_at(nas0,50));
==>
TRUE
```

5.2 PVS as a Functional Programming Language

PVSio supports the evaluation of uninterpreted functions by attaching Lisp code to PVS functions. This feature is called *semantic attachments* [2] and greatly extends the functionality provided by PVSio. For instance, semantic attachments enable the animation of real valued functions, such as trigonometric functions, square root, exponential, etc., and the execution of imperative features, such as input/output operations, within PVS functional specifications.

Semantic attachments are written in a file `pvs-attachments` in the working directory. For example, a semantic attachment for the PVS function `sqrt` can defined as follows.

```
(defattach |sqrt.sqrt| (x) (sqrt x))
```

In this definition, the first occurrence of `sqrt` refers to the name of the PVS theory where the function `sqrt` is defined. The Lisp expression (`sqrt x`) represents the application of the Lisp function `sqrt` to the variable `x`. This semantic attachment enables the animation of functions involving the uninterpreted PVS function `sqrt`.

```
<PVSio> sqrt(2);
==>
1.4142135

<PVSio> max_gs(nas0);
==>
450.0
```

Since the semantic attachment of the PVS function `sqrt` is the Lisp function `sqrt`, which implements a floating-point approximation of the square root function, the following evaluation should not be surprising.

```
<PVSio> sqrt(2)*sqrt(2)=2;
==>
FALSE
```

This example shows that evaluations in the presence of semantic attachments are not necessarily sound with respect to the PVS logic.

PVSio provides a PVS library of theories supported by predefined semantic attachments, e.g., `stdmath`, which provides attachments for floating point operations, `stdstr`, which provides attachments for string operations, `stdio`, which provides attachments for input/output operations, etc. This PVSio library extends the PVS formal specification language with basic programming language features. For example, PVS functions that pretty-print aircraft and NAS information can be defined as follows.

```
NASio : THEORY
BEGIN

  IMPORTING NAS3D

  ac2str(ac:Aircraft): string =
    LET s = ac`state`s,
        v = ac`state`v IN
      ac`id+
      " ("+s`x+"[m],"+s`y+"[m],"+s`z+"[m])"+
      " ("+v`x+"[m/s],"+v`y+"[m/s],"+v`z+"[m/s])"

  printac(ac:Aircraft) : void =
    print(ac2str(ac))

  printnas(nas:NAS) : void =
    FORALL(i:below(nas`acs`length)) :
      printac(nas`acs(i)) & print(newline)

  ...

END NASio
```

The operation + on strings is defined in `stdstr` and is attached to a Lisp function that concatenates strings. The function `print` is defined in `stdio` and is attached to a Lisp function that prints a string to standard output. The type `void`, which is defined in the theory `stdlang` as an alias to the type `bool`, is intended to be used as the type of procedures that do not return any particular value.

By convention, functions that use PVSio definitions are specified on a different theory separated from non-PVSio supported definitions. However, from the point of view of the type checker and the theorem prover, there is nothing special about `NASio`, i.e., PVSio-supported theories are considered as any other theory. The semantic attachments that support the evaluation of PVSio functions only exist within the PVSio interface. The following examples show animations of functions defined in `NASio`.

```
<PVSio> printac(find(nas0,"AC0"));
AC0 (-90000[m],0[m],6000[m]) (400[m/s],0[m/s],0[m/s])

<PVSio> printnas(nas0);
AC0 (-90000[m],0[m],6000[m]) (400[m/s],0[m/s],0[m/s])
AC1 (0[m],-78000[m],6000[m]) (0[m/s],300[m/s],0[m/s])
AC2 (-110000[m],0[m],5500[m]) (450[m/s],0[m/s],1[m/s])

<PVSio> printnas(nas_at(nas0,50));
AC0 (-70000[m],0[m],6000[m]) (400[m/s],0[m/s],0[m/s])
AC1 (0[m],-63000[m],6000[m]) (0[m/s],300[m/s],0[m/s])
AC2 (-87500[m],0[m],5550[m]) (450[m/s],0[m/s],1[m/s])
```

Simple applications can be written using PVSio. For example, the procedure `cdnas3dio` implements a program that

1. reads a filename `file` and a time `t`,
2. checks if a file `file` exists and, if this is the case, opens the file and loads it in an element `nas` of type `NAS`,
3. computes the projection of `nas` at time `t`, and
4. prints the identifiers of the aircraft that are in conflict in the projected NAS.

```
 readvect(t:Tokenizer) : [Vect3,Tokenizer] =
   LET t = accept_real(t), x = val_real(t),
       t = accept_real(t), y = val_real(t),
       t = accept_real(t), z = val_real(t),
       v : Vect3 = (x,y,z) IN
       (v,t)

 readac(s:string) : Aircraft =
   LET t  = go_next(str2tokenizer(s,empty_tokenizer)),
       id = last_token(t),
       (s,t) = readvect(t),
       (v,t) = readvect(t) IN
     (# id := id, state := (# s:=s,v:=v #) #)

 loadnas(file:string) : NAS =
   fmap_line(fopenin(file),empty,
                        LAMBDA(s:string,nas:NAS):
                           update(nas,readac(s)))

  cdnas3dio : void =
    LET file = query_line("Filename: ") IN
      IF fexists(file) THEN
        LET t = query_real("Time [sec]: "),
            nas = nas_at(loadnas(file),t) IN
          println("NAS:") &
          printnas(nas) &
          println("---") &
          FORALL(k:below(length(nas`acs)),
                 i:subrange(k+1,length(nas`acs)-1)):
            LET aco = nas`acs(k)`state,
                aci = nas`acs(i)`state IN
              IF cd3d?(aco`s-aci`s,aco`v-aci`v) THEN
                println("Conflict between "+nas`acs(k)`id+" and "+
                                            nas`acs(i)`id)
              ELSE skip
              ENDIF
      ELSE
         print("File "+file+" not found")
      ENDIF
```

The procedure `cdnas3dio` can be executed from the command line as a standalone application.[7]

```
$ pvsio examples@nasio:cdnas3dio
Filename:
xxxx
File xxxx not found

$ pvsio examples@nasio:cdnas3dio
Filename:
traffic.txt
Time [sec]:
0
NAS:
AC0 (-90000[m],0[m],6000[m]) (400[m/s],0[m/s],0[m/s])
AC1 (0[m],-78000[m],6000[m]) (0[m/s],300[m/s],0[m/s])
AC2 (-110000[m],0[m],5500[m]) (450[m/s],0[m/s],1[m/s])
---

$ pvsio examples@nasio:cdnas3dio
Filename:
traffic.txt
```

[7] Messages generated by the Lisp runtime have been suppressed.

```
Time [sec]:
100
NAS:
AC0 (-50000[m],0[m],6000[m]) (400[m/s],0[m/s],0[m/s])
AC1 (0[m],-48000[m],6000[m]) (0[m/s],300[m/s],0[m/s])
AC2 (-65000[m],0[m],5600[m]) (450[m/s],0[m/s],1[m/s])
---
Conflict between AC0 and AC1
Conflict between AC0 and AC2
Conflict between AC1 and AC2
```

The file `traffic.txt` contains the following information.

```
AC0 -90000 0 6000 400 0 0
AC1 0 -78000 6000 0 300 0
AC2 -110000 0 5500 450 0 1
```

5.3 Ground Evaluation in the Theorem Prover

PVSio provides a safe integration of the ground evaluator into the theorem prover. The proof command (`eval` *e*), where *e* is a PVS expression, evaluates the expression *e* and prints the result.

```
  |-------
{1}   max_z(nas_at(nas0, 600)) > 6000

Rule? (eval "max_z(nas_at(nas0, 600))")
max_z(nas_at(nas0, 600)) = 6100

 |-------
{1}   max_z(nas_at(nas0, 600)) > 6000
```

The proof command (`eval` *e*) never changes the current sequent, i.e., independently of the expression *e*, it behaves exactly as (`skip`). In contrast, the proof rule (`eval-expr` *e*) introduces the formula $e = \bar{e}$, where $\bar{e}$ is the ground evaluation of *e*, to the antecedent of the sequent.

```
Rule? (eval-expr "max_z(nas_at(nas0, 600))")

{-1}  max_z(nas_at(nas0, 600)) = 6100
  |-------
[1]   max_z(nas_at(nas0, 600)) > 6000
```

In this case, the resulting sequent is discharged by (`assert`).

The proof command (`eval-formula` *n*), where *n* is a formula number in the sequent, ground evaluates the formula and discharges the sequent if it reduces to **FALSE** in the antecedent or to **TRUE** in the consequent.

```
  |-------
{1}   max_z(nas_at(nas0, 600)) > 6000

Rule? (eval-formula 1)
Q.E.D.
```

A fundamental difference between the proof rule `eval` and the proof rules `eval-expr` and `eval-formula` is illustrated by the following example.

```
Rule? (eval "max_gs(nas0)")
max_gs(nas0) = 450.0

  |-------
{1}   max_gs(nas0) > 400

Rule? (eval-expr "max_gs(nas0)")
Function sqrt.sqrt is defined as a semantic attachment.
It cannot be evaluated in a formal proof.

  |-------
{1}   max_gs(nas0) > 400

Rule? (eval-formula 1)
Function sqrt.sqrt is defined as a semantic attachment.
It cannot be evaluated in a formal proof.

  |-------
{1}   max_gs(nas0) > 400
```

The evaluations performed by `eval` are exactly as the evaluations performed by PVSio, i.e, they use semantic attachments if these are available. This behavior is safe in the theorem prover since `eval` does not change the sequent in any way. On the other hand, semantic attachments are always disabled in the evaluations performed by `eval-expr` and `eval-formula` since these attachments are potentially unsound.

5.4 Computational Reflection

Since the proof rules `eval-expr` and `eval-formula` are sound, or at least as sound as the rest of the PVS system, a proof technique known as *computational reflection* [5] can be used in PVS to mechanically solve some kinds of problems.

Given a logical system $\mathcal{L}$, e.g., PVS, a computational reflection approach to solve problems of the form $\vdash_{\mathcal{L}} p$, where p belongs to a family P of formulas, consists in finding

– another family of formulas Q for which a decision procedure for $\vdash_{\mathcal{L}} q$, with q in Q, exists, and

– a transformation f that maps formulas p in P into formulas in Q, such that $\vdash_{\mathcal{L}} f(p) \Longrightarrow p$.

Therefore, a proof of $\vdash_{\mathcal{L}} p$ can be obtained by first discharging $\vdash_{\mathcal{L}} f(p)$ via the decision procedure for Q and then using the fact that $\vdash_{\mathcal{L}} f(p) \Longrightarrow p$. This approach is complete if $\vdash_{\mathcal{L}} f(p) \Longleftrightarrow p$. If the formulas in Q are ground, then the decision procedure is simply the ground evaluator. Furthermore, in a proof assistant such as PVS, which provides an expressive strategy language, the transformation function f can be a defined at the meta-theoretical level using the strategy language. This way, a mechanical approach to solve formulas in P becomes available.

For example, assume that an air traffic manager user is interested in proving theorems of the form `conflictnas3d?`(c), where c is a known constant of type `NAS` and `conflictnas3d?` is a predicate on elements of type `NAS` defined as follows.

```
conflictnas3d?(nas:NAS) : bool =
  EXISTS(k:below(length(nas`acs)),
         i:subrange(k+1,length(nas`acs)-1)):
      LET aco = nas`acs(k)`state,
          aci = nas`acs(i)`state IN
       conflict_3D?(aco`s-aci`s,aco`v-aci`v)
```

The predicate `conflict_3D?`, on a position `s` and velocity vector `v`, is defined in the theory `cd3d` of the PVS NASA Library `ACCoRD`. It specifies the existence of a time `t`, less than `T`, where the position `s + t*v` is in the interior of a cylinder of radius `D` and height `2*H` centered at the origin. It is recalled that `D` and `H` are given as parameters to the theory `cd3d`.

A direct proof of a formula of the form `conflictnas3d?`(c) requires finding instantiations for the existentially quantified variables `k` and `i` in the definition of `conflictnas3d?`, and `t` in the definition of `conflict_3D?`. Since the type of `t` is `nnreal`, the predicate `conflictnas3d?` cannot be ground evaluated.

An indirect approach to prove `conflictnas3d?`(c) proceeds by first showing the following general theorem.

```
cdnas3d : THEOREM
  FORALL(nas:NAS):
    cdnas3d?(nas) IFF conflictnas3d?(nas)
```

Theorem `cdnas3d` can be easily proved since the theory `cd3d` provides a lemma that states that the predicates `cd3d?` and `conflict_3D?` are equivalent. Therefore, proving `conflictnas3d?`(c) is equivalent to proving the ground Boolean expression `cdnas3d?`(c). The truth value of the expression `cdnas3d?`(c) can be found by using the ground evaluator.

In this case, all the elements for developing a mechanical approach to prove formulas of the form `conflictnas3d?`(c) by computational reflection are available. The strategy `conflictnas3d`, defined in the file `pvs-strategies` in the working directory, implements this approach.

```
(defstep conflictnas3d ()
2    (then (lemma ”cdnas3d”)
3          (inst? –1)
4          (assert)
5          (invoke (eval–formula $1n) (? * ”cdnas3d?”)))
6    ”Decision procedure for formulas of the form conflictnas3d?(nas),
7  where nas is a known constant element of type NAS.”
8    ”Discharging the predicate conflictnas3d?”)
```

In PVS, strategies are written in a proof scripting language that consists of proof commands, proof combinators, which provide control structure, and Lisp functions, which provide read-only access to the internal representation of proof sequents. The strategy `conflictnas3d` is defined in line 1 without any parameters. The body of the strategy, lines 2–5, starts with the proof combinator **THEN** that applies in sequence the proof commands given as arguments. The first command is (`lemma` ”`cdnas3d`”), which introduces the lemma as formula –1 in the current sequent. The proof command (`inst?` –1) automatically instantiates this formula. The resulting sequent is simplified with the proof command (`assert`). The last command in the proof script is the most interesting. It uses the proof rule `invoke`, which is defined in the Manip package [4], with the arguments (`eval–formula` $1n) and (? * ”`cdnas3d?`”). The second argument finds a formula in the sequent that matches the string ”`cdnas3d?`”. The first argument applies the proof rule `eval–formula` to the formula number of the formula specified by the second argument. Lines 6–7 specify the documentation string displayed when the command (`help conflictnas3d`) is issued. Line 8 specifies the string that is displayed when the strategy is used in the theorem prover.

The following examples illustrate the use of the strategy `conflictnas3d`.

```
{–1}  conflictnas3d?(nas0)
  |–––––––

Rule? (conflictnas3d)
Discharging the predicate conflictnas3d?,
Q.E.D.

  |–––––––
{1}   conflictnas3d?(nas_at(nas0, 50))

Rule? (conflictnas3d)
Discharging the predicate conflictnas3d?,
Q.E.D.
```

When a user-defined strategy such as `conflictnas3d` is introduced in the theorem prover, another strategy named `conflictnas3d$` is automatically introduced as well. This new strategy behaves exactly as `conflictnas3d`, but instead of being a blackbox, it automatically expands its definition when used inside the theorem prover. This way, proofs involving user-defined strategies can always be expanded to strategy-free proofs only supported by basic proof rules.

6 Conclusion

This paper illustrates the use of some advanced features in PVS through a simple formal model of a National Aerospace System. In this model, PVS has been used to perform a typical verification task from formal specifications to formal proofs and testing. The main PVS features covered by this task are dependent types and predicate subtyping for specifications, tail recursion, implicit induction, iterative functions, rapid prototyping, computational reflection via ground evaluation, and strategy writing.

PVS is under active maintenance and development by SRI International and members of the PVS community such as the Formal Methods group at NASA Langley. In this survey, several PVS enhancements and features have been omitted. For instance, record and tuple type extensions, structural record subtyping, co-inductive types, sum types, and theory interpretations [17] are recent enhancements to the specification language.[8] Packages for manipulation of algebraic expressions [4], simplification of real number expressions [11], and batch proving [10] are now part of the standard distribution of PVS. Powerful automated strategies for real-valued functions [3] and multivariate polynomials [12] are available as part of the PVS NASA Libraries. These strategies are fully written in the PVS strategy language and do not rely on any external oracle. More experimental extensions to the theorem prover include a random test generator based on work done in Haskell and Isabelle [13], a strategy based on the SMT (Satisfiability Modulo Theories) solver Yices, and simplification procedures for propositional and temporal logic based on a BDD (Binary Decision Diagrams) package. PVS tools under development and currently unavailable include, a generator of Java code from PVS functional specifications [6], a dimensional analysis tool, and a package for termination analysis based on work done in ACL2s [8].

Last but not least, libraries of theories ranging from fundamental mathematical developments to general frameworks for the analysis of fault-tolerant protocols and air traffic management systems are available from repositories such as the PVS NASA Libraries and the PVS libraries by the University of Seville[9]. Despite its age, the Program Verification System continues to be a solid system. The rich specification language, powerful theorem prover, large set of contributions, and pragmatic approach to formal verification distinguish PVS from other proof assistnat and make it particularly well-adapted to the formal analysis of safety-critical complex applications. For these reasons, PVS has played a major role on several formal verifications projects at NASA Langley for more than 10 years.

Acknowledgements The author are especially thankful to Sam Owre and Natarajan Shankar for their assistance with PVS during the preparation of this document.

[8] `http://pvs.csl.sri.com/pvs-release-notes/pvs-release-notes.html`.
[9] `http://www.glc.us.es/wiki/Theories#Theories_in_PVS`.

References

1. Archer, M., Vito, B.D., Muñoz, C.: Developing user strategies in PVS: A tutorial. In: Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics STRATA'03. NASA/CP-2003-212448, NASA LaRC,Hampton VA 23681-2199, USA (September 2003)
2. Crow, J., Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: Evaluating, testing, and animating PVS specifications. Tech. rep., Computer Science Laboratory, SRI International, Menlo Park, CA (March 2001)
3. Daumas, M., Lester, D., Muñoz, C.: Verified real number calculations: A library for interval arithmetic. IEEE Transactions on Computers 58(2), 226–237 (February 2009)
4. Di Vito, B.: A PVS prover strategy package for common manipulations. Technical Memorandum NASA/TM-2002-211647, NASA Langley Research Center (2002)
5. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK (1995)
6. Lensink, L., Smetsers, S., van Eekelen, M.C.J.D.: Generating verifiable Java code from verified PVS specifications. In: Goodloe, A., Person, S. (eds.) Proceedings of the 4th International Symposium NASA Formal Methods. Lecture Notes in Computer Science, vol. 7226, pp. 310–325. Springer (2012)
7. Lüttgen, G., Muñoz, C., Butler, R., Vito, B.D., Miner, P.: Towards a customizable PVS. Contractor Report NASA/CR-2000-209851, ICASE, Langley Research Center, Hampton VA 23681-2199, USA (January 2000)
8. Manolios, P., Vroon, D.: Termination analysis with calling context graphs. In: Lecture Notes in Computer Science. vol. 4144, pp. 401–414. Springer (2006)
9. Muñoz, C.: Rapid prototyping in PVS. Contractor Report NASA/CR-2003-212418, NASA, Langley Research Center, Hampton VA 23681-2199, USA (May 2003)
10. Muñoz, C.: Batch proving and proof scripting in PVS. Report NIA Report No. 2007-03, NASA/CR-2007-214546, NIA-NASA Langley, National Institute of Aerospace, Hampton, VA (February 2007)
11. Muñoz, C., Mayero, M.: Real automation in the field. Contractor Report NASA/CR-2001-211271, ICASE, Langley Research Center, Hampton VA 23681-2199, USA (December 2001)
12. Muñoz, C., Narkawicz, A.: Formalization of a representation of Bernstein polynomials and applications to global optimization. Journal of Automated Reasoning (2012), accepted for publication
13. Nipkow, S.B.T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) Software Engineering and Formal Methods (SEFM 2004). pp. 230–239. IEEE Computer Society (2004)
14. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) Proceeding of the 11th International Conference on Automated Deductioncade. Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer (June 1992)
15. Owre, S., Shankar, N.: Abstract datatypes in PVS. Contractor Report NASA/CR-97-206264, NASA, Langley Research Center, Hampton VA 23681-2199, USA (November 1997)
16. Owre, S., Shankar, N.: The formal semantics of PVS. Technical Report CSL-97-2, Computer Science Laboratory, SRI International (March 1999)
17. Owre, S., Shankar, N.: Theory interpretations in PVS. Technical report, SRI International, Menlo Park, CA (April 2001)

18. Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in PVS. IEEE Transactions on Software Engineering 24(9), 709–720 (September 1998), `http://www.csl.sri.com/papers/tse98/`
19. Shankar, N.: Efficiently executing PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA (November 1999)