SRI International

March 30, 2001

Evaluating, Testing, and Animating PVS Specifications

Judy Crow, Sam Owre, John Rushby, N. Shankar, and Dave Stringer-Calvert
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

This research was supported by NASA Langley Research Center under contract NAS1-20334.

Abstract

We explore ways to enhance the utility of PVS for evaluating, testing, and animating PVS specifications. The PVS ground evaluator is the focus of the work. We describe a mechanism to provide semantic attachments for PVS symbols while preserving soundness, and discuss strategies to provide a generic framework for integrating independently developed applications with PVS. We explore these capabilities in the current system, but conclude that more effective functionality requires extensions to PVS. Recommendations for these extensions are outlined.

Contents

List of Figures

Chapter 1

The PVS Ground Evaluator

The work described in this report exploits the PVS ground evaluator to enhance the utility of PVS for evaluating, testing, and animating PVS specifications. We begin the discussion with a brief functional overview of the PVS ground evaluator. Subsequent chapters discuss enhancements to provide semantic attachments for PVS symbols while perserving soundness, and strategies to provide a generic framework for integrating independently developed applications with PVS.

Specification languages like PVS are designed to be expressive rather than executable. However, a surprisingly large fragment of PVS turns out to be executable as a functional language; all ground expressions of ground type are evaluable.[1] The use of static analysis to determine safe destructive updates yields excellent efficiency; for certain types of applications, PVS serves effectively as a programming language.

The PVS ground evaluator consists of a translator from an executable subset of PVS into Common Lisp, a proof rule, and an evaluation environment. The proof rule provides evaluation of ground PVS expressions in the prover. The evaluation environment is an interactive read-eval-print loop that reads expressions from the user and returns the result of their evaluation. The translation and compilation of PVS expressions is performed lazily, that is, on demand; the first use of a particular definition will cause the evaluator to proceed more slowly than subsequent evaluations. The subset of PVS that is handled by the translation is large—the unexecutable fragments are uninterpreted functions, nonbounded quantification, free variables, and higher-order equalities. However, the evaluation of expressions is nonstrict, so expressions may be evaluable even if they contain nonevaluable subexpressions. Evaluator options include display of timing information, use of destructive updates, and conversion of the result back into PVS syntax. For example, ground evaluation

[1]Ground types do not contain any higher-order or uninterpreted types; they are formed from the base types `bool` and `integer` by means of tuples, enumeration types, and recursive datatypes. Ground expressions do not contain free variables, uninterpreted functions or constants, quantification over infinite domains, or equalities between higher-order terms (i.e., functions).

of the factorial function defined in Chapter 2, Section 2.1 with arguments 12 and 120, respectively, yields the following results.

```
<GndEval> "factorial(12)"
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  0 msec
; space allocation:
;  3 cons cells, 0 symbols, 0 other bytes, 0 static bytes
==>
479001600

<GndEval> "factorial(120)"
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  0 msec
; space allocation:
;  3 cons cells, 0 symbols, 6,216 other bytes, 0 static bytes
==>
66895029134491270575881180540903725867527463331380298102956713523016335572
44962989366874165271984981308157637893214090552534408589408121859898481114
389650005964960521256960000000000000000000000000000
```

The PVS ground evaluator was developed to speed up deduction in proofs containing ground expressions[2], and to support the animation, validation, and testing of PVS specifications by executing them on concrete data. In the following chapters, we explore ways to enhance the utility of the current ground evaluator for these activities. Our experiments suggest that more effective enhancements require extensions to PVS; we outline recommendations for these extensions in the concluding chapter.

[2]This capability is planned, but not currently implemented. Cf. Chapter 4.

Chapter 2

Semantic Attachments

Confronted with the need to demonstrate the truth of a proposition such as $\sin(\frac{13}{25}) < \frac{1}{2}$, it will have occurred to many users of theorem provers that it would be expedient to call a mathematical subroutine library, or a numerical or symbolic algebra system to evaluate $\sin(\frac{13}{25})$ rather than laboriously develop the properties of the sine function from some primitive definitional or axiomatic basis. More generally, arbitrary computer programs could be associated with certain function or predicate symbols and the theorem prover could just run the associated program whenever it needs to evaluate the function or predicate concerned. Weyhrauch [18] coined the term *semantic attachment* for such programs "attached" to predicate or function symbols.[1] Semantic attachments are used quite widely in artificial intelligence and in other applications where logic is viewed as a programming or query language. Their use in a theorem proving environment presents more of a challenge because of the need to maintain soundness: the result of executing a semantic attachment must be consistent with any properties that can be deduced from those defined or axiomatized for the symbol. One way to view the PVS ground evaluator is as a mechanism that constructs semantic attachments for PVS symbols that are guaranteed sound with respect to the PVS definitions for those symbols (which is why in future, the PVS evaluator will be used in proofs, as well as simply for evaluation). However, the programming techniques used in the PVS ground evaluator to attach Lisp code to PVS functions are available to any user of PVS and create the opportunity for users to provide their own semantic attachments to PVS symbols.

This opportunity is a powerful one, but also dangerous, so we begin by discussing how best to use attachments, then describe how to program them, and conclude this chapter with some illustrations of their use.

[1]Semantic attachments are used to evaluate ground instances of function or predicate applications; *universal attachments* [16] allow programs to be attached to any expression.

2.1 Semantic Attachments Used To Compute Values

The previous section motivated semantic attachments with an example involving $\sin(\frac{13}{25})$. We could attach the built-in Lisp function `sin` to the PVS symbol `sin` so that evaluation of the PVS term `sin(13/25)` executes the Lisp program `(sin (/ 13 25))`. This yields the internal Lisp result 0.49688 but PVS can only represent rational constants, so we have to change the program to `(rationalize (sin (/ 13 25)))`, which produces `1513/3045`. However, this is not the value of $\sin(\frac{13}{25})$—it is merely a close approximation to its value. Thus, if we have suitable definitions or axioms for `sin` and related functions in PVS (e.g., $\sin(x)^2 + \cos(x)^2 = 1$), we will be able to prove `sin(13/25) /= 1513/3045`, thereby contradicting the result of evaluation, and exhibiting unsoundness. For this reason, it is often better to define semantic attachments that explicitly approximate PVS functions rather than ones that have a precise definition. For example, the Lisp program above could be attached to a PVS function `sin_approx`, which is related to `sin` by the axiom `abs(sin(x) - sin_approx(x)) < delta` for some suitable `delta` (that should be specified in the documentation of the subroutine library providing the `sin` program used).

Because of the ease with which it can introduce unsoundness, PVS flags any proofs that make use of evaluations involving user-defined semantic attachments. But, in fact, there is little need for users to define their own semantic attachments for the purpose of evaluating mathematical functions: the PVS ground evaluator generates such efficient code that these functions can be defined directly in PVS. For example, Dutertre [8] and Gottliebsen [11] provide verified libraries of definitions and properties for many mathematical functions. In particular, Gottliebsen gives the following definition for `sin(x)`, where `even?` is a predicate defined in the PVS prelude that tests if its argument is even, `fac` is the factorial function, and `suminf` (not shown here) denotes an infinite summation.

```
fac(n) : RECURSIVE nat =
 IF n = 0 THEN 1 ELSE n * fac(n-1) ENDIF
MEASURE n

sin_ser(n) : real =
 IF even?(n) THEN 0 ELSE ((-1) ^ ((n - 1) / 2)) / fac(n) ENDIF

sin(x) : real = suminf(LAMBDA n : sin_ser(n) * (x ^ n))
```

From these, we can define the function `sin(n, x)` that sums just the first `n` terms of the series for `sin(x)` and thereby provides an approximation to its value. We also need a constructive definition for `even?`, and this is also shown below.

```
even?(n): RECURSIVE bool =
 IF n = 0 THEN true ELSIF n = 1 THEN false ELSE even?(n-2) ENDIF
MEASURE n

sin(n, x): RECURSIVE real =
 IF n = 0 THEN 0 ELSE sin_ser(n) * (x ^ n) + sin(n-1,x) ENDIF
MEASURE n
```

With these definitions, PVS evaluates `sin(4,13/25)` in less than a millisecond to yield `46553/93750`, and `sin(12,13/25)` in 1 millisecond to yield `47287619319628640249713/95169067382812500000000`. Results proved in Gottliebsen's library can be used to establish a relationship between `sin(n, x)` and `sin(x)`.

2.2 Semantic Attachments Used for Side Effects

We have seen that it is potentially dangerous, and seldom necessary, to employ user-defined semantic attachments to compute values for PVS functions: instead, it is generally preferable to give constructive definitions in PVS and allow the PVS ground evaluator to generate efficient and sound attachments directly from the PVS specification.

The most useful application for user-defined semantic attachments is in constructing PVS functions with side effects. Side effects are, of course, alien to the purely functional world of PVS specifications, but their use allows the provision of services such as I/O and persistent state that are lacking in the PVS ground evaluator.

An example is a function `print` with the following signature.

```
printing[T: TYPE+]: THEORY
begin
  print(a: T): bool = true
END printing
```

We can make a semantic attachment to `print` that causes the value of its argument to be printed whenever the function is invoked by the PVS ground evaluator. Now suppose that we would like to examine the values of the PVS built-in function `id` (the identity function) on the bounded type `upto(10)`. In the PVS ground evaluator, we can evaluate individual elements such as `id[upto(10)](3)` and receive the result `3`, but we cannot examine the whole function: the query `id[upto(10)]` yields the error message[2] "`Result not ground.  Cannot convert back to PVS.`" However, using `print`, we can write the following query

[2]The query actually produces a closure, which is not ground and cannot be translated back into PVS. Cf. Section 3.3.

```
<GndEval> "FORALL (x:upto(10)): print(id(x))"
```

and receive the following output.

```
 0 1 2 3 4 5 6 7 8 9 10
```

This works because the PVS ground evaluator evaluates `FORALL` over a bounded type by enumerating the elements of the type and evaluating the quantified expression for each element in turn (using a Lisp loop) until it finds one that is `FALSE`, in which case it returns `FALSE`, or reaches the end of the enumeration, in which case it returns `TRUE`. Since `print` always returns `TRUE`, the example query enumerates the whole type and causes the corresponding values of the `id` function to be printed. Several more examples of semantic attachments with side effects are described later.

2.3 Programming an Attachment

To attach Lisp code to a PVS function, it is first necessary to register the name of the function as one having an attachment. This is done by the following Lisp form. Vertical bars are used to build case sensitive symbol names; the default is uppercase.

```
(push (mk-name '|print| nil '|printing|) *pvs2cl-primitives*)
```

Here `print` and `printing` should be replaced by the function and theory name, respectively.

Then, the attachment itself is defined as a Lisp defun.

```
(defun |PVS_print| (x) (format t "~a" x) t)
```

The name of the defun corresponding to PVS function `xxx` is `|PVS_xxx|` if the function takes a single argument, or `|PVS__xxx|` if it takes more than one argument.[3] The argument(s) to the defun will be the Lisp encodings of the arguments to the PVS function, and the defun must return the Lisp encoding of the intended result of the PVS function. The Lisp encodings of the PVS boolean values are `t` and `nil`, so the defun above returns the encoding of `true`, as required for consistency with the PVS specification given for `print`.[4] Notice that this attachment prints the value of the *Lisp encoding* of the argument to the PVS print function. To print the value as a PVS expression, we need a rather more complicated defun[5]

```
(defun |PVS__print| (x y)
   (format t "~a" (cl2pvs x (pc-typecheck (pc-parse y 'type-expr)))) t)
```

[3]This is not strictly correct, but is adequate for the programming purposes described here.

[4]The Lisp function `format` returns `nil` and prints its third argument according to the specification in its second argument when its first argument is `t`.

[5]The definition as stated should be used with care, since, as noted in Section 2.2, `cl2pvs` may fail; expressions that are not ground cannot be translated from Lisp back into PVS.

which is attached to the PVS function

```
pvsprint(a:T, s:string): bool = true
```

and has the effect of printing its first argument as a value of the type given as a string in its second argument (e.g., `print(id[upto(10)](3), "nat")`).[6] It is convenient to place all the Lisp code in the file `.pvs.lisp` located either in the home directory or the current context as this is automatically loaded by PVS as it starts up.[7]

2.4 Applications

As mentioned in the opening chapter, one of the basic motivations for the PVS ground evaluator is to support validation of PVS specifications by allowing constructive specifications to be tested by running them on concrete (ground) data. The read-eval-print loop of the ground evaluator allows the user to enter a PVS ground expression (in quotes), and then prints the result of evaluating that expression. This simple interaction has several limitations for the purposes of testing a specification. First, it provides only a single evaluation at a time: we can evaluate a function, for example, applied to specific arguments, but we cannot easily iterate over all (or several) values of its arguments. We could write a higher-order recursive function that builds a list containing the results of iterating over one argument, but it becomes quite difficult to do this for several arguments and, of course, we would have no control over the way the resulting list is printed, so the output could be difficult to interpret. Second, many PVS functions return complex data structures such as functions, sets, or records of these. The PVS ground evaluator does not regard any result involving a function as ground and will not print such values. Third, many PVS specifications define a collection of functions that maintain and access a system state. As PVS is a purely applicative language, these functions take the state as one of their arguments, and (possibly) return the modified system state as a value. Thus, the system state that results from the application of many operations is represented by a long series of nested function applications—and it is tedious to type these into the PVS ground evaluator. What we would like to say is "take the state that resulted from the last operation, and apply this operation to

[6]It is something of a kludge that the type must be given (particularly as a string) because this must surely already be known to the PVS system; the explanation is that at the point that `|PVS__pvsprint|` is invoked, its first argument has been evaluated and type information has been lost. The type is known to the PVS system at the point where it compiles the call to `|PVS__pvsprint|`, but causing it to supply the information would require a modification to PVS—and the point of the present discussion of attachments is to describe methods that allow users to perform simple augmentations without modifying the core PVS system. The improved development environment proposed for the ground evaluator in Chapter 4 would include this modification.

[7]If there is a `.pvs.lisp` file in both the home directory and current context, the file in the home directory is loaded before the one in the current context.

it": in other words, we would like to be able to save the results of PVS function evaluations in the state of the ground evaluator, and to be able to reference those saved values in later evaluations. The mechanisms for semantic attachments described above are sufficient to provide these and several other useful enhancements that improve the support for validation provided by the PVS ground evaluator. In the following sections we illustrate the utility of these mechanisms by applying them to two examples.

2.4.1 Printing and Iteration

The simple attachments for printing described in Sections 2.2 and 2.3 provide control over the way values are printed out, allow the values of functions to be printed, and also allow systematic enumeration of the arguments to a function. Those attachments need to be augmented a little to produce a generally useful collection of basic printing functions. A suitable library is the following

```
printing[T : TYPE+]: THEORY
BEGIN
  printf(a:T, typ: string, fmt: string): bool = true
    % attachment prints a as type typ with
    % Lisp format string fmt

  print(a:T, typ: string): bool = printf(a, typ, "~a ")

  dumpf(a:T, fmt: string): bool = true
    % attachment prints Lisp representation of a with
    % Lisp format string fmt

  dump(a:T): bool = dumpf(a, "~a ")

END printing

printstrings: THEORY
BEGIN
  IMPORTING printing[real]
  prints(s:string): bool = true
    % attachment prints Lisp format string s
  space: bool = prints(" ")
  tab: bool = prints("~0,8T")
  newline: bool = prints("
")
  printreal(a:real): bool = dumpf(a,"~g")
  readreal(a:string) : rational
    % attachment reads a number expressed in decimal from the string

END printstrings
```

The primitive functions `printf`, `dumpf`, and `prints` have the following attachments.

```
(push (mk-name '|printf| nil '|printing|) *pvs2cl-primitives*)
(defun |PVS__printf| (a typ fmt)
 (not (format t fmt (cl2pvs a (pc-typecheck (pc-parse typ 'type-expr))))))

(push (mk-name '|dumpf| nil '|printing|) *pvs2cl-primitives*)
(defun |PVS__dumpf| (a fmt) (not (format t fmt a)))

(push (mk-name '|prints| nil '|printstrings|) *pvs2cl-primitives*)
(defun |PVS_prints| (s) (not (format t s)))
```

Notice that `dumpf` is a slightly augmented version of the first attachment described for `print` in Section 2.3, while `printf` is a slightly augmented version of the second attachment described in that section: `dumpf` "dumps" the Lisp representation of its first argument, while `printf` prints it as a PVS value of the type given as a string in its second argument. The final argument to both `printf` and `dumpf` is a Lisp format string that determines how their first arguments will be printed; format strings are described in the Lisp Manual [17]. The simpler functions `print` and `dump` supply a default format string.

The `prints` function in the `printstrings` theory prints its single argument as a Lisp format string: if this is a string with no ~ directives, it will simply be printed out; ~ directives can be used to obtain control over spacing (as in the `tab` function). The functions `space`, `tab`, and `newline` print the characters suggested by their names.

The only real numbers that can be constructed in PVS are actually rationals, and are represented as such (i.e., as a pair of integers). The `printreal` function prints PVS reals as fixed or floating point real numbers. Using this, the sine examples in Section 2.1 can be changed to read `printreal(sin(4,13/25))`, which produces `0.49656534`, and `printreal(sin(12,13/25))`, which produces `0.49688014`. The dual capability is provided by the function `readreal`, which takes a string containing some representation of a real number as its argument, and returns it as a rational to PVS. Thus, for example, `readreal("3.14159")` returns `9918/3157`. The attachment for `readreal` is the following.

```
(push (mk-name '|readreal| nil '|printstrings|) *pvs2cl-primitives*)
(defun |PVS_readreal| (a) (rationalize (read-from-string a)))
```

We can illustrate these printing functions by applying them to a moderately complicated example. The `safer` example was introduced in a NASA guidebook [4,5] and later used to illustrate the animation features of VDM-SL [1]. The version used here is taken from a paper by Di Vito [7] whose PVS files are available at http://shemesh.larc.nasa.gov/people/bld/safer/. One of the main functions in this specification is `grip_command` (in file `model.pvs`), whose signature is

```
grip_command(grip: hand_grip_position,
             mode: control_mode_switch): six_dof_command
```

where the associated types are defined as follows.

```
axis_command:   TYPE = {NEG, ZERO, POS}

hand_grip_position: TYPE =
        [# vert, horiz, trans, twist: axis_command #]

control_mode_switch:      TYPE = {ROT, TRAN}

tran_axis:      TYPE = {X, Y, Z}
rot_axis:       TYPE = {roll, pitch, yaw}

tran_command:   TYPE = [tran_axis -> axis_command]
rot_command:    TYPE = [rot_axis  -> axis_command]

six_dof_command:   TYPE = [# tran: tran_command, rot: rot_command #]
```

The specification of the `grip_command` command function is quite complicated and involves many nested functions. It would be useful to enumerate the values of this function over its entire input space.

To start, we need to be able to print a `six_dof_command`, which is a record containing two functions. As noted earlier, the PVS ground evaluator does not print function values, but we can accomplish our purpose using the print attachments as follows.

```
print_six_dof(c:six_dof_command): bool =
    prints("TRAN: [ ") AND
    (FORALL (a:tran_axis): print(c`tran(a), "axis_command")) AND
    prints("] ~0,4T ROT: [ ") AND
    (FORALL (a:rot_axis): print(c`rot(a), "axis_command"))
    AND prints("]")
```

Notice that the general technique for printing a series of values is to exploit the fact that the printing attachments all return the Boolean value *true*, and can therefore be composed by conjunction: the ground evaluator evaluates each conjunct in turn seeking one that will falsify the overall expression. Because the `axis_command`s range from three to four characters in length, we use the Lisp format string `~0,4T` to tab to the nearest multiple of four characters between the `TRAN` and `ROT` `axis_commands`.

Now we can iterate over the arguments to `grip_command` and print the corresponding `six_dof_command` as follows.

```
print_grip_command: bool =
  FORALL (a:axis_command):
   FORALL (b:axis_command):
    FORALL (c:axis_command):
     FORALL (d:axis_command):
      FORALL (r:control_mode_switch):
        printf(a, "axis_command", "Grip input: ~5a ") AND
        printf(b, "axis_command", "~5a") AND
        printf(c, "axis_command", "~5a") AND
        printf(d, "axis_command", "~5a") AND
        printf(r, "control_mode_switch", "Mode: ~5a  Output: ") AND
        print_six_dof(
          grip_command((# vert:= a, horiz:= b, trans:= c, twist:= d #), r)
        ) AND
        newline
```

Here, we use another way to deal with the varying lengths of the strings: the Lisp format string `~5a` prints in the default format in a field of width 5. In the PVS ground evaluator, we can now type "print_grip_command" and receive the following 162 lines of output.

```
Grip input: NEG   NEG  NEG  NEG  Mode: ROT   Output: TRAN: [ NEG ZERO ZERO ]   ROT: [ NEG NEG NEG ]
Grip input: NEG   NEG  NEG  NEG  Mode: TRAN  Output: TRAN: [ NEG NEG NEG ]     ROT: [ ZERO NEG ZERO ]
Grip input: NEG   NEG  NEG  ZERO Mode: ROT   Output: TRAN: [ NEG ZERO ZERO ]   ROT: [ NEG ZERO NEG ]
Grip input: NEG   NEG  NEG  ZERO Mode: TRAN  Output: TRAN: [ NEG NEG NEG ]     ROT: [ ZERO ZERO ZERO ]
Grip input: NEG   NEG  NEG  POS  Mode: ROT   Output: TRAN: [ NEG ZERO ZERO ]   ROT: [ NEG POS NEG ]
Grip input: NEG   NEG  NEG  POS  Mode: TRAN  Output: TRAN: [ NEG NEG NEG ]     ROT: [ ZERO POS ZERO ]
Grip input: NEG   NEG  ZERO NEG  Mode: ROT   Output: TRAN: [ NEG ZERO ZERO ]   ROT: [ NEG NEG ZERO ]
Grip input: NEG   NEG  ZERO NEG  Mode: TRAN  Output: TRAN: [ NEG ZERO NEG ]    ROT: [ ZERO NEG ZERO ]
⋮

Grip input: POS   POS  POS  ZERO Mode: ROT   Output: TRAN: [ POS ZERO ZERO ]   ROT: [ POS ZERO POS ]
Grip input: POS   POS  POS  ZERO Mode: TRAN  Output: TRAN: [ POS POS POS ]     ROT: [ ZERO ZERO ZERO ]
Grip input: POS   POS  POS  POS  Mode: ROT   Output: TRAN: [ POS ZERO ZERO ]   ROT: [ POS POS POS ]
Grip input: POS   POS  POS  POS  Mode: TRAN  Output: TRAN: [ POS POS POS ]     ROT: [ ZERO POS ZERO ]
```

2.4.2 Saving State

We can save a PVS value in the Lisp system state using a function PVS `write` defined as follows

```
state[T: TYPE]: THEORY
BEGIN
  write(a:T): bool = true
    % attachment saves a in state
END state
```

with the following attachment.

```
(push (mk-name '|write| nil '|state|) *pvs2cl-primitives*)
(defun |PVS_write| (x) (progn (setq *pvsstate* x) t))
```

This attachment simply writes the Lisp encoding of the argument to `write` into the global variable `*pvsstate*` and returns the value `true`.

The corresponding operation to read the value of `*pvsstate*` back again should be a 0-ary function (i.e., a constant), but the PVS ground evaluator does not provide the necessary coding hooks for attachments to constants, so we define function `innerread` that takes a superfluous argument, and then introduce the constant `read` by means of a PVS definition.

```
  innerread(i:nat): T     % attachment returns saved state
  read: T = innerread(3)  % argument is ignored
```

The attachment for `innerread` is the following.[8]

```
(push (mk-name '|innerread| nil '|state|) *pvs2cl-primitives*)
(defun |PVS_innerread| (x) *pvsstate*)
```

We illustrate these attachments using the `phone_3` example from the PVS tutorial [6, pp. 16–20]. The PVS specification files are available at `ftp://pvs.csl.sri.com/pub/pvs/examples/wift-tutorial/`. This specification maintains a “phone book,” which is a simple database associating names with sets of phone numbers.

```
phone_3 : THEORY
BEGIN
  N: TYPE                        % names
  P: TYPE                        % phone numbers
  B: TYPE = [N -> setof[P]]      % phone books
  nm, x: VAR N
  pn: VAR P
  bk: VAR B
```

[8]Of course, the Lisp definition for the attachment depends on the application in which it is used. For example, Muñoz [15] has defined an attachment for write that takes a second parameter indicating where the value is to be written; his version supports a model of user memory implemented as a hash list.

```
  emptybook(nm): setof[P] = emptyset[P]

  FindPhone(bk, nm): setof[P] = bk(nm)

  AddPhone(bk, nm, pn): B = bk WITH [(nm) := add(pn, bk(nm))]

  DelPhone(bk,nm): B = bk WITH [(nm) := emptyset[P]]

  DelPhoneNum(bk,nm,pn): B = bk WITH [(nm) := remove(pn, bk(nm))]
END phone_3
```

To make this specification suitable for validation by evaluation, we need to instantiate the uninterpreted types `N` and `P` (of names and phone numbers, respectively) with suitable ground types. The elegant way to do this would be to make `N` and `P` into parameters to the theory, and then instantiate the theory with suitable concrete types. Unfortunately, `AddPhone` is considered unevaluable because it operates on uninterpreted types, even though we will only attempt to evaluate an instance on concrete types. It is therefore necessary to modify the type definitions for `N` and `P` directly in the specification. We do this as follows.

```
N: TYPE = {alice, bob, carol}
P: TYPE = upto(10)
```

Next we need a function to print out a set of phone numbers (the result of `FindPhone` is a set).

```
IMPORTING printing, printstrings

printphones(s: setof[P]): bool =
  prints(" ") AND
  (FORALL pn: IF member(pn, s) THEN print(pn,"P") ELSE true ENDIF) AND
  prints("")
```

Then we can type queries to the PVS evaluator such as the following

```
printphones(
  FindPhone(
    AddPhone(
     AddPhone(
      AddPhone(emptybook,alice,1),
     bob,2),
    carol,3),
  bob))
```

and receive the result `{ 2 }`. If we now wonder what happens to the "state" when `bob`'s phone numbers are deleted following the three `AddPhone`s, we must retype all this text again (actually, the Emacs command `M-p` recalls the previous query).

```
printphones(
  FindPhone(
   DelPhone(
    AddPhone(
     AddPhone(
      AddPhone(emptybook,alice,1),
     bob,2),
    carol,3),
   bob),
  bob))
```

Using the **read** and **write** functions described above, we can make this kind of exploration rather more convenient. The first query can be split into two as follows.

```
  write(
    AddPhone(
     AddPhone(
      AddPhone(emptybook,alice,1),
     bob,2),
    carol,3))

  printphones(FindPhone(read, bob))
```

The second query then becomes much simplified as follows.

```
  write(DelPhone(read,bob))

  printphones(FindPhone(read, bob))
```

We can ease the task of entering test cases still further by defining "testing" versions for each of the functions in the specification; these versions omit the phone book as an explicit argument and instead access its value implicitly through `read` and `write` calls to the state. For example, the testing versions of `DelPhone` and `FindPhone` are called `delphone` and `findphone`, respectively, and are defined as follows.

```
delphone(nm): bool = LET bk = read IN write(DelPhone(bk,nm))

findphone(nm): bool = LET bk = read IN printphones(FindPhone(bk,nm))
```

The testing functions are all derived from their parent functions in a similar manner, so we can define higher-order "lifting" functions to describe this transformation. We will need a different higher-order function for each signature of base function; here is the lifting function for the signature `[B,N,P->B]`.

```
lift((f: [B,N,P->B]), nm, pn): bool =
     LET bk = read[B], v = f(bk,nm,pn) IN write(v)
```

Then we can define the testing versions `addphone` and `delphonenum` of `AddPhone` and `DelPhoneNum`, respectively, as follows.

```
addphone(nm,pn): bool = lift(AddPhone,nm,pn)

delphonenum(nm,pn): bool = lift(DelPhoneNum,nm,pn)
```

We can also define a function `printbook` that prints an entire phone book, and a testing version `printb` that prints the phone book stored in the state.

```
printbook(bk): bool =
  FORALL nm:
    print(nm,"N") AND tab AND printphones(FindPhone(bk,nm)) AND newline

printb: bool = LET b = read[B] IN printbook(b)
```

With these definitions, we can conduct the test described earlier by performing the following evaluations in sequence (these can either be entered into the evaluator one at a time, or as a single conjunction).

```
write(emptybook)
addphone(alice, 1)
addphone(bob, 2)
addphone(carol, 3)
findphone(bob)
delphone(bob)
findphone(bob)
```

The first `findphone` prints the result { `2` } and the second { }, as we should expect. The further sequence

```
addphone(alice, 5) AND delphone(carol) AND printb
```

produces the following result.

```
alice   { 1 5 }
bob     { }
carol   { }
```

The `phone_3` and `safer` examples discussed in this section illustrate the enhanced support provided by semantic attachments for validation of PVS specifications in the ground evaluator. In the next chapter, we further exploit this mechanism to implement a graphical user interface for the ground evaluator.

Chapter 3

Integrating Applications with PVS

Graphical visualization is useful for communicating in a readily assimilable form the properties and behaviors captured by a formal specification. Although graphical formats have special merit when presenting a specification to an audience unfamiliar with formal notations or to a person or group new to a particular application domain, graphical tools provide effective interfaces to formal models for both novice and expert alike. Pressing a "button" on a flight autopilot display to command a mode change [14] and animating a 3-dimensional model to visualize thruster settings for a propulsive backpack system [1] exemplify the accessibility and effectiveness of graphical I/O. Although the mechanisms described in this section may be adapted to provide interfaces to other PVS components, we focus exclusively on Graphical User Interfaces (GUIs) for the ground evaluator. Our objective is to provide generic support for interface capabilities to enhance the evaluator's utility for testing, animating, documenting, and communicating formal specifications.

To accommodate interfaces that interact with PVS in various architectural configurations, as well as interfaces developed in a variety of graphical toolkits, we begin by considering design issues relevant to a general model of interaction. We then narrow the focus, illustrating design choices that instantiate the general framework with a particular model of interaction and a graphical Tcl/Tk interface.[1] The discussion is intended to provide those interested in integrating independently developed components and, more particularly, GUIs into PVS with a firm idea of the design space for such an undertaking, as well as suggested approaches and a generic framework to facilitate the task.

[1]Tk is a toolkit that adds about 35 Tcl commands to support the creation and manipulation of widgets in a graphical user interface. Hereafter, we refer simply to Tcl.

3.1 Combining PVS with Other Components

PVS is a Common-Lisp-based system accessed through a customized Emacs interface that runs as an interactive Lisp process and may be driven by any program that viably replaces the Emacs interface. We do not consider alternatives such as foreign function calls—e.g., as used to interface the PVS model checker—or interfaces that exploit PVS batch mode. Foreign function calls are attractive because they are a component of the PVS image and therefore provide the fastest and most tightly coupled communication. However, they also introduce a host of implementation issues, one of the most challenging of which is garbage collection. PVS batch mode is built directly on the underlying Emacs batch mode described in the GNU Emacs Manual [10]. With the possible exception of creating canned demonstrations, the utility of PVS batch mode for external component interfaces is not clear. For the remaining alternatives, combining PVS with independently developed components or providing GUIs for PVS opens a design space that may be explored along several dimensions. The major design decisions concern model of communication, context of interaction, and level of integration. We concentrate first and in most depth on communication models.

3.1.1 Communication Models

Independently developed components and graphical interfaces may be added to PVS in essentially two ways, depending on whether interprocess communication is implemented following the current model, or an alternate one. In either case, the resulting model may be seen as a variant of the following general model.

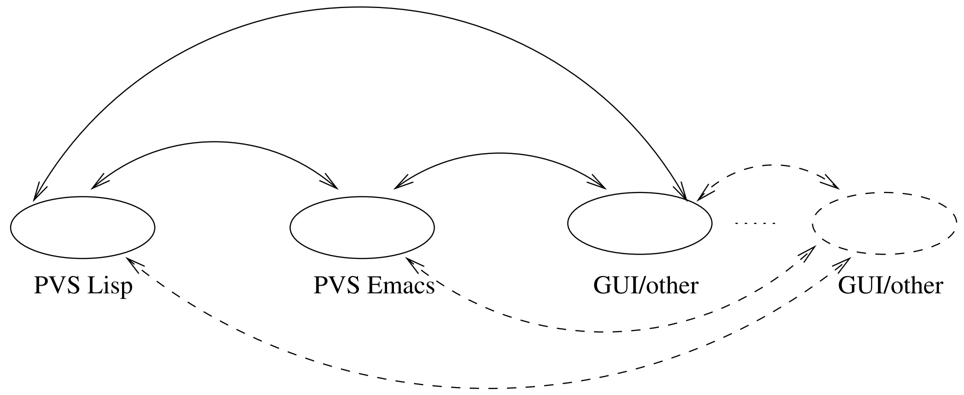


Figure 3.1: General Model of PVS Process Communication

3.1.1.1 A General Model of PVS Process Communication

The most general model of PVS interaction would provide full connectivity, with processes communicating directly as shown in Figure 3.1. Extensibility of this model is suggested by dashed lines outlining components and their connectivity. This form of distributed communication is not currently supported in PVS, and would require rethinking and enhancing the current messaging protocol, and possibly also the APIs for PVS components such as the parser, typechecker, and prover. The generality and inherent flexibility of this approach is appealing, but the actual utility of the fully general model is an open question. Alternate models of communication may be derived from this general model by restricting connectivity in various ways. For example, the current model of communication may be viewed as a variant in which connectivity is restricted and (communication) control centralized in a single component.

3.1.1.2 The Current Model of PVS Process Communication

The current model of PVS process communication, which provides the context for the work described in this chapter, appears in Figure 3.2. In this model,

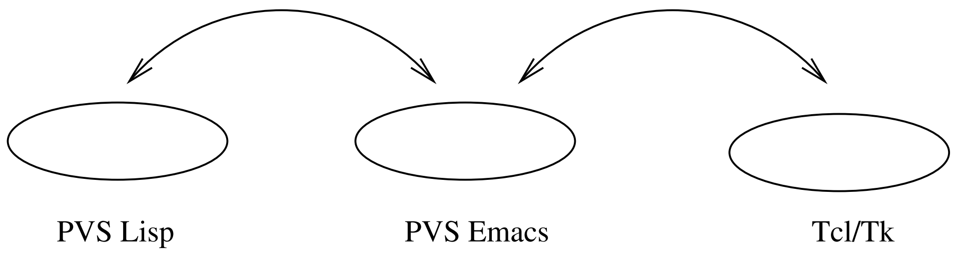


Figure 3.2: Current Model of PVS Process Communication

all communication between the PVS Lisp process and the Tcl process passes through PVS Emacs. More specifically, there is an output filter for each of the sibling processes—`pvs-output-filter` and `pvs-tcl-output-filter` for PVS Lisp and Tcl, respectively—that identifies the message type and calls the appropriate function. The message format is `:<pvs-action> arg_1.. .arg_n :<end-pvs-action>`, where `pvs-action` is one of `msg`, `log`, `warn`, `err`, `qry`, `buf`, `yn`, `bel`, `loc`, `mod`, `pmt`, `dis`, `wish`, `eval`, `evaln`. The sibling processes use the same Lisp functions to output and log messages, including error messages; to query the user for input or yes/no responses; to beep the user; locate a PVS theory or object; modify a buffer; prompt the user for input; and so forth. `eval` evaluates an expression in Emacs Lisp; the result is read by the PVS Lisp process or sent to the Tcl process, depending on the output filter and the corresponding function invoked.

Used only by the Tcl process, `evaln` evaluates an expression in Emacs Lisp, but does not return the result to the Tcl process. Message types used only by the Lisp process include `warn` messages used to collect PVS parse and typecheck warning messages, and `wish` messages used to send a string to the sibling Tcl process.

Communication between processes is logged and may be viewed in the Emacs buffer `pvs`.[2] The fragment of the log shown below illustrates the prover communicating via PVS Emacs with a Tcl process in response to the command `M-x x-prover-commands`. The latter brings up an interactive Tcl display of PVS prover commands, including user-defined commands, if any. The prover commands are first dumped to a temporary file along with the name of the Tcl function (`show-proof-commands`) that specifies and displays the Tcl window. The file is then input by the main Tcl program (`wish` for "windowing shell"). `catch` and `source` are Tcl commands that provide error handling, and evaluate Tcl commands from a file, respectively. The Tcl `exec` command is used to run the UNIX `rm` program on the temporary file.

```
rec:{NIL PVS(47): }

  sent:{(x-prover-commands)}

  rec:{:pvs-wish catch {source /tmp/pvs-18278.p1};
        exec rm -f /tmp/pvs-18278.p1 :end-pvs-wish
  NIL
  PVS(48): }
```

Communication between PVS components that run in the PVS Lisp (sub)process, including the parser, typechecker, prover, and prettyprinter, is similarly mediated by PVS Emacs. The partial log shown below records the message sequence initiated by a user request to typecheck theory `fcp_demo`. The `NIL`s in the messages represent optional arguments.

```
  rec:{PVS(59): }

  sent:{(typecheck-file "fcp_demo" nil nil nil)}

  rec:{ :pvs-msg Parsing fcp_demo :end-pvs-msg}

  rec:{:pvs-msg fcp_demo parsed in 0.39 seconds :end-pvs-msg}

  rec:{:pvs-buf fcp_demo.tccs&NIL&NIL&NIL&NIL :end-pvs-buf}

  rec:{:pvs-buf fcp_demo.lisp&NIL&NIL&NIL&NIL :end-pvs-buf}
```

[2]Appendix A provides a brief tour of the buffers created and used by PVS.

```
    rec:{:pvs-buf fcp_demo.ppe&NIL&NIL&NIL&NIL :end-pvs-buf}

    rec:{:pvs-msg Typechecking fcp_demo :end-pvs-msg}

    rec:{:pvs-err fcp_demo&/export/u1/homes/.. ./&
          Typecheck error&/tmp/pvs-18278.p4&45 30 :end-pvs-err}

    rec:{PVS(60): }
```

The message traffic indicates that parsing completed successfully in 0.39 second, but that typechecking detected an error, which was recorded in a temporary file. The `pvs-err`, `end-pvs-err` brackets are intercepted by `pvs-process-filter`, which passes the error to the PVS error handler, `pvs-error`, which in turn calls `pvs-error*` to display the error to the user and delete the temporary file. If parsing completes successfully, the three ancillary buffers associated with the file are removed (they may contain outdated information); this is reflected in the `:pvs-buf` entries.

There have been a handful of applications that integrate independently developed components with PVS. We look briefly at three examples, each developed by PVS users working within the basic structure of the current model. Buth's integration of Pamela with PVS [2] and the Rockwell Collins visualizations of PVS ground evaluator output [14] both served to motivate the current project; although their approaches differ, these two examples suggest the challenges of combining PVS with independently developed applications and GUIs. The third example, Dunstan et al.'s Maple/PVS Project [13], illustrates a potential solution to these challenges. In this regard, it shares the objectives of our work, while differing in approach. Although Dunstan et al. jettison PVS Emacs whereas we retain it, both efforts seek to exploit the current model of interaction to provide a general framework for integrating new components with PVS.

Visualization for PVS Ground Evaluator: The Flight Guidance System (FGS) visualization developed by Rockwell Collins [14] was developed primarily to determine whether graphical interfaces could be used to promote and facilitate discussion between pilots, human factors experts, and system designers [14, p. 7]. The Collins prototype consists of three graphical displays, one for the Electronic Flight Instrumentation System (EFIS), the primary flight display in the cockpit; one for the Flight Control Panel (FCP), the primary user interface with the Flight Control System; and one for the display of high-level mode information for the Flight Director, Autopilot, and vertical and lateral flight modes.[3] The initial strategy was to generate PVS specifications automatically from the visualization models. That strategy was ultimately abandoned; the PVS specifications were generated

[3]Broadly speaking, the mode of a Flight Guidance System system component refers to the state of that component. For example, the mode of the Autopilot may be *engaged* or *disengaged*.

by hand and their consistency with the visualization models checked via manual review [14, p. 12]. The implementation[4] consists of a “PVS Controller” written in Java that oversees communication between the PVS Ground Evaluator and the graphical displays, which are written in Delphi. A second Java process maintains the FGS state. The Java Controller and Delphi visualizations are PC based; the PVS code, its “wrapper” (a C program), and the Java state administrator run on Unix. As Figure 3.3 shows, this implementation omits PVS Emacs; communication

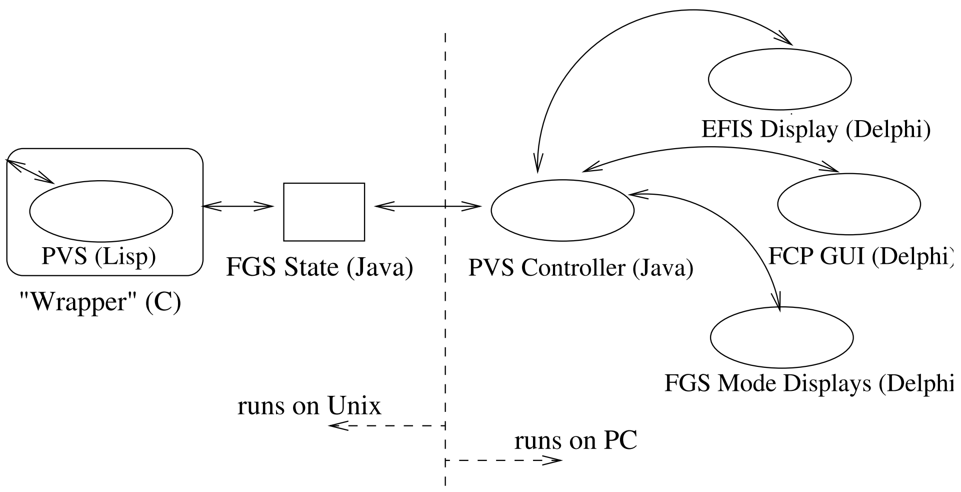


Figure 3.3: Communication Model for Rockwell Collins FGS Visualization

between PVS and the GUI is handled by the Java controller, in concert with the Java FGS state administrator, and the C wrapper that filters input to and output from PVS. Although the Collins GUI yielded a highly effective proof of concept, its utility as an exemplar for other GUI developers is limited by its inherent complexity and application-specific design choices.

Pamela+PVS: Pamela was originally developed to establish the partial correctness of VDM specifications. Subsequent changes and extensions have yielded a more general system for formal verification of sequential software. The purpose of Buth’s study is to exploit PVS to provide enhanced proof support for Pamela [3]; Pamela processes specifications and calculates proof obligations (a variant of strongest postconditions), which are then submitted to PVS. File-oriented and non-interactive, Pamela is implemented in C for Unix platforms. The Pamela+PVS interface uses PVS Emacs to process communication between Pamela and PVS, as shown in Figure 3.4. A quick visual comparison of the current communication

[4]As far as we know, there are no published accounts of the implementation. The architecture presented here has been inferred from unpublished notes generously supplied by Steve Miller.

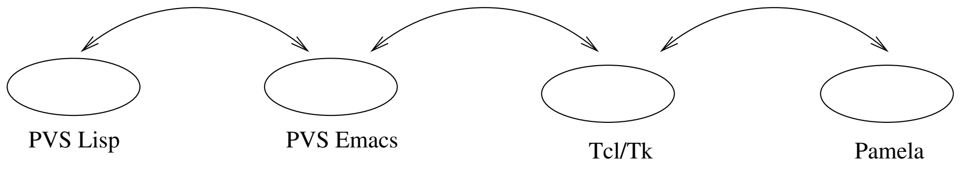


Figure 3.4: Communication Model for PAMELA+PVS

model in Figure 3.2 and the PAMELA+PVS model in Figure 3.4 confirms that Buth's system is a straightforward extension of the current model, which allowed her to use or adapt existing PVS-Tcl interfaces. One of the objectives of Buth's study was to encourage reuse by others interested in exploiting the PVS prover as a back end. This objective is realized via sound software-engineering practices, including thorough documentation, rather than through general-purpose interface tools or a generic framework.

Maple/PVS Project: An interesting variant of the current model results if we manage the communication with something other than PVS Emacs. Although the Rockwell Collins visualization similarly detours PVS Emacs, the work of Martin Dunstan and his colleagues at the University of St. Andrews, who are developing a Maple/PVS interface, provides a more generic example of the opportunities presented by this approach. Maple [12] is a Computer Algebra System that offers extensive libraries of symbolic computation algorithms, arbitrary-precision numerics, and graphics capabilities.[5] The Maple/PVS project is exploring strategies to allow Maple users to solve problems that require theorem technology, optionally without interacting directly with a theorem prover. The decision to use the PVS theorem prover for this project was motivated by ongoing work in real analysis in PVS [11]. As Figure 3.5 suggests, Dunstan has replaced PVS Emacs with a Tcl "PVS Controller," effectively a graphical interface that exists largely to effect communication between Maple and PVS, but that also displays PVS output and accepts user commands for PVS. All communication is initiated by Maple. The controller includes a lexical analyzer that filters PVS output, yielding a readily parsable, whole-line format in which each line is tagged with a unique 2-character code identifying its content. The analyzer is generated via FLEX (Fast Lexical Analyzer Generator) and may be used or adapted by others interested in interfacing applications to PVS in a non-Emacs framework.

[5]Originally developed by members of the University of Waterloo Symbolic Computation Group (SCG), Maple is currently sold by Waterloo Maple Software (WMS). Research and development involves collaboration between SCG, WMS, and several research organizations, including the Numerical Algorithms Group (NAG) (`http://www.nag.com`), which collaborated on the extensive set of numerical algorithms included in Maple 6.

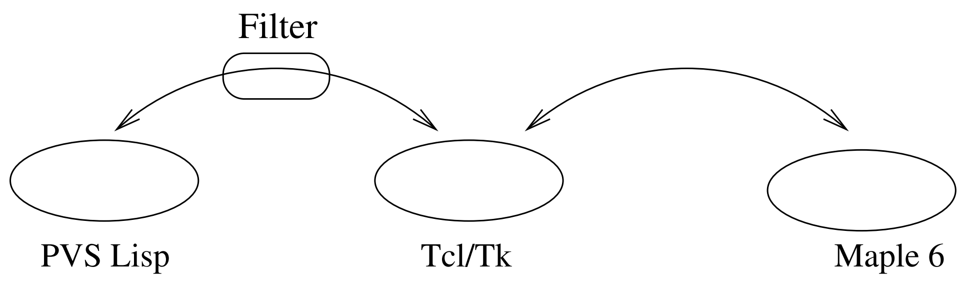


Figure 3.5: Communication Model for Maple/PVS

The continuum along which these three examples lie reflects generality of the approach, with the FGS visualization being the least and the Maple interface the most general. All three exemplify the current model of communication in which a central "controller" mediates communication between a PVS process and one or more independently developed application processes. Both the FGS visualization and the Maple interface use a non-Emacs controller. PAMELA+PVS provides an adaptable exemplar, and the Maple-PVS interface offers a reasonably general framework.

3.1.2 Context and Level of Interaction

The second and third of the design dimensions identified earlier delineate the context necessary to support interaction with PVS, and the level and bandwidth at which this interaction takes place. The two are interrelated; communication of a single proof obligation may be accomplished relatively easily via a prover API function such as `prove-formula` or `prove-formula-decl`, whereas communication via large files containing multiple proof obligations potentially introduces response time and prover state explosion issues, as noted in [2, p.19], and therefore requires more careful design of the communication channel(s).

GUIs, such as those in Maple/PVS and in PAMELA+PVS, that simply funnel input to PVS and display the results do not need to preserve state from one PVS invocation to another. However, GUIs such as the FGS that drive displays reflecting state variables need to maintain state either in PVS (cf. Section 2.4.2) or in the interface or both. The state displayed in the FGS visualizations provides context for the PVS ground evaluator, and, conversely, the results returned by the ground evaluator drive updates to the displays. In the Collins interface, the state vector is maintained in Java. The phone book GUI described in Section 3.2 maintains state exclusively in PVS.

3.2 Implementing a GUI for the Current PVS System

Although we have experimented with several graphical interface languages, including Glade, GTK, and Perl/Tk, graphical user interfaces for the current PVS system (e.g., for proof display or proof command selection) continue to be written in Tcl. For this reason, the GUI development described in this chapter also takes place in Tcl, which allows us to exploit existing Tcl support in the PVS system.

Assuming the current model of PVS-GUI interaction, implementing a GUI involves creating functionality at each of the three nodes in Figure 3.2. PVS Emacs Lisp code defines the interface between the sibling PVS Lisp and GUI components. This code typically creates a new PVS command, initializes the GUI, provides a communication layer between PVS and the GUI, and so forth. PVS Lisp code implements the PVS end of the interaction, for example, providing a wrapper for the ground evaluator that accepts input expressions from the Emacs Lisp interface and returns to the interface the results of the ground evaluation. The GUI code is self-explanatory; it implements the graphical display for a given interface. The PVS system uses standard naming conventions; Emacs Lisp code files use a `.el` extension, Lisp files use a `.lisp` extension, and GUI files use an extension that reflects the chosen graphical interface language, for example, `.tcl`.

To illustrate the functionality and suggest an implementation for the Emacs Lisp, Lisp, and GUI interface components, we develop a Tcl GUI for the `phone_3` example introduced in Section 2.4.2 of Chapter 2. We first revisit the design choices outlined in Section 3.1 above, explicitly framing our choice of communication model, and context and level of interaction.

For largely pragmatic reasons, principally the relatively low overhead and pedagogical utility of adopting the existing model of PVS-GUI communication, our graphical `phone_3` interface communicates with PVS via PVS Emacs. The `phone_3` GUI sends single expressions—for example, `AddPhone(emptybook, alice, 8)`[6]—to PVS via PVS Emacs, and PVS Emacs sends the Tcl interface the resulting phone book state if the ground evaluation succeeds. If the ground evaluation does not succeed—for example, if the user misspells a name, resulting in a type error—PVS Emacs sends the Tcl interface a value indicating that the attempted evaluation left the phone book state unchanged.

As a further example of the PVS attachment mechanisms discussed in Chapter 2, we use semantic attachments to modify and access the phone book state. The state is maintained exclusively in PVS, and passed via Emacs to the Tcl interface for display. This design decision is viable because the state in this example is small; hence, the overhead of passing it to the graphical interface is low. We look first at two

[6]The actual PVS expression is formatted in Emacs Lisp rather than Tcl, as explained in Section 3.2.3.

modest changes to the PVS `phone_3` theory and associated semantic attachments, motivated primarily by GUI requirements on the PVS state.

We add the following function to theory `phone_3` to appropriately initialize the phone book:

```
InitPhone(bk): bool = (FORALL nm: write(bk WITH [(nm) := emptybook(nm)]))
```

and we remove the call to `tab` in `printbook`, since formatting is now handled by the graphical interface.

```
  printbook(bk): bool =
    FORALL nm: print(nm,"N") AND printphones(FindPhone(bk,nm))
```

We next modify the Lisp defuns that define the semantic attachments for the print functions; in the context of the phone book GUI, instead of displaying information to the user, the print functions accumulate ground evaluator results in a global whose value is returned via PVS Emacs to the Tcl interface.

```
   (setq *totcl* nil)

   (defun |PVS__printf| (a typ fmt)
      (setq *totcl* (append *totcl*
            (list(cl2pvs a (pc-typecheck (pc-parse typ 'type-expr))))))

   (defun |PVS_prints| (s)
      (setq *totcl* (append *totcl* (list s))))
```

For convenience, we define the left and right braces as semantic attachments. Since, as noted in Section 2.4.2, the ground evaluator does not allow attachments to constants, we use the superfluous argument workaround described in that section.

```
   lbrace(b: bool) : bool
   rbrace(b: bool) : bool
     % argument is ignored, attachment appends {
   leftbrace: bool = lbrace(true)
     % argument is ignored, attachment appends }
   rightbrace: bool = rbrace(true)
```

The corresponding semantic attachments are straightforward.

```
  (push (mk-name '|lbrace| nil '|printstrings|) *pvs2cl-primitives*)
  (defun |PVS_lbrace| (x) (setq *totcl* (append *totcl* (list '{))))

  (push (mk-name '|rbrace| nil '|printstrings|) *pvs2cl-primitives*)
  (defun |PVS_rbrace| (x) (setq *totcl* (append *totcl* (list '}))))
```

We turn now to the three GUI components mentioned above, each of which corresponds to a node in the diagram in Figure 3.2. When reading this discussion, it is helpful to keep in mind the underlying model of communication sketched in the introductory remarks to Section 3.2, in which all communication is initiated by the Tcl process in response to user input and consists largely of Tcl requests sent to the PVS ground evaluator via Emacs Lisp, with the results returned again via Emacs Lisp to the Tcl process. We begin with the central node, PVS Emacs, and describe first a generic framework agnostic with respect to GUI implementation language.

3.2.1 PVS Emacs

The generic PVS Emacs framework defines a PVS command to start up a GUI for a given PVS theory, and loads the Emacs Lisp, Lisp, and GUI source files that implement the graphical interface. The framework shown below exploits the Emacs "hook" mechanism, which provides a way to specify functions to be run by Emacs on particular occasions.[7] For example, we use the hook mechanism to run a specified GUI initialization function whenever the PVS `M-x x-gui` command is issued and the theory named in the command by the user has been typechecked. Hook variables are set via the built-in `add-hooks` function, and run via the corresponding `run-hooks` function or one of its variants such as `run-hooks-with-args` or `run-hooks-with-args-until-failure`. Comments marked with `#` precede globals whose values need to be set appropriately for each application.

```
;; # SPECIFY GLOBALS for interface GUI, .lisp, and .el source
(defvar pvs-gui-lisp-source '(" .. . ") )
(defvar pvs-gui-el-source   '(" .. .") )
(defvar pvs-gui-source      '(" .. .") )

;; # SPECIFY THE LOAD PATH
(add-to-list 'load-path "/.. .")

;; # ADD INTERFACE-INIT FN(S) TO GUI INIT HOOK
(add-hook 'gui-init-hook 'gui-init)

(defun gui-load-file (file)
   ; load file(s) for graphical user interface
   .. .
   )
```

[7]Online descriptions of Emacs Lisp functions, including the hook functions, may be displayed via the Emacs `C-h d` or `C-h f` commands.

```
(defun gui-init ()
   ; load el, lisp, GUI sources
   (mapc 'load pvs-gui-el-source)
   (mapc (function (lambda (f)(pvs-send (format "(load  \"%s\") f))))
          pvs-gui-lisp-source)
   (mapc 'gui-load-file pvs-gui-source)
   )


(defpvs x-gui x-display (theoryname)
   "Start the gui for theory theoryname"
   (interactive (complete-theory-name "Gui for theory:"))
   (if (not (getenv "DISPLAY"))
       (message "DISPLAY not set, cannot run gui")
       (if (pvs-send-and-wait
              (format "(typechecked\? \"%s\")" theoryname) nil 'tc nil)
           (run-hooks 'gui-init-hook)
           (message "Please typecheck %s and reissue x-gui command"
                    theoryname))))
```

The generic framework instantiated for the Tcl phone book GUI appears below. The application-specific version differs only in the values given to the globals, the inclusion of the function to load Tcl files into the Tcl process, and the calls (to `ensure-pvs-wish` and `tcl-send-string*`) that ensure an existing Tcl process and unmap the current Tcl window (if any), respectively. For transparency, we have retained the function `gui-load-file`, although in the spirit of Occam's razor it would be preferable to eliminate it and call `tcl-load-file` directly in the final `mapc` in `gui-init`.

```
;; # SPECIFY GLOBALS for interface .tcl, .lisp, .el source
(defvar pvs-gui-lisp-source '("gui-groundeval.lisp") )
(defvar pvs-gui-el-source   '("pb-interface.el") )
(defvar pvs-gui-tcl-source  '("pb.tcl") )

;; # TO SPECIFY THE LOAD PATH
(add-to-list 'load-path "/.. ./gui")

;; # ADD TCL INTERFACE-INIT FN TO GUI INIT HOOK

(add-hook 'gui-init-hook 'tcl-gui-init)

(defun gui-load-file (file)
   (tcl-load-file file)
   )
```

```
(defun tcl-gui-init ()
   ; load el, lisp, tcl sources
   (mapc 'load pvs-gui-el-source)
   (mapc (function (lambda (f)(pvs-send (format "(load  \"%s\")" f))))
         pvs-gui-lisp-source)
   (ensure-pvs-wish)
   (tcl-send-string* "wm withdraw .")
   (mapc 'gui-load-file pvs-gui-tcl-source)
   )

(defpvs x-gui x-display (theoryname)
   "Start the gui for theory theoryname"
   (interactive (complete-theory-name "Gui for theory:"))
   (if (not (getenv "DISPLAY"))
       (message "DISPLAY not set, cannot run gui")
       (if (pvs-send-and-wait
              (format "(typechecked\? \"%s\")" theoryname) nil 'tc nil)
           (run-hooks 'gui-init-hook)
           (message "Please typecheck %s and reissue x-gui command"
                    theoryname))))
```

In addition to the Emacs Lisp code obtained by instantiating the generic GUI framework, we need application-specific code that minimally implements the following functions:

- provide initialization
- convert between PVS and GUI language data structures
- mediate communication between PVS and the GUI

We include portions of the code that implements this functionality for the phone book example to suggest the flavor of Emacs Lisp mediation. We reproduce entire functions, but focus primarily on communication between Emacs Lisp and the Lisp and Tcl (sub)processes.

The initialization function for the phone book GUI, `init-gevaluator`, is called via Emacs Lisp from the Tcl process (cf. Section 3.2.3). The function `ilisp-send` sends a string to the buffer on an inferior Lisp process and returns the result.[8] We use it to pass to PVS Lisp a conjunction whose evaluation initializes the PVS ground evaluator for GUI interaction in the context of the specified theory, and invokes the GUI API function to the ground evaluator, `gui-gevaluate`, to evaluate a PVS expression that initializes the PVS state. The PVS state must be initialized to avoid a PVS `read` operation (cf. Section 2.4.2) on an undefined state.

[8]This account is somewhat simplified; `ilisp-send` has several optional arguments that determine, e.g., whether the command is executed without waiting for results or the result is returned, whether control remains in the inferior Lisp buffer, and so forth.

```
(defun init-gevaluator (theory)
  ;; check that neither prover nor evaluator currently invoked
  (confirm-not-in-checker)
  ;; indicate ground evaluator active
  (pvs-evaluator-busy)
  ;; save PVS buffers
  (save-some-pvs-buffers)
  ;; bury all temporary windows
  (pvs-bury-output)
  ;; initialize ground evaluator and *pvsstate*
  (ilisp-send
     (format "(and (gui-init-gevaluator \"%s\")

                   (gui-gevaluate \"InitPhone(emptybook)\"))" theory)
     nil 'pr t))
```

The second role of the Emacs Lisp code noted above was conversion between PVS and GUI data structures. Given the highly application-specific nature of this code, we limit the discussion to a single illustrative example. Recall that the Tcl interface sends expressions consisting of phone book operations to the PVS ground evaluator. The following function converts a phone book operation and its arguments (if any) received from the Tcl interface into the corresponding PVS expresssion. The formatted expression is passed to PVS Lisp via the Emacs Lisp function `Geval-for-Tcl` (cf. below).

```
;; phone book operators
(defvar *addop       'AddPhone)
(defvar *findop      'FindPhone)
(defvar *delnamop    'DelPhone)
(defvar *delnumop    'DelPhoneNum)
(defvar *printnums   'printphones)
(defvar *printbk     'printb)

(defun pb-tcl-to-pvs (op arg1 arg2)
   ;; format phone book operations for ground evaluator
   ;; pb-format formats operations that write PVS state
   ;; revisit formatting if PVS Phone3 fns are modified
   (case op
      (add    (pb-format *addop arg1 arg2))
      (find   (format "\"%s(read[B],%s) AND %s\"" *findop arg1 *printbk))
      (delete (if arg2 (pb-format *delnumop arg1 arg2)
                       (pb-format *delnamop arg1 arg2)))
      (print  (format "\"%s\"" *printbk))
      (t nil)))
```

```
(defun pb-format (op arg1 arg2)
  (if arg2
      (format "\"write(%s(read[B], %s, %s)) AND printb\"" op arg1 arg2)
      (format "\"write(%s(read[B], %s)) AND printb\"" op arg1)))
```

The final role of the Emacs Lisp code is to mediate communication between the GUI process and the PVS Lisp process. The following Emacs Lisp function passes phone book operations and their arguments from Tcl to PVS (with appropriate formatting), and returns the resulting state from PVS to Tcl. Note the two uses of `ilisp-send`. This function is called from Tcl via the function `emacs-eval` (cf. Section 3.2.3), and the results returned to Tcl.

```
(defun GEval-for-Tcl (pbOp pbName pbNbr)
   ;; reset lisp var that returns PVS state
   (ilisp-send (format "(setq %s nil)" *pvstotcl*))
   ;; format and
   (let* ((to-pvs (pb-tcl-to-pvs pbOp pbName pbNbr))
          (from-pvs (if to-pvs
                        (ilisp-send
                           (format "(gui-gevaluate %s)" to-pvs)))))
         (pb-pvs-to-tcl from-pvs)))
```

This ends our discussion of the Emacs Lisp code minimally required to implement a Tcl GUI in the current communication model. We turn now to a discussion of the interface code in the PVS Lisp and Tcl (sub)processes, beginning with the former.

3.2.2 PVS Lisp

The PVS Lisp code for the phone book GUI consists of modified versions of the PVS ground evaluator API functions. The modifications are not specific to the phone book example; they would be useful for any GUI that requires the ability to evaluate single expressions in the PVS ground evaluator. The main differences between the GUI API functions to the ground evaluator and the standard API functions involve input from and output to the user. The ground evaluator reads a PVS expression input by the user and, depending on the values of various control flags, outputs tccs, timing information, and so forth, in addition to the result. In the context of a GUI, ground evaluator interaction with the user is processed by the interface; input is received from the GUI and output returned to the GUI for display. The ground evaluator currently ignores type correctness conditions (TCCs), but does check whether the user wishes to proceed in the presence of unproven TCCs. At present, the modified ground evaluator code for GUI applications similarly ignores TCCs.

The modified ground evaluator interface consists of three functions:

- `gui-init-gevaluator`—initialization function called by Emacs Lisp (cf. Section 3.2.1) that establishes a given PVS theory as the context for ground evaluation and checks again that the theory has been typechecked. One of two API functions for the modified ground evaluator interface.
- `gui-gevaluate`—once the ground evaluator has been initialized, the API function used for all subsequent GUI interaction with the evaluator. Sets evaluation parameters and calls `gui-gevaluate*`.
- `gui-gevaluate*`—top-level function that sequences evaluation. The typechecked input expression is translated to Lisp, evaluated, and the results translated back into PVS. Currently, only ground results are retranslated into PVS.

There is no interprocess communication in any of these functions; hence, we choose one as an exemplar, but reproduce the code without comment. `*gui-geval-current-theory*` is a global that holds the name of the theory on which the GUI has been invoked. Other globals are similarly delimited by `*`.

```
(defun gui-gevaluate (expression &optional theoryname)
   (let ((*current-theory* (or (get-theory theoryname)
                               (get-theory *gui-geval-current-theory*)))
         (gui-geval-result 'none))
        (if *current-theory*
            (let ((*generate-tccs* 'all)
                  (*current-context*
                    (or (saved-context *current-theory*)
                        (context nil)))
                  (*in-evaluator* t)
                  (*destructive?* t)
                  (*convert-back-to-pvs* t))
                 (setq gui-geval-result (gui-gevaluate* expression)))
            (pvs-message "Gui Eval: Unknown theory: ~a" theoryname))
        (pvs-emacs-eval "(pvs-evaluator-ready)")
        gui-geval-result)
)
```

The final interface component in the phone book example is the Tcl GUI itself.

3.2.3 Tcl Phone Book GUI

A snapshot of the Tcl phone book interface appears in Figure 3.8. The two-part window includes a panel of five buttons on the left, and a larger display area on the right. Buttons correspond to basic phone book operations. An additional button closes (i.e., iconifies) the phone book window; reopening the window clears the display area. The darkened button to the left of the `print` label in Figure 3.8 indicates that `print` is the most recently selected button. It is not currently possible to quit PVS from the phone book interface. Selecting one of the phone book operations brings up a dialogue box that prompts for the requisite arguments, as indicated in Figures 3.6 and 3.7, which show the dialogues initiated by selecting the `print` and `add` buttons, respectively. The result of evaluating the PVS expression `printb`,

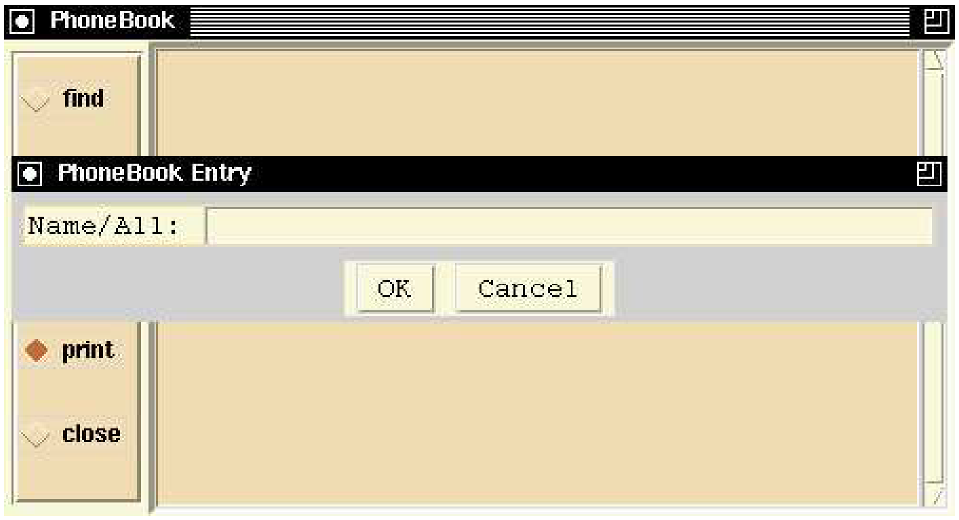
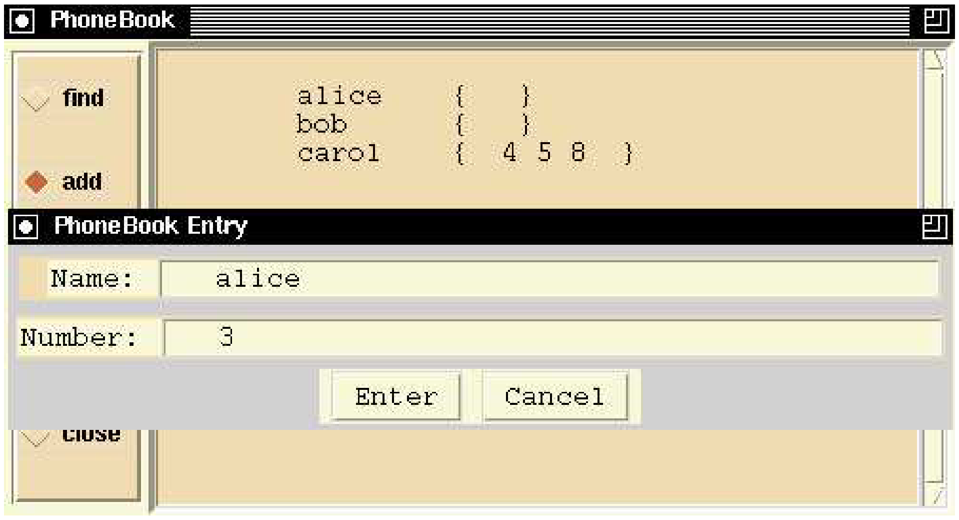


Figure 3.6: `print` Command Dialogue Box

which corresponds to the `print` command with user-supplied argument "all," appears in Figure 3.8. Figure 3.9 shows the result of evaluating a PVS expression corresponding to the user dialogue in Figure 3.7.

The Tcl procedures responsible for formatting and sending a request to PVS via Emacs Lisp are shown below. The PVS Tcl procedure `emacs-eval` outputs (to the standard output stream) a formatted expression of the form `:pvs-eval <exp> :end-pvs-eval`, which is processed by the Emacs Lisp filter `pvs-tcl-output-filter` resulting in a call to `pvs-tcl-emacs-eval` where `<exp>` is evaluated and the result returned to the Tcl process. Note that although we explicitly call `emacs-eval` from Tcl to evaluate an expression in Emacs Lisp, we do not explicitly call an analogous function on the result; the result of the evaluation is automatically returned to the Tcl process via a call to the Emacs Lisp function `tcl-send-string*` in `pvs-tcl-emacs-eval`. For the phone book GUI, the expression sent from the Tcl process to Emacs Lisp via `emacs-eval` is `(GEval-for-Tcl $evalstr)`; for example, the value of the expression corresponding to the `add` in Figure 3.7 would be `(GEval-for-Tcl 'add 'alice '3)`. As we have seen in Section 3.2.1, the Emacs Lisp function `GEval-for-Tcl` formats its arguments and



```
alice    {    }
bob      {    }
carol    {  4 5 8  }
```

Name: alice

Number: 3

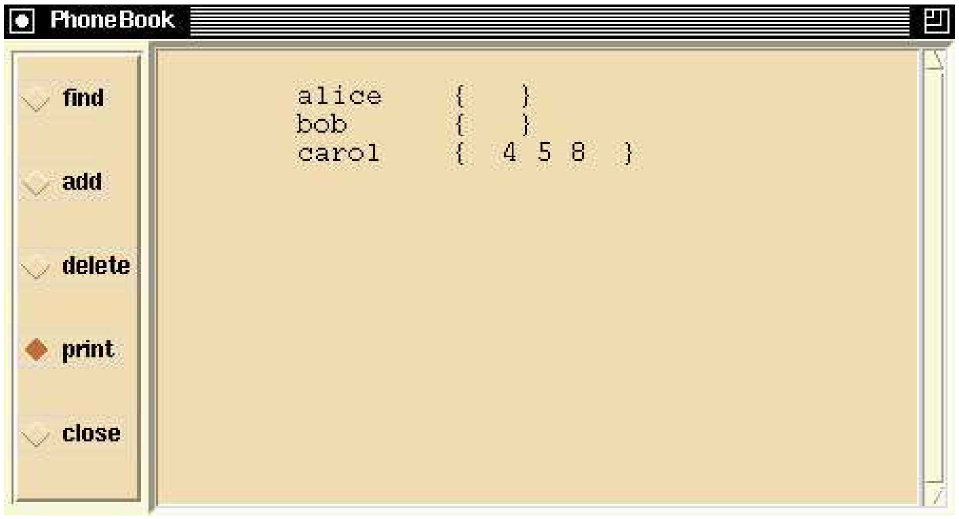
Figure 3.7: `add` Command Dialogue Box

sends the resulting PVS expression to the ground evaluator GUI API function `gui-gevaluate` (cf. Section 3.2.2) in the inferior Lisp process, where the expression is evaluated and the resulting value returned via Emacs Lisp to Tcl.

```
proc ToPvsEmacs  evalstr
# send operation and associated arguments to PVS,
# call pbupdate to update display with result
   set result [emacs-eval "(GEval-for-Tcl $evalstr)"]
   pbupdate $result


proc PbEnter
   # format operation, argument string,
   # call ToPvsEmacs to send string via Emacs to PVS unless
   # op is ‘‘close’’ which is handled locally (in Tcl)
   global PbName PbNbr PbOp noNbr CloseOp
   if  [string compare $PbOp $CloseOp] == 0   PbClose  else
        if  [string compare $PbNbr $noNbr] == 0
             ToPvsEmacs "‘$PbOp ‘$PbName nil"
             ToPvsEmacs "‘$PbOp ‘$PbName ‘$PbNbr"
```

In the phone book example, we use a small subset of the communication functions available in PVS Lisp and Emacs Lisp, and the PVS Tcl support code. Readers



```
alice    {    }
bob      {    }
carol    {  4 5 8  }
```

Figure 3.8: Tcl Display Following PVS Evaluation of `printb`

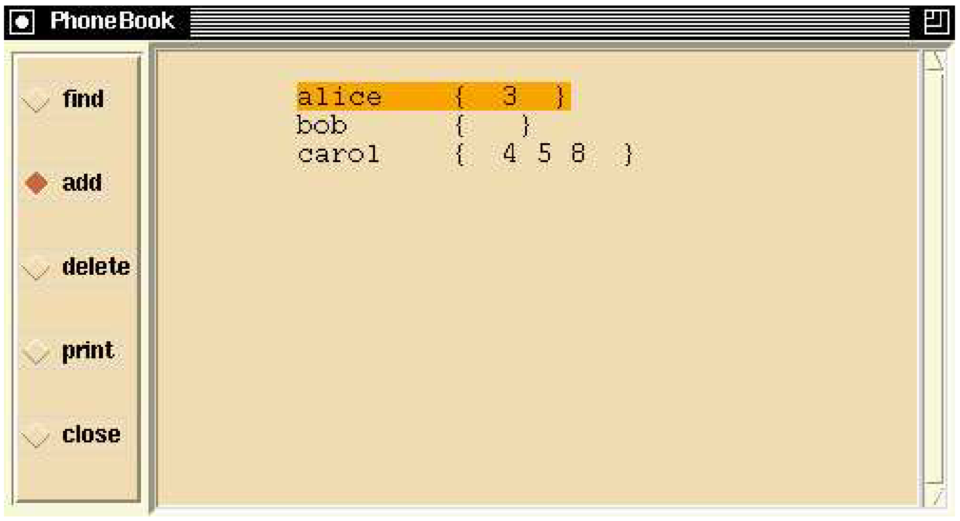
interested in a more complete account may refer to Appendix A, which provides an annotated listing of most of the available communication functions.

3.3 Discussion

The phone book GUI sketched in this chapter serves reasonably well as an example for introducing a generic framework for GUI development and illustrating an approach to interfacing independently developed components, such as GUIs, with PVS. However, the phone book example per se is arguably a rather poor choice; the specification is highly functional (even `setof[P]`, the set of phone numbers is a function)[9] and therefore a better example for the theorem prover than the ground evaluator. For this reason, we were pleasantly surprised to find that developing this small, highly functional example has exposed interesting issues, one of which we discuss below. We also briefly touch on one of the more pragmatic issues encountered while developing the Phonebook GUI.

The first issue relates to the distinction between writing programs and writing specifications, and the dual nature of executable specifications. Given `P: TYPE = upto(10)` of phone numbers, evaluating expressions such as `AddPhone(read[B], alice, 9)` yield subtype TCCs of the form `9 <= 10`. This TCC is clearly trivial and would be discharged summarily if TCCs were not optionally ignored in the current ground evaluator. The more interesting case arises when we evaluate an expression

[9]In PVS, sets are represented as predicates, i.e., as functions of type `[ t -> bool ]`.



```
alice     {  3  }
bob       {    }
carol     {  4 5 8  }
```

Figure 3.9: Tcl Display Following PVS Evaluation of `write(AddPhone(read[B], alice, 3)) AND printb`

such as `AddPhone(read[B], alice, 11)`, which generates a similar, but unprovable, subtype TCC. Ground evaluation of the expression `write(AddPhone(read[B], alice, 11)) AND printb` reflects no change of state, for reasons that become clear if we look carefully at the definitions of `AddPhone` and the PVS prelude function `add`, reproduced below. Note that `sets[P].add` returns a nonempty set whose elements are of type `P`.

```
AddPhone(bk, nm, pn): B = bk WITH [(nm) := add(pn, bk(nm))]

sets [T: TYPE]: THEORY
BEGIN
  set: TYPE = setof[T]
  x, y: VAR T
  a, b, c: VAR set
  member(x, a): bool = a(x)
  empty?(a): bool = (FORALL x: NOT member(x, a))
  emptyset: set = x | false
  nonempty?(a): bool = NOT empty?(a)
  add(x, a): (nonempty?) = y | x = y OR member(y, a)
  .. .
END sets
```

In a specification environment that includes uninterpreted types, strong typing and TCC generation yield precisely the expressibility and correctness guarantees we want. However, in an execution environment, the resulting loss of transparency suggests that we may need to think carefully about how to integrate TCC evaluation into the execution of PVS expressions. In our example, the subtype TCC `11 <= 10` indicates a type incompatibility somewhere in the specification; the execution, however, proceeds "normally." This raises questions about the role of TCCs in an execution environment and about the kind of information they should convey.

The second issue is more pragmatic. In the ground evaluator, functions, as well as uninterpreted types and certain records and tuples, are represented in the translation as closures. For example, evaluating the PVS expression `AddPhone(read[B], alice, 9)` in the ground evaluator yields `#<Closure (INTERNAL AddPhone!_2 0)>`. As noted previously, there is currently no support in the PVS evaluator for translating Lisp closures into PVS functions. The phone book example detours this issue by exploiting the attachment mechanism to maintain a PVS state and provide accessors that apply closures to yield displayable values. In principle, it should be possible to convert a closure to a function by creating variable(s) of the argument type(s) and applying the closure. However, since variables are not ground expressions, this approach would require *real* symbolic evaluation, a capability not yet available in PVS, although one proposed, along with others, in the following chapter.

Chapter 4

Summary and Future Work

This report has explored ways to enhance the utility of PVS for evaluating, testing, and animating PVS specifications. Our focus has been the PVS ground evaluator. We have discussed semantic attachments as a mechanism for extending the use of the ground evaluator to a larger class of applications, for example, to those for which printing, iteration, or maintaining and accessing state is essential. We have also discussed strategies and a generic framework for integrating independently developed applications with PVS in general, and the ground evaluator in particular. The work reported here has genuine utility, but nonetheless represents relatively crude experiments in the context of extant ground evaluator capabilities. Accordingly, future work can take one of two directions: we can continue our efforts to enhance the current ground evaluator, or we can design a more productive development environment, including a new evaluator API, drawing on the work reported here to suggest desired functionality and preferred implementation.

Earlier chapters of this report have suggested enhancements to the current ground evaluator that reflect functionality equally desirable in the existing ground evalutor and any redesign. The list below pairs those suggestions with two additional capabilities. An enhanced ground evaluator would

1. Optionally evaluate TCCs: TCCs generated via evaluation of ground expressions should themselves be ground and, hence, be evaluable in the ground evaluator.

2. Allow the use of evaluation in proofs: The result of ground evaluation should be factored into the state of the proof and reflected in the proofchain analysis. User-supplied attachments should be distinguished in the proof record from evaluator-generated attachments.

3. Evaluate ground instantiations of generic theories: As noted in Chapter 2, there should be a mechanism to provide evaluation of `phone[N, P: TYPE]` instantiated with `N: TYPE = {alice, bob, carol}` and `P: TYPE = upto(10)`.

4. Support (possibly generic) modifications to the generated Lisp code: This is useful—for example, for testing, where coverage data could be acquired via the addition of a count function that tracks frequency of execution for each code fragment.

The potential impact of these items differs. Item 4 is particularly useful for applications such as those discussed in this report, whereas items 1 and 3 potentially benefit a broader class of applications by improving the basic functionality of the ground evaluator. Item 2 has the most general utility; it enhances both the capability of the prover and that of the ground evaluator.

To facilitate GUI development for the ground evaluator, we are also considering development of a "meta-gui," that is, a GUI generator that would allow the user to instantiate a particular model of interaction and generate a GUI for a specific application in a specified graphical interface language. A generic tool of this kind could be developed in Glade [1] or other interface toolkit and would be highly useful for users integrating independently developed tools with PVS—for example, the Pamela front end to PVS or the Maple/PVS interface—as well as for those interested in testing PVS specifications and rapidly prototyping graphical interfaces for PVS.

The GUI generator, as well as items 1 through 4 above, would clearly enhance the utility of the ground evaluator, as would other additions to its basic functionality. Arguably, however, limitations imposed by the ground evaluator's current read-eval-print loop circumscribe our ability to provide desired functionality in a general way consistent with the aesthetic and underlying philosophy of the PVS language. Consider, for example, the PVS specifications that manipulate state and provide printing. What do the functions in theories `printing`, `printstrings`, and `state` mean in a PVS specification? In a very real sense, they are in semantic limbo, occupying a half-world between PVS and Lisp. It is virtually impossible to implement cleanly GUIs or interfaces to independently developed applications that use the ground evaluator. The application-specific workarounds implied in the discussions of the Rockwell Collins FGS visualization, Pamela+PVS, and the Maple/PVS Project reflect the problems inherent in trying to handle state, and I/O in PVS in general, and the ground evaluator in particular. Implications of the work described in this report for future work point most compellingly to a new read-eval-print loop realized as part of an integrated development environment for the ground evaluator.

The desired functionality of the new development environment would in many ways resemble that of a good Lisp debugger (e.g., the Lucid and CMU Lisp debug-

[1]Glade is a free user interface builder for GTK+ and Gnome released under the GNU License [9].

gers). In addition to the capabilities of the current read-eval-print loop, the new environment would provide the ability to

- assign values to variables, including structured variables
- print results, including the ability to print closures whose arguments have types on finite domains
- iteratively evaluate an expression over all possible values in the (finite) type domains of specified variables
- report test coverage data, including statistical information on function execution and coverage
- provide feedback from static analysis, e.g., indicate to the user those parts of the specification that are not single threaded
- extend current evaluation techniques to symbolic evaluation
- access generated functions, including the ability to look at and manipulate the Lisp translation yielded by evaluation of a PVS expression
- provide a facility analogous to the Emacs `C-M-x` command that allows the user to change a function dynamically
- provide step and trace facilities, i.e., allow the user to walk through the evaluation of an expression one step at a time or to trace specified functions encountered during expression evaluation

The ability to calculate and predict behaviors of a model using judicious combinations of deduction and validation techniques would be significantly increased by the enhancements and extensions to the PVS ground evaluator proposed here. The work described in this report provides the basis for this proposal, as well as pragmatic strategies for increasing the utility of the current PVS ground evaluator.

Bibliography

[1] Sten Agerholm and Peter Gorm Larsen. Modeling and validating SAFER in VDM-SL. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM' 97: Fourth NASA Langley Formal Methods Workshop*, pages 51–64, NASA Langley Research Center, Hampton, VA, September 1997. Available at http://atb-www.larc.nasa.gov/Lfm97/proceedings/. 10, 17

[2] Bettina Buth. An Interface between PAMELA and PVS. Unpublished Report, August 1998. 21, 24

[3] Bettina Buth. PAMELA + PVS. In Rudolph Berghammer and Yassine Lakhnech, editors, *Tool Support for System Specification, Development, and Verification*, pages 62–76, Malente, Germany, June 1998. Proceedings published in May 1999. 22

[4] Judith Crow et al. *NASA Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion*. NASA Office of Safety and Mission Assurance, Washington, DC, 1995. Available at http://eis.jpl.nasa.gov/quality/Formal_Methods/. 10

[5] Judith Crow et al. *NASA Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume II: A Practitioner's Companion*. NASA Office of Safety and Mission Assurance, Washington, DC, 1997. Available at http://eis.jpl.nasa.gov/quality/Formal_Methods/. 10

[6] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available, with specification files, at http://www.csl.sri.com/wift-tutorial.html. 12

[7] Ben L. Di Vito. High-automation proofs for properties of requirements models. *Software Tools for Technology Transfer*, 3(1):20–31, September 2000. 10

[8] Bruno Dutertre. Elements of mathematical analysis in PVS. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, Volume 1125 of Springer-Verlag *Lecture Notes in Computer Science*, pages 141–156, Turku, Finland, August 1996. 4

[9] `http://glade.pn.org`. 40

[10] `http://www.gnu.org/manual/emacs/index.html`. 18

[11] Hanne Gottliebsen. Transcendental functions and continuity checking in PVS. In Mark Aargaard and John Harrison, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, Volume 1869 of Springer-Verlag *Lecture Notes in Computer Science*, pages 197–214, Portland, OR, August 2000. 4, 23

[12] `http://www.maplesoft.com`. 23

[13] `http://www-theory.dcs.st-and.ac.uk/info/example/`. 21

[14] Steven P. Miller and James N. Potts. Detecting mode confusion through formal modeling and analysis. NASA Contractor Report CR-1999-208971, NASA Langley Research Center, Hampton, VA, January 1999. (Work performed by Rockwell Collins, Inc.). 17, 21, 22

[15] César Muñoz, January 2001. Personal communication. 12

[16] Karen L. Myers. Hybrid reasoning using universal attachments. *Artificial Intelligence*, 67(2):329–375, June 1994. 3

[17] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, Bedford, MA, second edition, 1990. 9

[18] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13(1 and 2):133–170, April 1980. 3

Appendix A

PVS System Buffers and Communication Functions

This appendix contains a description of PVS Emacs buffers and an annotated list of PVS functions used for interprocess communication.

A.1 PVS System Buffers and Associated Functions

In addition to the Emacs buffers that display PVS specification files, the PVS system minimally creates seven other buffers. For example, an Emacs `C-x C-b` command immediately after PVS starts up on the file `fcp_demo.pvs` yields the following list of buffers, where "." marks the selected buffer, "*" denotes a buffer that has been modified, but not saved, and "%" indicates a read-only buffer. We discuss buffers associated with PVS only; others buffers exist, including the initial `*scratch*` buffer (used for evaluating Lisp expressions in Emacs) and the `*Messages*` buffer (used to log Emacs activity), but these buffers do not play a role specific to PVS.[1]

```
 MR Buffer           Size  Mode              File
 -- ------           ----  ----              ----
.   fcp_demo.pvs     6625  PVS /homes/.. ./fcp_demo.pvs
    *scratch*           0  Lisp Interaction
 *  *Messages*       1345  Fundamental
 *  PVS Log           206  PVS View
    *ilisp-send*        0  Lisp
 *  *pvs*             611  ILISP
 *  pvs              1687  Fundamental
  % PVS Welcome      2278  Text
 *% *Buffer List*     400  Buffer Menu
```

[1]There is some overlap in the entries in the `PVS Log` buffer and the `*Messages*` buffer, but the latter logs Emacs activity, whereas the former records PVS activity.

Certain PVS commands, such as `new-pvs-file`, `help-pvs` (alternately, `pvs-help`), `prettyprint-expanded`, and `show-tccs`, create additional buffers. Buffers for the latter two have extensions `.ppe` and `.tccs`, respectively. `help-pvs` displays a summary of PVS commands in the PVS Help buffer. There is also a class of display commands, e.g., `x-theory-hierarchy`, `x-show-proof`, `x-show-current-proof`, `x-prover-commands`, and `x-prove`, that create a window and an Emacs buffer on a Tcl/Tk process. The following buffer-list shows the default PVS Tcl/Tk process buffer, `*tcl-pvs*`, created in response to the PVS `M-x x-gui` command.

```
 MR Buffer           Size  Mode         File
 -- ------           ----  ----         ----
.*  *pvs*            1261  ILISP
 *  *tcl-pvs*         392  Inferior Tcl
    fcp_demo.pvs     6625  PVS /homes/.. ./fcp_demo.pvs
    *scratch*           0  Lisp Interaction
 *  *Messages*    1533  Fundamental
 *  PVS Log           475  PVS View
    *ilisp-send*        0  Lisp
 *  pvs              6786  Fundamental
  % PVS Welcome      2278  Text
 *% *Buffer List*     194  Buffer Menu
```

A.2 PVS Interprocess Communication Functions

In the report, we frequently mention the current model of PVS process communication (cf. Figure 3.2). In this section, we provide an annotated list of the PVS Emacs functions that implement communication between PVS and Tcl/Tk, beginning with functions that support communication from PVS to Tcl. Unless otherwise noted, these functions may be found in file `pvs-tcl.el`.

- To run an inferior wish (Tcl) from PVS Emacs: `(ensure-pvs-wish)`. This function checks to see if an inferior wish is running and, if not, starts one.
- To send a string to an inferior Tcl process: `(tcl-send-string* <string>)`.
- To load a file into the inferior Tcl process: `(tcl-load-file <file> &optional and-go)` (in file `tcl.el`). The optional argument indicates whether or not to switch to the Tcl buffer after loading the file.
- To evaluate a string in the inferior Tcl process: `(tcl-eval <string>)`. (Uses `tcl-send-string*`.)
- To evaluate an expression in PVS Emacs and send the result to Tcl (using `tcl-send-string*`): `(pvs-tcl-emacs-eval <form>)`.

- To evaluate an expression in PVS Emacs (for use in PVS Emacs): `(pvs-tcl-emacs-eval-nowait <form>)`.

Communication from Tcl to PVS uses the following functions. Unless otherwise noted, these functions may be found in file `pvs-ilisp.el`.

- To evaluate an expression in PVS Emacs and return the result to the Tcl process: `(emacs-eval <arg>)`. Outputs to standard output stream (`stdout`) a string of the form `":pvs-eval $arg :end-pvs-eval"` and returns result via `stdin` (in file `pvs-support.tcl`).

- To evaluate an expression in PVS Emacs (result not returned): `":pvs-evaln $arg :end-pvs-evaln"` (in file `pvs-support.tcl`).

- To send a string to the inferior Lisp process: `(ilisp-send <string> &optional <message> <status> <and-go> <handler>)`. This function sends a string to the ilisp process. If `<and-go>` is `nil`, it prints `<message>`, sets `<status>`, and returns the result. If `<and-go>` is `true`, control switches to the ilisp process, where `<message>` and result are displayed. Other options for the value of `<and-go>` include `dispatch` (command is executed without waiting for results) and `call` (a call is generated). `ilisp-send` initializes the ilisp process (from the files in `ilisp-load-inits`) if it is the first ilisp command executed. If there is an error, `comint-errorp` is set to `t` and the error is handled by `<handler>`.

- To send a string to the PVS ilisp process (result is not returned): `(pvs-send <string> &optional <message> <status>)`. Calls `pvs-send*`, which in turn calls `ilisp-send` with `and-go` set to `dispatch`.

- To send a string to the PVS ilisp process and get the result: `(pvs-send-and-wait <string> &optional <message> <status> <expected>)`. Identical to `pvs-send` except that the result is returned. Prior to return, the result is processed using the value of `<expected>` (e.g., `bool`, `list`, `string`, `dont-care`).

- To send a string to PVS ilisp via a file: `(pvs-file-send-and-wait <string> &optional <message> <status> <expected>)`. Results in the following call: `(pvs-send-and-wait "(write-to-temp-file <string> t)" <message> <status> 'tmp-file)`. After the call, the file is deleted and its associated buffer killed. If the type of the result does not match the value of `<expected>`, an error buffer pops up indicating the expected return value(s) and the result.