PVS System Guide

Version 3.2 • September 2004

S. Owre
N. Shankar
J. M. Rushby
D. W. J. Stringer-Calvert
{Owre,Shankar,Rushby,Dave_SC}@csl.sri.com
http://pvs.csl.sri.com/

SRI International
Computer Science Laboratory • 333 Ravenswood Avenue • Menlo Park CA 94025

The initial development of PVS was funded by SRI International. Subsequent enhancements were partially funded by SRI and by NASA Contracts NAS1-18969 and NAS1-20334, NRL Contract N00014-96-C-2106, NSF Grants CCR-9300044, CCR-9509931, and CCR-9712383, AFOSR contract F49620-95-C0044, and DARPA Orders E276, A721, D431, D855, and E301.

Contents

Chapter 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 manual describes the system, including the system commands, the computing environment, how to get and install PVS, customization, and a short tutorial on Emacs. The complete set of manuals for the PVS system consists of this manual, the language reference [3], and the prover guide [7]. There are also several supporting technical reports: the formal semantics of PVS [5], an advanced tutorial [6], and a description of the abstract datatypes mechanism [4]. All of these manuals (and much more!) are available online at http://pvs.csl.sri.com/

The rest of this chapter provides a broad overview of PVS; the facilities provided by the system are discussed in the order you are likely to encounter them.

The PVS Environment

PVS runs on Sun 4 (Sparc) workstations using Solaris 2 or higher and PC systems running Redhat Linux. PVS is implemented in Common Lisp, but it is not necessary to know Lisp to effectively use the system.[1] PVS runs best using the X window system, though it is not required. The Emacs (Gnu Emacs or XEmacs) editors provide the interface to PVS; familiarity with Emacs and access to the GNU Emacs manual [8] (usually available as an info file) are desirable. A brief introduction to Emacs is provided in Appendix A on page 79 of this manual. The LaTeX generating facilities require a good understanding of the LaTeX document preparation system [1]. If you have Tcl/Tk available, there are PVS interfaces provided that display proof trees, theory hierarchies, and proof commands. Instructions for obtaining and installing the PVS system as well as Emacs, X windows, LaTeX, and Tcl/Tk may be found at http://pvs.csl.sri.com.

[1] The only exception to this is in writing complex prover strategies.

The PVS 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 rich set of built-in types and type-constructors, as well as a powerful notion of subtype. Specifications can be constructed using definitions or axioms, or a mixture of the two.

Specifications are logically organized into parameterized *theories* and *datatypes*. Theories are linked by *import* and *export* lists. Specifications for many foundational and 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 the PVS language reference [3].

Specification Files and the PVS Context

PVS specifications are ordinary ASCII text files prepared and modified using a text editor—usually the Emacs editor that acts as the interface to PVS. A PVS specification consists of any number of such files, each of which contains one or more theories or datatypes. PVS specification files have the `.pvs` extension.

Each specification file has associated with it a proof file (with the `.prf` extension) that saves the proof scripts generated during proof attempts on formulas contained in the associated PVS specification file. In addition, the system generates binary representations of the typechecked specification files (with the `.bin` extension) that speed up retypechecking when a PVS session is resumed in the same context.

The set of files and theories constituting a specification, together with various items of status information, comprise a PVS *context*. The PVS context retains information about the state of a specification and verification from one PVS session to the next. This information is primarily kept in the `.pvscontext` file that is associated with each PVS context. It keeps track of which formulas have been proved, and which binary files are valid by keeping track of the write dates associated with the various files.

PVS contexts are closely related to directories, and the term *context* is used in this document to refer to either the PVS context or its 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 that 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, so for most purposes a PVS context and its containing directory can be treated synonymously.

PVS Libraries

PVS has a library facility that allows files and theories from one PVS context to be used in another, thus allowing for general reuse, and making it easier to standardize theories that are frequently used. There are two ways that the library facility can be used: by explicitly importing a theory from a different PVS context within a specification, or by issuing a command that effectively extends the prelude.

The PVS User Interface

You interact with the PVS system through a customized Emacs. It is expected, though not required, that editing of specifications is performed with this editor. Using other editors is quite painful, as they cannot directly interact with the underlying Lisp image.

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 you through various display mechanisms provided by Emacs or Tcl/Tk.

The PVS interface allows a certain amount of parallel activity. For example, you 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, you 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. You 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 generally 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. There are commands available that attempt to prove the TCCs using built-in prover strategies. You may choose when to prove the TCCs, but until they are proved the theory that generated them is not considered to be typechecked.

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 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 will need to be typechecked in order to guarantee consistency. This happens automatically as the need arises.

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 (Section 3.5.6, page 35).

There is some incremental typechecking that goes on at the theory level. When a typecheck command is issued on a PVS file that has been modified, the file is first parsed, and the resulting abstract syntax is compared to the previous abstract syntax. If they are the same, the theory is not retypechecked. Otherwise it typechecks as usual. Comments, added or deleted whitespace, and certain kinds of expression transformations (such as changing `a + 1` to `+(a,1)`) will thus not trigger retypechecking.

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. This manual describes how to enter the prover and some of the commands for obtaining and editing proof information. Details on the proof checker commands may be found in the prover guide.

Status and Proof Chain Analysis

The PVS system provides several commands for determining the status of specification elements such as theories and formulas. You can, for example, inquire whether a theory has been typechecked or whether a specific formula has been proved.

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; 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 subscripts 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. The typeset specifications are also of value during the development of a formal specification and verification, as they allow direct comparison with existing, informal presentations and analyses.

Display Commands

There are a few commands available for displaying graphical information using an interface to the Tcl/Tk system. These include the display of proof trees, theory hierarchies, and prover commands. These displays are interactive; for example the

proof tree display is updated as a proof is developed, and clicking on a theory in the theory hierarchy display pops up an Emacs buffer containing that theory specification.

Other Commands

There are other miscellaneous commands are not easily categorized, such as commands for sending bug reports, interrupting PVS, getting help, and some commands that help in editing PVS files.

Chapter 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 normal usage of PVS. You will get the most out of this section if you are sitting in front of a terminal with PVS installed.[1]. In the following we assume some familiarity with UNIX and Emacs. If you are unfamiliar with Emacs you may want to look at the introduction in Appendix A on page 79.

Start by going to a UNIX shell and creating a working directory (using `mkdir`). Next, change (`cd`) to this working directory and start up PVS by typing `pvs`.[2] This command executes a shell script which runs Emacs, loads the necessary PVS Emacs extensions, and starts the PVS lisp image as a subprocess. See Chapter 4 on page 63 for further details on the `pvs` command and its parameters. 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. When you are ready to exit PVS, type the key sequence `C-x C-c`.

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 holding down the `Meta` key,[3] then pressing `x`. Release the `Meta` key, then type `prove` (or `pr`) and press 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

[1] If you don't have it installed, see the instructions at http://pvs.csl.sri.com

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

[3] Most keyboards provide a `Meta` key (hence the `M-` prefix). On the SUN4, this key is labeled ◇; IBM style keyboards tend to use the `Alt` key. The `Meta` key is like the shift key—to use it simply hold the `Meta` key down while typing another key. If your keyboard does not have a `Meta` key, you can press the `Escape` key for the same effect. Note that the `Escape` key does not act as a shift, but is pressed and released before the command, e.g. `Escape` followed by `x` followed by `pr`.

need to be pressed when giving the key binding form. In PVS all commands and abbreviations are invoked by first typing 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 an argument and specify a default; if the default is the desired one, you can simply type the `Return` key.

To begin, type `M-x pvs-help` (`C-c h`) for an overview of the commands available in PVS, and use `C-v` and `M-v` to browse the help file and get a feel for the commands provided by PVS. Type `q` to exit the help buffer. If you are running Emacs under X windows, you should see a menu bar across the top of the window, including a `PVS` entry. If you move the mouse cursor over this entry, and press the left mouse button, a menu will be displayed that also shows all the PVS commands (including the help commands). This menu may also be used to invoke the commands, though most users prefer to learn the keyboard commands as this is generally faster. When discussing the PVS commands we will not mention the PVS menu, but you should be aware that all of the PVS Emacs commands are available as menu entries.

2.1 Creating the Specification

Now let's develop a small specification. Figure 2.1 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 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 is explained below. The final declaration is a formula which gives the closed form of the sum.

The `sum` theory may be introduced to the system in a number of ways, all of which create a file with a `.pvs` extension.[4] The most common ways are:

1. Simply use `M-x find-file` (`C-x C-f`), or `M-x find-pvs-file` (`M-x ff`, `C-c C-f`), provide `sum.pvs` for the file name and type in the specification.[5]

2. Use the `M-x new-pvs-file` command (`M-x nf`) to create a new PVS file, and type `sum` when prompted for a file name. Then simply type the specification into the buffer (a basic template will be provided).

3. Since the file is included in the distribution in the `Examples` subdirectory of the main PVS directory, it can be imported with the `M-x import-pvs-file`

[4] The file does not have to be named `sum.pvs`, it simply needs the `.pvs` extension.
[5] If there is already a file called `sum.pvs` in the current context, this will load that file.

```
sed_form: THEOREM sum(n) = n * (n + 1)/2

sum
```

```
 sum.pvs        14:06 0.02   (PVS :ready)--L10--All----------------
d_form.2 :

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

 (postpone)
oning closed_form.2.

d_form.1 :

-----
 sum(0) = 0 * (0 + 1) / 2

 (expand "sum")
ding the definition of sum,
simplifies to:
d_form.1 :

-----
 0 = 0 / 2

 (assert)
ifying, rewriting, and recording with decision procedures,
```

Figure 2.1: The `sum` Specification in Emacs

command (`M-x imf`). Use the `M-x whereis-pvs` command to find the path of the main PVS directory.

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

2.2 Parsing and Typechecking

Once the `sum` specification is displayed in the current buffer, it can be parsed with the `M-x parse` (`M-x pa`) command, which checks the syntactic consistency of the specification and 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.

The next step is to typecheck the file by typing `M-x typecheck` (`M-x tc`, `C-c C-t`), which checks for semantic errors, such as undeclared names and ambiguous types. After `sum` has been typechecked, a message is displayed in the minibuffer indicating that two 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 in general it is a good idea to view TCCs to convince yourself that they are provable before moving on to other proofs in your specification. TCCs can be viewed using the `M-x show-tccs` (`M-x tccs`, `C-c C-q s`) command, the results of which are shown in Figure 2.2 below.

```
% Subtype TCC generated (at line 7, column 32) for  n - 1
    % expected type  nat
  % unfinished
sum_TCC1: OBLIGATION
  FORALL (n: nat): NOT n = 0 IMPLIES n - 1 >= 0;

% Termination TCC generated (at line 7, column 28) for  sum(n - 1)
  % unfinished
sum_TCC2: OBLIGATION
  FORALL (n: nat): NOT n = 0 IMPLIES id[int](n - 1) < id[int](n);
```

Figure 2.2: 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.

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.)

2.3 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 prover is ready to accept input. The commands needed to prove this theorem constitute only a very small subset of the commands available to the prover. In fact, for this proof all that is actually needed is the single command `(induct-and-simplify "n")`, which is a more powerful strategy. For more information on these and other prover commands consult the prover guide [7].

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. An empty antecedent is equivalent to `true`, and an empty consequent is equivalent to `false`, so if both are empty the sequent is `false`. Every proof in PVS starts with 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, called the *current* sequent. 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 on to the proof. We will prove this formula by induction on `n`. To do this, type `(induct "n")`.[6] This is not an Emacs command, rather it is typed directly at the prompt, including the parentheses. As indicated, two subgoals are generated; the one displayed is the base case, where `n` is `0`. To see the inductive step, type

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

`(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`).

Three extremely useful Emacs key bindings 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 bindings out; they are easier to use than to explain. Thus to type the second postpone command above, you can either type `M-p` or type `(po` followed by `M-s`. Section 3.5.7 on page 36 describes further useful shortcut commands for the prover.

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`, we issue the `(assert)` command, which applies the decision procedures of the prover to simplify the consequent to `TRUE`, completing the proof of this subgoal. The prover then automatically 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[7], 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.[8]

The final step is to invoke the PVS decision procedures, which can automatically decide certain fragments of arithmetic. This is done by typing `(assert)`. 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 by rewriting (if any auto-rewrites have been given) and by using the underlying decision procedures.

[7] The exclamation point differentiates this command from the `skolem` command, where you provide the new constant names.

[8] 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 [7].

These arithmetic and equality procedures are the main workhorses of most PVS proofs.

The proof is now complete, and is saved in the `sum.prf` file. The buffer from which the `prove` command was issued is then redisplayed if necessary, and the cursor is placed on the formula that was just proved. The entire proof transcript is shown below. Yours may be slightly 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 on formula 1,
this yields  2 subgoals:
closed_form.1 :

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

Rule? (postpone)
Postponing 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? (postpone)
Postponing closed_form.2.

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}   0 = 0 / 2

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

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!)
Skolemizing,
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}   1 + sum(j!1) + j!1 = (2 + j!1 + (j!1 * j!1 + 2 * j!1)) / 2

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of closed_form.2.

Q.E.D.


Run time  = 0.81 secs.
Real time = 223.01 secs.
```

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

2.4 Status

Now type `M-x status-proof-theory` (`M-x spt`) and you will see a buffer that displays the three 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 that the proof is complete, and that it depends on the TCCs and the `nat_induction` axiom, as well as some definitions and TCCs provided by the prelude.

2.5 Generating LaTeX

In order to try out this section, you must have access to LaTeX and a TeX previewer such as `xdvi`.

Type `M-x latex-theory-view` (`M-x ltv`). You will be prompted for the theory name to which you should 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 which PVS assumes that you want to use the same previewer. You can set the previewer automatically, by adding the following line to your `~/.pvsemacs` file.

```
(setq pvs-latex-viewer "previewer")
```

sum: THEORY
BEGIN

n: VAR $\mathbb{N}$

sum(n): RECURSIVE $\mathbb{N} = (\text{IF } n = 0 \text{ THEN } 0 \text{ ELSE } n + \text{sum}(n-1) \text{ ENDIF})$
MEASURE id

closed_form: THEOREM $\text{sum}(n) = \frac{(n \times (n+1))}{2}$

END sum

Figure 2.3: Theory `sum` with default translations

After a few moments the previewer will pop up displaying the `sum` theory, as shown in Figure 2.3. Note that `*` has been translated as $\times$ and `LAMBDA` as λ. These and other translations are built into PVS; you may also specify translations for keywords and identifiers by providing a substitution file named `pvs-tex.sub`, that contains commands to customize the LaTeX output. For example, if the substitution file contains the two lines

```
THEORY key 7 {\large\textbf{\textrm{Theory}}}
sum    1   2 {\sum_{i = 0}^{#1} i}
```

the output will look like Figure 2.4. See Section 3.9.4 on page 47 for more details.

sum: Theory
BEGIN

n: VAR $\mathbb{N}$

$\sum_{i=0}^{n} i$: RECURSIVE $\mathbb{N} = (\text{IF } n = 0 \text{ THEN } 0 \text{ ELSE } n + \sum_{i=0}^{n-1} i \text{ ENDIF})$
MEASURE id

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

END sum

Figure 2.4: Theory `sum` with additional translations

Chapter 3

PVS Commands

This chapter contains descriptions for all PVS commands; the commands are grouped according to function. A summary of the information in this chapter is also provided in the buffer displayed by the `M-x pvs-help` command. 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 C-t`. The behavior of many of the commands can be modified by providing an argument, and many of the commands work on regions.[1] 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`.

[1] See Section 4.9 of [8] for details on providing arguments to commands, and Section 9 for creating and manipulating regions.

3.1 Exiting PVS

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

The `exit-pvs` command first saves the context information (see the `save-context` command) and then exits PVS. If there is a proof in progress, the system will not exit, but will instead output a message asking you to exit the prover, thus giving you the opportunity to save the proof before exiting.

The `suspend-pvs` command suspends the Emacs process, except under X-windows, where the command has no effect. The system first asks whether the context should be saved; if you answer `yes` the `save-context` command is invoked prior to suspending PVS. This may take a while, as the `save-context` may have to save any number of files, depending on what has changed in the context. 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.[2]

3.2 Getting Help

Command	*Aliases*	*Function*
`help-pvs`, `pvs-help`	`C-c h`	Display the PVS help buffer
`help-pvs-bnf`, `pvs-help-bnf`	`C-c C-h b`	Display the pvs grammar
`help-pvs-language`, `pvs-help-language`	`C-c C-h l`	Display help for the PVS language
`help-pvs-prover`, `pvs-help-prover`	`C-c C-h p`	Display help for the prover commands
`help-pvs-prover-command`, `pvs-help-prover-command`	`C-c C-h c`	Display help for prover command
`help-pvs-prover-strategy`, `pvs-help-prover-strategy`	`C-c C-h s`	Displays the specified prover strategy
`x-prover-commands`		Displays the prover commands in a Tcl/Tk window
`help-pvs-prover-emacs`, `pvs-help-prover-emacs`	`C-c C-h e`	Display help for prover emacs commands
`pvs-release-notes`,	`C-c C-h r`	Display PVS release notes

The `help-pvs` 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. These are built in to Emacs, and may be used to get help for any Emacs command or key sequence, not just PVS commands.

[2] This assumes you are running the `csh` or `tcsh` shell. To restart under a shell lacking job control, use the UNIX command `ps` to determine the process id (*pid*) and then do `kill -CONT pid`.

The `help-pvs-bnf` command provides the PVS grammar in BNF form, and the `help-pvs-language` command displays a summary of the PVS language with examples in the `Language Help` buffer.

The `help-pvs-prover` command displays the documentation string for all of the prover commands in the `Prover Help` buffer. The `help-pvs-prover-command` displays the documentation string for the specified command, and the `help-pvs-prover-strategy` command provides the arguments, definition, format string, and documentation string for the specified command. The latter is useful for finding out exactly what a strategy does, or for defining your own strategies based on existing ones. If you are running under the X window system, `x-prover-commands` provides an easy interface to get help for individual prover commands.

The `help-pvs-prover-emacs` command displays a summary of the commands that provide a convenient Emacs interface to the PVS prover. This is discussed in more detail in Section 3.5.7, page 36. The help text appears in the `Prover Emacs Help` buffer.

The `pvs-release-notes` command displays the release notes for the running version of PVS. The text appears in the `PVS Release Notes` buffer.

3.3 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

PVS specification files are edited using the standard Emacs editing commands. Appendix A, page 79 gives a brief introduction to the most useful Emacs commands for editing PVS files.

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`" or "`No earlier 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.

3.4 Parsing and Typechecking

3.4.1 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. Note that only files (with extension `.pvs`) may be parsed. When a file is parsed, it becomes a part of the context if it wasn't already, and any proofs that have been saved for the file are reinstated. If the file being parsed has a valid `.bin` file, then this file is loaded instead (this will result in the file being typechecked as well as parsed).

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. Note that the argument is usually not needed, as changes to the file are automatically detected by the system and the file is reparsed in that case.

When an error is detected, the file is displayed with the cursor at the location where the error was detected, which is frequently after the actual source of the error. In addition, the `PVS Error` buffer is displayed with an explanatory error message. You may need to consult the language manual for details on the grammar.

Certain language features may result in the parser producing theory messages. See the `show-theory-messages` command (page 22) for details.

3.4.2 Typechecking

Command	*Aliases*	*Function*
`typecheck`	`tc`, `C-c C-t`	Typecheck theories in current buffer *Arg:* Force reparsing and retypechecking
`typecheck-importchain`	`tci`	Typecheck importchain of theories *Arg:* Force reparsing and retypechecking
`typecheck-prove`	`tcp`	Typecheck theories, proving TCCs *Arg:* Force reparsing and retypechecking
`typecheck-prove-importchain`	`tcpi`	Typecheck importchain of theories, proving TCCs *Arg:* Force reparsing and retypechecking

Typechecking a PVS specification checks semantic constraints, determines the types of expressions, and resolves names (see the language manual [3]). 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. Without the argument, `typecheck` and `typecheck-importchain` are the same. With the argument, `typecheck` only reparses and retypechecks the current file, while `typecheck-importchain` forces reparsing and retypechecking of the entire import chain of the theories of the current file.

Forcing a file to be retypechecked is done primarily for development and debugging, as is the case for reparsing. If you have typechecked a set of PVS files, made some changes and found an error on retypechecking that shouldn't have occurred, try forcing a typecheck of the file where the error occurred. If that doesn't help, try forcing with `typecheck-importchain`. The error should disappear after that, unless it is a true typecheck error. If it is not a simple typecheck error, send a bug report to `pvs-bugs@csl.sri.com`.

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 theory `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, and then attempts to prove the generated TCCs. 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. The TCC proof attempts are made with built-in prover strategies (selected according to the type of TCC generated). These strategies basically expand all definitions in the TCC, and repeatedly skolemize, perform heuristic instantiation, lift `IF`s, and invoke the decision procedures.[3] As explained in the prover guide, you may redefine the `tcc` strategies; usually to extend their capabilities.

The `typecheck-prove-importchain` command typechecks the file, and attempts to prove the TCCs of all the theories on the import chain that have not already been attempted. Providing an argument forces the retypechecking of the import chain.

The `typecheck-prove` commands can take some time, especially if there are a lot of TCCs. This can be controlled in a number of ways:

[3]The TCC strategies are variants of a powerful strategy called `(grind)`, which is useful for more than just TCCs.

Use these commands sparingly. Our experience is that TCCs should be analyzed whenever a new specification is created, significantly modified, or is nearing completion. At these times it pays to use the `typecheck-prove` command and to look at the TCCs that weren't subsequently proved, and check that they at least seem provable. After minor changes, we find it best to use just `typecheck` and defer consideration of the TCCs until later.

Define your own TCC strategy. The prover guide describes techniques for defining your own strategies, and you may change existing ones, such as the `tcc` strategy to be more efficient for your particular specifications. Changing the `tcc` strategy should probably be done in the `pvs-strategies` file in the current context, especially if it is tailored to the specifications in that context.

Use judgements to cut down on the number of TCCs. The language manual describes how to do this.

Use `NONEMPTY_TYPE` or `CONTAINING` in type declarations This is also described in the language manual.

When typechecking is completed, a message is displayed, indicating the total number of TCCs generated along with a breakdown of the number proved, subsumed[4], and unproved.

3.4.3 Typechecking Information

Command	*Aliases*	*Function*
`show-theory-warnings`		Show typechecker warnings for the given theory
`show-pvs-file-warnings`		Show typechecker warnings for the given file
`show-theory-messages`		Show typechecker messages for the given theory
`show-pvs-file-messages`		Show typechecker messagess for the given file
`show-theory-conversions`		Show conversions for the given theory
`show-pvs-file-conversions`		Show conversions for the given file

In the process of typechecking a specification, various warnings and informative messages may be produced. These are associated with the theory that produced them, and saved so they may be perused. Warnings may indicate a possible problem. For example, if the typechecker cannot determine that a datatype is nonempty, it produces a warning. There is nothing wrong with having an empty datatype, but if at some point it isn't proved to be nonempty, a lot of time may be wasted proving formulas that are vacuously true. Informative messages do not indicate anything is wrong, but the information may be of interest. For example, using the `TYPE+` keyword generates an existence axiom, and this is treated as an informative message.

[4] 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.

Conversion messages[5] have been separated out of the warnings. Conversions may be applied to make an expression type correct. This is not always what the user intended, so the show conversions commands are provided to make it easy to look at the conversions that have been applied.

Note that typechecking not only reports the number of TCCs generated, but also the generation of any warnings, messages, and conversions. While in the prover, these messages are generated interactively.

3.5 Proving

The prover is described in full in the prover guide [7], here we describe the Emacs interface to the prover, including commands for invoking the prover, editing and rerunning proofs, displaying proof information, some useful keyboard shortcuts for the prover, and managing multiple proofs.

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`, `x-prove`, `step-proof` and `x-step-proof` commands lead to prover interaction; the other commands simply rerun proof scripts that have been previously generated.

PVS keeps track of the status of formulas within and across sessions. The status may be one of four values; "untried" means that no proof has been attempted, "proved" means that the proof has been completed, "unchecked" means that a proof has been completed, but that the specification has been modified since the proof attempt, and "unfinished" means that a proof has been attempted, but not yet completed. Formulas labelled as "proved" will be "complete" or "incomplete". The status is only "complete" when all formulas (including TCCs) upon which the proof is dependent have been completed.

Modifying a specification causes the proof status of all proved formulas to revert to "unchecked," although the proof scripts are retained.[6]

[5] See the Language Guide[3] for details of conversions.

[6] 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.

3.5.1 Proving a Single Formula

Command	*Aliases*	*Function*
`prove`	`pr`, `C-c p`	Prove formula pointed to by cursor
`x-prove`	`xpr`, `C-c C-p x`	Start proof along with X display
`step-proof`	`prs`, `C-c C-p s`	Set up proof stepper for current formula
`x-step-proof`	`xsp`, `C-c C-p X`	Combines `x-prove` and `step-proof`
`redo-proof`	`prr C-c C-p r`	Redo the proof of formula at cursor *Arg:* don't display the proof
`prove-next-unproved-formula`	`prnext`, `C-c C-p n`	Start proof on next unproved formula

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.[7] If the formula has already been proved, then you will be asked whether the proof should be retried; a `no` answer ends the `prove` command. Otherwise, if the formula has an associated proof script, you will be 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.[8] At this point you are interacting with the prover, and certain commands will be unavailable until the prover is exited.[9]

The `x-prove` command is exactly like the `prove` command, except that it also pops up a window in which the proof tree is represented graphically. See section 3.11, page 52 for more details. If you are not running under X windows, then a warning message will be displayed and the command will be treated as a `prove` command.

The `step-proof` command is used to initiate the proof stepper, and is invoked in the same way as the `prove` command. Two buffers are displayed, one showing the sequent (the `*pvs*` buffer) and the other showing the proof script associated with the formula, if any (the `Proof` buffer). Section 3.5.10, page 39 explains how to use the proof stepper.

The `x-step-proof` command combines the `x-prove` and `step-proof` commands.

The `prove-next-unproved-formula` command invokes the prover on the next unproved formula at or beyond the current cursor position. If the formula already

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

[8] The system tries to keep as much of the proof visible as possible by redisplaying the screen so that the `Rule?` prompt is at the bottom of the window. This feature is not always desirable (*e.g.*, over a slow modem connection), and may be turned off by setting the Emacs variable `*pvs-maximize-proof-display*` to nil.

[9] 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.

has a proof, you will be asked whether to go ahead and run it or to start anew. Note that starting a new proof will not delete the old proof unless you allow the prover to overwrite it at the end of the proof session.

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.

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[10] 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 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.

[10] While in the prover you may freely move around in the `*pvs*` buffer or move to any other buffer to examine specifications or perform ordinary editing functions.

3.5.2 Proving Sets of Formulas

Command	*Aliases*	*Function*
`prove-theory`	`prt`, `C-c C-p t`	Rerun unproved proofs in theory
		Arg: include those already proved
`prove-theories`		Rerun proofs in specified theories
		Arg: include those already proved
`prove-pvs-file`		Rerun unproved proofs in current file
	`prf`, `C-c C-p f`	*Arg:* include those already proved
`prove-importchain`		Rerun prove-theory on `IMPORT` chain
	`pri`, `C-c C-p i`	*Arg:* include those already proved
`prove-importchain-subtree`		Rerun prove-theory on specified subtree
	`pris`	of `IMPORT` chain
		Arg: include those already proved
`prove-proofchain`		Rerun proofs on formulas in proofchain
	`prp`, `C-c C-p p`	*Arg:* include those already proved
`prove-formulas-theory`		Try unproved formulas with specified strategy
	`prft`	*Arg:* attempt proved formulas as well
`prove-formulas-pvs-file`		Try unproved formulas with specified strategy
	`prff`, `C-c C-p U`	*Arg:* attempt proved formulas as well
`prove-formulas-importchain`		Try unproved formulas with specified strategy
	`prfi`	*Arg:* attempt proved formulas as well
`prove-formulas-importchain-subtree`		Try unproved formulas with specified strategy
	`prfs`	*Arg:* attempt proved formulas as well
`prove-tccs-theory`		Try unproved TCCs with specified strategy
	`prft`	*Arg:* attempt proved TCCs as well
`prove-tccs-pvs-file`		Try unproved TCCs with specified strategy
	`prff`, `C-c C-p U`	*Arg:* attempt proved TCCs as well
`prove-tccs-importchain`		Try unproved TCCs with specified strategy
	`prfi`	*Arg:* attempt proved TCCs as well
`prove-tccs-importchain-subtree`		Try unproved TCCs with specified strategy
	`prfs`	*Arg:* attempt proved TCCs as well
`prove-untried-theory`		Try untried proofs with specified strategy
	`prut`, `C-c C-p u`	*Arg:* attempt TCCs as well
`prove-untried-pvs-file`		Try untried proofs with specified strategy
	`pruf`, `C-c C-p U`	*Arg:* attempt TCCs as well
`prove-untried-importchain`		Try untried proofs with specified strategy
	`prui`	*Arg:* attempt TCCs as well
`prove-untried-importchain-subtree`		Try untried proofs with specified strategy
	`prus`	*Arg:* attempt TCCs as well

Proof scripts can be rerun using the `prove-theory`, `prove-pvs-file`, `prove-importchain`, `prove-importchain-subtree` 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, import chain subtree, or proof chain, respectively. The import chain of a theory is simply the transitive closure of the `IMPORTING`s including those implicit in a theory declaration. The `prove-importchain-subtree` command takes additional theory name arguments and excludes these theories and their subtree from the importchain. The proof chain of a given formula is the transitive closure of the formulas used in the proof of that formula. 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 Section 3.16).

The `prove-theories` command prompts for theory names (with completion) one at a time, until an empty theory name is provided, and then runs `prove-theory` on each of these.

The commands `prove-formulas-theory`, `prove-formulas-pvs-file`, `prove-formulas-impor` `prove-formulas-importchain-subtree` `prove-tccs-theory`, `prove-tccs-pvs-file`, `prove-tccs-importchain`, and `prove-tccs-importchain-subtree` `prove-untried-theory`, `prove-untried-pvs-file`, `prove-untried-importchain`, and `prove-untried-importchain-subt` are all similar, but allow a given strategy to be applied to all applicable formulas.

For the `prove-formulas` commands, all unproved formulas that are not TCCs or axioms or postulates are attempted with the provided strategy, which defaults to `(grind)`. The `prove-tccs` commands are similar, but only attempt unproved TCCs, and the default strategy is `(tcc)`. With an argument, the already proved formulas are also attempted. If a given proof attempt succeeds, then it replaces any existing proof. If it fails and the given formula already has a proof, then the original proof is kept. Otherwise the new proof is associated with the formula. Thus after these commands all attempted formulas will have proofs associated with them. The strategy is any acceptable single prover command, as in the following example.

```
(then (grind :if-match nil) (inst?) (grind))
```

The `prove-untried` commands are similar, but they only affect formulas that have no associated proof, and providing an argument attempts TCCs that have no proofs as well. To apply a strategy to just the untried TCCs, redefine the `tcc` in your `pvs-strategies` Note that after any of these commands, all attempted formulas will have associated proofs, so issuing the same command with a different strategy will have no effect.

3.5.3 Selecting Decision Procedures

Command	*Function*
`set-decision-procedure`	Set the default decision procedure
`prove-theory-using-default-dp`	Rerun unproved proofs in specified theory using default decision procedures *Arg:* include those already proved
`prove-theories-using-default-dp`	Rerun proofs in specified theories using default decision procedures *Arg:* include those already proved
`prove-pvs-file-using-default-dp`	Rerun unproved proofs in current file using default decision procedures *Arg:* include those already proved
`prove-importchain-using-default-dp`	Rerun prove-theory on `IMPORT` chain using default decision procedures *Arg:* include those already proved
`prove-importchain-subtree-using-default-dp`	Rerun prove-theory on subtree of `IMPORT` chain using default dec. procedures *Arg:* include those already proved
`prove-proofchain-using-default-dp`	Rerun proofs on all formulas in proof chain using default decision procedures *Arg:* include those already proved

These commands have no effect if PVS was invoked with the -force-decision-procedures switch; see Section 4.1

The currently available decision procedures are `shostak` and `ics`. Much of the prover was built around the Shostak decision procedure,[11] ICS is a new decision procedure that can be run stand alone or included as a library. See `http://ics.csl.sri.com` for more. The prover manual discusses how the decision procedures are used; here we simply describe the commands for selecting them.

The decision procedure interface provides a set of methods that make it easy to add new decision procedures, as long as they satisfy the basic API. When a new decision procedure is added, it's name is made available to be used as a decision procedure.

The `set-decision-procedure` command sets the default decision procedure to be used in subsequent proofs. When a single formula is attempted that doesn't have a proof, the default decision procedure is automatically used. If it already has a proof that was developed using a different decision procedure, the prover prompts for whether to use the default or stay with the original decision procedure. When a proof is saved, the decision procedure used during the proof is saved as well. For the prover commands such as `prove-theory`, the proofs are each attempted with the decision procedure they were developed with. The remaining commands allow existing proofs to be rerun using the default decision procedures, and otherwise behave exactly as

[11] This was developed by Rob Shostak in the late 70s. Since then it has undergone many refinements.

the similarly named commands defined in the previous section.

Note that setting the decision procedure does not affect an ongoing proof. The decision procedures generally have different ways of storing state and processing it, and a proof may only be run with a single decision procedure. However, the decision procedure API is flexible enough to allow methods to be defined that, for example, run two different decision procedures in parallel and compare their results, or spawn two subprocesses and use the result of the first one to finish.

3.5.4 Editing and Viewing Proofs

Command	*Aliases*	*Function*
`edit-proof`	`show-proof`	Edit the proof of the indicated formula
`install-proof`	`C-c C-i`	Install proof on the indicated formula
`install-and-step-proof`	`C-c s`	Install proof on a formula and step
`install-and-x-step-proof`	`C-c x`	Install proof on formula, display, and step
`remove-proof`		Remove proof associated with a formula
`show-proof-file`		Edit the proofs of the indicated PVS file
`show-orphaned-proofs`		Edit the orphaned proofs
`show-proofs-theory`		Show all proofs of a theory
`show-proofs-pvs-file`		Show all the proofs of a PVS file
`show-proofs-importchain`		Show all proofs of importchain of a theory
`install-pvs-proof-file`		Installs proof file for typechecked theory
`display-proofs-formula`		Display the (multiple) proofs associated with this formula
`display-proofs-theory`		Display the (multiple) proofs of the formulas in the theory
`display-proofs-pvs-file`		Display the (multiple) proofs of the formulas in the PVS file
`load-pvs-strategies`		Loads a pvs-strategies file
`set-print-depth`		Sets print depth for printing sequents
`set-print-length`		Sets print length for printing sequents
`set-print-lines`		Sets number of lines to print for each sequent formula
`set-rewrite-depth`		Sets the print depth for rewrite messages
`set-rewrite-length`		Sets the print length for rewrite messages
`dump-sequents`		Save unproved sequents to a file
`toggle-proof-prettyprinting`		Toggles the prettyprinting of proof files

Every formula of a specification for which a proof has been attempted has an associated proof script that 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 (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,[12] 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

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

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 using `install-proof`.

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

```
;;; Proof for formula sum.closed_form
;;; developed with old decision procedures
(""
 (INDUCT "n")
 (("1" (EXPAND "sum") (ASSERT))
  ("2" (SKOLEM!) (FLATTEN) (EXPAND "sum" +) (ASSERT))))
```

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. In each case, the new proof only becomes the default, the old proofs are still available and may be manipulated by means of the `display-proofs-formula` command, which allows the default proof to be reset. If no proof is being edited (*i.e.*, there is no `Proof` buffer), an error is reported.

The proof may also be installed using the `install-and-step-proof` or `install-and-x-step-pr` commands, both of which install the proof and initiate the proof stepper; the latter also displays the proof tree.

Checkpoints may be added to the `Proof` buffer obtained by the `edit-proof` command. To add a checkpoint, position the cursor and type `C-c a`. The checkpoint is indicated by a double exclamation point (`!!`). Any number of checkpoints may be added. When the proof is installed using `C-c C-i`, these are changed to the `checkpoint` proof rule, and branches of the proof that do not have a checkpoint on them are wrapped in a `just-install-proof` proof rule. When this proof is rerun, it will run until it hits a `checkpoint`, and then prompt for a prover command. When it hits a `just-install-proof`, it simply installs the given commands and marks that branch as proved. This allows the prover to quickly get to the next checkpoint, without attempting to reprove branches that do not have checkpoints in them. When a proof that has `just-install-proof` rules in it is finished, the prover asks whether the proof should be rerun, as the formula will not be considered proved until the proof is rerun.

To remove a checkpoint from the `Proof` buffer, position the cursor at the checkpoint and type `C-c r`. To remove all checkpoints, type `C-c DEL`.

In addition to the above, the key bindings for browsing and the prover emacs (TAB) commands are available in a `Proof` buffer.

The `remove-proof` command is used to remove the proof associated with the specified formula. The primary use for this is to remove proofs from axioms for which a proof attempt has been made.

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. Note that invoking `edit-proof` on the formula currently being proved will display the proof script stored with the formula, if there is one. To display the current proof script, use the `show-current-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 `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. You may select a proof script for editing, or simply view the script in a pop-up 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 pop-up buffer, but may not be edited. To edit a proof script, position the cursor and type "`s`." This will create or use the `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, creating *orphaned* proofs. Orphaned proofs are saved in the `orphaned-proofs.prf` file. In some cases, the system will recognize that an orphaned proof should be reattached to a formula, and will ask whether it should go ahead.

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`" pops up a `View Proof` buffer, and typing a "`q`" exits the orphaned proof buffer.

The commands `show-proofs-theory`, `show-proofs-pvs-file`, and `showproofs-importchain` display all of the proofs of the associated theory, PVS file, or importchain in a buffer named `Show Proofs`, which is in `PVS View` mode.

The `install-pvs-proof-file` command prompts for a PVS file name, and reads in the corresponding proof file, replacing any proofs that may have been loaded or developed. This command is needed in order to get a new proof file accepted in a

context. When specification files are parsed and/or typechecked, the corresponding proof files are read in. After that the system will not pay any attention to changes made to the proof file, but simply update it as changes are made that affect proof status. This command allows you to modify the file or copy a new one in and get it installed.

The `load-pvs-strategies` command loads the strategies files from your home directory, imported libraries, and the current context. This command is only needed when a new strategy is being developed during a proof; when a proof is started the system checks whether any of the strategy files have changed and automatically loads them if they have. See the prover guide [7] for details on the contents of the `pvs-strategies` files.

The `set-print-depth`, `set-print-length`, and `set-print-lines` commands control how much of an expression is displayed in a sequent. If the print depth and length are set to 0 and the lines is `NIL`, then the entire sequent is displayed. This is the default. If the depth is set to a positive integer, then any subterms at that depth are replaced by a pound sign (`#`). Similarly, if the length is set to a positive integer, then any subterms beyond the specified length are replaced by three periods (`...`). The length and depth of an expression are not easy to define, because it is related to the abstract syntax used by the prettyprinter. In general, expressions separated by commas have a length, while subterms[13] are deeper by one than the containing terms. If the print lines is set to a number n, then only the first n lines of each formula of the sequent is displayed, and remaining lines are replaced by two periods (`..`). Note that all these commands are also rules in the prover that otherwise behave as a `SKIP`, so it is easy to adjust the printout interactively.

The `set-rewrite-depth` and `set-rewrite-length` commands control how much information to output when printing the results of automatic rewrites. Normally, both the rule name and the expression being rewritten are displayed in the proof commentary when an auto-rewrite is triggered. The value should be a positive number or `NIL`. If it is a positive number, then any subexpression at that depth or length will be replaced by a pair of periods (`..`) or three periods (`...`) respectively. If it is 0 (zero), then only the rule name is displayed. If it is `NIL`, then there is no bound.

The `dump-sequents` command indicates that any incomplete proof attempt should save the remaining unproved sequents to file. If the proof is for formula `foo` from theory `th`, then the file containing the unproved sequents is named `th-foo.sequents`. If the formula is proved, then no file is generated, and any file left from an earlier attempt on this formula is removed.

The `toggle-proof-prettyprinting` command toggles whether to prettyprint the proof file (with extension .prf) associated with a PVS file. Prettyprinted files are easier to read, edit, and email, but they take a lot longer to generate. By default, proof files are prettyprinted.

The commands for exhibiting proofs can get confusing. To summarize, only the

[13] For example, the operator and arguments are subterms of an application.

`display-proofs-` commands support multiple proofs, while the others just show the default proofs. The `show-proof-file` and `show-orphaned-proofs` commands provide listings that are similar to those produced by the `display-proofs-` commands, but

3.5.5 Displaying Proof Information

Command	*Function*
`show-current-proof`	Display the current proof
`show-last-proof`	Displays printout of most recent proof *Arg:* make it brief
`set-proof-backup-number`	Set number of backup proof files to retain *Arg:* number of files to retain
`show-proof-backup-number`	Show number of backup proof files retained
`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
`show-expanded-sequent`	Display the sequent in expanded form *Arg:* also expand names from the prelude
`show-skolem-constants`	Display the Skolem constants and their types
`explain-tcc`	Display the explanation for a TCC
`usedby-proofs`	Display formulas whose proofs refer to the declaration at the cursor
`pvs-set-proof-parens`	Control parentheses display in proofs
`pvs-set-proof-prompt-behavior`	Indicates the kind of prompting at the end of a proof; one of `:ask`, `:overw`
`pvs-set-proof-default-description`	Sets a default description string for saved proofs

These commands work only while an interactive proof is being developed, *i.e.*, after the `prove` command. The `show-current-proof` command shows the current proof in the `*Proof*` buffer 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 parts of it to other branches using the `rerun` prover command, as described in the prover guide[7].

The `show-last-proof` command provides a display of the commentary and subgoals associated with the most recently completed proof in the `Proof Display` buffer. This version does not contain the `undo`, `skip`, or `postpone` steps and provides a clean version that shows the commentary and subgoals. This printout is useful in trying to summarize the proof for publication. With an argument, many of the sequents are suppressed, and within a sequent, formulas which haven't changed since the previous sequent display are elided.

The `set-proof-backup-number` command indicates the number of backups to be kept for proof files. If the argument is 0, then no backups are kept. If it is 1, then before the `.prf` file is written, the old copy is retained with extension `.prf~`. For larger arguments, that number of old `.prf` files are retained with the extension `.prf.~x~`, with increasing values of `x`. For example, if the argument is 3, and backup

files `foo.prf.~3~`, `foo.prf.~4~`, and `foo.prf.~5~` exist, when the next backup is created `foo.prf.~3~` is removed and `foo.prf.~6~` is created. The default value is 1, and PVS will revert to this behaviour on each invocation. Thus, it is recommended that this command be placed in the file `.pvsemacs` in your home directory, e.g.:

```
(set-proof-backup-number 5)
```

The current number of proof files being retained is reported by the `showproof-backup-number` command.

The `ancestry` command displays the branch of the proof from the root to the current sequent in the `Ancestry` buffer, and the `siblings` command displays the siblings of the current sequent in the `Siblings` buffer, 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. These formulas are displayed in the `Hidden` buffer. Each formula is displayed with a number which may be referred to in the `reveal` prover command (see the prover guide [7]).

The `show-auto-rewrites` command displays the auto-rewrite rules that are in effect for the current sequent. The rules are displayed in the `*Auto-Rewrites*` buffer, 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 a clash and two or more rewrite rules are applicable, the most recently introduced one is applied first.

The `show-expanded-sequent` command displays the current sequent in the `Expanded Sequent` buffer, with each variable, constant and operator expanded to its full type, including the theory and its parameters, unless they are from the current theory or the prelude. With an argument, prelude names are also expanded. `show-skolem-constants` displays the type of all skolem constants introduced in the current proof in the `Proof Display` buffer. Normally names from the prelude are not expanded, an argument expands these as well.

A TCC subgoal is marked as such in a proof. Invoking the `explain-tcc` command provides some explanation for why the TCC was generated, giving the type of TCC, and the expression which caused its generation.

The `usedby-proof` command provides a list of formulas whose proofs refer to the given declaration. This works by looking through the formulas of all the currently typechecked theories of the current context; in particular, for prelude or library declarations it will not locate all formulas that ever referred to the declaration, as this information would be difficult to maintain and be of marginal use. The buffer generated by the `usedby-proof` command is the same as that for the `find-declaration` command, with the same key-bindings for viewing and going to the listed declarations.

The `pvs-set-proof-parens` command asks whether to show parentheses, and if so, sets a variable indicating that sequents should be displayed with full parenthesization. This is mostly useful for proofs involving large arithmetic terms, where it may otherwise be difficult to figure out whether a given rewrite rule should apply.

The introduction of multiple proofs changed the way PVS handles the end of a proof session. When a proof attempt is ended, either by quitting or successfully completing the proof, the proof is checked for changes. If any changes have occured, the user is queried about whether to save the proof, and whether to overwrite the current proof or to create a new one. If a new proof is created, the user is prompted for a proof identifier and a description. At the end of any given proof a number of questions may be asked:

- `Would you like the proof to be saved?`
- `Would you like to overwrite the current proof?`
- `Please enter an id`
- `Please enter a description:`

The `pvs-set-proof-prompt-behavior` command allows you to control this behavior. The possible values for the prompt behavior are:

`:ask`	the default; all four questio
`:overwrite`	similar to earlier PVS versi
`:add`	asks if the proof should be

The `pvs-set-proof-default-description` command allows you to set a default description string. It is used if the prompt is anything but `:ask`, or if the empty string (i.e., just hitting Return) is provided when a description is asked for. It defaults to the empty string.

3.5.6 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. 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 you to make adjustments to theories precisely when the need for such an adjustment is 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 declarations if they are visible, *i.e.*, exported,[14] 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: the `Modify Declaration` 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. Thus these commands should be viewed as a convenient 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.

3.5.7 Prover Emacs Commands

The prover commands can be somewhat tedious to type in, especially the simple ones that are used regularly, such as `assert`, `grind` and `skosimp*`. C. Michael Holloway of NASA Langley created an extension to Emacs to relieve some of the tedium, and was kind enough to make these extensions available to PVS. This section describes those extensions in three subsections: General Commands, Prover Commands, and Proof Stepper Commands.

3.5.8 General Commands

Command	*Aliases*	*Function*
`pvs-prover-any-command`	`TAB TAB`	Insert (prompted for) command
`pvs-prover-quotes`	`TAB '`	
`pvs-prover-wrap-with-parens`	`TAB C-j`	

[14] 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.

The `pvs-prover-any-command` prompts for a command (with completion), and inserts it in the prover buffer with the cursor positioned for additional arguments. This command is provided for those prover commands that do not have an Emacs key binding associated with them.

The `pvs-prover-quotes` command makes it easier to give PVS types and expressions, by inserting a pair of double quotes around the current cursor location. The `pvs-prover-wrap-with-parens` command wraps a given prover command in parentheses and send it to the prover. You must be at the end of the prover input to use this command.

3.5.9 Prover Commands

These commands simply prompt for any arguments, and then apply the specified prover command to those arguments. After all the arguments, if any, have been given the command is immediately executed by the prover. Not all prover commands are represented below, and even for those that are given below not all arguments are prompted for. Commands with complex arguments are generally easier to type in directly, using the `M-x pvs-prover-any-command` command if desired. The `M-p`, `M-n`, and `M-s` keys are particularly useful in this case, as a mistyped prover command can easily be brought back and corrected, or a complex command that is used frequently may be easily brought back.

The prover command associated with the following Emacs commands should be obvious. Details for any given command may be found by typing `C-h d` followed by the command name, e.g., `pvs-prover-auto-rewrite`.

Command	*Aliases*	*Function*
pvs-prover-apply-extensionality	TAB E	
pvs-prover-assert	TAB a	
pvs-prover-auto-rewrite	TAB A	
pvs-prover-auto-rewrite-theory	TAB C-a	
pvs-prover-bddsimp	TAB B	
pvs-prover-beta	TAB b	
pvs-prover-case	TAB c	
pvs-prover-case-replace	TAB C	
pvs-prover-decompose-equality	TAB =	
pvs-prover-delete	TAB d	
pvs-prover-do-rewrite	TAB D	
pvs-prover-expand	TAB e	
pvs-prover-extensionality	TAB x	
pvs-prover-flatten	TAB f	
pvs-prover-grind	TAB G	
pvs-prover-ground	TAB g	
help-pvs-prover-command	TAB H	
pvs-prover-hide	TAB C-h	
pvs-prover-iff	TAB F	
pvs-prover-induct	TAB I	
pvs-prover-induct-and-simplify	TAB C-s	
pvs-prover-inst	TAB i	
pvs-prover-inst-question	TAB ?	
pvs-prover-lemma	TAB L	
pvs-prover-lift-if	TAB l	
pvs-prover-model-check	TAB M	
pvs-prover-musimp	TAB m	
pvs-prover-name	TAB n	
pvs-prover-postpone	TAB P	
pvs-prover-prop	TAB p	
pvs-prover-quit	TAB C-q	
pvs-prover-replace	TAB r	
pvs-prover-replace-eta	TAB 8	
pvs-prover-rewrite	TAB R	
pvs-prover-skolem-bang	TAB !	
pvs-prover-skosimp	TAB S	
pvs-prover-skosimp-star	TAB *	
pvs-prover-split	TAB s	
pvs-prover-tcc	TAB T	
pvs-prover-then	TAB C-t	
pvs-prover-typepred	TAB t	
pvs-prover-undo	TAB u	

3.5.10 Proof Stepper Commands

Command	*Aliases*	*Function*
pvs-prover-one-proof-step	TAB 1	
pvs-prover-many-proof-steps	TAB @	
pvs-prover-undo-one-proof-step	TAB U	
pvs-prover-undo-many-proof-steps	TAB C-u	
pvs-prover-skip-one-proof-step	TAB #	

The proof stepper is invoked with the `step-proof` or `x-step-proof` command, though it may be used after a proof is begun simply by putting the cursor on the formula in the specification and typing `M-x edit-proof`, which pops up the `Proof` buffer. When this buffer is available, the proof stepper may be used. The proof stepper keeps track of the current position within the `Proof` buffer, and when invoked from the `*pvs*` buffer, sends the next command(s) from the `Proof` buffer to the prover, changing the current position to point to the next command. When `step-proof` is invoked, the current position is at the beginning of the buffer. You may go to the `Proof` buffer and edit it or change position within it, and the stepper will then use the new information. The `pvs-prover-one-proof-step` command just invokes the next single command in the proof buffer. The next command in this sense is not necessarily simple, for example the next command may be

```
(apply (then* (skosimp*) (expand "foo") (lift-if) (ground)))
```

in which case the entire apply is invoked, not the individual components.

The `pvs-prover-many-proof-steps` prompts for the number of proof steps, and iterates the `pvs-prover-one-proof-step` command that many times.

The `pvs-prover-undo-one-proof-step` undoes the last command, and backs up one position in the `Proof` buffer. The `pvs-prover-undo-many-proof-steps` command prompts for the number of steps to undo, and has the same effect as invoking `pvs-prover-undo-one-proof-step` that many times. The difference between these and the `pvs-prover-undo` command is that the latter does not change the position of the cursor within the `Proof` buffer.

The `pvs-prover-skip-one-proof-step` skips the next proof step.

If you are using a recent version of Emacs, then the next prover command should be highlighted in the `Proof` buffer. All of the commands of this section move the highlight the appropriate direction. The highlight does not always point to the correct location; in particular, if you go to the `Proof` buffer, move the cursor, and go back to the `*pvs*` buffer, then the highlight is not moved, but the next command is relative to the cursor position, not the highlight. The highlight is only accurate right after one of these commands.

3.6 Prettyprinting

Command	*Aliases*	*Function*
`prettyprint-theory`	`ppt`, `C-c C-q t`	Prettyprint theory
`prettyprint-pvs-file`	`ppf`, `C-c C-q f`	Prettyprint PVS file
`prettyprint-declaration`	`ppd`, `C-c C-q d`, `C-M-q`	Prettyprint declaration
`prettyprint-region`	`ppr`, `C-c C-q r`, `C-M-\`	Prettyprint region
`prettyprint-theory-instance`	`ppti`, `C-c C-q i`	Prettyprint theory instance
`pvs-set-linelength`		Set prettyprinting line length

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 "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-pvs-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 are generally lost during prettyprinting.[15]

The `prettyprint-theory-instance` command prettyprints the given theory instance, which is a theory name, generally including actual parameters and/or mappings. It is primarily used to show the results of a theory instance involving complex actuals and/or mappings. Given a theory name, for example, `th[int, 2]{{ c := 13 }}`, a new buffer `th.ppti` is created with the contents of `th`, but with formals and uninterpreted declarations substituted for. A second theory must be provided for context, in order to typecheck the actuals and the mappings. The theory name is typechecked in this context, which may lead to a type error. Note that the theory instance may not be a stand alone theory, as the substitutions may point to declarations that are not visible to the original theory.

The `pvs-set-linelength` command sets the line length used to control prettprinting. The default is the width (in characters) of the starting window.

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

3.7 Viewing TCCs

Command	*Aliases*	*Function*
`prettyprint-expanded`	`ppe`, `C-c C-q e`	Prettyprint expanded theory in new buffer
`show-tccs`	`tccs`, `C-c C-q s`	Show the TCCs of the specified theory *Arg:* Show only unproved TCCs
`show-declaration-tccs`		Show the TCCs of the specified declaration *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 before 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 (including the expanded definitions of inline ADTs and conversions), 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 with the extension `.tccs` 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 of TCCs are initiated exactly as described above.

The `show-declaration-tccs` command pops up a buffer with the name *`theory.declid`*`.tccs`, displaying just the TCCs of the specified declaration. Proofs of any displayed TCCs may be initiated in the usual way, simply by moving the cursor to the formula to be proved and invoking the `prove` command.

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` and `show-declaration-tccs` commands are faster to process and include 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.

3.8 PVS Files and Theories

3.8.1 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
`view-prelude-theory`	`vpt`	List prelude theory
`view-library-file`	`vlf`	List library file
`view-library-theory`	`vlt`	List library theory

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.[16]

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, as long as their names are changed; theory names of the prelude may not be reused. Viewing the prelude is useful for finding out what types, constants, and formulas are available, for seeing paradigmatic examples of specifications, and for trying out the prover on some readily available formulas.

The `view-library-file` and `view-library-theory` commands operate in a similar manner to the `view-prelude-file` and `view-prelude-theory` commands. They allow for completion on those libraries which are imported into the current context, and will pop up a buffer containing the contents of the file, moving the cursor to the beginning of the specified theory for `view-library-theory`. Giving an argument to `view-library-file` allows for completion on all of the distributed libraries as well (i.e. those in the `lib` subdirectory of the PVS installation) whether they are imported into the current context or not.

The `view-library-file` and `view-library-theory` commands may not report all of the theories which have been imported into the context if the specification files in the context have not yet been typechecked. A warning message will be printed to

[16] Note that `find-pvs-file` and `find-theory` will only find files and theories that are in the current PVS context

this effect if there are no imported libraries found.

3.8.2 Creating New Files and Theories

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

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. The template normally includes comments indicating the form of formal parameters and the assumings section; with an argument a minimal template is used that simply gives the beginning and end of the specified theory.

3.8.3 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

The commands described here allow files and theories to be imported from other contexts. The `import-pvs-file` command prompts for a source file (including directory, but omitting the `.pvs` extension) and a target file (a new PVS filename without directory or extension) and copies the former to the latter, and places the file in the current context. In addition, the corresponding proof file is copied.

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.

3.8.4 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.

3.8.5 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
`save-pvs-buffer`		Saves the current buffer to file

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

The `save-pvs-buffer` command copies the contents of the current buffer to the specified file name, without renaming the buffer. This command should be used for buffers that have no associated file instead of the Emacs `write-file` command, which does rename the buffer.

3.8.6 Mailing PVS Files

Command	*Aliases*	*Function*
`smail-pvs-files`		Send a set of PVS files by e-mail
`rmail-pvs-files`		Read a set of PVS files sent by `smail-pvs-files`

These commands make it easy to send and receive sets of PVS files. At least two messages are sent: one that is composed by you, to explain the contents of the following message(s), and the rest which are the files tarred, compressed, and translated to ascii. If the resulting file is large, then it is also split into smaller pieces that are mailed separately.

The `smail-pvs-files` command prompts for a root file, an e-mail address (defaults to `pvs-bugs@csl.sri.com`, or the last address used in this session), a `CC:` list, and a subject line. A mail buffer is then popped up so that you can compose your message. When you have completed your message, type `C-c C-c` to send it.[17] At that

[17] If you change your mind about sending a message, simply kill (`C-x k`) the `*mail*` buffer.

point the patch revision number is added to the end of your message, the PVS files in the import chain of the root file are collected along with the associated proof files and the files `pvs-strategies`, `pvs-tex.sub`, `~/pvs-strategies`, `~/pvs-tex.sub`, and `~/.pvsemacs`. Those files collected from your home directory will be put in a newly created directory named `PVSHOME`. Then all of these file will be sent using `tarmail`, which uses `tar`, `compress`, `btoa`, and `split` to send the files collected, splitting them into multiple parts if necessary. A buffer is popped up showing the result of the `tarmail` command; you should look this over to verify that all of the desired files are included, and that there are no errors. Use `C-z 1` to remove this buffer. After the files are sent, the `PVSHOME` directory is deleted.

The `rmail-pvs-files` command unpacks mail sent by `smail-pvs-files`. To use this, first create a new directory in which to install the files, and, using your favorite mailer, copy the files to the new directory with extensions corresponding to the message order, *e.g.*, `mail.01`, `mail.02`, etc. If there is just one file, leave the extension off. Then invoke `M-x rmail-pvs-files` and give the root file name when prompted (*e.g.*, `mail`). The mail files will be unpacked using `untarmail`, and a pop-up buffer will be displayed showing the files that have been unpacked. If a directory named `PVSHOME` has been created, it will contain the PVS files from the home directory of the person that sent the mail. If these are needed, they should be copied or merged into the corresponding files in your home directory. Check that the patch version number that appears at the bottom of the first (readable) mail message matches the patch revision number in the `PVS Welcome` buffer. If they don't match, the sender or receiver (or both) should update their PVS installations (see http://pvs.csl.sri.com fro details).

3.8.7 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
		Arg: overwrites existing files without asking
`edit-pvs-dump-file`		Edit a PVS dump file

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. A secondary purpose is to make global edits, *e.g.*, changing the name of a constant or formula throughout all of the `.pvs` and `.prf` files.

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.

The `edit-pvs-dump-file` command makes it easy to edit a dump file created by `dump-pvs-files`. This is useful when you wish to send just a subset of the theories in the import chain. Note that the system uses `$$$` followed by the file name as a separator; if these are modified files may be merged randomly when they are undumped. The dump file buffer is put in outline mode, with these separators treated as headings. The `hide-body` (`C-c C-t`) command will show just these separators, making it easy to remove entire files. See the Emacs manual for more details on outline mode.

3.9 PVS Output

3.9.1 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`[18] in gaudy mode producing two column rotated output, add the following lines to your `~/.pvsemacs` file:

```
(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.[19] 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.

3.9.2 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

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

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

These commands send the specified theories to the printer, using 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.

3.9.3 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
`alltt-proof`	`alp`, `C-c C-a p`	Format last proof for LaTeXalltt *Arg:* make it brief

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. 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.

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

3.9.4 Generating LaTeX Output

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

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. The default filename is the name of the formula last

proved, with a `.tex` extension. If there was no proof attempted in the session, then the system will ask for a proof to be rerun. The proof is written verbosely, unless an argument is given, in which case it provides a brief printout of the proof, and only the changed sequent formulas are printed..

In addition to the generated specification and proof files, a file named `pvs-files.tex` is generated that includes all of the files generated in the last invocation of one of these commands. The purpose of the `pvs-files.tex` file is two-fold: to facilitate LaTeXing and printing the theories, and to illustrate the inclusion of these files in a document. It is best to create your own top-level file to include the others, as the `pvs-files.tex` file is overwritten with each latex output command.

These commands make use of the prettyprinter, which uses the linelength to determine where to make line breaks. For prettyprinting in Emacs buffers, this is set according to the size of the window if in X windows, or to 80 otherwise. For prettyprinting specifications for LaTeX, however, there is no easy way to determine the "right" value, as the page width, number of columns, font size, etc., all contribute to the determination of the actual value. The `latex-set-linelength` command allows you to set the linelength according to your needs. The default value is 100, which generates reasonable looking specifications for `pvs-files.tex`.

To process the generated files you must include the `pvs.sty` style file, located in the main PVS directory. (see the LaTeX manual [1] for details on including style files).

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 ∀. 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 or your home directory. Each row of this file specifies an identifier, its kind and arguments, an estimated length (in `em`s), and the substitution. For example,

```
THEORY key 7 {\large\textbf{\textrm{Theory}}}
sum 1 2 {\sum_{i=0}^{#1} i}
sum (2 1) 2 {\sum_{i=#1}^{#2} #3}
sum [2] 1 {sum_{#1}^{#2}}
th.sum 1 2 {\bigoplus_{i=0}^{#1} i}
```

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.

Identifiers may also include a theory name, for example, `groups.+`. This allows the limited number of ASCII operators to be mapped to different LaTeX symbols. Note that actuals may not be included as this would require typechecking the substitution file.

The kind field may be specified as one of the symbols `id` or `key`, a number, a parenthesized sequence of 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 `sum[int,3]` would be translated to $\text{sum}^{3}_{\text{int}}$.

When both the argument form and the actuals form are provided in the `pvs-tex.sub` file and both are given in the specification, the argument form is used. Thus using the above translation, `sum[int,3](a+1,b)(n)` would be translated to $\sum_{i=a+1}^{b} n$.

The argument form also pertains to declarations; so a declaration of the form

```
i,j,n: VAR int
sum(i,j)(n): RECURSIVE int = ...
```

will be nicely printed, whereas the equivalent declaration

```
sum(i,j:int)(n:int): RECURSIVE int = ...
```

will still use the argument form, but will have types included, and

```
sum: RECURSIVE [int, int -> [int -> int]] = (LAMBDA ... )
```

will be translated using the id form.

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. If the length field is a hyphen, then the length is taken to be the length of the identifier.

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 your 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.

3.10 Generating HTML

Command	*Aliases*	*Function*
`html-pvs-file`		Generate HTML for a PVS file
`html-pvs-files`		Generate HTML for all files imported by a given PVS file

These commands generate HTML files corresponding to PVS files. These can be generated in place, or in a specified web location. This is driven by setting a Lisp variable `*pvs-url-mapping*`, as described below.

The in place version creates a `pvshtml` subdirectory for each context and writes HTML files there. This is done by copying the PVS file, and adding link information so that comments and whitespace are preserved. Note that there is no `html-theory` command. This is not an oversight; in creating the HTML file links are created to point to the declarations of external HTML files. Hence if there was as way to generate HTML corresponding to both theory and PVS file, it would be difficult to decide which a link should refer to.

HTML files can be generated in any order, and may point to library files and the prelude. Of course, if these files do not exist then following these links will produce a browser error. The `html-pvs-files` command will attempt to create all files that are linked to, failure is generally due to write permission problems.

Usually it is desirable to put the HTML files someplace where anybody on the web can see them, in which case you should set the `*pvs-url-mapping*` variable. It's probably best to put this in your `/.pvs.lisp` file in your home directory so that it is consistently used. This should be set to a list value, as in the following example.

```
(setq *pvs-url-mapping*
      '("http://www.csl.sri.com/~owre/"
        "/homes/owre/public_html/"
        ("/homes/owre/pvs-specs" "pvs-specs" "pvs-specs")
        ("/homes/owre/pvs3.2" "pvs-specs/pvs3.2" "pvs-specs/pvs3.2")
        ("/homes/owre/pvs-validation/3.2/libraries/LaRC/lib"
         "pvs-specs/validation/nasa"
         "pvs-specs/validation/nasa")))
```

The first element of this list forms the base URL, and is used to create a `<base>` element in each file. The second element is the actual directory associated with this URL, and is where the `html-pvs-file` commands put the generated files. The rest of the list is composed of lists of three elements: a specification directory, a (possibly relative) URL, and a (possibly relative) HTML directory. In the above example, the base URL is http://www.csl.sri.com/~owre/, which the server associates with `/homes/owre/public_html`. The next entry says that specs found in (a subdirectory of) `/homes/owre/pvs-specs` are to have relative URLs corresponding to `pvs-specs`, and relative subdirectories similarly. Thus a specification in `/homes/owre/pvs-specs/tests/conversions/` will have a corresponding HTML

file in `/homes/owre/public_html/pvs-specs/test/conversions/` and correspond to the URL http://www.csl.sri.com/~owre/pvs-specs/test/conversions/. In this case, PVS is installed in `/homes/owre/pvs3.2`, and thus references to the prelude and distributed libraries (such as finite sets), will be mapped as well. Note that in this example, all the relative structures are the same, but it doesn't have to be that way.

The `*pvs-url-mapping*` is checked to see that the directories all exist, though currently no URLs are checked (if anybody knows a nice way to do this from Lisp, please let us know). If a subdirectory is missing, the system will prompt you for each subdirectory before creating it. A `n` or `q` answer terminates processing without creating the directory, a `y` creates the directory and continues, and a `!` causes it to just create any needed directories without further questions.

If a `*pvs-url-mapping*` is given, it must be complete for the file specified in the `html-pvs-file` command. In practice, this means that your PVS distribution must be mapped as well. PVS will complain if it is not complete; in which case simply add more information to the `*pvs-url-mapping*` list.

No matter which version is used, the generated HTML (actually XHTML) file contains a number of `<span>` elements. These simply provide a way to add `class` attributes, which can then be used in Cascading Style Sheet (CSS) files to define fonts, colors, etc. The classes currently supported are:

```
span.comment
span.theory
span.datatype
span.codatatype
span.type-declaration
span.formal-declaration
span.library-declaration
span.theory-declaration
span.theory-abbreviation-declaration
span.variable-declaration
span.macro-declaration
span.recursive-declaration
span.inductive-declaration
span.coinductive-declaration
span.constant-declaration
span.assuming-declaration
span.tcc-declaration
span.formula-declaration
span.judgement-declaration
span.conversion-declaration
span.auto-rewrite-declaration
```

See the `<PVS>/lib/pvs-style.css` file for examples. This file is automatically copied to the base directory if it doesn't already exist, and it is referenced in the generated HTML files. Most browsers underline links, which can make some operators difficult to read, so this file also suppresses underlines. This file may be edited to suit your own taste or conventions.

Both the `html-pvs-file` commands take an optional argument. Without it, many of the common prelude operators are not linked to. With the argument all operators get a link. Overloaded operators not from the prelude still get links.

3.11 Display Commands

Command	*Aliases*	*Function*
`x-theory-hierarchy`		Display the `IMPORTING` chain from a theory
`x-show-proof`		Display the proof of specified formula in an X window
`x-show-current-proof`		Display the current proof in an X window

These commands provide graphical displays by making use of the Tcl/Tk [2] system. To use these commands you must be running the X Window System and have the `DISPLAY` environment variable correctly set. In addition, you must have Tcl/Tk (version 7.3/3.6 or later) installed, and the `wish` command must be in your path before you start PVS (or the variable `pvs-wish-cmd` must be set to the full pathname of `wish`).

The `x-show-current-proof` command creates a window showing the current proof tree. Every sequent in the tree is represented by a ⊢ symbol. The proof commands used to create the tree will also be shown between the ⊢ symbols. This tree will be automatically updated after every proof command.

To see the full text of a given sequent, click on the ⊢ symbol. The ⊢ will acquire a number, and a new numbered window will pop up containing the text of the sequent. Proof commands which are longer than a certain customizable length (see below) are abbreviated; the full command can be seen by clicking on the abbreviation. Figure 3.1 shows an example proof display, in which the `case` rule and two sequents have been clicked on. The look of your display will probably be different, depending on the window manager you use and the defaults you set up for it.

Colors are used to display status information about the proof. These colors may be specified using the X resource database (i.e., in your `.Xresources` or `.Xdefaults` file). Stipple patterns may be specified instead of colors; a stipple pattern is specified as `@file`, where `file` is either an absolute pathname of a file in X bitmap format or the special bitmap name `gray`.[20]

The current resources and their defaults are:

[20] If `file` is not an absolute path, it is looked up in the `wish` subdirectory of the PVS directory, which contains the `gray` bitmap.

Figure 3.1: A Proof Display Example

Resource name	*Color default*	*Monochrome default*
`pvs.windowbackground`	wheat	white
`pvs.displaybackground`	white	white
`pvs.displayforeground`	black	black
`pvs.activedisplaybackground`	mediumslateblue	black
`pvs.activedisplayforeground`	white	white
`pvs.buttonbackground`	lightblue	white
`pvs.buttonforeground`	black	black
`pvs.activebuttonbackground`	steelblue	black
`pvs.activebuttonforeground`	white	white
`pvs.troughcolor`	sienna3	black
`pvs.currentColor`	DarkOrchid	black
`pvs.circleCurrent`	yes	yes
`pvs.tccColor`	green4	black
`pvs.doneColor`	blue	@gray
`pvs.ancestorColor`	firebrick	black
`pvs.abbrevLen`	35	35
`pvs.displayfont`	lucidasanstypewriter-bold-12	
`pvs.buttonfont`	lucidasanstypewriter-10	
`pvs.proof.geometry`	*none*	
`pvs.theory-hierarchy.geometry`	*none*	
`pvs.prover-commands.geometry`	*none*	

The `foreground` color is used for things that aren't otherwise specified below. The `currentColor` is used for the current sequent in the proof tree. The `ancestorColor` is used for all the ancestors of the current sequent, up to the root. The `doneColor` is used for sequents which have been proved. The `tccColor` is used for TCC's. When `pvs.circleCurrent` is set, the current sequent in the proof tree is circled.

Proof commands which are longer than `abbrevLen` characters are abbreviated.

If the Emacs variable `pvs-x-show-proofs` is not `NIL`, then `prove` automatically calls `x-show-proof`. This can be set in your `.pvsemacs` file.

The `x-theory-hierarchy` command prompts for a theory name and displays the `IMPORTING` hierarchy rooted at that theory. In a complex hierarchy, it can be difficult to follow the lines; to make this easier, when you move the mouse onto a theory identifier, all the lines connecting that theory to other theories turn the `highlight` color. Clicking on a theory identifier will bring up the theory in Emacs. Figure 3.2 shows an example of the theory hierarchy for the `finite_sets` library, as produced from clicking on the `Gen PS` button and selecting `portrait`.

The remainder of this section applies to both `x-show-proof` and `x-theory-hierarchy`.

The layout in the windows created by these commands can be manually edited. The editing commands are accessed by holding down the `Control` key while pressing mouse buttons. In a proof window, pressing `Control`-button 1 and dragging moves a whole proof subtree, while `Control`-button 2 moves a single sequent. In a theory

Figure 3.2: The Theory Hierarchy for the `finite_sets` Library

hierarchy window, `Control`-button 1 moves a theory. (Note that most proof commands will do a relayout.) Once the layout is to your liking, the `Gen PS` button will generate a PostScript file which contains the contents of the window. The filename will be briefly displayed below the buttons.

The `Config` button will bring up a menu which will let you customize the horizontal and vertical separations used by the automatic layout for the current window. These can also be customized with the resource database.

Resource name	*Default*
`pvs*proof*xSep`	10
`pvs*proof*ySep`	20
`pvs*th-hier*xSep`	50
`pvs*th-hier*ySep`	100

3.12 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
`pvs-remove-bin-files`		Remove the `.bin` files of the current context
`pvs-dont-write-bin-files`		Inhibit writing or loading of `.bin` files
`pvs-do-write-bin-files`		Allows writing and loading of `.bin` files (default)
`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. If the requested directory does not exist, and the Emacs you are running supports `make-directory`, then PVS offers to make a new one, including parent directories if necessary. If the command fails for any reason, then the current context stays the same.

The `save-context` command saves the current state of the session in the context file `.pvscontext`. In addition, any PVS files that have been typechecked will generate a binary (`.bin`) file, unless there is already a current one saved, or the `dont-write-bin-files` command has been invoked.

Under normal circumstances, binary (`.bin`) files corresponding to the specification (`.pvs`) files are updated or created as needed. These binary files contain type

information, so that loading a binary file has the same effect as typechecking the corresponding PVS file, but is generally much faster. The down side is that binary files take more disk space. If that is a problem then use the `pvs-dont-write-bin-files`, which neither loads nor creates binary files. This can be added to your `.pvsemacs` file, by adding the line

```
(pvs-dont-write-bin-files)
```

The `pvs-do-write-bin-files` undoes the effect of the `pvs-dont-write-pvs-files`, and is not needed normally. The `pvs-remove-bin-files` command may be used to remove the binary files that have been created.

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

3.13 Library Commands

Command	*Aliases*	*Function*
`load-prelude-library`		Extend the prelude from the specified context
`remove-prelude-library`		Remove the specified context from the prelude

The `load-prelude-library` command prompts for a context pathname (*i.e.*,directory), and extends the prelude with all of the theories that make up that context. Note that the theories that make up the context are defined by the `.pvscontext` file in the associated directory—there may be specification files in the same directory that are not a part of the context. The files that make up the context are typechecked if necessary, and internally the prelude is extended. All of the theories of the current context are untypechecked, as they may not typecheck the same way in the extended prelude. The PVS context is updated to reflect that the prelude has been extended. Thus the next time this context is entered, the prelude will automatically be extended (by typechecking the libraries if necessary).

This is just one of two means of gaining access to theories of a different context (short of copying them). For an alternative approach see the language guide [3].

The `remove-prelude-library` command removes the specified library from the prelude. It reverts all the theories of the current context to untypechecked to guarantee that no theories depend on the removed library. Note that the built-in prelude may not be removed this way.

3.14 Browsing

Command	*Aliases*	*Function*
`show-declaration`	`M-.`	Show declaration of symbol at cursor
`goto-declaration`	`M-'`	Go to declaration of symbol at cursor
`find-declaration`	`M-,`	Search for declarations of given symbol
`whereis-declaration-used`	`M-;`	Search for declarations which reference identifier
`whereis-identifier-used`	`C-M-;`	Search for declarations which reference identifier
`list-declarations`	`M-:`	Produce list of declarations in import chain
`show-expanded-form`	`C-.`	Show expanded form of term containing region *Arg:* also expand names from the prelude

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 the symbol or name at the cursor. Positioning the cursor on a name in the specification and typing `M-.` yields a pop-up 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. Note that when used on a record accessor it will display the declaration of the record rather than just the record field.

The `goto-declaration` command goes to the declaration associated with the symbol or name at the cursor. It pops up a buffer containing the theory associated with the declaration, and positions the cursor at the declaration.

The `find-declaration` command 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 entity denoted by a given identifier. The related `whereisidentifier-used` command generates a list of all references to a *textually identical* identifier, which may or may not result from the same declaration, due to overloading and multiple declarations. The `list-declarations` command generates a listing of all the declarations in the import chain of the specified theory. For all of these commands, the resulting buffer behaves exactly as described for `find-declaration`.

The `show-expanded-form` command displays the expanded form of the term containing the region in the `Expanded Form` buffer. Each variable, constant and operator is expanded to its full name including the theory name and its parameters, unless they are from the current theory or the prelude. With an argument, prelude names are also expanded. If the region is not defined, the current cursor location is used instead.

3.15 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
`status-importbychain`	`stb`, `C-c C-s b`	Status of theories in import by chain

These commands provide information regarding the status of the specified theories. The status information for a theory indicates whether it is parsed or typechecked, and provides the number of formulas, the number proved, the number of TCCs generated, and the number of TCCs proved. Note that the number of formulas does not include the TCCs.

The number of theory warnings and messages is also displayed. See the `show-theory-warnings` and `show-theory-messages` on page 22 for more information on these commands.

The `status-theory` command provides the status of the specified theory in the minibuffer. The `status-pvs-file`, `status-importchain`, and `statusimportbychain` commands display the information in the `PVS Status` buffer with a line for each theory. Using any of these commands on the `sum` theory yields

```
sum is typechecked: 1 formula, 1 proved; 2 TCCs, 2 proved; 0 warnings; 0 msgs
```

The `show-theory-warnings` and `show-theory-messages` (page 22) may be used to see any warnings or messages.

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

3.16 Proof Status

Command	*Aliases*	*Function*
`status-proof`	`sp`, `C-c C-s p`	Status of formula at cursor
`status-proof-theory`	`spt`	Status of formulas in theory *Arg:* provide timing information
`status-proof-pvs-file`	`spf`	Status of formulas in PVS file *Arg:* provide timing information
`status-proof-importchain`	`spi`	Status of formulas on importchain *Arg:* provide timing information
`status-proofchain`	`spc`	Proofchain of formula at cursor
`status-proofchain-theory`	`spct`	Proofchain of specified theory
`status-proofchain-pvs-file`	`spcf`	Proofchain of current file
`status-proofchain-importchain`	`spci`	Proofchain of importchain

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`, `untried`, `unfinished`, or `unchecked`. Untried means that the proof has not yet been attempted. 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 commands `status-proof-theory`, `status-proof-pvs-file`, and `status-proof-importchain` use the `PVS Status` buffer to display the proof status for all of the formulas within the theory, PVS file, or the import chain respectively. With an argument, these commands display timing information as well.

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 (for example, using lemma $\mathcal{A}$ to prove $\mathcal{B}$ and vice-versa). Because judgements are used implicitly, they may be included in the analysis even if they are not actually used.

The commands `status-proofchain-theory`, `status-proofchain-pvs-file`, and `status-proofchain-importchain` provide the proof chain analysis for each formula of the theory, PVS file, and import chain of the specified theory, respectively, in the `PVS Status` buffer.

3.17 Environment Commands

Command	*Aliases*	*Function*
`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
`pvs-log`		Display the `PVS Log` buffer
`status-display`		Display the `PVS Status` buffer
`pvs-status`		Find out if PVS is busy
`pvs`		Start the PVS process
`pvs-load-patches`		Load new PVS patches

The `whereis-pvs` command is used to determine the directory where the PVS system resides. This is useful for finding the example specifications and files that are part of the PVS distribution.

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.

Most of the messages that appear in the minibuffer are kept in the `PVS Log` buffer, stamped with the time. The `pvs-log` command simply pops up the `PVS Log` buffer so that you may view it.

The `status-display` command simply displays the `PVS Status` buffer. This is the buffer used for most of the status commands.

The `pvs` command is what is used to actually start PVS after the Emacs files have all been loaded. It is provided as a user command because there are times when the PVS lisp subprocess has been killed and you wish to start up that process while keeping the same Emacs session.

The `pvs-load-patches` command reloads the patches. This is useful when new patches have been installed, and you wish to load them without exiting the system and starting up again.

3.18 Interrupting PVS

Command	*Aliases*	*Function*
`pvs-status`		Find out if Lisp is busy
`pvs-interrupt-subjob`	`C-c C-c`	Interrupt PVS (lisp) process
`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: background commands 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—usually a beep and `Quit` appears in the minibuffer. This command interrupts Emacs, but has no effect on any PVS commands that are still running. After Emacs responds go to the end of the `*pvs*` buffer, and type `C-c C-c`. If Lisp is able to respond, you should see the message

```
Error: Received signal number 2 (Keyboard interrupt)
  [condition type: INTERRUPT-SIGNAL]

Restart actions (select using :continue):
 0: continue computation
 1: Return to Top Level (an "abort" restart)
[1c] PVS(22):
```

You can then type `:continue 0` to keep going as it was never interrupted, `(restore)` if you are in the middle of an ongoing proof and want to continue from the state prior to the last *atomic* prover command (see the prover guide [7]), or `:continue 1` or `:reset` to abort to the top level.

The Lisp process may not be able to respond to the interrupt right away, especially if it has started garbage collection. If you really want to interrupt it, type more `C-c C-c` interrupts; after about six of them it is supposed to respond regardless. This is not recommended in general as it can leave the Lisp process in an unstable state. Unfortunately, we have seen Allegro Common Lisp get into a state where it is completely unresponsive, even after several interrupts and waiting for hours for a response. This is rare, but if it happens the only recourse is to kill the process and start up a new PVS session. See below for how to do this while allowing Emacs to continue.

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 as if you typed `C-c C-c`, as described above. If `reset-pvs` is issued somewhere other than the `*pvs*` buffer, you are 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.

If you wish to kill the PVS Lisp process, while keeping your current Emacs session, simply go to the `*pvs*` buffer and kill it `kill-buffer C-x k`, then run `pvs` and the PVS Lisp process will restart. All your other Emacs buffers are unaffected by this.

Chapter 4

Customizing PVS

PVS is a complex system, and utilizes many subsystems, including Lisp, Emacs, the X window system, and Tcl/Tk. You can control aspects of these subsystems by a combination of command-line arguments, environment variables, and various files. In this section we discuss some aspects of the customization of these subsystems as they relate to PVS.

4.1 Invoking PVS

PVS is invoked from a shell script named `pvs` in the PVS directory—this is a text file, and may be examined or copied and modified to suit your taste. The script is a Bourne shell script, and requires `/bin/sh` to execute correctly.[1]

PVS accepts a number of command-line arguments, as well as using environment variables. The command-line arguments specific to PVS are

`-h | -help | --help` - Print a brief description of the command line options and exit.

`-lisp` ***lispname*** - Specifies which lisp to use. The lisp image used for PVS is then `pvs-`*`lispname`*, which should be located in a directory determined by the machine architecture. See Section 4.3, page 66 for details.

`-redhat` ***redhat-release*** - Specifies the release of the Redhat Linux operating system you are using (different PVS binaries are required for libc5 and glibc C libraries). PVS attempts to discover this for itself, but if the wrong binary is chosen you can specify `4` or `5` using this argument. Note that Redhat 6 uses the glibc libraries, which corresponds to the value `5`.

`-runtime` - This is only needed at SRI, where the development version of the system is used by default. With this option the runtime image is used instead.

[1] On some systems, `/bin/sh` is linked to the `bash` shell; this works as well.

`-emacs` ***emacsname*** - Specifies the Emacs to use; see below for details.

`-decision-procedures` ***new|old*** - Sets the default decision procedures to be used in proofs. See Section 3.5.3, page 28 for details.

`-force-decision-procedures` ***new|old*** - Forces the chosen decision procedure to be used regardless of the default decision procedure setting or which decision procedures were used in developing a proof. Note that with this option there is no way to switch between the new and old decision procedures.

`-nw` - Tells Emacs not to use its special interface to X.

`-batch` - Run PVS in batch mode. See chapter 5, page 69 for details.

`-timeout:` In batch mode, this causes typechecking and individual proof attempts to be interrupted after the given number of seconds.

`-nobg:` Normally PVS starts in the background (with the & control operator). This starts it in the foreground.

`-raw:` This runs PVS without Emacs. This is only useful for front ends, which must do the same initialization as done by the Emacs interface.

`-v` ***number*** - Specifies verbosity level for PVS batch mode. See Chapter 5, page 69 for details.

`-q` - A standard emacs option to inhibit loading of the users `.emacs` file, but extended in PVS to inhibit loading of the users `.pvsemacs`, `.pvsxemacs-options` and `.pvs.lisp` files on startup.

`-patchlevel` ***level*** - Specifies which patch files to load. Level `none` loads no patch files. Level `rel` loads the file `patch2` from your PVS directory, which usually contains the release versions of PVS patches. Other valid levels are `test` (loads the files `patch2` and `patch2-test`) and `exp` (loads the files `patch2`, `patch2-test` and `patch2-exp`). This option is mainly used for PVS development.

Any other command-line arguments are passed directly to the underlying Emacs, including those for X windows—these are discussed below.

In addition, the PVS script uses the environment variables `PVSLISP`, `PVSEMACS`, and `PVSXINIT`, which may be set in your `.cshrc` or `.login` file to specify the defaults you prefer. If both the environment variable and the corresponding command-line argument are given, the command-line argument takes precedence. The `PVSXINIT` variable is described in Section 4.4, page 66.

4.2 Emacs

The PVS system uses Emacs as its user interface, and provides a number of files that extend Emacs for use with PVS. For historical reasons, there are currently a number of Emacs editors available. Because we wanted PVS to be freely available, we have chosen to concentrate on just Gnu Emacs and XEmacs, which are also freely available. To find out what version of Emacs you are using, start up Emacs and type `M-x emacs-version`. We try to keep up with new releases of emacs and if necessary patch files will be made available to support the new Emacs.

By default, the system uses `emacs`, which is assumed to be in your path when you start up PVS. You may specify a different Emacs as specified above. When you start PVS, is assumed (in order to supply X resources in the correct format) that if the name of the emacs command contains the character "`x`" then you are using XEmacs.

PVS loads your `~/.emacs` file first (assuming you have not specified the `-q` option as described on page 64), followed by *PVSPATH*`/emacs/go-pvs.el`, which determines which version of emacs you are running and then loads the rest of the PVS emacs files, including ILISP. At this point you may receive an error from PVS saying that your Emacs version is unknown. PVS does not support Emacs 18 (or earlier), but we try to keep up with new Emacs versions as they are released. Finally, the `~/.pvsemacs` is loaded. If you are running XEmacs, the `.pvsemacs` file will load XEmacs options from the `.pvsxemacs-options` file instead of the standard `.xemacs-options` file, as some are incompatible with standard XEmacs.

In loading the files in this order, PVS functions and key bindings will overwrite any conflicting ones defined in your `.emacs` file. `.pvsemacs` is the file to use to override the key bindings and definitions given by PVS. This approach was taken to ensure that the behavior of PVS by default follows the user guide, but can be readily modified to suit your taste.

One file that is worth noting is the *PVSPATH*`/emacs/emacs-src/pvs-abbreviations.el` file, where the abbreviations for many of the PVS commands are given. You may define your own abbreviations for commands you use a lot that don't currently have abbreviations, by adding the appropriate lines in your `.pvsemacs` file. For example, adding

```
(pvs-abbreviate 'show-tccs 'st)
```

will make `M-x st` an abbreviation for `M-x show-tccs` in addition to those already defined. Note that you cannot redefine a name which is already in use.

If you would like to byte-compile your Emacs customizations, create a separate file, byte-compile it, and load it from your `.pvsemacs`. Generally the kinds of forms provided in a `.pvsemacs` file are simply variable settings and minor function definitions, and are not worth byte-compiling. It is only worthwhile if a function is being (re)defined that will be invoked noninteractively and frequently, for example, if you want to modify the way the process filter works.

4.3 The PVS Image

PVS currently runs under Allegro Common Lisp on a number of different platforms. PVS is provided as a Common Lisp image, meaning that it includes both the Lisp runtime system and the PVS programs, so you do not need to have Allegro installed on your system.

There is usually just one PVS image available at a given site, and if the system is properly installed, nothing further needs to be done. If more than one image is available, and the default one is not the desired one, then it can be specified using either command-line arguments or environment variables. Invoking PVS with

```
pvs -lisp lucid -image pvs-lucid-sun4
```

will use the `pvs-lucid-sun4` image. Note that `-lisp lucid` must be specified, so that the Emacs interface can be set up properly. For linux, also see the `-redhat` option on page 63.

Alternatively, the environment variables `PVSLISP` and `PVSIMAGE` may be set to get the same effect. Note that command-line arguments take precedence.

After the PVS lisp image has started, it loads in the patch files as specified by the `-patchlevel` argument and then loads the file `.pvs.lisp` from your home directory. This file can be used to provide lisp customizations on a per user basis and for overriding definitions in the patch file.

4.4 Window Systems

PVS was built primarily for the X window system, though it can be run from a terminal interface. When run under X windows with the supported versions of Emacs, the resource name will be set to `PVS`, and the window and icon names will be set to `PVS@`*host*, where *host* is the host name of the system on which PVS was invoked. These may be modified by adding command-line arguments or setting the `PVSXINIT` environment variable.

You may customize the title and icon names by defining the function `pvs-title-string` in your `.pvsemacs` file taking no arguments and returning a string to be used as the title. This function is invoked at startup, and whenever the context is changed. For example, the following provides the name of the pvs path, the patch level (`N` for none, `R` for released, `T` for test, and `E` for experimental), the hostname, and the last two components of the current context.

```
(defun pvs-title-string ()
  (format "%s%s%s:%s/"
      (trailing-components pvs-path 1)
    (cond ((stringp (cadddr *pvs-version-information*)) "E")
          ((stringp (caddr *pvs-version-information*)) "T")
          ((stringp (cadr *pvs-version-information*)) "R")
          (t "N"))
```

```
(let ((host (car (string-split ?. (getenv "HOSTNAME")))))
  (format "@%s" host))
(trailing-components *pvs-current-directory* 2)))
```

For example, this might generate pvs2.3N@photon:lib/finite_sets/.

It is difficult to get a single setting for all of the Emacs versions; the following table gives the arguments needed to set the resource, window, and icon names for the various versions.

Emacs	Resource	Window	Icon
emacs19	`-rn`	`-name`	
emacs19.29 (and later, including emacs20)	`-name`		
xemacs	`-name`	`-wn`	`-in`

Note: in emacs19, if `-rn` is not given, then `-name` is used for the resource name as well. Emacs19.29 and later will give an error if the `-rn` argument is given.

The window name is the name used in the title bar of the PVS window, the icon name is the name used in the icon, and the resource name is the name referred to in the `.Xdefault` or `.Xresource` file that controls the defaults for X clients. An example entry for PVS in one of these files might be

```
! PVS defaults
PVS.geometry: 80x63-0-0
PVS*pointerColor: Red
PVS*Font: *courier-medium-r-normal--12*
```

See the man pages for `X` and `emacs`, as well as the news and info pages for the version of Emacs you are using for more details on X resources.

The `PVSXINIT` environment variable may be set[2] to a string of command-line arguments that are then appended to the defaults described above. You can also change the default resource, window, and icon names, simply by adding them to this variable (or by including them in the command-line arguments). Note that you should make certain that the version of Emacs you are using matches the command-line arguments as shown in the footnote. You can tell that there is a mismatch when you start up PVS and find buffers with names matching command-line arguments, *e.g.*, `-in` or `PVS@acorn`.

[2]Generally environment variables are set in your shell startup file, *e.g.*, `.profile` or `.cshrc`.

Chapter 5

Running PVS in Batch Mode

To support validation runs, PVS supports a *batch mode*, which means that specifications and proofs being processed are not displayed. In batch mode there is no direct interaction with PVS; it simply processes whatever files or functions are provided and terminates after completing the last of them. PVS batch mode is built directly on the underlying Emacs batch mode described in Section A.2 of the GNU Emacs Manual [8].

If PVS is invoked in batch mode from a shell, then it may be interrupted (using `C-c`), suspended (`C-z`), or run as a background job. The output may be redirected to a file or piped to another command.[1]

To run PVS in batch mode, simply include the '`-batch`' option in your call to PVS. In addition, you should include one or more Emacs source files and/or a Emacs or PVS function to run, using the '`-l`' or '`-load`' option to load a file, and the '`-f`' or '`-funcall`' option for a function. For example:

```
pvs -batch -l test.el
pvs -batch -f pvs-version
```

Note that the function option is severely limited, as it only allows a function name. This means that only functions that take no arguments may be provided, for example, `pvs-version` or `whereis-pvs`.

Running PVS in batch mode does cause your `~/.emacs` file to be loaded, in contrast to running Emacs in batch mode. If you want to suppress the loading of your `.emacs`, include the '`-q`' option, for example:

```
pvs -batch -q -l test.el
```

In batch mode PVS suppresses messages by default, though you can print your own messages. You can also control the amount of printout using the verbose option, '`-v`', and providing a level number ranging from `0` to `3`. The following table summarizes the levels.

[1] The Emacs batch option actually sends its output to `stderr` rather than `stdout`; the `pvs` shell script redirects this to `stdout`, as this is generally more useful and easier to work with.

```
(pvs-message "Proving stamps2")
(change-context "~/pvs/test")
(let ((current-prefix-arg t))
  (prove-pvs-file "stamps2"))
```

Figure 5.1: Batch File Example

level	**printout**
0	User-defined `pvs-message`s only
1	Messages normally sent to the echo area and PVS errors
2	Status buffers
3	Proof replays

The `pvs-message` function is much like the Emacs `message` function, but the message will get printed no matter what the level number is. If you want to print out only when the level is `1` or higher, use `message` instead. Both take a control string and an arbitrary number of arguments. An example is shown in Figure 5.1.

The file provided to the load option ('`-l`' or '`-load`') is an ordinary Emacs file, and usually has an `.el` extension. Inside this file you can invoke any PVS commands you want, though many of them only make sense interactively. For example, the `prove` command expects the cursor to be positioned at a given formula, which is difficult (though not impossible) to do in batch mode. The various Tcl/Tk commands available will not run at all because there is no X display associated with PVS running in batch mode. The most useful commands to run in batch mode are listed in Table 5.1. In that table, a *filename* is a PVS file name without the `.pvs` extension, a *theoryname* is the name of a theory in the current context, and a *directory* is a Unix pathname. These must all be given as strings (enclosed in double quotes). The *length* and *depth* arguments are integers, and do not need any special treatment. PVS Emacs commands are given in Emacs lisp syntax; for example,

```
(parse "foo")
(set-print-depth 3)
(save-context)
```

An example of the contents of a batch file is shown in Figure 5.1. This file consists of three commands. It prints the message "`Proving stamps2`", changes to the `~/pvs/test` context, and then reruns all the proofs of the specification file `stamps2.pvs`. Note that `current-prefix-arg` is set to `t` to ensure that the proofs are retried. This is equivalent to using `C-u` interactively. While PVS is running in batch mode, two possible kinds of error may be encountered. An Emacs error comes from badly formed batch files or nonexistent functions. These errors will cause the system to stop immediately, and the error will be displayed if the level number is

Command	**Arguments**
`parse`	*filename*
`typecheck`	*filename*
`typecheck-importchain`	*filename*
`typecheck-prove`	*filename*
`typecheck-prove-importchain`	*filename*
`prove-theory`	*theoryname*
`prove-pvs-file`	*filename*
`prove-importchain`	*theoryname*
`set-print-depth`	*depth*
`set-print-length`	*length*
`set-rewrite-depth`	*depth*
`set-rewrite-length`	*length*
`alltt-theory`	*theoryname*
`alltt-pvs-file`	*filename*
`alltt-importchain`	*theoryname*
`latex-theory`	*theoryname*
`latex-pvs-file`	*filename*
`latex-importchain`	*theoryname*
`latex-set-linelength`	*length*
`change-context`	*directory*
`save-context`	
`pvs-remove-bin-files`	
`pvs-dont-write-bin-files`	
`pvs-do-write-bin-files`	
`status-theory`	*theoryname*
`status-pvs-file`	*filename*
`status-importchain`	*theoryname*
`status-importbychain`	*theoryname*
`status-proof-theory`	*theoryname*
`status-proof-pvs-file`	*filename*
`status-proof-importchain`	*theoryname*
`status-proofchain-theory`	*theoryname*
`status-proofchain-pvs-file`	*filename*

Table 5.1: Commands available for validation

```
(pvs-validate
  "stamps2.log"
  "~/pvs/test"
  (pvs-message "Proving stamps2")
  (set-rewrite-depth 0)
  (let ((current-prefix-arg t))
    (prove-pvs-file "stamps2")))
```

Figure 5.2: Example Use of `pvs-validate`

nonzero. A PVS error generates an error message (for a nonzero level number) and abandons the current command, but allows the system to go on to the next command.

If an emacs error is encountered that reports 'entering debugger' when run with verbosity level 3, the full commands of the emacs debugger are available[2]. A useful command to discover where your validation script encountered the error is:

```
e (progn (set-buffer "*Backtrace*")(buffer-string))
```

Another potential pitfall is that PVS may appear to hang. If this happens, try running with verbosity level 3 as it is likely that PVS is awaiting user input (usually a yes/no). You may respond to such prompts from the shell.

5.1 Validation Runs

A validation run is simply a batch run in which the `pvs-validate` macro is used in the batch file. Given a *log file* name, a directory, and a sequence of PVS Emacs commands, `pvs-validate` will change context to the specified directory and run the commands, collecting the output into the log file. It then compares the new results to the previous ones, and reports whether there were any significant differences. An example of the use of `pvs-validate` is shown in Figure 5.2.

Any number of `pvs-validate` forms may be used, and they may be freely intermixed with other Emacs or PVS commands. When the sequence of commands associated with an invocation of `pvs-validate` is complete, the log file is compared to the previous version, if it exists. At this point PVS will report one of three messages:

- `Nothing to compare` *`log`* `to` - the log file has not been generated before this run.

[2]See the Emacs manual[8] for details.

- `No significant changes in log` - the current run does not differ significantly from the last one. A significant difference is one that involves more than timing differences. For example, the message `proved in 27 seconds` is not significantly different from `proved in 31 seconds`.

- `Differences found since last run` - differences were found. The following line indicates the two log files that should be compared to see where they differ.

This is normally all the output provided by PVS while processing a `pvs-validate` macro, though you can get more information by including the '`-v`' option as described above.

With minor exceptions, the log files contain the same information as obtained with the '`-v 3`' option, but only for the commands of the given `pvs-validate` macro. In comparing log files, timings are ignored.[3]

When a difference is reported, you can find out what the differences actually are by starting up (an interactive) PVS, and bringing up the two files in a split window.[4] Then use `M-x pvs-compare-validation-windows`, which works much like the Emacs `compare-windows` command, and will position the cursor at the point where the two files differ. Again, differences in timing are ignored. After analyzing the difference, you can move the pointer in each buffer to the next position where they are the same, and run `M-x pvs-compare-validation-windows` again to get to the next difference. In this way you can quickly analyze all the differences since the last validation run.

The log files are maintained under RCS [9], using the Emacs *Version Control* interface [8]. The first time a validation run is made from a given directory, an RCS subdirectory is created to keep the directory from being cluttered with RCS files. If this is the first validation run for a given log file, then the log file is created and registered to RCS. In subsequent runs, the log file is compared to the previous version, which will have a name including the version number, for example, `stamps2.log.~1.8~`. If the comparison shows no significant differences, then the file is subsequently deleted.

Note that the log files are all kept in the directory from which PVS was run, and changing context will not affect that. This makes it easy to maintain a single directory that controls the validation for several different contexts.

5.2 Example Validation Run

Here is an example of a validation run for a very simple specification.

[3] In the future we may want to compare timings and report those that are significantly different, but in order for this to work properly we must get CPU times rather than real times, and make sure that we are keeping track of the machine used for the previous validation run For now we are only concerned with functional correctness.

[4] In detail, start up PVS, use `C-x C-f` to visit the first file, use `C-x 2` to split the window vertically, and then use `C-x C-f` again to bring in the second file.

5.2.1 The Specification

The specification is in the file `stamps.pvs`:

```
stamps : THEORY
 BEGIN
  i, n3, n5: VAR nat
  stamps: LEMMA (FORALL i: (EXISTS n3, n5: i+8 = 3*n3 + 5*n5))
 END stamps
```

5.2.2 The Validation File

The file `stamps.el` has the validation commands. In this case we are simply going to reprove the formulas of the specification file (there is only one):

```
(pvs-validate
 "stamps.log"
 "~/pvs-specs/validation"
 (pvs-message "Proving stamps")
 (let ((current-prefix-arg t))
   (prove-pvs-file "stamps")))
```

5.2.3 The Validation Run

Here is the validation run, with level number 1. This shows the messages that normally appear in the echo area at the bottom of the Emacs window (these messages are sent to `stdout`):

```
% ./pvs -batch -l stamps.el -v 1
Started initializing ILISP
Finished initializing pvsallegro
Loading compiled patch file /project/pvs/patch2.fasl
Context changed to ~/pvs-specs/validation/
Checking out ~/pvs-specs/validation/stamps.log...
Checking out ~/pvs-specs/validation/stamps.log...done
PVS Version 2.3 (No patches loaded)
Context changed to ~/pvs-specs/validation/
Proving stamps
Parsing stamps
stamps parsed in 0.02 seconds
Typechecking stamps
stamps typechecked in 0.02s: No TCCs generated
Rerunning proof of stamps
Using old decision procedures
```

```
Proving stamps.stamps.
Proving stamps.stamps..
Proving stamps.stamps...
Proving stamps.stamps....
Proving stamps.stamps.....
Proving stamps.stamps......
Proving stamps.stamps.......
stamps proved in 2.20 real, 0.58 cpu seconds
stamps: 1 proofs attempted, 1 proved in 2.20 real, 0.58 cpu seconds
Checking out ~/pvs-specs/validation/stamps.log.~1.3~...
Checking out ~/pvs-specs/validation/stamps.log.~1.3~...done
No significant changes in stamps.log
Checking in ~/pvs-specs/validation/stamps.log...
Checking in ~/pvs-specs/validation/stamps.log...done
```

5.2.4 The Log File

The resulting log file `stamps.log` is shown here. This will be used for comparison to in subsequent validation runs.

```
PVS Version 2.3 (No patches loaded)
Context changed to ~/pvs-specs/validation/
Proving stamps
Restoring theories from stamps.bin
Restored file stamps (stamps) in 0.57 seconds
Rerunning proof of stamps
Using old decision procedures

stamps :

  |-------
{1}    (FORALL i: (EXISTS n3, n5: i + 8 = 3 * n3 + 5 * n5))

Proving stamps.stamps.
Rerunning step: (INDUCT "i")
Proving stamps.stamps..
Inducting on i,
this yields  2 subgoals:
stamps.1 :

  |-------
{1}    (EXISTS (n3: nat), (n5: nat): 0 + 8 = 3 * n3 + 5 * n5)

Rerunning step: (INST + 1 1)
```

```
Instantiating the top quantifier in + with the terms:
 1, 1,
this simplifies to:
stamps.1 :

  |-------
{1}    0 + 8 = 3 * 1 + 5 * 1

Rerunning step: (ASSERT)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of stamps.1.

stamps.2 :

  |-------
{1}    (FORALL (j: nat):
         (EXISTS (n3: nat), (n5: nat): j + 8 = 3 * n3 + 5 * n5)
             IMPLIES (EXISTS (n3: nat), (n5: nat):
                         j + 1 + 8 = 3 * n3 + 5 * n5))

Rerunning step: (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
stamps.2 :

{-1}    j!1 + 8 = 3 * n3!1 + 5 * n5!1
  |-------
{1}    (EXISTS (n3: nat), (n5: nat): j!1 + 1 + 8 = 3 * n3 + 5 * n5)

Rerunning step: (CASE "n5!1 = 0")
Case splitting on
Proving stamps.stamps...
   n5!1 = 0,
this yields  2 subgoals:
stamps.2.1 :

{-1}    n5!1 = 0
[-2]    j!1 + 8 = 3 * n3!1 + 5 * n5!1
  |-------
[1]    (EXISTS (n3: nat), (n5: nat): j!1 + 1 + 8 = 3 * n3 + 5 * n5)

Proving stamps.stamps....
Rerunning step: (INST + "n3!1 - 3" 2)
Instantiating the top quantifier in + with the terms:
```

```
Proving stamps.stamps.....
 n3!1 - 3, 2,
this yields  2 subgoals:
stamps.2.1.1 :

[-1]    n5!1 = 0
[-2]    j!1 + 8 = 3 * n3!1 + 5 * n5!1
  |-------
{1}    j!1 + 1 + 8 = 3 * (n3!1 - 3) + 5 * 2

Rerunning step: (ASSERT)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of stamps.2.1.1.

stamps.2.1.2 (TCC):

[-1]    n5!1 = 0
[-2]    j!1 + 8 = 3 * n3!1 + 5 * n5!1
  |-------
{1}    n3!1 - 3 >= 0

Rerunning step: (ASSERT)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of stamps.2.1.2.


This completes the proof of stamps.2.1.

stamps.2.2 :

[-1]    j!1 + 8 = 3 * n3!1 + 5 * n5!1
  |-------
{1}    n5!1 = 0
[2]    (EXISTS (n3: nat), (n5: nat): j!1 + 1 + 8 = 3 * n3 + 5 * n5)

Proving stamps.stamps......
Rerunning step: (INST + "n3!1 + 2" "n5!1 - 1")
Instantiating the top quantifier in + with the terms:
Proving stamps.stamps.......
 n3!1 + 2, n5!1 - 1,
this yields  2 subgoals:
stamps.2.2.1 :
```

```
[-1]    j!1 + 8 = 3 * n3!1 + 5 * n5!1
  |-------
[1]    n5!1 = 0
{2}    j!1 + 1 + 8 = 3 * (n3!1 + 2) + 5 * (n5!1 - 1)

Rerunning step: (ASSERT)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of stamps.2.2.1.

stamps.2.2.2 (TCC):

[-1]    j!1 + 8 = 3 * n3!1 + 5 * n5!1
  |-------
{1}    n5!1 - 1 >= 0
[2]    n5!1 = 0

Rerunning step: (ASSERT)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of stamps.2.2.2.


This completes the proof of stamps.2.2.


This completes the proof of stamps.2.

Q.E.D.
stamps proved in 19 seconds
stamps: 1 proofs attempted, 1 proved in 19 seconds


 Proof summary for theory stamps
    stamps.....................................proved - complete
    Theory totals: 1 formulas, 1 attempted, 1 succeeded.

Grand Totals: 1 proofs, 1 attempted, 1 succeeded.
```

Appendix A

Introduction to Emacs

The PVS system uses the GNU Emacs system as its user interface. To make effective use of PVS, you must become familiar with at least the basic Emacs commands. This section provides an introduction to Emacs that should allow you to get started in PVS right away. This Appendix introduces enough of the basic ideas and commands of Emacs to use PVS, but to become effective you really should consult the GNU Emacs Manual [8]. It is also quite helpful to run through the online tutorial. To do this, start up PVS or Emacs, type `C-h t`, and follow the instructions.

Emacs provides the primary interface to the PVS system. We chose Emacs as our interface for a number of reasons. First, it is freely available, and runs on a large number of different platforms. It is also quite popular; on Unix systems the `vi` editor is probably the only editor that is used more than Emacs, but it is too limited to use as a general-purpose interface. In particular, it has no support for running subprocesses.

Emacs is an extremely flexible editor, and includes a built in programming language (Emacs Lisp) which makes it easy to increase its functionality. There is a cost associated with this. First, Emacs is rather large, and takes longer to start up than smaller editors such as `vi`. Emacs is also quite complex, with a large number of commands and associated key bindings that are not easy to learn.

Emacs is significantly different than other editors. In most editors, you start the editor, get a file, make some changes, save the file, and exit. There is a tendency to think in terms of "leaving" the current file in order to go to the next. To handle multiple files in a single session usually requires extra care and some specialized commands. For example, `vi` can only focus on one file at a time, with one alternate.

In Emacs multiple buffers may be open at once, and as many can be made visible as your screen allows. Unlike other editors, buffers are not all associated with files. It is not unusual to have over a hundred buffers associated with a single Emacs session. It is also quite normal to have the same Emacs session up for weeks at a time.[1] When you are done editing and saving a given file, you do not exit from that buffer, you

[1] Some people have even been known to use Emacs as their shell.

simply go on to the next one.

Unlike `vi`, there is no command mode. By default an Emacs buffer is in insert mode, and most keys on the keyboard simply insert themselves. Emacs has a large number of interactive commands, any of which may be bound to a key or key sequence.[2] Any interactive command may be invoked by typing `M-x` followed by the command. Recall that `M-x` is gotten by holding down the `Meta-` key and typing an `x`. If you don't have a key labeled `Meta-`, then look for and try the `Alt` or ◇ keys. If you really don't have a `Meta-` key, then the `Esc` key will do, but in this case you must release the `Esc` key before typing the `x`.

Commands may be bound to key sequences, in order to make typing easier. For example, to page down repeatedly by typing `M-x scroll-up` over and over would get quite tedious, so the key sequence `C-v` was bound to this command. This and most of the key bindings of Emacs are not particularly mnemonic, but once learned they are extremely effective. With a little practice you will find that you don't think about what key sequence is needed to get a particular effect—your hands just do it automatically.

Each buffer in Emacs has an associated major mode, and any number of minor modes. The major mode indicates what kind of a buffer it is, and generally defines the key bindings and functions associated with the buffer. This is usually determined from the file extension, so for example the file `foo.pvs` is in `pvs-mode`, while a file `foo.c` would normally be in `c-mode`. Minor modes modify the major mode. Examples include `auto-fill-mode` and `overwrite-mode`.

When you start up PVS, you will see the `PVS Welcome` buffer, which takes up most of the window. Toward the bottom of the window you will see a line in inverse video; this is the mode line. The last line of the window is the minibuffer. If you are running Emacs version 19 (or later) under X windows, then you will see a menu line at the top of the window, and a scroll bar to the right. If you display more than one buffer in the window, then the bottom of each buffer will have a mode line displaying information for that buffer. There will still be only one minibuffer, however.[3]

The mode line provides information relating to the buffer above it. The first five characters indicate whether the buffer is read-only, and whether the buffer has changed relative to the file. If you see `---%%-`, then the file is read-only, and you will not be allowed to modify it. Sometimes this is set when you have copied a file from somewhere else, and you think you should be able to make modifications. In that case, use the `toggle-read-only` command, make your changes, and save the file. Emacs may still ask whether it should try to save the file anyway, go ahead and answer yes in this case.

If the mode line is 5 dashes (`-----`), then the file can be modified but has not yet

[2] It turns out that typing a letter key actually invokes the command `self-insert-command`.

[3] In Emacs 19 (and later versions), it is possible to have multiple windows, called frames, associated with a single Emacs session. In this case, each frame by default has its own copy of the minibuffer. See the Emacs manual for more details.

been changed. Once modified, the mode line changes to `--**-`. If you did not intend to modify the file, then use the `undo` commands described below to undo your change. If there are a few changes, you may need to repeat the `undo` command until they are all backed out. If there are a lot of changes, then `M-x revert-buffer` may be used to restore the buffer from the original file. The other information in the mode line is the buffer name, possibly the time, the mode of the buffer in parentheses, and the amount of the buffer currently displayed. Like everything else in Emacs, the mode line is customizable; see the Emacs manual for details.

The minibuffer is used to display messages, echo longer commands as they are typed in, and prompt for arguments. Many of these arguments support *completion*, which means that you can type the start of an argument and type a `TAB` to have it automatically filled in. Emacs will fill in as much as is unique, and then wait for more input. If it is ambiguous already, Emacs will pop up a buffer with the possible completions in it. You can force it to show all possible completions by typing a `?`. Not all arguments support completion, but it is usually worthwhile to try typing a `TAB` after typing the start of an argument to see if completion is supported; if it is then you will either get a pop up buffer or a (partial) completion of what you typed. Otherwise you will simply get a `TAB` inserted.

Each buffer has associated with it a current *region*, which is defined by two different locations within the buffer, called *point* and *mark*. Point is normally the cursor position, so any of the cursor motion commands automatically move point. Mark is not directly displayed; it is set by command, and does not move until another mark setting command is issued. There are a large number of Emacs commands that work on regions, though by far the most common usage is for cutting and pasting operations.

A.1 Leaving Emacs

Command	*Aliases*	*Function*
`save-buffers-kill-emacs`	`C-x C-c`	Kill Emacs

This command exits Emacs, after first prompting whether to save each modified file.

A.2 Getting Help

Command	*Aliases*	*Function*
`info`	`C-h i`	Read Emacs documentation
`help-with-tutorial`	`C-h t`	Display the Emacs tutorial
`command-apropos`	`C-h a`	Show commands matching a string
`describe-key`	`C-h k`	Display name and documentation a key runs
`describe-function`	`C-h f`	Display documentation for function
`describe-bindings`	`C-h b`	Display a table of key bindings

These commands provide help. When you type the `C-h` prefix key, you are prompted for the next key, and can type `?` to find out all the possibilities—only a few are described here.

The `info` command brings up a buffer containing the Emacs online documentation. Type `m` followed by a topic name to view the info page for that topic, or click mouse button 2 over the highlighted name.

The `help-with-tutorial` command brings up an Emacs tutorial. This tutorial is interactive, inviting you to try out the commands as it describes them. If you are new to Emacs, you should try to go through this at least once.

The `command-apropos` command displays a list of those commands whose names contain a specified substring. This is helpful if you know only part of a command name, or suspect there is some command available for performing some task, but do not know what it might be named. For example, you might do an `C-h a` on `mail` to find out what mail commands are available. If you know the beginning of a command, it is usually better to simply start typing the command and use the completion mechanism.

The `describe-key` and `describe-function` commands provide the same information, but one prompts for a key and the other for a command (with completion). The key is not necessarily a single keystroke, as some keystrokes are defined to be prefix keys. In this case the key typed so far will be displayed in the minibuffer, and the function description will not be given until a completed key sequence has been typed.

The `describe-bindings` command displays the key bindings in effect in a separate buffer. Many of these key bindings are specific to the buffer mode, so issuing this command from different buffers will generally lead to different results.

A.3 Files

Command	*Aliases*	*Function*
`find-file`	`C-x C-f`	Read a file into Emacs
`save-buffer`	`C-x C-s`	Save a file to disk

The file commands are needed to read a file into Emacs and save the changes. The `find-file` creates a new buffer with the same name as the file and reads the file contents into it. Completion is available on the file name, including the directory. If the file does not exist, then an empty buffer is created. Note that the buffer is not the same as the file, and changes made to the buffer are not reflected in the file until the file is saved.

The `save-buffer` command saves the current buffer to file. If the current buffer is not associated with a file, you are prompted to give a file name.

A.4 Buffers

Command	*Aliases*	*Function*
`switch-to-buffer`	`C-x b`	Select another buffer
`list-buffers`	`C-x C-b`	List all buffers
`kill-buffer`	`C-x k`	Kill a buffer

The `switch-to-buffer` command is used to switch control from one buffer to another. When you type the command, you will be prompted for a new buffer to switch to, and a default will be given. If the default is the right one, simply type the return key. Otherwise type in the name of the buffer you desire. Completion is available. If the buffer specified does not already exist, then it is created.

The `kill-buffer` command is used to remove a buffer. Completion is available. Note that some buffers have processes associated with them, and killing that buffer also kills the associated process. In particular, the `*pvs*` buffer is associated with the PVS process.

The `list-buffers` command lists all the buffers currently available, along with an indication of whether the buffer has changed, its size, its major mode, and its associated file, if any.

A.5 Cursor Motion commands

Command	*Aliases*	*Function*
`forward-char`	`C-f`	Move forward one character
`backward-char`	`C-b`	Move backward one character
`forward-word`	`C-f`	Move forward one word
`backward-word`	`C-b`	Move backward one word
`next-line`	`C-n`	Move down one line vertically
`previous-line`	`C-p`	Move up one line vertically
`beginning-of-line`	`C-a`	Move to the beginning of the line
`end-of-line`	`C-e`	Move to the end of the line
`beginning-of-buffer`	`M-<`	Move to the beginning of the buffer
`end-of-buffer`	`M-<`	Move to the end of the buffer

These are largely self explanatory; the best way to get used to what they do is to simply try them out. Note that, depending on your Emacs environment, you may have appropriate key bindings for the arrow, Home, PageUp, etc. keys.[4]

[4] As described above, you can find out what these are bound to by typing `C-h k` followed by the key.

A.6 Error Recovery

Command	*Aliases*	*Function*
`keyboard-quit`	`C-g`	Abort partially typed or executing command
`undo`	`C-x u`, `C-_`	Undo one batch of changes
`revert-buffer`		Revert the buffer to the file contents
`recenter`	`C-l`	Redraw garbaged screen

`C-g` is used if you start to type a command and change your mind, or a command is running and you want to abort it. Sometimes it takes two or three invocations before it has the desired effect. For example if you started an incremental search, the first `C-g` erases some of the input and the second actually quits the incremental search.

The `undo` command is the normal way to undo changes made to the current buffer. If you undo twice in a row, then the last two changes are undone. In this manner you can eventually undo all the changes made to a buffer. Once you type something other than an undo, all the previous undo commands are treated as changes that themselves can be undone.

If you made a large number of changes to a file buffer and simply want to go back to the original file contents, use `M-x revert-buffer`. Note that if you have changed the file and saved it, then reverting will bring back the saved version, not the earlier one.

A.7 Search commands

Command	*Aliases*	*Function*
`isearch-forward`	`C-s`	Incremental search forward
`isearch-backward`	`C-r`	Incremental search backward

These search through the text for a specified string. The search is incremental in that it starts searching as soon as a character is typed in, finding the first occurrence of the string typed in so far. If the string can't be found, the minibuffer changes its prompt from `I-search:` to `Failing I-search:`. If it finds the string, but you wish to go on to the next (previous) occurrence, type another `C-s` (`C-r`). To terminate the search, type the Enter key, or any other Emacs command. Consult the Emacs manual for other useful options available for search.

A.8 Killing and Deleting

Command	*Aliases*	*Function*
`delete-next-character`	`C-d`	Delete next character
`delete-backward-char`	`DEL`	Delete previous character
`kill-word`	`M-d`	Kill word
`backward-kill-word`	`M-DEL`	Kill word backwards
`kill-line`	`C-k`	Kill rest of line
`kill-region`	`C-w`	Kill region
`copy-region-as-kill`	`M-w`	Save region a killed text without killing

These commands delete or kill the specified entities. The difference between killing and deleting is that a killed entity is copied to the kill ring, and can be *yanked* later, while deleted entities are not. The kill ring is a stack of blocks of text that have been killed, with the most recent kills at the top. The kill ring is not associated with any specific buffer, and in this respect acts much like a *clipboard* found in most window systems.

The `delete-backward-char` command is frequently mapped onto the `Backspace` key instead; you may need to experiment with this. If you want it mapped to the `Backspace` key, it is usually easier to map it outside of Emacs, for example using the `xmodmap` command. This is because by default the `Backspace` key and the `C-h` key are indistinguishable by Emacs, and the `C-h` key is used for accessing various Emacs help functions.

The `kill-line` command kills from the current cursor location to the end of the line, unless it is already at the end of the line, in which case it kills the newline, thus merging the current line with the following one.

The `copy-region-as-kill` command is similar to the `kill-region` command, but does not actually kill any text. This is useful when trying to copy text from a file for which you do not have write access, since Emacs will not let you modify such a buffer without first changing its read-only status.

A.9 Yanking

Command	*Aliases*	*Function*
`yank`	`C-y`	Yank last killed test
`yank-pop`	`M-y`	Replace last yank with previous kill

The `yank` command puts the text of the most recent kill command into the buffer at the current cursor position. Note that the usual way to move text from one place to another in Emacs is to kill it, move the cursor to the new location, and yank it. Because the kill ring is globally used, this works across buffers as well.

The `yank-pop` command may only be used after the `yank` command, and has the effect of replacing the yanked text with earlier killed text from the kill ring.

A.10 Marking

Command	*Aliases*	*Function*
`set-mark-command`	`C-@`, `C-SPC`	Set mark here
`exchange-point-and-mark`	`C-x C-x`	Exchange point and mark

The `set-mark` command sets the mark to the current cursor position. Since point is also at the current cursor position, this defines an empty region initially. As the cursor is moved away from the mark position, the region grows. Note that the relative positions of mark and point do not matter; the region is defined as the text between these two positions.

`C-x C-x` is used to exchange the point and mark positions, moving the cursor to where mark was last set, and setting mark to the old cursor position. Doing this again puts mark and point back where they started. This is useful for checking that the region is as desired, before doing any destructive operations.

Bibliography

[1] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley Publishing Company, Reading, MA, 2 edition, 1994. 1, 48

[2] John K. Ousterhout. *Tcl and the TK Toolkit*. Professional Computing Series. Addison-Wesley, 1994. 52

[3] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. 1, 2, 21, 23, 57

[4] Sam Owre and Natarajan Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Extensively revised June 1997; Also available as NASA Contractor Report CR-97-206264. 1

[5] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997. 1

[6] John Rushby and David W. J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report SRI-CSL-95-10, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1995. Revised, July 1996. Available, with specification files, at `http://www.csl.sri.com/csl-95-10.html`. 1

[7] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. 1, 11, 12, 23, 32, 33, 34, 61

[8] Richard M. Stallman. *GNU Emacs Manual*. Free Software Foundation, 675 Massachusetts Ave., Cambridge, MA, 13th edition, July 1997. 1, 17, 69, 72, 73, 79

[9] Walter F. Tichy. *RCS—A System for Version Control*. Department of Computer Sciences, Purdue University, West Lafayette, IN, July 1985. 73

Index