Getting Started with a Framework for Verifying Software Transactional Memory Algorithms

Mohsen Lesani Victor Luchangco Mark Moir

September 2012

Abstract

This document is intended to help readers start to use our framework for formally verifying software transactional memory algorithms using the PVS verification system. A paper describing technical aspects of this framework, and some of the challenges encountered and lessons learned in using the framework, was presented at CONCUR 2012. This document describes more mundane aspects, such as how to explore the framework using PVS for readers not already familiar with it, and provides brief descriptions of the files contained in the framework.

1 Introduction

This document briefly describes a framework for formally verifying transactional memory (TM) algorithms that we presented at CONCUR 2012 [7], and gives some tips that we hope enables readers to start using this framework quickly and easily. Our framework is formalized using the PVS verification system [1, 11], and, for readers not already familiar with PVS, we describe how to get started using it quickly, and provide pointers to where to get it. Although we do not assume familiarity with PVS, this document is not a tutorial on PVS; such tutorials are available at the PVS website [1, 2].

2 Overview of the framework

The framework provides three kinds of files used by PVS:

- *specification files* (`.pvs` suffix) contain definitions and lemmas provided by the framework;
- *proof files* (`.prf` suffix) contain saved proofs of lemmas; and
- the `pvs-strategies` files defines *strategies* used in the proofs.

Most users will be interested primarily in the specification files. Proof files are generated and updated automatically by PVS, and should not be edited directly. Strategies are user-defined routines that capture common patterns in proofs. We do not discuss strategies further in this document; for more information, please refer to PVS documentation (e.g., [2]).

Definitions and lemmas are specified in the PVS language, which is a typed higher-order logic. PVS generates proof obligations for every lemma, as well as for *type-correctness conditions* (TCCs) induced by certain expressions. For example, a function call induces the TCC that the function is defined on its argument. Every lemma and TCC of our framework has been verified, and its proof is contained in the appropriate proof file (i.e., the one with the same name as the specification file in which the lemma appears).

Definitions and lemmas in PVS are packaged into *theories*, which are the basic unit of modularity in PVS. A theory can use definitions and lemmas specified in another theory by *importing* that theory. A single specification file may contain multiple theories, and the name of a theory need not correspond to the name of the file that contains it. A theory may import any theory contained in a specification file in the same directory (in our framework, all the files are in a single directory). In our framework, every specification file other than `FiniteSequences.pvs` contains a single theory with the same name as the file.

Our framework is based on I/O automata [9] and simulation proof techniques [10]. As discussed in our CONCUR paper [7], we found it useful to refine standard automata theory by splitting the definition of an automaton into a *basic automaton*, which specifies the "internal structure" of the automaton, and a *view*, which maps *actions* to external *events*.

Upon this foundation, we define several automata that model TM specifications and algorithms. In particular, we formalize TMS1 and TMS2 from our previous work on specifying transactional memory [4], *opacity* by Guerraoui and Kapalka [5, 6] modeled as an automaton [8], and the NOrec algorithm of Dalessandro et al. [3]. As described in our CONCUR paper, we define several equivalent automata for TMS2, and also "intermediate" automata for proving that NOrec implements TMS2. We also define several simulation relations of various kinds to prove that certain automata implement others. We model transactional memory as acting on a single data object, and we also formalize the sequential specification of such objects.

In addition, we provide several theories that extend the theories of finite sequences and partial functions provided in the PVS prelude. (The PVS prelude is a collection of PVS theories shipped with PVS whose definitions are implicitly available in any user-defined theories.) In particular, we define several simple functions and many lemmas, especially for finite sequences, that we found useful in our proofs. We hope that these extensions will be helpful to others, even those not interested in verifying transactional memory.

Finally, our framework includes several "main" files, which provide starting points for verifying the framework or parts of it, using the `prove-import-chain` command of PVS (see below).

2.1 The specification files

We now list the specification files included in the framework.

- `Automaton` `Simulations` `SimulationComposition`
  `View` `BackSimulations`
  `AutomatonWithView` `HistoryMappings`

  These files formalize the basic automata theory used in the framework.

- `SequentialSpecifications` `RWObject`

  These files define sequential semantics for data objects, and read-write memory in particular.

- `TMEvent`

  This file defines the type of external events for TM automata.

- `TMS1` `Opacity` `TMS1Action` `StatusValue` `TMS2` `TxnOrdTMS2` `TxnOrdTMS2WithFailures` `ReservationTMS2` `TxnOrdTMS2Augmented` `NOrec` `NOrecAtomicCommitValidate` `NOrecDerived` `NOrecPaperPseudocode` `Lock`

  These files define basic automata, and also lemmas and invariants for these automata. (The `TMS1Action` and `StatusValue` files define types for actions and possible values for the "status" of a transaction for both `TMS1` and `Opacity`. `Lock` defines the lock datatype used by `NOrec` and `NOrecDerived`.)

- `TMS2WithView` `TxnOrdTMS2WithFailuresWithView` `ReservationTMS2WithView` `NOrecWithView` `NOrecAtomicCommitValidateWithView` `NOrecDerivedWithView` `NOrecPaperPseudocodeWithView`

  These files define the views for the automata above. (There are no files for `TMS1` and `Opacity` because their view is just the identity function. There is no file for `TxnOrdTMS2` because its view is the same as for `TMS2`.)

- `TxnOrdTMS2ToTMS2`
  `ReservationTMS2ToTxnOrdTMS2WithFailures`
  `TxnOrdTMS2WithFailuresToTxnOrdTMS2`
  `TxnOrdTMS2ToReservationTMS2`
  `OpacityToTMS1`
  `TxnOrdTMS2AugmentedToOpacity`
  `TxnOrdTMS2ToTxnOrdTMS2Augmented`
  `NOrecToTMS2`
  `NOrecAtomicCommitValidateToTMS2`
  `NOrecDerivedToNOrecAtomicCommitValidate`
  `NOrecToNOrecDerived`
  `NOrecPaperPseudocodeToNOrec`
  `NOrecExtraProperties`

  These files define simulation relations between the automata above. (`NOrecToTMS2` just combines the results of the three subsequent files. `NOrecExtraProperties` derives an invariant of `NOrec` from an analgous invariant of `NOrecAtomicCommitValidate`. See our CONCUR paper for more details [7].)

- `FiniteSequences` `Serializations` `Flatten` `Unfold` `PartialFunctions`

  These files extend theories about finite sequences and partial functions from the PVS prelude.

- `Main` `Main_Preliminaries` `Main_Automata` `Main_TMSpecs` `Main_NOrec`

  These files serve as starting points for verifying the framework.

3 Using the framework

Using the framework requires PVS (and Emacs) to be installed. For pointers on how to acquire and install PVS, see Appendix A.

Start PVS in the directory containing the framework (i.e., the directory with the specification and proof files). For example, if this directory is `~/framework`, then go to `~/framework` and type `pvs` at the prompt:

`~/framework$ pvs`

This starts Emacs with PVS running (in the `*pvs*` buffer).

In the rest of this section, $\mathbb{C}$, $\mathbb{M}$, and $\mathbb{T}$ denote the control, meta, and tab keys respectively. PVS can be run interactively or in batch mode.

Although we discuss aspects of PVS in this section, it is neither a tutorial nor a definitive reference for PVS. Interested readers should consult the PVS documentation available on the PVS website [1], which includes a tutorial [2] and a guide to the prover [12]. However, note that this documentation is rather dated, and so is not entirely accurate for the latest version of PVS.

3.1 Running proofs

Open the file `Automata.pvs`, and look at lemma `lengthTruncate`. There is a saved proof for this lemma. (Proofs of lemmas in `Automaton.pvs` are saved in `Automaton.prf`.) To see this proof, place the cursor on `lengthTruncate` lemma and type

$\mathbb{M}$–`x show-proof`

A buffer named `Proof` (henceforth, the *proof buffer*) is opened, displaying the saved proof. (A proof in PVS is a script consisting of a sequence of commands to the prover.)

The PVS prover runs all proofs in the `*pvs*` buffer. To run the saved proof of a lemma, place the cursor on the lemma (in the specification file) and press

$\mathbb{C}$–`c p`

At the prompt, type `yes` when you are asked whether to rerun the existing proof. (Type `no` to write a new proof.) This starts the proof in the `*pvs*` buffer. As the proof is correct, it finishes successfully. If there were no saved proofs for the lemma, this step would be skipped and you would be immediately prompted to prove the lemma in the `*pvs*` buffer.

We can also step through the proof: Go to the lemma `lengthTruncate` again and press

$\mathbb{C}$–`c` $\mathbb{C}$–`p s`

This loads the proof buffer with the saved proof, which consists of four commands. It also makes the `*pvs*` buffer current, and prompts for a prover command. Above the prompt, the current sequent is displayed. We must establish that the conjunction of formulas above the turnstile implies the disjunction of the formulas below it.

To execute the next command from the proof buffer, press

$\mathbb{T}$ `1`

This executes the command following the cursor in the proof buffer, and advances the cursor in the proof buffer to the next command. Thus, it can be used repeatedly to run consecutive commands. To run the next k command consecutively, press

$\mathbb{C}$–`u` k $\mathbb{T}$ `1`

The proof buffer can be edited, and the cursor can be moved inside the proof buffer to mark a different command as the next command.

Note that 𝕋 `1` executes whatever command follows the cursor in the proof buffer. Thus, one can load the proof of one lemma into the proof buffer (e.g., by 𝕄–`x show-proof`), and then use it to prove a different lemma, provided that the proof buffer is not overwritten (e.g., by pressing ℂ–`c` ℂ–`p s` to start proving the other lemma).

To issue a command directly (rather than running one from the proof buffer), either type it at the prompt in the `*pvs*` buffer or use a command shortcut. Shortcuts for some of the commands we use most frequently are:

𝕋 `a`	`(assert)`	𝕋 `g`	`(ground)`	𝕋 `p`	`(prop)`	𝕋 `=`	`(decompose-equality)`
𝕋 `e`	`(expand)`	𝕋 `G`	`(grind)`	𝕋 `s`	`(split)`	𝕋 `?`	`(inst?)`
𝕋 `f`	`(flatten)`	𝕋 `l`	`(lift-if)`	𝕋 `S`	`(skosimp)`		

See the PVS prover guide [12] for a description of these commands.

Some commands generate multiple proof obligations as subgoals, leading proofs to have a tree structure. The `(postpone)` command (shortcut: 𝕋 `P`) causes the prover to switch to the next incomplete branch in the proof tree. The `(undo)` command (shortcut: 𝕋 `u`) removes the last command in the current branch of the proof tree. Note that this is *not* the previous command issued if that command completed some branch of the tree. If the undone command had some completed branches, those branches are also removed from the tree. A mistaken `(undo)` command can be reversed using `(undo undo)`, but only if this is done immediately afterwards (i.e., with no intervening command).

PVS provides a graphic display of the proof tree. To see the current proof tree, type

𝕄–`x x-show-current-proof`

This tree is updated as you issue more commands, so that it always reflects the current state of the proof. This is helpful to see proof branches in large proofs. Clicking on the turnstiles that appear between commands opens a window that displays the sequent at that point in the proof. Long commands are abbreviated in the proof tree; clicking on them opens a window showing the full command.

Certain commands, particularly `(grind)`, may run for a long time—indeed, they may never terminate (or at least run longer than we have been willing to wait). You can interrupt such a command and return to the point immediately before executing it by typing

ℂ–`c` ℂ–`c` ℂ–`d`

To quit a proof, use the `(quit)` command (shortcut: 𝕋 `q`). If you have issued a sequence of commands other than the saved proof, you are prompted whether the proof should be saved. In this case, type `no` to preserve the existing proof.

A theory (and any theories it imports) must be type-checked before it can be proved. This is done automatically when a proof in the theory is attempted, but it can also be done explicitly by typing

𝕄–`x typecheck-prove`

TCCs are automatically generated constraints (lemmas) to enforce type soundness. Some of them are automatically discharged and some are not. To see the TCCs of this theory, type

𝕄–`x show-tccs`

or

ℂ–`c` ℂ–`q s`

3.2 Rerun the proofs in batch

To run all the saved proofs for a theory, place the cursor anywhere within the theory and press

ℂ–`c` ℂ–`p t`

After executing all the proofs, the status of all lemmas and TCCs of the theory is displayed in a new buffer called `PVS Status`. The status of a lemma or TCC may be "untried", "unfinished", "proved - incomplete" or "proved - complete". A lemma is "proved - incomplete" if it was proved but used lemmas that are not "proved - complete". To get this status again later, type

𝕄–`x status-proof-theory`

To prove all the theories that a theory directly or indirectly imports, type

ℂ–`c` ℂ–`p i`

which runs all the proofs and then brings a summary of the status of the theories in the import chain. To prove all theories in the entire framework, invoke this command on the `Main` theory. To see this status again later, type

𝕄–`x status-proofchain`

3.3 Get information

To quickly find the definition of a function in a specification file, place the cursor on a usage of this function and press

𝕄–.

For example, the `lengthTruncate` lemma uses the `truncate` function. In this case, the definition is right above the lemma but it could be far away, even in a different (imported) theory. Placing the cursor on the call to `truncate` in `lengthTruncate` and pressing 𝕄–. opens a new buffer with the definition of `truncate`. Conversely, to see where a function is used, place the cursor over the function and press

𝕄–;

Placing the cursor on the `truncate` function and pressing 𝕄–; opens a new buffer with a list of the places where `truncate` is used, including the lemma `lengthTruncate`.

To see the theories in the PVS prelude, type

𝕄–`x view-prelude-file`

To get the list of all PVS prover commands, type

𝕄–`x help-pvs-prover`

In PVS, definitions can be *overloaded*; that is, there can be many definitions for the same name. PVS resolves overloading by context, and causes type-checking to fail if it cannot do so. When displaying sequents, however, it typically just shows the plain overloaded names without the context. Thus, two expressions that appear to be identical might differ because an overloaded name might be resolved differently. To see how user-defined overloading is resolved, type

𝕄–`x show-expanded-sequent`

This does not show the resolution of names overloaded in the prelude. To see these, press

ℂ–`u` 𝕄–`x show-expanded-sequent`

(Note that infix operators are expanded into binary functions, so that an expanded sequent might look quite different from the unexpanded sequent.)

3.4 Issues with PVS

We encountered some difficulties in using PVS, many of which were resolved with help from Sam Owre. For some others, we developed workarounds, which we discuss here.

First, sometimes PVS generates extra TCCs (for which it cannot, of course, find any proofs) or complains about names already being defined. We believe that this has to do with how PVS stores intermediate computations, both internally and in `.bin` files. When this happens, we find it easiest to "reboot" PVS as follows: Kill the `*pvs*` buffer by typing

$\mathbb{C}$–`x k *pvs*`

(Emacs will ask whether you are sure you want to kill the PCS process.) Then delete any `.bin` files in the directory. Then restart PVS by typing

$\mathbb{M}$–`x` pvs

We notice that PVS typically gets confused about names in theories that use theory aliases, and intend to excise such use from our files. However, our current version of the framework still has many such uses.

Second, we discovered a case in which PVS failed to generate certain TCCs, allowing unsound proofs. Fortunately, Sam Owre fixed this problem and provided us a patch (see Appendix A.1), which should be included in the next release of PVS.

Third, PVS allows theories to be organized into "libraries" (i.e., directories), to provide an extra level of modularity. We would like to exploit this functionality by having separate libraries for the basic automata theory, the TM specifications, and the automata related to NOrec, and theories that extend the PVS prelude theories of finite sequences and partial functions. (This classification is embodied in the different "main" files we provide.) However, we have not been able to get PVS to consistently confirm that all proofs are complete when they depend on lemmas defined in theories imported from other libraries. Thus, for now, we keep all the files in a single library.

A Installing PVS

PVS is developed and maintained by SRI International. It can be downloaded from their website [1], along with instructions on how to install it. Our framework was (and is being) developed using PVS 5.0 running on Linux on a 32-bit Intel machine. For licensing reasons, we use the SBCL version. To install this version, download `pvs-5.0-ix86-Linux-sbclisp` from the PVS website, and unzip and untar it in the directory in which you want to install PVS. Let's say that directory is `~/PVS`:

```
~/PVS$ tar -zxvf pvs-5.0-ix86-Linux-sbclisp.tgz
```

Now, run the relocation script

```
./~/PVS/bin/relocate
```

and add the PVS folder to your path

```
~/$ export PATH=$PATH:~/PVS
```

You should now be able to run pvs

```
~/$ pvs
```

A.1 PVS patch

As mentioned above, there is a bug in PVS that caused it to fail to generate certain type-correctness conditions. Sam Owre provided the following patch, which should be added to the `.emacs` file.

```
(in-package :pvs)

(defmethod make-implicit-conversion ((conv expr) ctype ex)
 (let ((nexpr (copy-untyped ex)))
   (change-class ex 'implicit-conversion)
   (setf (argument ex) nexpr)
   (setf (types nexpr) nil)
   (setf (operator ex)
  (raise-actuals (copy (expr (from-conversion ctype))) 1))
   (typecheck* ex nil nil nil)))
```

A.2 Shortcuts

Some users, especially those not familiar with Emacs, may find the Emacs commands confusing. Such users may find it useful to remap many common commands of Emacs and PVS to more familiar keys. Such a remapping is done by the `.emacs` file that can be retrieved from

`http://cs.ucla.edu/~lesani/downloads/.emacs`

Follow the simple instructions in the `.emacs` file to install it. The shortcut keys it provides are listed at the beginning of the file.`.emacs` file.

References

[1] PVS website. `http://pvs.csl.sri.com/`.

[2] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.

[3] Luke Dalessandro, Michael Spear, and Michael Scott. NOrec: Streamlining STM by abolishing ownership records. In *Symposium on Principles and Practice of Parallel Programming*, January 2010.

[4] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 2012.

[5] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Symposium on Principles and Practice of Parallel Programming*, February 2008.

[6] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2010.

[7] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *International Conference on Concurrency Theory (CONCUR)*, September 2012.

[8] Mohsen Lesani, Victor Luchangco, and Mark Moir. Putting opacity in its place. In *Workshop on Theory of Transactional Memory*, July 2012.

[9] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Symposium on Principles of Distributed Computing*, pages 137–151, August 1987.

[10] Nancy Lynch and Frits Vaandrager. Forward and backward simulations part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[11] Sam Owre, Natarajan Shankar, and John Rushby. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752, June 1992.

[12] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*, November 2001.