Using PVS in Batch Mode

Sam Owre
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Owre@csl.sri.com
Phone: +1 (650) 859-5114

September 8, 1997

1 Introduction

The Prototype Verification System (PVS) was developed as a research tool for exploring techniques of specification and theorem proving. PVS was made freely available in 1993, and has been steadily growing in popularity, as witnessed by the hundreds of members of the PVS mailing list and the increasing number of papers that reference it [2]. As expected for a system of this size and complexity, various versions and patches have been released since the original distribution. However, while new releases have corrected problems and added new features, they have occasionally introduced errors that cause existing specifications and proofs to fail. Therefore, we have developed batch and validation facilities within PVS that makes it easy to rerun system commands and compare the results to earlier runs, and have put together the foundations for a validation suite. This provides the foundation for rigorous regression testing that will be applied to future PVS releases.

2 Running PVS in Batch Mode

To support validation runs, PVS now supports *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 [3].

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.

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

[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 1: Batch File Example

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 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 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 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

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 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 2: Example Use of `pvs-validate`

nonexistent functions. These errors will cause the system to stop immediately, and the error will be displayed if the level number is 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.

3 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 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.
- `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.[2]

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.[3] 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 [4], using the Emacs *Version Control* interface [3, Section 14.7]. 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

[2]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.

[3]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.

a single directory that controls the validation for several different contexts.

4 The Validation Suite

Though the validation facility is available for general use, it was primarily developed to aid in test and validation of new releases of PVS. Currently, the validation suite consists of the prelude, the distributed libraries, and a few libraries developed at NASA and elsewhere. We plan on systematically making the specifications submitted by users reporting bugs a part of the validation suite, to ensure that reported bugs are indeed fixed, and stay fixed through subsequent releases. Finally, we plan on generating specifications that purposely explore different features of PVS, for example, specifications that exercise all possible nonterminals of the parser, or various datatype combinations. Note that because PVS error messages are logged but do not cause validation runs to be aborted, it is possible to make validation runs that are expected to trigger errors. Thus we can easily make negative tests as well as positive ones.

To provide better handling of bug reports, we have installed the *GNATS Problem Report Management System* [1]. Messages submitted to `pvs-bugs@csl.sri.com` are monitored, and those that report new bugs are submitted to the GNATS system. Those that relate to earlier bug reports are appended to the earlier problem report, to ensure that the related messages are all kept together. The specifications associated with the bug reports are kept in a directory tree whose names make it easy to relate the specification to the bug report. When a bug report is submitted to GNATS, the user that sent the report is notified of the submission and given a bug report number which they can refer to in followup messages. Before being submitted, the bug reports are cleaned up to remove signatures and messages unrelated to the bug. We provided a web page that displays the status of bug reports at `http://www.csl.sri.com/htbin/pvs/pvs-bug-list` so that PVS users can see if a bug they wish to report has already been submitted, and can follow the status of their submitted bug reports.

The prelude, libraries, test specifications, and specifications associated with bug reports make up the PVS validation suite. All future PVS releases will be regression tested against this test suite before being distributed.

References

[1] Jeffrey M. Osier. *Keeping Track: Managing Messages with GNATS.* Free Software Foundation, 675 Massachusetts Ave., Cambridge, MA, October 1993.

[2] John Rushby. *PVS Bibliography.* Menlo Park, CA. Constantly updated; available at `http://www.csl.sri.com/pvs-bib.html`.

[3] Richard M. Stallman. *GNU Emacs Manual.* Free Software Foundation, 675 Massachusetts Ave., Cambridge, MA, 10th edition, July 1994.

[4] Walter F. Tichy. RCS—a system for version control. *Software—Practice & Experience*, 15:637–654, July 1985.

A Example Validation Run

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

A.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
```

A.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")))
```

A.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
Loading compiled patch file /project/pvs/patch2-test.fasl
Context changed to ~/pvs-specs/validation/
Checking out ~/pvs-specs/validation/stamps.log...
```

```
Checking out ~/pvs-specs/validation/stamps.log...done
Context changed to ~/pvs-specs/validation/
Proving stamps
Restoring theories from stamps.bin
Restored file stamps (stamps) in 6.41 seconds
Rerunning proof of stamps
Proving stamps.stamps.
Proving stamps.stamps..
Proving stamps.stamps...
Proving stamps.stamps....
Proving stamps.stamps.....
Proving stamps.stamps......
Proving stamps.stamps.......
stamps proved in 19 seconds
stamps: 1 proofs attempted, 1 proved in 19 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
```

A.4 The Log File

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

```
Context changed to ~/pvs-specs/validation/
 "~/pvs-specs/validation/"

T
Restoring theories from stamps.bin
 Restored file stamps (stamps) in 6.41 seconds
Rerunning proof of stamps

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.
```