User Guide for the PVS Specification and Verification System (Beta Release)

S. Owre and N. Shankar and J. M. Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025
{Owre,Shankar,Rushby}@csl.sri.com

March 31, 1993

Contents

1 Introduction

The Prototype Verification System (PVS) provides an integrated environment for the development and analysis of formal specifications, and supports a wide range of activities involved in creating, analyzing, modifying, managing, and documenting theories and proofs. This section provides a broad overview of PVS; the facilities provided by the system are discussed in the order a new user is likely to encounter them.

Environment

PVS runs on SUN 3 and SUN 4 (SPARC) workstations using the SUNOS or *Solaris* variants of the UNIX operating system. PVS is implemented in Common Lisp, but it is not necessary to know Common Lisp to effectively use the system.[1] The system was developed in Lucid Common Lisp and has been ported to Allegro, AKCL , and CMU Common Lisp. PVS runs under any window system that supports GNU EMACS. The GNU EMACS display editor provides the interface to PVS; familiarity with EMACS and access to the GNU EMACS manual [4] are desirable. The LaTeX generating facilities require a good understanding of the LaTeX document preparation system [1]. Instructions for installing the PVS system are given in Appendix A.

Language

The specification language of PVS is built on higher-order logic; *i.e.*, functions can take functions as arguments and return them as values, and quantification can be applied to function variables. There is a fairly rich set of built-in types and type-constructors, as well as a rather powerful notion of subtype. Specifications can be constructed using definitions, axioms, or a mixture of the two.

Specifications are logically organized into *theories* and *datatypes*, which can be parameterized with semantic constructs. Theories can be linked by import and export lists. Specifications for certain standard theories are preloaded into PVS as *prelude* theories that are always available and do not need to be explicitly imported. Details on the PVS language may be found in [2].

Specification Files and Context

PVS specifications are ordinary ASCII text prepared and modified using a text editor (usually the GNU EMACS editor that acts as the interface to PVS). A PVS specification consists of any number of files, each of which contains one or more theories or datatypes.

[1]The only exception to this is in writing complex prover strategies, though even this limited use will be unnecessary in future versions of PVS.

The set of files and theories constituting a specification, together with various items of status information, comprise a PVS *context*. Contexts retain information about the state of a specification and verification from one PVS session to another. Contexts are closely related to directories, and the term "context" is used in this document to refer to either the context or to the associated directory. Note that the directory may contain files other than those produced by or for PVS, but these are not considered to be a part of the context.

During a PVS session, there is always a *current context* in which the activities of PVS take place. For example, typechecking of a specification file is allowed only if the file is a part of the current context. There are commands for changing the current context during a PVS session, so that it is unnecessary to exit PVS just to change contexts. Because contexts are associated with UNIX directories there can be at most one PVS context in a directory, but a directory that contains a PVS context may also contain arbitrary additional files.

User Interface

The facilities of the PVS system are accessed through a customized GNU EMACS, which henceforth we will simply refer to as EMACS. It is expected, though not required, that editing of theory text is performed with this editor. Instructions are issued to PVS by means of EMACS commands. For example, in order to perform a proof, the cursor is positioned at a formula declaration in the EMACS buffer and the EMACS command `M-x prove` or the key sequence `C-c p` is issued. PVS returns information to the user through various display mechanisms provided by EMACS.

The PVS interface allows a certain amount of parallel activity. For example, the user can continue editing theories or perform any other activity supported by EMACS while PVS is typechecking a series of theories or performing a lengthy proof. Also, the user need not wait for one PVS activity to finish before issuing another command; most commands are queued for execution in the order they were issued, but certain status and other "short" commands preempt any ongoing analyses, perform their function, and then return the system to its previous activity.

Prettyprinting

The PVS prettyprinter rearranges the layout of PVS specification text into a standard, regular format. The commands allow the prettyprinting of files, theories, regions, or individual declarations. The user can choose whether to prettyprint specification text, but output from PVS itself is always prettyprinted.

Parsing

The parser checks theories for syntactic consistency and builds an internal representation that is used by other components of the system. When errors are detected by the parser or other components of PVS, the cursor is placed at the location where the error was detected and an error message is displayed in a pop-up window.

Typechecking

The PVS typechecker analyzes theories for semantic consistency and adds semantic information to the internal representation built by the parser. The type system of PVS is not algorithmically decidable; theorem proving may be required to establish the type-consistency of a PVS specification. The theorems that need to be proved are called *type-correctness conditions* (TCCs). TCCs are attached to the internal representation of the theory and displayed on request. Some TCCs are proved during typechecking; TCCs whose proofs require assistance from the user can be done at the user's convenience.

The PVS system automatically tracks the status of theories (whether they have been changed, parsed, typechecked etc.) and also takes care of the dependencies among theories. For example, if the specification text of a theory is changed and then a command is issued that requires semantic information, PVS will first parse and typecheck the theory automatically. More subtly, if the text of a theory that is used by the current theory is changed, both theories may need to be typechecked in order to guarantee consistency.

It is often necessary to make changes in theories on which long chains of other theories depend, and frequent reparsing and retypechecking of such theory chains can be very time-consuming. Therefore PVS provides commands which allow limited additions and modifications of declarations without requiring that the associated theories be retypechecked.

Browsing

Specifications can be quite large and involve many theories and files, and it can become difficult to remember all the identifiers declared, their locations, definitions, and uses. PVS provides facilities for displaying or visiting the declaration of an identifier indicated by the cursor, for displaying all references to an identifier, and for producing a cross-reference of all declared identifiers.

Proving

PVS provides a powerful interactive proof checker with the ability to store and replay proofs. PVS can be instructed to perform a single proof, or to rerun all the proofs in a given theory, all the proofs of all the lemmas used in a proof, or all the proofs in an entire specification. More information on the prover may be found in the prover reference manual [3].

Status and Proof Chain Analysis

The PVS system provides several commands for determining the status of specification elements such as theories and formulas. For example, the user can inquire whether a theory has been typechecked or whether a proof has been performed.

Proof chain analysis is an important form of status report. An individual theorem is considered proved when it is the conclusion of a successful proof, but this is a local notion of "proved;" the result is a true theorem only if all the lemmas appearing in its proof have themselves been proved or stated as axioms or definitions, and all TCCs have been discharged. Proof chain analysis assures that all of the aforementioned obligations are discharged; in addition to recording whether or not the proof chain is sound, the output of this analysis also identifies the axiomatic foundation of the given theorem.

Generating Output

When a formal specification and verification is complete, it is usually desirable to present it to others in as readable a form as possible. PVS provides commands for generating LaTeX versions of the specifications and proofs that can be included in typeset documents. The output produced can be controlled by user-supplied tables so that mathematical notation, including infix and mix-fix symbols and sub and superscripts, can be created easily. This customized prettyprinting facility makes it possible to reproduce the notation standard to some branch of mathematics or computer science, thereby assisting peer review of the formal specification. These typeset specifications are also of value to users during the development of a formal specification and verification, since they allow direct comparison with existing, informal presentations and analyses.

2 A Brief Tour of PVS

In this section we introduce the system by developing a theory and doing a simple proof. This will introduce the most useful commands and provide a glimpse into the philosophy behind PVS. You will get the most out of this section if you are sitting in front of a terminal with PVS installed (see Appendix A). In the following we assume familiarity with Sun Unix and GNU EMACS.

Start by going to a UNIX shell and creating a working directory (using `mkdir`). Next, connect (`cd`) to your working directory and start up PVS by typing `pvs`.[2] This command executes a shell script which runs GNU EMACS, loads the necessary PVS EMACS extensions, and starts the PVS lisp image as a subprocess.[3] After a few moments, you should see the welcome screen indicating the version of PVS being run, the current directory, and instructions for getting help. You may be asked whether you want to create a new context in the directory; answer `yes` unless it is the wrong directory or you don't have write permission there, in which case you should answer `no` and provide an alternative directory when prompted.

PVS uses EMACS as its interface by extending EMACS with PVS functions, but all the underlying capabilities of EMACS are available. Thus the user can read mail and news, edit non-PVS files, or execute commands in a shell buffer in the usual way.

In the following, PVS EMACS commands are given first in their long form, followed by an alternative abbreviation and/or key binding in parentheses. For example, the command for proving in PVS is given as `M-x prove` (`M-x pr`, `C-c p`). This command can be entered by typing the `Escape` key, then an `x`[4] followed by `prove` (or `pr`) and the `Return` key. Alternatively, hold the `Control` key down while typing a `c`, then let go and type a `p`. The `Return` key does not need to be pressed when giving the key binding form. In PVS all commands and abbreviations are preceded by a `M-x`; everything else is a key-binding. In later sections we will refer to commands by their long form name, without the `M-x` prefix. Some of the commands prompt for a theory or PVS file name and specify a default; if the default is the desired theory or file, you can simply type the `Return` key.

To begin, type `M-x pvs-help` (`C-h p`) for an overview of the commands available in PVS (type `q` to exit the help buffer). To exit PVS, use `M-x exit-pvs` (`C-x C-c`).

PVS specifications consist of a number of files, each of which contains one or more theories. Theories may import other theories; imported theories must either be one

[2] You may need to include a pathname, depending on where and how PVS is installed.

[3] All the GNU EMACS (and X-Windows or Emacstool) command line flags can be added to the `pvs` command and passed through as appropriate; the `-q` flag inhibits loading of the user's `.emacs` initialization file, and should be used if difficulties are encountered starting PVS or if there appear to be conflicts in keybindings.

[4] Many keyboards provide a `Meta` key (hence the `M-` prefix), and this may be used instead. On the SUN3, the `Meta` key is normally labeled `Left` and on the SUN4, it is labeled ◇. The `Meta` key is like the shift key; to use it simply hold the `Meta` key down while typing another key.

of the built-in theories of PVS (the *prelude*) or the files containing them must be in the same directory.[5] There is also an internal database of theories which have been introduced to the system which is referred to as a *context*. PVS maintains the state of the context between sessions by means of the `.pvscontext` file, which contains information about the state of the files and their associated theories. Specification files in PVS all have a `.pvs` extension. As specifications are developed, their proofs are kept in files of the same name with `.prf` extensions. The `.pvscontext` and `.prf` files are not meant to be modified by the user. Other files used or created by the system will be described as needed. You may move to a different context (*i.e.*, directory) using the `M-x change-context` command, which is analogous to the UNIX `cd` command.

Now let's develop a small specification. Figure 1 shows a specification for summation of the first n natural numbers, as it appears in EMACS. The `sum` specification is in the top window, and a proof is in progress in the bottom. The mode line indicates that PVS is `ready` for a command.

This simple theory has no parameters and contains three declarations. The first declares `n` to be a variable of type `nat`, the built-in type of natural numbers. The next declaration is a recursive definition of the sum of the function `sum(n)` whose value is the sum of the first `n` natural numbers. Associated with this definition is a *measure* function, following the `MEASURE` keyword, which will be explained below. The final declaration is a formula which gives the closed form of the sum.

Creating the Specification

The `sum` theory may be introduced to the system in a number of ways, all of which create a file with a `.pvs` extension,[6] which can be done by

1. using the `M-x new-pvs-file` command (`M-x nf`) to create a new PVS file, and typing `sum` when prompted. Then type in the `sum` specification.

2. Since the file is included on the distribution tape in the `Examples/tutorial` subdirectory of the main PVS directory, it can be imported with the `M-x import-pvs-file` command (`M-x imf`). Use the `M-x whereis-pvs` command to find the path of the main PVS directory.

3. Finally, any external means of introducing a file with extension `.pvs` into the current directory will make it available to the system; for example, using `vi` to type it in, or `cp` to copy it from the `Examples/tutorial` subdirectory.

[5] PVS does support soft links, thus supporting a limited capability for reusing theories.
[6] The file does not have to be named `sum.pvs`, it simply needs the `.pvs` extension.

Figure 1: The `sum` Specification

The first two options display the specification in a buffer. The third option requires an explicit request such as a built-in GNU EMACS file command (like `M-x find-file`, `C-x C-f`), or the `M-x find-pvs-file` (`M-x ff` or `C-c C-f`) command. The latter is more useful when there are multiple specification files, as it supports completion on just the specification files, ignoring other files that you or the system have created in the directory.

Parsing

Once the `sum` specification is displayed, it can be parsed with the `M-x parse` (`M-x pa`) command, which creates the internal abstract representation for the theory described by the specification. If the system finds an error during parsing, an error window will pop up with an error message, and the cursor will be placed in the vicinity of the error. If you didn't get an error, introduce one (say by misspelling the `VAR` keyword), then move the cursor somewhere else and parse the file again (note that the buffer is automatically saved). Fix the error and parse once more. In practice, the parse command is rarely used, as the system automatically parses the specification when it needs to.

Typechecking

The next step is to typecheck the file by typing `M-x typecheck` (`M-x tc`, `C-c t`), which checks for semantic errors, such as undeclared names and ambiguous types. When `sum` has been typechecked, a message is displayed in the minibuffer indicating that three TCCs were generated. These TCCs represent *proof obligations* that must be discharged before the `sum` theory can be considered typechecked. The proofs of the TCCs may be postponed indefinitely, though it is a good idea to view them to see if they are provable. TCCs can be viewed using the `M-x show-tccs` command, the results of which are shown in Figure 2 below.

```
% Subtype TCC generated for n - 1
  % unproved
sum_TCC1: OBLIGATION (FORALL n: NOT n = 0 IMPLIES n - 1 >= 0)

% Termination TCC generated for sum
  % unproved
sum_TCC2: OBLIGATION (FORALL n: NOT n = 0 IMPLIES n - 1 < n)

% Subtype TCC generated for 2
  % unproved
closed_form_TCC1: OBLIGATION 2 /= 0
```

Figure 2: TCCs for Theory `sum`

The first TCC is due to the fact that `sum` takes an argument of type `nat`, but the type of the argument in the recursive call to `sum` is integer, since `nat` is not closed under subtraction. Note that the TCC includes the condition `NOT n = 0`, which holds in the branch of the `IF-THEN-ELSE` in which the expression `n - 1` occurs.

The second TCC is needed to ensure that the function `sum` is total, *i.e.*, terminates. PVS does not directly support partial functions, although its powerful subtyping mechanism allows PVS to express many operations that are traditionally regarded as partial. The measure function is used to show that recursive definitions are total by requiring the measure to decrease with each recursive call.

The final TCC is due to the fact that division in PVS is declared to be a function of type `[number, nonzero_number -> number]`, so the denominator must be nonzero.

These TCCs are trivial, and in fact can be discharged automatically by using the `M-x typecheck-prove` (`M-x tcp`) command, which attempts to prove all TCCs that have been generated. (Try it).

Proving

We are now ready to try to prove the main theorem. Place the cursor on the line containing the `closed_form` theorem, and type `M-x prove` (`M-x pr` or `C-c p`). A new buffer will pop up, the formula will be displayed, and the cursor will appear at the `Rule?` prompt, indicating that the user can interact with the prover. The commands needed to prove this theorem constitute only a very small subset of the commands available to the prover; more details can be found in the prover guide [3].

First, notice the display, which consists of a single formula (labeled {`1`}) under a dashed line. This is a *sequent*; formulas above the dashed lines are called *antecedents* and those below are called *consequents*. The interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the consequents. Either or both of the antecedents and consequents may be empty.[7] In our case, we are trying to prove a single consequent.

The basic objective of the proof is to generate a *proof tree* of sequents in which all of the leaves are trivially true. The nodes of the proof tree are sequents, and while in the prover you will always be looking at an unproved leaf of the tree. The *current* branch of a proof is the branch leading back to the root from the current sequent. When a given branch is complete (*i.e.*, ends in a proved leaf), the prover automatically moves on to the next unproved branch, or, if there are no more unproven branches, notifies you that the proof is complete.

Now back to the proof. We will prove this formula by induction on `n`. To do this, type `(induct "n")`.[8] This is not an EMACS command, rather it is typed directly at

[7] An empty antecedent is equivalent to `true`, and an empty consequent is equivalent to `false`, so if both are empty the sequent is unprovable.

[8] PVS expressions are case-sensitive, and must be put in double quotes when they appear as arguments in prover commands.

the prompt, including the parentheses. This generates two subgoals; the one displayed is the base case, where `n` is `0`. To see the inductive step, type `(postpone)`, which postpones the current subgoal and moves on to the next unproved one. Type `(postpone)` a second time to cycle back to the original subgoal (labeled `closed_form.1`).[9]

To prove the base case, we need to expand the definition of `sum`, which is done by typing `(expand "sum")`. After expanding the definition of `sum`, the prover applies the underlying decision procedures to simplify the consequent to `TRUE`, which completes the proof of this subgoal, and the system moves on to the next subgoal, which is the inductive step.

The first thing to do here is to eliminate the `FORALL` quantifier. This can most easily be done with the `skolem!` command[10], which provides new constants for the bound variables. To invoke this command type `(skolem!)` at the prompt. The resulting formula may be simplified by typing `(flatten)`, which will break up the consequent into a new antecedent and consequent. The obvious thing to do now is to expand the definition of `sum` in the consequent. This again is done with the `expand` command, but this time we want to control where it is expanded, as expanding it in the antecedent will not help. So we type `(expand "sum" +)`, indicating that we want to expand `sum` in the consequent.[11]

The final step is to send the proof to the PVS decision procedures, which can automatically decide certain fragments of arithmetic. This is done by typing `(assert)`.[12] The proof is now complete, the system may ask whether to save the new proof, and whether to display a brief printout of the proof. You should answer `yes` to these questions just to see how they work. After responding to these questions, the buffer from which the `prove` command was issued is redisplayed if necessary, and the cursor is placed on the formula that was just proved. The entire proof transcript is shown

[9]Three extremely useful EMACS key sequences to know here are `M-p`, `M-n`, and `M-s`. `M-p` gets the last input typed to the prover; further uses of `M-p` cycle back in the input history. `M-n` works in the opposite direction. To use `M-s`, type the beginning of a command that was previously input, and type `M-s`. This will get the previous input that matches the partial input; further uses of `M-s` will find earlier matches. Try these key sequences out; they are easier to use than to explain.

[10]The exclamation point differentiates this command from the `skolem` command, where the new constants have to be provided by the user.

[11]We could also have specified the exact formula number (here `1`), but including formula numbers in a proof tends to make it less robust in the face of changes. There is more discussion of this in the prover guide [3].

[12]The `assert` command actually does a lot more than decide arithmetical formulas, performing three basic tasks:

- it tries to prove the subgoal using the decision procedures.
- it stores the subgoal information in an underlying database, allowing automatic use to be made of it later.
- it simplifies the subgoal, again utilizing the underlying decision procedures.

These arithmetic and equality procedures are the main workhorses to most PVS proofs. You should learn to use these effectively in a proof.

below. Yours may be different, depending on your window size and the timings involved.

```
closed_form :

  |-------
{1}   (FORALL (n: nat): sum(n) = (n * (n + 1)) / 2)

Rule? (induct "n")
Inducting on n,
this yields  2 subgoals:
closed_form.1 :

  |-------
{1}   sum(0) = (0 * (0 + 1)) / 2

Rule? (postpone)
Which takes us to,
closed_form.2 :

  |-------
{1}   (FORALL (j: nat):
         sum(j) = (j * (j + 1)) / 2
           IMPLIES sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2)

Rule? (postpone)
Which takes us to,
closed_form.1 :

  |-------
{1}   sum(0) = (0 * (0 + 1)) / 2

Rule? (expand "sum")
Expanding the definition of sum
this simplifies to:
closed_form.1 :

  |-------
{1}   TRUE

which is trivially true.

This completes the proof of closed_form.1.

closed_form.2 :

  |-------
{1}   (FORALL (j: nat):
         sum(j) = (j * (j + 1)) / 2
           IMPLIES sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2)

Rule? (skolem!)
For the top quantifier in 1, we introduce Skolem constants: (j!1)
this simplifies to:
closed_form.2 :

  |-------
{1}   sum(j!1) = (j!1 * (j!1 + 1)) / 2
        IMPLIES sum(j!1 + 1) = ((j!1 + 1) * (j!1 + 1 + 1)) / 2

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
```

```
closed_form.2 :

{-1}   sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   sum(j!1 + 1) = ((j!1 + 1) * (j!1 + 1 + 1)) / 2

Rule? (expand "sum" +)
Expanding the definition of sum
this simplifies to:
closed_form.2 :

[-1]   sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   (IF j!1 + 1 = 0 THEN 0 ELSE j!1 + 1 + sum(j!1) ENDIF)
        = (j!1 * j!1 + 2 * j!1 + (j!1 + 2)) / 2

Rule? (assert)
Invoking decision procedures,
this simplifies to:
closed_form.2 :

{-1}   sum(j!1) = (j!1 * j!1 + j!1) / 2
  |-------
{1}   TRUE

which is trivially true.

This completes the proof of closed_form.2.

Q.E.D.


Run time  = 5.62 secs.
Real time = 58.95 secs.
```

A brief version of the just completed proof can be generated by the Emacs command `M-x show-last-proof`.

Status

Now type `M-x status-proof-theory` (`M-x spt`) and you will see a buffer which displays the four formulas in `sum`, along with an indication of their proof status. This command is useful to see which formulas and TCCs still require proofs. Another useful command is `M-x status-proofchain` (`M-x spc`), which analyzes a given proof to determine its dependencies. To use this, go to the `sum.pvs` buffer, place the cursor on the `closed_form` theorem, and enter the command. A buffer will pop up indicating whether the proof is complete, and that it depends on the TCCs and the `nat_induction` axiom.

Generating LaTeX

In order to try out this section, you must have access to LaTeX and a TeX previewer, such as `vitex` or `dvitool` (for SunView), or `xdvi` (for X-windows). Otherwise this section may be skipped.

Type `M-x latex-theory-view` (`M-x ltv`). You will be prompted for the theory name—type `sum`, or just `Return` if `sum` is the default. You will then be prompted for the TeX previewer name. Either the previewer must be in your path, or the entire pathname must be given. This information will only be prompted for once per session, after that PVS assumes that you want to use the same previewer.

sum : THEORY
 BEGIN
 n : VAR nat
 sum(n) : RECURSIVE nat $=$
  (IF $n = 0$ THEN 0 ELSE $n + \mathrm{sum}(n - 1)$ ENDIF)
   MEASURE $(\lambda\ n : n)$
 closed_form : THEOREM $\mathrm{sum}(n) = (n \times (n + 1)) / 2$
 END sum

Figure 3: Theory `sum`

After a few moments the previewer will pop up displaying the `sum` theory, as shown in Figure 3. Note that `LAMBDA` has been translated as λ. This and other translations are built into PVS; the user may also specify translations for keywords and identifiers by providing a substitution file, `pvs-tex.sub`, which contains commands to customize the LaTeX output. For example, if the substitution file contains the three lines

```
THEORY key 7 {\large\bf Theory}
sum    id  2 {\sum_{i = 0}^{\star_1} i}
sum    1   2 {\sum_{i = 0}^{#1} i}
```

the output will look like Figure 4. See the section on Generating LaTeX Output in Chapter 3 for details.

sum : **Theory**

BEGIN

n : VAR nat

$\sum_{i=0}^{\star 1} i(n)$: RECURSIVE nat $=$ (IF $n = 0$ THEN 0 ELSE $n + \sum_{i=0}^{n-1} i$ ENDIF)
    MEASURE $(\lambda\ n :\ n)$

closed_form : THEOREM $\sum_{i=0}^{n} i = (n \times (n + 1)) / 2$

END sum

Figure 4: Theory `sum`

3 PVS Commands

This chapter contains descriptions for all PVS commands; the commands are grouped according to function and each group appears in an easily referenced section. A summary of the information in this chapter is also provided on the PVS reference card. The information in this chapter is best absorbed after reading and experimenting with the brief tour provided in Chapter 2.

Each of the following sections begins with a table summarizing the commands discussed in that section; each table entry gives the full name of the command, available aliases and/or key bindings, a brief description, and the effect of providing command arguments. Commands are invoked by typing `M-x` followed by the command name or its abbreviation, or by using a (less mnemonic) key sequence. For example, the `typecheck` command can be invoked by typing `M-x typecheck` or one of the alternate forms `M-x tc` or `C-c t`. The behavior of many of the commands can be modified by providing an argument.[13] For example, preceding the `typecheck` command with a `C-u` or `M-1` forces the file to be reparsed and typechecked, even if it has already been typechecked. Each command that takes an argument has a second line prefixed by *Arg:* that describes the effect of the argument.

Many PVS commands are appropriate at either the file or theory level; yielding two different commands. For example, the command for creating a new PVS file is `new-pvs-file`, while the command `new-theory` creates a template for a new theory within the current PVS file. In general, a command *foo* that applies to both files and theories will have a version named `M-x` *foo*`-pvs-file` and one named `M-x` *foo*`-theory`.

Exiting PVS

Command	*Aliases*	*Function*
`suspend-pvs`	`C-x C-z`	Suspend PVS
`exit-pvs`	`C-x C-c`	Terminate PVS session

The `suspend-pvs` command suspends the EMACS process, except under X-windows, where the command has no effect. The suspended job can be restarted from the UNIX shell in which it was suspended by first determining the job number (using the UNIX command "`jobs`") and then typing "`fg %`*n*", where *n* is the job number.

The `exit-pvs` command first saves the context information (see the `save-context` command) and then exits PVS permanently. If there is a proof in progress, the system will not exit, but will instead provide a message; thus allowing the user to exit the prover and save the proof.

[13]See Section 4.9 of [4] for details on providing arguments to commands.

Editing PVS Files

Command	*Aliases*	*Function*
`forward-theory`	`M-]`	Move forward to beginning of next theory
`backward-theory`	`M-[`	Move backward to beginning of previous theory
`find-unbalanced-pvs`	`C-c ]`	Find unbalanced delimiters
`comment-region`	`C-c ;`	Comment out all lines in the current region
		Arg: Uncomment all lines in the current region

The `forward-theory` and `backward-theory` commands are used to move to different theories within a single PVS file. The cursor is moved to the beginning of a theory; if there are no preceding or following theories to move to, the message "`No more theories`" is displayed and the cursor remains unchanged.

The `find-unbalanced-pvs` command checks whether there are any unbalanced parentheses (`(, )`), square brackets (`[, ]`), curly braces (`{, }`), or `BEGIN-END` pairs. If none are found, the message "`All delimiters balance`" is displayed. Otherwise the cursor is left at the token for which there is no match and a corresponding message is displayed.

The `comment-region` command inserts the comment character (`%`) at the beginning of every line in the specified region. To uncomment a region, simply provide an argument to the command, and all commented lines within the region will be uncommented.

Parsing

Command	*Aliases*	*Function*
`parse`	`pa`	Parse file in current buffer
		Arg: Forces the file to be reparsed

Parsing a PVS specification accomplishes two things: first, it checks that the specification is syntactically correct, *i.e.*, satisfies the PVS grammar, and second, it builds the internal abstract grammar data structures. The `parse` command is not normally used, as typechecking will automatically parse the file if required as well. Note that only files (with extension `.pvs`) may be parsed. When a file is parsed, it becomes a part of the context, and any proofs that have been saved for the file are reinstated.

Parsing is invoked by moving the cursor to a buffer containing a file in the current context, and issuing the `parse` command. While parsing the file, the minibuffer displays the message "`Parsing` *foo*." If there is no error, the message "*foo* `parsed in` # `seconds`" is displayed. If the file has not changed since the last time it was parsed, the message "*foo* `is already parsed`" is displayed. To force reparsing, provide an argument to the parse command.

When an error is detected, the file is displayed with the cursor at the location where the error was detected, which is usually later than the actual source of the error. In addition, a popup buffer is displayed with the error message. Most error messages are self-explanatory, though they may require a glance at the grammar (see [2]).

Typechecking

Command	*Aliases*	*Function*
`typecheck`	`tc`, `C-c t`	Typecheck file in current buffer
		Arg: Forces reparsing and retypechecking
`typecheck-prove`	`tcp`	Typecheck file in current buffer, proving TCCs
		Arg: Forces reparsing and retypechecking

Typechecking a PVS specification checks semantic constraints, determines the types of expressions, and resolves names (see [2]). Typechecking is invoked much like parsing, and automatically parses the file if necessary. Errors are indicated in the same manner as for parsing, although the cursor is usually more accurately positioned at the error. As in parsing, an argument to the command forces reparsing and retypechecking.

The typechecker will automatically attempt to typecheck any theories appearing in `IMPORTING` clauses. If the theories appear in the current context, then the associated file is typechecked, otherwise PVS tries to find a file with the same name as the theory. For example, in typechecking

```
IMPORTING foo[int]
```

the current context (reflected in the context file `.pvscontext`) is searched for a file known to contain `foo`. If no such file is found, then the file `foo.pvs` is sought. If that also cannot be found, the system complains and the desired file must be manually located (or created) and typechecked.

The `typecheck-prove` command typechecks the file, attempting to prove the TCCs as they are generated. If the file is already typechecked, but the TCC proofs have not yet been attempted, then they are attempted in the order they were generated.

When typechecking is completed, a message is displayed, indicating the total number of TCCs generated along with a breakdown of the number proved, subsumed[14], and unproved.

[14] A TCC is subsumed if there is an earlier TCC which implies it. PVS uses a simple syntactic test, so not all possible subsumptions will be determined.

Prove[15]

Command	*Aliases*	*Function*
prove	pr, C-c p	Prove formula pointed to by cursor
redo-proof	C-c C-p r	Redo the proof of formula pointed to by cursor *Arg:* don't display the proof
prove-theory	prt, C-c C-p t	Rerun unproved proofs in specified theory *Arg:* include those already proved
prove-pvs-file	prf, C-c C-p f	Rerun unproved proofs in current file *Arg:* include those already proved
prove-importchain	pri, C-c C-p i	Rerun prove-theory on IMPORT chain *Arg:* include those already proved
prove-proofchain	prp, C-c C-p p	Rerun proofs on all formulas in proof chain *Arg:* include those already proved

The prover may be applied to a single formula, all formulas in a theory, all formulas in the import chain of a theory, all formulas in a PVS file, or all formulas in the proof chain of a given formula. Only the `prove` command leads to prover interaction; the other commands simply rerun proof scripts that have been generated previously.

To invoke the prover on a single formula, move the cursor to any part of the desired formula and type the `prove` command. The formula may be in a PVS file, a buffer generated by the `prettyprint-expanded` command (with extension `.ppe`), a buffer generated by the `show-tccs` command (with extension `.tccs`), or a prelude buffer produced by one of the `view-prelude` commands.[16] If the formula has already been proved, then the user is asked whether the proof should be retried; a `no` answer ends the `prove` command. Otherwise, if the formula has an associated proof script, the user is asked whether to rerun the proof or start over. In either of these two cases, the proof is displayed in the `*pvs*` buffer. If the proof script terminates before completing the proof or if no script was requested, the prover will prompt for a command, which should be typed directly into the `*pvs*` buffer at the `Rule?` prompt. At this point the user is interacting with the prover, and certain commands will be unavailable until the prover is exited.[17] After asking whether to print a summary or save the proof script the prover exits automatically when a proof is successfully completed. If at any time you want to exit the prover, go to the bottom of the `*pvs*` buffer[18] and type `(quit)` to the `Rule?` prompt. If there is no such prompt, type `C-c C-c` and `(restore)` to get to the prompt. Once the prover is exited, control is returned

[15]The prover is described in full in [3], here we simply describe the commands used to invoke the prover and to run it in batch mode.

[16]Of course, the prelude formulas have already been proved; this facility allows the user to explore the proofs.

[17] Specifically, the commands `parse`, `typecheck`, `prove`, `change-context`, `exit-pvs`, and all of the prove commands of this section are unavailable while the prover is active.

[18]While in the prover you may freely move to any other buffer to examine specifications or perform ordinary editing functions.

to the buffer from which the prover was invoked, with the cursor positioned at the beginning of the formula being proved. Do not kill the `*pvs*` buffer, as this will also kill the associated PVS process.

The `redo-proof` command is invoked exactly like the prove command, but simply reruns the proof with no questions asked. An error is signaled if the indicated formula has no associated proof. In addition, if an argument is provided, the proof will not be displayed interactively—instead the proof is processed in the background, and the status of the proof is provided in the minibuffer when the attempt is completed.

PVS keeps track of the status of formulas within and across sessions. The status may be one of four values; "unproved" means that no proof has been attempted, "proved" means that the proof is complete, and "unfinished" means that a proof has been attempted, but not yet completed. Modifying a specification causes the proof status of all formulas to revert to "unchecked," although the proof scripts are retained.[19] Proof scripts can be rerun using the `prove-theory`, `prove-pvs-file`, `prove-importchain` and `prove-proofchain` commands, which simply rerun the proof scripts, if any, for all of the formulas of the theory, its PVS file, *import chain*, or *proof chain*, respectively.[20] These commands skip formulas that have no proof scripts, and normally skip formulas which already have status "proved;" providing an argument to the command forces PVS to reprove all formulas that have proof scripts. When any of these commands finish processing, the corresponding proof status command is automatically invoked to display the results (see the Proof Status Commands described below).

Editing Proofs

Command	*Aliases*	*Function*
`edit-proof`		Edit the proof of the indicated formula
`install-proof`	`C-c C-i`	Install the edited proof on the indicated formula
`revert-proof`		Revert to previous proof of the indicated formula
`show-proof-file`		Edit the proof file of the indicated PVS file
`show-orphaned-proofs`		Display orphaned proofs

Every formula of a specification for which a proof has been attempted has an associated proof script, which reflects the commands used during the proof attempt. Proof scripts may be edited using the `edit-proof` command. This command is invoked on the formula declaration at the cursor; the formula may occur in a specification buffer

[19] PVS currently tracks the consequences of changes rather coarsely: any change in a file reverts all the proofs in that file, and all those in theories that depend on that file (and so on, transitively) to the "unchecked" state. Later versions of PVS will provide more fine-grained tracking of changes.

[20] The import chain of a theory is simply the transitive closure of the `IMPORTs` (including those implicit in a theory declaration—see the language reference [2]). The proof chain of a given formula is the transitive closure of the formulas used in the proof of that formula.

(with extension `.pvs`), a prettyprint-expanded buffer (with extension `.ppe`), a show-tccs buffer (with extension `.tccs`), or a buffer generated by one of the `view-prelude` commands. When the `edit-proof` command is invoked, it creates a buffer with the name `Proof` containing the relevant proof script,[21] which may then be edited using the standard EMACS editing commands. Editing proof scripts is a convenient way to handle modifications made to a specification, and allows the same proof script to be revised and used for many similar formulas. The `Proof` buffer normally persists until the next time the `edit-proof` command is invoked, allowing the same proof script to be attached to different formulas.

A proof script records a tree of prover commands that will generate a proof of the given formula. Although the proof tree does not record verbatim the commands originally typed to the prover, the proof script should be easy to understand. For example, the `Proof` buffer of the formula `closed_form` in the `sum` example would contain

```
("" (INDUCT "n")
    (("1" (EXPAND "sum") (PROPAX))
     ("2" (SKOLEM 1 ("j!1"))
          (FLATTEN)
          (EXPAND "sum" +)
          (ASSERT)
          (PROPAX))))
```

When editing is complete, the proof script may be attached either to the original, or to a different formula using the `install-proof` command. If this command is invoked in the `Proof` buffer, it attaches the new proof script to the original formula and offers to rerun the proof. The proof script may also be attached to any other formula by invoking `install-proof` in a `.pvs`, `.ppe`, or `.tccs` buffer, in which case the script is attached to the formula at the cursor. The previous proof script for the chosen formula is saved, and may be retrieved by means of the `revert-proof` command, which swaps the previous script with the current one. If no proof is being edited, but a proof is being viewed in the `View Proof` buffer, then that proof will be installed on the indicated formula.

If a proof is in progress, proofs may still be edited, but the prover must be exited before the edited proof may be attached to a formula. An error is signaled if there is no proof associated with the formula targeted for the `edit-proof` command. Note that editing the proof of the formula currently being proved will display the previous proof script, if there is one. To display the current proof script, use the `show-proof` command described below.

As noted above, each specification file (with extension `.pvs`) has an associated proof file of the same name with a `.prf` extension. This file contains the proof scripts for all of the formulas of the specification file whose proofs have been attempted. The

[21] If the formula has no proof script, an empty `Proof` buffer is created.

`show-proof-file` command allows you to browse a proof file, and select or view any of the associated proof scripts. A `Proofs File` buffer is created with a line for each proof script in the file. The user may select a proof script for editing, or simply view the script in a popup buffer. This command may be used to look at the proof file of any context or PVS file—in this respect it is analogous to the `import-pvs-file` command.

To view a proof script, place the cursor on the desired line, and type "`v`." The proof script will be displayed in a popup buffer, but may not be edited. To edit a proof script, position the cursor and type "`s`." This will create a `Proof` buffer which may be edited and attached to formulas exactly as described above.

While developing a specification, some theorems or even entire theories may be moved around or deleted, leaving the attached proofs *orphaned*. Orphaned proofs are saved in the `orphaned-proofs.prf` file and are still available. In some cases, the system will recognize that an orphaned proof should be attached to a formula, and will do so after querying the user.

The `show-orphaned-proofs` command provides access to the orphaned proofs file by means of an `Orphaned Proofs` buffer that displays the formula name, theory name, and file name associated with each orphaned proof. A given proof may be selected by moving the cursor to the line and typing "`s`," which pops up the `Proof` buffer. This buffer is the same as the one generated by the `edit-proof` command, except that there is no default formula, so that `install-proof` (`C-c C-i`) will not work from the `Proof` buffer. Typing a "`d`" on a proof line deletes the corresponding entry from the orphaned proof file, typing a "`v`" will pop up a `View Proof` buffer, and typing a "`q`" exits the orphaned proof buffer.

Displaying Proof Information

Command	*Aliases*	*Function*
`show-proof`		Display the current proof
`show-last-proof`		Displays brief printout of most recent proof
`ancestry`		Display the ancestry of the current sequent
`siblings`		Display the siblings of the current sequent
`show-hidden-formulas`		Display the hidden formulas in the current sequent
`show-auto-rewrites`		Display the currently used auto-rewrite rules

These commands work only while an interactive proof is being developed. The `show-proof` command shows the current proof in the same format as the `edit-proof` command, but the displayed proof may not be edited. The primary use of this facility is for reviewing the development of a proof in progress and applying it to other branches using the `rerun` prover command.

The `show-last-proof` command provides an abbreviated version of the commentary and subgoals associated with the most recently completed proof. This version does not contain the `undo` or `postpone` steps and provides a clean version that shows the commentary and subgoals (at the branch points in the proof). This printout is useful in trying to summarize the proof for publication.

The `ancestry` command displays the branch of the proof from the root to the current sequent, and the `siblings` command displays the siblings of the current sequent, where the siblings are those sequents of the proof tree which share the same parent.

The `show-hidden-formulas` command displays the formulas that have been hidden in the current branch of the proof. Each formula is displayed with a number which may be referred to in the `reveal` proof command (see [3]).

The `show-auto-rewrites` command displays the auto-rewrite rules that are in effect for the current sequent, in reverse of the order in which they were introduced *i.e.*, the most recently introduced ones first. The order is significant since if there is clash and two or more rewrite rules are applicable, the most recent one is applied first.

Adding and Modifying Declarations

Command	*Aliases*	*Function*
`add-declaration`		Add declarations to a PVS theory
`modify-declaration`		Modify the indicated declaration body

Declarations are normally added and modified directly in a specification buffer; the system determines the differences and updates the corresponding internal structures accordingly. This can be quite expensive, as any theories which import a modified theory must be retypechecked.[22] However, there are two commands that allow declarations to be added and modified without causing retypechecking. This is especially important during proof development, when these commands allow the user to make adjustments to theories precisely when the need for such an adjustment is usually discovered.

The `add-declaration` command inserts new declarations before the declaration at the cursor. When invoked, it pops up an empty buffer named `Add Declaration`. Declarations may be typed in and edited just as in a specification buffer. When editing is completed, the new declarations may be installed by typing `C-c C-c`. The new declarations are parsed, typechecked, and checked for uniqueness; if an error is discovered it is reported in the usual way. If there is no error, the declarations are inserted above the declaration located at the cursor when the `add-declaration` command was invoked. If a proof is in progress, it will have access to the new

[22] In future versions of PVS, incremental typechecking will be available, which will greatly reduce the amount of retypechecking needed. The commands described here will still be useful during proof development.

declarations if they are visible, *i.e.*, exported,[23] declarations of a theory used by the theory whose formula is being proved, or they occur in the same theory and precede the formula being proved.

The `modify-declaration` command is used to modify the body of a constant or formula declaration; modifying the signature of a constant or any other kind of declaration is not permitted because these modifications have potentially non-local ramifications. This command is similar to the `add-declaration` command: a buffer pops up containing the declaration at the cursor, and the modified declaration is installed by typing `C-c C-c`. If the modified declaration typechecks and maintains the same id and signature, it is installed in the theory and is immediately available for use in a proof. Otherwise the cursor is placed in the vicinity of the error and a message is displayed indicating the nature of the error.

Both `add-declaration` and `modify-declaration` update the buffer containing the affected theory and mark the buffer as unchanged; the system considers the affected theory typechecked. However, the checks cannot guarantee that everything is sound; for example, any proofs done using a declaration that was later modified will need to be reproved, and any theory which uses a theory to which declarations have been added should eventually be retypechecked, as ambiguities may have inadvertently been introduced.[24] Thus these commands should be viewed as an expedient way to explore proofs; they should not be used in the "validation" phase of the verification. Proofs constructed when either of these commands is successfully used are marked unchecked; *i.e.*, the proofs will need to be rerun to change their status to proved.

Prettyprinting

Command	*Aliases*	*Function*
`prettyprint-theory`	`ppt`	Prettyprint theory in current buffer
`prettyprint-file`	`ppf`	Prettyprint file in current buffer
`prettyprint-declaration`	`ppd`, `C-M-q`	Prettyprint declaration at the cursor
`prettyprint-region`	`ppr`, `C-M-\`	Prettyprint declarations within the region

These commands are used to prettyprint portions of a specification using the built-in formatting rules. The prettyprinted sections replace the originals in the specification buffers, which are then marked as unmodified. If the prettyprinted version is not the desired one, the EMACS commands `undo` or `revert-buffer` may be used to return to the earlier state. Prettyprint commands are used primarily to

[23] See the Language Reference for a definition of exported declarations. In short, formal parameters and variable declarations may never be exported, and, by default, everything else is exported.

[24] In practice, all such ambiguities may be resolved by including the theory name with the ambiguous id, *e.g.*, `mytheory.foo` instead of `foo`.

"clean-up" after adding new declarations or making a significant change to an existing declaration.

The `prettyprint-theory` command prettyprints the specified theory, and the `prettyprint-file` prettyprints all the theories of the specified file; if the file has only one theory, then these are equivalent. The `prettyprint-declaration` command prettyprints the declaration at the cursor and the `prettyprint-region` command prettyprints all the declarations within the specified region.

Note that comments may be moved around, or even occasionally lost during prettyprinting. This is because comments are associated internally with the following identifier (not keyword), and in prettyprinting the comment is output just before the associated identifier. In general this does not present a problem for comments between declarations, but comments in the middle of expressions tend to migrate and comments after the last theory tend to disappear.[25]

Viewing TCCs

Command	*Aliases*	*Function*
`prettyprint-expanded`	`ppe`	Prettyprint expanded theory in new buffer
`show-tccs`		Show the TCCs of the specified theory *Arg:* Show only unproved TCCs

As described in the introduction, the typechecker may generate obligations called *type-correctness conditions* (TCCs), which must be discharged before the corresponding theory is considered type correct. PVS does not insist that TCCs be taken care of during typechecking; it simply stores the TCCs in the internal form of the theory, as if they were declared after the declaration which spawned them. At some point it is necessary to view and prove the TCCs, which is accomplished by means of the commands described below.

The `prettyprint-expanded` command provides a view of the entire theory, with the TCCs inserted as described above. When this command is invoked, it prompts for a theory name, and then pops up a buffer containing the expanded theory. The name of the buffer is derived from the theory name, with the extension `.ppe`. The buffer is read-only, and may not be parsed or typechecked, although proofs of any displayed TCCs or other formulas may be initiated in the usual way, simply by moving the cursor to the formula to be proved and invoking the `prove` command.

The `show-tccs` command pops up a buffer displaying just the TCCs. PVS prompts for the theory name and the name of the buffer is derived from the theory name with the extension `.tccs`; the buffer is read-only. Proofs are initiated exactly as described above.

[25]The problem of disappearing comments will probably be corrected eventually, but it is not currently one of our priorities.

The advantage to using the `prettyprint-expanded` command is that TCCs are shown in context, so it is easy to determine their derivation. On the other hand, the `show-tccs` command is faster to process and includes information about the proof status in comments associated with each TCC.

When the theory associated with either of these buffers is reparsed or retypechecked, the buffers are killed, to ensure that all displayed information is current.

Finding Files and Theories

Command	*Aliases*	*Function*
`find-pvs-file`	`ff`, `C-c C-f`	Find buffer containing named PVS file
`find-theory`	`ft`	Find buffer containing named theory
`view-prelude-file`	`vpf`	List prelude file in buffer in read-only mode
`view-prelude-theory`	`vpt`	List prelude theory in buffer in read-only mode

The `find-pvs-file` command finds or creates a buffer containing the specified file and makes it the current buffer. The file should be specified by filename only; *i.e.*, the directory and `.pvs` suffix should not be given. The `find-theory` command determines the PVS file containing the specified theory, does a `find-pvs-file` for that file, and puts the cursor at the start of the specified theory. If the theory cannot be found an appropriate error message is displayed.

PVS has a number of built-in theories which provide the primitive types, constants, and formulas of the language. These built-in theories reside in the *prelude* file. The `view-prelude-file` command displays the prelude file in a buffer in read-only mode. The `view-prelude-theory` command displays a specified prelude theory in read-only mode. Completion is supported; to find out what prelude theories are available, hit the space bar when prompted for a theory name. Prelude displays are strictly informative; although they resemble a normal PVS specification, they do not belong to the current context and therefore may not be parsed or typechecked. Proofs may be attempted as described in the `prove` command description. Prelude theories may be copied to a new buffer and modified, if their names are changed; theory names of the prelude may not be reused. There are three reasons for viewing the prelude: to find out what types, constants, and formulas are available, to see paradigmatic examples of specifications, and to try out the prover on some readily available formulas.

Creating New Files and Theories

Command	*Aliases*	*Function*
`new-pvs-file`	`nf`	Create PVS buffer containing named theory
`new-theory`	`nt`	Create named theory in current buffer

The `new-pvs-file` command prompts for a new file name, creates an associated buffer, and inserts a template for a theory with the given name. The `new-theory` command prompts for a theory name and puts the template in the current buffer, thus adding a new theory to the associated file. These commands are merely conveniences; a new PVS file may be created simply by using `find-file`, giving the new file name (with the `.pvs` extension), and typing in the theory. Similarly, a new theory may be added to a given PVS file simply by typing the theory in at an appropriate place in the file. In these cases, the theories and files are unknown to the context until they are parsed.

Importing Files and Theories

Command	*Aliases*	*Function*
`import-pvs-file`	`imf`	Import a text file as a PVS file
`import-theory`	`imt`	Import a theory into the current buffer

In the current version of PVS, all theories aside from the prelude must be available in the current context. The commands described here allow files and theories to be imported from other contexts. The `import-pvs-file` command prompts for a source file (a complete filename including directory) and a target file (a new filename without directory or extension) and copies the former to the latter, adds a `.pvs` extension, and places the file in the current context.

The `import-theory` command is similar, but prompts for a theory within the source as well as the source; the theory is copied after the current theory in the current PVS buffer. It is an error to invoke this command from any buffer other than a `.pvs` buffer.

Deleting Files and Theories

Command	*Aliases*	*Function*
`delete-pvs-file`	`df`	Delete PVS file from the context
		Arg: Delete the file from the directory
`delete-theory`	`dt`	Delete theory from PVS file

The `delete-pvs-file` command deletes a specified PVS file from the context, which means that all included theories are removed from the context, and any theories which depend on them are marked as untypechecked. Note that the file is not actually deleted, but simply removed from the context, so theory names declared in the file may be reused. To delete the file, a command argument must be supplied, in which case all of the associated proofs are copied to the orphaned proof file.

The `delete-theory` command deletes a theory from the file which contains it, removes it from the context, untypechecks any dependent theories, and copies any proofs to the orphaned proof file. Note that using standard EMACS commands to delete the theory from a PVS file and reparsing the file will have the same effect.

Saving Files

Command	*Aliases*	*Function*
`save-pvs-file`	`C-x C-s`	Save PVS file in current buffer
`save-some-pvs-files`	`ssf`	Save modified PVS files

PVS files are usually saved automatically at certain points, *e.g.*, prior to parsing, typechecking, or proving. The save commands allow the user to explicitly request the saving of files. These commands are almost identical to the EMACS commands `save-buffer` and `save-some-buffers`, except that they work only with PVS buffers.

Dumping Files

Command	*Aliases*	*Function*
`dump-pvs-files`		Write files in IMPORT chain to file
`undump-pvs-files`		Break dump file into separate PVS files

The `dump-pvs-files` and `undump-pvs-files` commands allow entire specifications and their associated proofs to be saved to, and restored from, single text files. The primary purpose of these commands is to allow complete specifications to be communicated conveniently from one place to another, e.g., by electronic mail. The `dump-pvs-files` command prompts for the name of a PVS file and a file pathname, and dumps the specification text and proofs of all theories on the import chain of the theories of the specified theories to the given file. `undump-pvs-files` prompts for a file pathname and performs the inverse process, importing all theories whose specification text is present in the named file. Both commands ask for confirmation prior to overwriting an existing file.

Printing Buffers and Regions

Command	*Aliases*	*Function*
`pvs-print-buffer`		Print buffer contents
`pvs-print-region`		Print region contents

These PVS commands are used to send buffers to the printer and replace the EMACS `lpr-buffer` and `lpr-region` commands, whose behavior they default to. This behavior can be modified by setting the `pvs-print-command`, `pvs-print-switches`, and `pvs-print-title-switches` EMACS variables. For example, to use `enscript`[26] in gaudy mode producing two column rotated output, add the following lines to your `~/.emacs` file:

[26] `enscript` is one of many print commands that provide better support and more options for postscript printers than the default `lpr` command.

```
(setq pvs-print-command "enscript")
(setq pvs-print-switches '("-G" "-2" "-r"))
(setq pvs-print-title-switches '("-b" "-J"))
```

The `pvs-print-command` must be a single print command; pipes are not allowed.[27] The `pvs-print-switches` variable contains a list of switches for the print command. The `pvs-print-title-switches` contains switches that each expect a name; the name provided to each of these switches is the name of the buffer in which the command was invoked.

Printing Files and Theories

Command	*Aliases*	*Function*
`print-theory`	`ptt`	Send theory to printer
`print-pvs-file`	`ptf`	Send PVS file to printer
`print-importchain`	`pti`	Send theories in import chain to printer

These commands send the specified theories to the printer, import the `pvs-print-buffer` and `pvs-print-region` commands. Multiple theories are concatenated into a single buffer, separated by page breaks, and then printed, thereby saving paper on systems that print a burst page with each print job.

Generating `alltt` Output

Command	*Aliases*	*Function*
`alltt-theory`	`alt`, `C-c C-a t`	Format theory for LaTeX alltt environment
`alltt-pvs-file`	`alf`, `C-c C-a f`	Format theories of file for LaTeX alltt
`alltt-importchain`	`ali`, `C-c C-a i`	Format theories in import chain for LaTeX alltt

These commands allow a specification to be inserted into a LaTeX document in an `alltt` environment. The `alltt` environment is defined in a style file included with the standard LaTeX distribution.[28] It is similar to the `verbatim` environment but allows a little more flexibility—see the `alltt.sty` file for details.

For each theory *foo* within the specified set of theories, the file *foo*`-alltt.tex` is created, which can then be inserted in a document in an `alltt` environment. The only differences between the `alltt` file and the original theory are that the braces ({ and }) are preceded by \ so they will not be interpreted by LaTeX, and tabs are replaced by spaces.

[27] To handle pipes, create a shell script somewhere in your path and set `pvs-print-command` to its name.

[28] It is also provided with PVS, in case your site has an older or nonstandard LaTeX.

Generating LaTeX Output

Command	*Aliases*	*Function*
`latex-theory`	`ltt`, `C-c C-l t`	Create LaTeX text for theory
`latex-pvs-file`	`ltf`, `C-c C-l f`	Create LaTeX text for theories of PVS file
`latex-importchain`	`lti`, `C-c C-l i`	Create LaTeX text for theories in import chain
`latex-proof`	`ltp`, `C-c C-l p`	Create LaTeX text for last proof *Arg:* make it brief
`latex-theory-view`	`ltv`, `C-c C-l v`	Create LaTeX text for theory, LaTeX and view

The first three commands generate LaTeX output for theories to be included in a document. If one of these commands is invoked in a PVS specification, a new file with the name of the theory and the `.tex` extension is generated in the current context for each specified theory.

The `latex-proof` command asks for a filename, and generates a LaTeX text file for the last proof attempted. If there was no proof attempted in the session, then the system will suggest that a proof must be rerun. The proof is written verbosely, unless an argument is given, in which case it provides a brief printout of the proof.

In addition to these files, a file named `pvs-theories.tex` is generated, which `includes` all of the files generated in the last invocation of one of these commands. The purpose of the `pvs-theories.tex` file is two-fold: to facilitate LaTeXing and printing the theories, and to illustrate the inclusion of these files in a document.

In order to LaTeX files which include the generated files, include the `pvs.sty` style file in your `documentstyle` command (see [1] for details). The `pvs.sty` file is included in the main PVS directory; if properly installed it should also be available in the main TeX inputs directory; if not, include the full pathname (see the `pvs-theories.tex` file).

While generating a LaTeX file for a theory, the system automatically makes substitutions for many of the built-in symbols; for example, `FORALL` is translated to the symbol $\forall$. This capability can be extended to user-defined symbols by means of a substitution file that specifies how to translate specified identifiers and keywords of a given specification.

To specify your own substitutions, create a new file named `pvs-tex.sub` in the current context. Each row of this file specifies an identifier, its kind and arguments, an estimated length (in `ems`), and the substitution. For example,

```
THEORY  key     9  {\large\bf Theory}
sum     id      2  {\sum_{i=0}^{\star_1} i}
sum     1       2  {\sum_{i=0}^{#1} i}
sum     (2)(1)  2  {\sum_{i=#1}^{#2} #3}
foo     [2]     1  {foo_{#1}^{#2}}
```

The identifier is an identifier or keyword in the specification; keywords must be given in upper case, and identifiers must match the case given in the specification.

The kind field may be specified as one of the symbols `id` or `key`, a number, a sequence of parenthesized numbers (*e.g.*, `(2)(1)(3)`), or a number in square brackets. The symbol `key` is used for keywords only; the others are used for translating identifiers. If the kind is simply `id`, substitution is done for occurrences of the identifier that have no arguments or actual parameters provided. If the kind is a number, the substitution is performed when the identifier appears with the specified number of arguments. A kind field specified as a sequence of parenthesized numbers corresponds to a curried function. For example, the fourth entry in the above set of substitutions would be used for an occurrence of `sum(a + 1, b)(n)` in the specification. When the number is in square brackets, the translation is used whenever the name appears with that number of actual parameters. With the substitutions given above, an occurrence of `foo[int,3]` would be translated to $\mathrm{foo}^{3}_{\mathrm{int}}$.

The length field specifies the expected size of the substitution, excluding arguments. The size is given in `ems`; an `em` is roughly the size of an `m` in the current font. This number does not have to be accurate, it is used by the underlying prettyprinting routines to determine the placement of line breaks.

The final field gives the substitution. The arguments, if any, are substituted for `#1`, `#2`, etc. in the order given. For example, in the expression "`sum(a + 1, b)(n)`," `a + 1` would be substituted for `#1`, `b` for `#2`, and `n` for `#3`.

The substitution file overrides substitutions provided in the default substitution file located in the PVS directory. In addition, a `pvs-tex.sub` file in the user's home directory overrides the default, but does not override substitutions specified in the current context. Finally, a substitution file for a specific theory may be provided; if the theory name is `foo`, then the substitution file `foo.sub` overrides all of the above when the theory `foo` is being processed.

Context Commands

Command	*Aliases*	*Function*
`list-pvs-files`	`lf`	Display a list of PVS files in current context
`list-theories`	`lt`	Display a list of theories in current context
`change-context`	`cc`	Switch to a new context
`save-context`	`sc`	Save the current context
`context-path`	`cp`	Display pathname of current context

The `list-pvs-files` and `list-theories` commands prompt for a directory, default is to the current directory; if there is a PVS context in the given directory, these commands list the PVS files or theories in that context. The resulting buffer is in a special mode, which allows the file/theory to be viewed (by typing a "`v`"), selected (by typing a "`s`") or imported (by typing an "`i`"). A file or theory may only be selected if it is in the current context, and may only be imported if it is not. Importing a theory from the list of theories will import the associated file.

The `change-context` command is similar to the "`cd`" command in UNIX; it saves the context (see below), and changes the working directory to the specified one. The PVS Welcome buffer is then displayed indicating the new directory.

The `save-context` command saves the current state of the session in the context file `.pvscontext`. In addition, all proofs are saved in proof (`.prf`) files. This command is called automatically during parsing.

The `context-path` command uses the minibuffer to display the directory path associated with the current context.

Browsing

Command	*Aliases*	*Function*
`show-declaration`	`M-.`	Show declaration of symbol at cursor
`find-declaration`	`M-,`	Search for declarations of given symbol
`whereis-declaration-used`	`M-;`	Search for declarations which reference symbol
`list-declarations`	`M-:`	Produce list of declarations in import chain

These commands browse a specification consisting of several PVS files and theories, providing information about where entities are declared and used. All of these commands browse the prelude as well as user files.

The `show-declaration` command is used to determine the declaration associated with a name. Positioning the cursor on a name in the specification and typing `M-.` yields a popup buffer displaying the declaration. This command is useful to determine the type of a name, or the resolution determined by the typechecker for an overloaded name.

The `find-declaration` takes a name and returns a list of all the declarations with that name, the default name is the one under the cursor. Each row in the display specifies the declaration name, its kind/type, and the theory to which it belongs. Declarations in this list may be viewed by placing the cursor on the row of interest and typing "`v`." Typing "`s`" will read in the associated file and position the cursor at the declaration. A "`q`" quits and removes the declaration buffer.

The `whereis-declaration-used` command generates a list of declarations which reference the specified name. The `list-declarations` command generates a listing of all the declarations in the import chain of the specified theory. For both of these commands, the resulting buffer behaves exactly as described above.

Theory Status

Command	*Aliases*	*Function*
`status-theory`	`stt`, `C-c C-s t`	Status of specified theory (parsed, etc.)
`status-pvs-file`	`stf`, `C-c C-s f`	Status of theories of current file
`status-importchain`	`sti`, `C-c C-s i`	Status of theories in import chain of theory
`importchain`	`ic`	Display the import chain of theory
`importbychain`	`ibc`	Display the import by chain of theory

These commands provide information regarding the status of the specified theories. The status information for a theory indicates whether it is in the context, whether it is parsed or typechecked, and a breakdown of the formulas whose proofs have been attempted and proved.

The `status-theory` command provides the status as a single line in the minibuffer. The `status-pvs-file` and `status-importchain` commands provide a new buffer with a line for each theory, unless the specified file or theory is not in the current context, which is noted in the minibuffer.

The `importchain` and `importbychain` commands display the IMPORT chains of the specified theory, indented to indicate the tree structure. The `importchain` command works recursively down the `IMPORT`s, displaying the status of each theory unless it has been displayed earlier in the buffer. The `importbychain` works in the opposite direction.

Proof Status

Command	*Aliases*	*Function*
`status-proof`	`sp`, `C-c C-s p`	Status of formula at cursor
`status-proof-theory`	`spt`	Status of all formulas in theory
`status-proof-importchain`	`spi`	Status of all formulas on import chain
`status-proofchain`	`spc`	Analyze proof chain of formula at cursor

These commands provide the status of the proofs of the indicated formulas. The `status-proof` command uses the minibuffer to display the proof status of the formula at the cursor. The status can be one of `proved`, `unproved`, `unfinished`, or `unchecked`. Unfinished means that the proof has been attempted, but is not complete. Unchecked means that the proof was successful at one point, but that some changes have been made that may invalidate the proof.

The `status-proof-theory` and `status-proof-importchain` commands use the `PVS Status` buffer to display the proof status for all of the formulas within the theory or the import chain.

The `status-proofchain` command provides a proof chain analysis of the formula at the cursor and displays it in the `PVS Status` buffer. The proof chain analysis indicates whether the formula has been proved, and analyses the formulas used in the proof to insure that the proof is complete; lemmas used in the proof are proved, and sound, *i.e.*, there are no circularities (*e.g.*, using lemma $\mathcal{A}$ to prove $\mathcal{B}$ and vice-versa).

Environment Commands

Command	*Aliases*	*Function*
`help-pvs`	`C-h p`	Display the PVS help buffer
`help-pvs-language`		Display help for the PVS language
`help-pvs-prover`		Display help for the prover commands
`whereis-pvs`		Display the root PVS directory
`pvs-version`		Display current version of PVS and underlying LISP
`pvs-mode`		Put current buffer in PVS mode
`remove-popup-buffer`	`C-z 1`	Remove the popup buffer

The `pvs-help` command displays a summary of PVS commands in the `PVS Help` buffer. Help may be obtained for an individual command by typing `C-h f` followed by the command or its abbreviation, or by typing `C-h k` followed by the key sequence that invokes the command.

The `help-prover` command provides a summary of all of the prover commands in the `Prover Help` buffer.

The `whereis-pvs` command is used to determine the directory where the PVS system resides along with example specifications and files which may be copied and modified to customize the PVS environment.

The `pvs-version` command displays the current version of PVS.

The `pvs-mode` command puts the current buffer in PVS mode. This command is not normally needed; buffers with a `.pvs` extension and buffers created by PVS are automatically put in the proper mode.

Popup buffers are used in PVS to display temporary information, such as PVS error messages, possible completions, and help messages. The `remove-popup-buffer` command removes any visible popup buffer. Most commands automatically remove the popup buffers. Popup buffers can be recognized by the occurrence of "`^Z1 bury`" in the status line of the buffer.

Interrupting PVS

Command	*Aliases*	*Function*
`pvs-status`		Find out if Lisp is busy
`keyboard-quit`	`C-g`	Abort current EMACS command
`reset-pvs`	`C-z C-g`	Abort PVS and resynchronize

Many PVS commands run in the background, allowing other editing activities to proceed concurrently. The effect of issuing new commands while another command is running depends on the command: another background command is placed on the command queue. Other (nonbackground commands) interrupt the currently running command, execute, and return control to the interrupted command. The EMACS status line indicates the abbreviation of the command that is currently running, if any, or `ready`. The `pvs-status` command provides information about both the currently running command and the command queue.

To interrupt PVS for any reason, the following procedure is recommended. First, if the keyboard is not responding, type the built-in EMACS command `keyboard-quit` (`C-g`); it may need to be struck a few times before there is any response. This command interrupts EMACS, but has no effect on any PVS commands that are still running.

The `reset-pvs` command aborts any ongoing activity in PVS; its effects depend on whether it is issued from the `*pvs*` buffer or from some other buffer. In the former case, `reset-pvs` simply interrupts PVS, invoking the LISP debugger. The options here are to abort to the top level (by typing `:reset` in Allegro, and `:a` in Lucid), or, in the prover, to type `(restore)` which restores the proof to the state prior to the last *atomic* prover command (see [3]). If `reset-pvs` is issued somewhere other than the `*pvs*` buffer, the user is asked whether to reset PVS in case the command was typed accidentally; if not, the current command is aborted and the command queue is emptied.

A Obtaining and Installing the System

To install the PVS system, you must have a Sun 4 (sparc) system running the SunOS 4.1 operating system (or later). At least 12 Mbytes of real memory and 50 Mbytes of swap space are needed. For good performance, we recommend at least 24 Mbytes of real memory and a Sparc2 or better.

PVS is written in Common Lisp, and has been ported to Lucid, Allegro, AKCL, and CMULisp.[29] It uses the Common Lisp Object System (CLOS), and hence runs slowly in the latter two systems (where CLOS has not been fully integrated). The Lucid version is the fastest, but requires a Lucid license. AKCL also requires a license, but in this case the license is free.

PVS uses Gnu Emacs as its interface. Most sites already have this installed; if not, Gnu Emacs is readily available by FTP from a number of sites (such as `prep.ai.mit.edu`) or from the Free Software Foundation. PVS was developed using Emacs version 18.57; earlier versions generally work as long as the same basic functions are provided. To make use of the LaTeX and `alltt` output commands, LaTeX must also be available. A LaTeX previewer must be available for the `latex-theory-view` command.

To obtain the PVS system, send a request to `pvs-request@csl.sri.com`, and we will provide further instructions for obtaining a tape; there is also a distribution fee. If you want to avoid the distribution fee, and have access to the internet, you can obtain the Allegro Lisp version by anonymous ftp as follows. First carry out steps (1) and (2) below. Next ftp to `ftp.csl.sri.com`, login `anonymous`, password your e-mail address. Then type "`binary`", followed by "`cd pub/pvs`" and "`get pvs-01.tar.Z`". You are now ready to install PVS.

1. Create a new directory to serve as the root for the new PVS system and `cd` to it.

2. Make certain that at least 40 Mbytes of space are available on the disk by typing "`df .`"; this command displays the amount of free disk space available (the "`.`" indicates the current directory).

3. If you have a tape, mount it on an available tape drive and type "`tar xvf /dev/rst0`". You may need to login to the machine connected to the tape drive; if so, make sure you `cd` to the right directory on that machine.

4. If you used ftp to obtain PVS, simply move the `pvs-01.tar.Z` to the directory, `uncompress` `pvs-01.tar.Z` and run "`tar xf pvs-01.tar`" to extract the system. If you need the space, you may now remove `pvs-01.tar`.

[29] CMULisp is only supported on sparcs.

5. Edit the file `pvs` that appears in the directory, setting `PVSPATH` to the full pathname of the directory.

6. Copy this file to a system directory to allow general access to the PVS system.

7. If you obtained the system by anonymous ftp, or by some other means that did not require you to sign the PVS License Agreement beforehand, print out the file `Doc/license.ps` (or `Doc/license.dvi`), read it, have it signed by someone who can legally bind your institution, and return it to

   Judith Burgess, Librarian
   Computer Science Laboratory
   SRI International
   333 Ravenswood Avenue
   Menlo Park, CA 94025, USA
   or fax to (+1) 415/859-2844

   For questions, email `Burgess@csl.sri.com`, or call (+1) 415/859-5924.

   You may not use PVS until you have signed and returned the PVS License Agreement to SRI. Basically, this agreement (a) preserves SRI's copyright in PVS, (b) allows you to further distribute PVS, subject to further propagating its licensing requirements, and providing you do not charge for it (other than distribution costs, and those must be less than those we charge), (c) recognizes that PVS is an experimental system, and is therefore provided "as is" and with absolutely no warranty of any kind, and (d) requests that you notify us periodically of the use you are making of PVS, send us copies of your publications that reference PVS, and send us bug reports (though we are not obligated to fix bugs or provide maintenance).

8. If you plan to make use of the LaTeX facilities of PVS, it would be a good idea to copy the `.sty` and `.tex` files to the TeX inputs directory; these are general-purpose files, and can be used outside of PVS. You can obtain LaTeX and TeX by anonymous ftp from labrea.stanford.edu.

To start the system, `cd` to a directory in which you have write permission and invoke the newly installed `pvs` command. PVS will automatically bring up a new Emacs window if started under X windows or SunView. If you encounter difficulties, you can force the system to run in the current window by using the `-notool` switch. In this case, you should probably also use the `-q` switch, which inhibits automatic loading of your `.emacs` file.

If you have a postscript printer, you can print the PVS documentation using the `.ps` files (they must first be `uncompressed`); otherwise you will need a LaTeX `dvi-to-yourprinter` program. The documentation files can be found in the /Doc subdirectory. The documentation includes a reference card, tutorial, user guide, language manual, and prover manual. Illustrative examples are provided in the /Examples directory. Several good examples also appear in the prelude file /lib/prelude.pvs.

If you have questions or suggestions, send email to `pvs-bugs@csl.sri.com`. To get on the PVS mailing list, send a message to `pvs-request@csl.sri.com`.

References

[1] Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley Publishing Company, Reading, MA, 1985.

[2] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Draft).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[3] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual (Draft).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[4] Richard Stallman. *GNU Emacs Manual.* Free Software Foundation, 675 Massachusetts Ave., Cambridge, MA, 6th edition, March 1987.

Index