Proof Mate: an Interactive Proof Helper for PVS (tool paper)

Paolo Masci[1] and Aaron Dutle[2]

[1] National Institute of Aerospace*
paolo.masci@nianet.org
[2] NASA Langley Research Center
aaron.m.dutle@nasa.gov

Abstract. This paper presents Proof Mate, an interactive proof helper for the PVS verification system. The helper is integrated in VSCode-PVS, the Visual Studio Code extension for PVS. It extends the capabilities of VSCode-PVS by introducing new functionalities for suggesting proof commands, sketching proof attempts, and repairing broken proofs during interactive proof sessions. This work further aligns VSCode-PVS to the functionalities provided by modern development tools, with the ultimate aim to facilitate the adoption of formal methods in engineering practices and education.

Tool available at: https://github.com/nasa/vscode-pvs

1 Introduction

The capabilities of formal methods tools have classically been measured by aspects such as the expressiveness of the specification language, the level of automation, and the scalability of the analysis when dealing with complex systems. In recent years, an additional metric started to play an important role, linked to the usability of the tool front-end. The current generation of proof engineers, and likely future generations, favor graphical front-ends over command line versions. Functionalities like auto-completion, integrated help, and point-and-click interactions are now considered baseline features that any modern tool front-end is expected to provide.

Developers of formal methods tools are upgrading the front-end of their tools to meet this new baseline. An example is VSCode-PVS [4], which upgrades the Emacs front-end of PVS [7] to Visual Studio Code, a mainstream open-source code editor widely popular in the developer community. VSCode-PVS provides editor functionalities such as autocompletion, hover information, live diagnostics, interactive proof tree visualizer and editor, among several others.

This work introduces Proof Mate, a new interactive tool for VSCode-PVS that further extends the capabilities of the PVS front-end with new functionalities for proof development, proof editing and proof repair.

* Research by the first author is supported by the National Aeronautics and Space Administration under NASA/NIA Cooperative Agreement NNL09AA00A.

2 Theorem Proving in PVS and VSCode-PVS

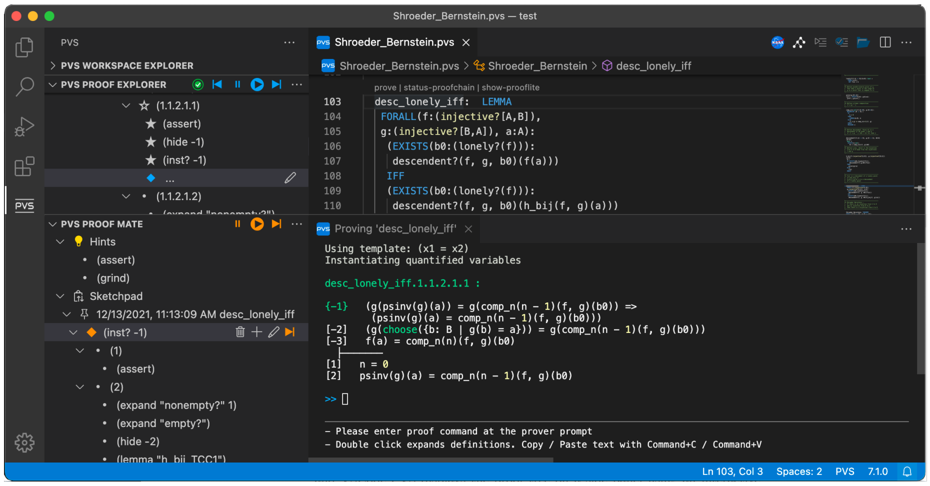
The Prototype Verification System (PVS [7]) is an interactive theorem prover (ITP) based on a sequent calculus for classical higher-order logic, used extensively by NASA Langley Research Center's formal methods team (see, e. g., [2, 6]). Specifications and properties are written in a human-readable ".pvs" file, but contrary to many other ITPs, proofs are stored in a separate proof file using an internal representation [5], and not generally intended for direct reading or editing. Proofs are constructed interactively in PVS using proof commands, which are applied to a *sequent.* A sequent has the structure $A_1..A_n \vdash C_1..C_n$, where A_i are called antecedent formulas, and C_i are consequent formulas. A proof command manipulates the sequent (with some commands branching to several sequents). A branch is closed (i.e., proven) when an antecedent is false, a consequent is true, or the same formula appears in the antecedent and consequent. An example proof command is `assert`, which expands and simplifies definitions. Proofs can consist of many branches and hundreds of proof commands, stored (essentially) as a list.

While the original PVS Emacs interface allows for *viewing* a proof in either text or tree form, *editing* a proof in this form is difficult even for expert users, and cannot be performed during an interactive proof session. VSCode-PVS [4] is a new front-end that integrates PVS in the Visual Studio Code editor. In VSCode-PVS, proof commands are entered in the Prover Console and displayed as a proof tree in a side panel called Proof Explorer (see Figure 1). Proof Explorer improves the viewing and navigation of proofs by incorporating a collapsible, file-system like view. Edit operations in Proof Explorer, however, are intentionally constrained, because the proof tree shown is intended to always reflect the proof structure computed by PVS. This way, the user knows exactly what will be saved in the proof file at the end of a proof session.

3 Proof Mate

Proof Mate extends the capabilities of VSCode-PVS by introducing new functionalities for suggesting proof commands, sketching proof attempts, and repairing broken proofs during interactive proof sessions. The tool is integrated in the front-end as a side panel characterized by interactive tree views, inline actions, and a toolbar (see Figure 1). Proof Mate has a similar look and feel to Proof Explorer, but because it is not tied directly to the proof being attempted, offers much more flexibility to experiment with and write proof segments.

Suggesting proof commands. Proof Mate provide hints for proof commands during a proof session, while the proof engineer is proving a theorem. Hints are selected using heuristics based on common proof patterns in PVS. The heuristics are encoded into templates which ensure that the selected commands are applicable. One example is: "if a consequent formula starts with `FORALL` or an antecedent starts with `EXISTS`, then recommend skolemization commands (i.e., `skosimp*` or `skeep`)." Another example is: "if a formula has the form `expr = IF`



```
desc_lonely_iff:  LEMMA
 FORALL(f:(injective?[A,B]),
 g:(injective?[B,A]), a:A):
  (EXISTS(b0:(lonely?(f))):
   descendent?(f, g, b0)(f(a)))
  IFF
  (EXISTS(b0:(lonely?(f))):
   descendent?(f, g, b0)(h_bij(f, g)(a)))
```

```
Using template: (x1 = x2)
Instantiating quantified variables

desc_lonely_iff.1.1.2.1.1 :

{-1}   (g(psinv(g)(a)) = g(comp_n(n - 1)(f, g)(b0)) =>
        (psinv(g)(a) = comp_n(n - 1)(f, g)(b0)))
[-2]   (g(choose({b: B | g(b) = a})) = g(comp_n(n - 1)(f, g)(b0)))
[-3]   f(a) = comp_n(n)(f, g)(b0)
  |-------
[1]   n = 0
[2]   psinv(g)(a) = comp_n(n - 1)(f, g)(b0)

>> 

- Please enter proof command at the prover prompt
- Double click expands definitions. Copy / Paste text with Command+C / Command+V
```

Fig. 1. Proof Mate (lower left panel), Proof Explorer (upper left panel), Editor (upper right panel) and Prover Console (lower right panel) in VSCode-PVS.

`expr THEN expr ENDIF`, recommend commands for lifting the innermost contiguous branching structure out to the top level (i.e., `lift-if`).” When none of the other heuristics are matched, general simplification procedures are recommended (i.e., `assert` or `grind`). The hints are automatically computed by Proof Mate during interactive proof sessions, every time a new sequent is returned by PVS. A tooltip providing a brief description of the proof command is shown when hovering the mouse on a recommendation. Point-and-click interactions can be used to select a recommendation and send it to the prover console for execution.

Sketching proof attempts. Proof Mate provides a sketchpad that can be used by proof engineers to create and edit *proof clips*. While the look and feel of the sketchpad resembles that of Proof Explorer, proof clips shown in the sketchpad are designed to reflect proof ideas in the mind of the proof engineer developing the proof, as opposed to mirroring the proof tree internally created by PVS. Because of this, proof clips can be edited freely. Multiple proof clips can be created and stored in the sketchpad. Each clip is automatically labeled with a timestamp or a custom name provided by the proof engineer. Edit operations allowed on sketchpad clips include renaming, addition, deletion, and copy/paste of proof commands and proof branches. Copy/paste operations are also allowed between the sketchpad and Proof Explorer. All operations can be performed with point-and-click interactions. Inline action buttons are provided for frequent operations. Proof clips are maintained across different proof sessions, allowing re-use of proof sketches created for other proofs. Interactive controls are available for executing proof commands and playback of proof clips.

Repairing broken proofs. A PVS proof may break for various reasons, ranging from changes introduced by the proof engineer in the PVS specification under

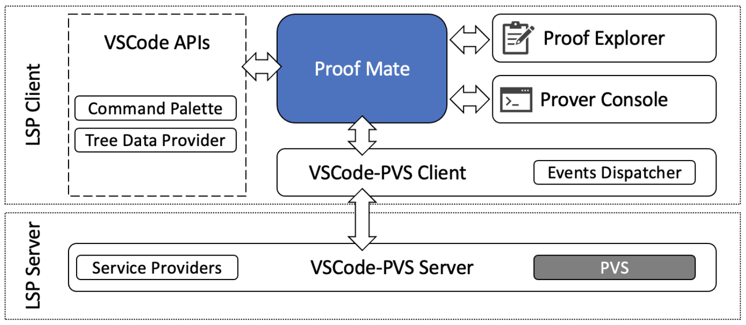


Fig. 2. Proof Mate Architecture (arrows indicate exchange of data or events).

analysis (e.g., updated definitions or refactoring of terms), to enhancements introduced by the PVS developers in the prover engine (e.g., when a new version of PVS is released). When a proof breaks, PVS automatically discards a fragment of the proof structure. Proof Explorer, which is designed to reflect the proof structure internally stored by PVS, automatically prunes sections of the proof tree corresponding to the part discarded by PVS. While this ensures consistency between Proof Explorer and PVS, the net result is that a fragment of the proof is effectively lost. Proof Mate seamlessly detects these situations and saves the proof fragments that would otherwise be lost in the sketchpad. Proof engineers can inspect the fragments saved in the sketchpad to understand what caused the break and edit/execute the fragments to repair the proof. Figure 1 shows a situation where a proof, that was previously complete, broke during a proof re-run. In the original proof, PVS was generating two sub-goals (i.e., two branches) after `(inst? -1)`. In the proof re-run, PVS is not generating sub-goals. In this situation, PVS discards the two branches and, consequently, Proof Explorer automatically prunes all nodes after `(inst? -1)`. Proof Mate saves the pruned fragments in the sketchpad, as a clip rooted in `(inst? -1)` — this provides a visual cue that can help proof engineers map the content of the sketchpad with that of Proof Explorer. In this example, the repair action involved executing the first command in the first branch of the proof clip (i.e., `assert`).

4 Architecture and Implementation

The high-level architectural diagram shown in Figure 2 illustrates how Proof Mate is integrated in VSCode-PVS and the Language Server Protocol[3] (LSP) architecture. Being a front-end module, Proof Mate is part of the client side of the LSP architecture. It communicates with three VSCode-PVS components.

VSCode-PVS Client is used for sending a request to execute a proof command to the VSCode-PVS server through the LSP connection. The client is also used for receiving notifications about changes in the proof structure, in particular

[3] https://microsoft.github.io/language-server-protocol

deletion of nodes and proof branches. These events are used by Proof Mate for seamless detection and handling of broken proofs.

Proof Explorer provides a shared clipboard that is used by Proof Mate when performing copy/paste operations from/to Proof Explorer.

Prover Console provides APIs for writing text programmatically in the console. These API are used by Proof Mate to provide feedback to the user when, e.g., point-and-click interactions with Proof Mate trigger the execution of a command.

The *VSCode APIs* are used by Proof Mate for creating the visual elements of the view, as well as to link the view to the global command palette and clipboard of the Visual Studio Code editor.

Implementation. Proof Mate is entirely implemented in TypeScript, a version of the JavaScript language annotated with type information that can be statically checked for type correctness. A class `ProofMate` implements the functionalities of the module. The class inheritance mechanism is used to define the class as an extended version of Proof Explorer and build on existing code. Overall, the implementation of the Proof Mate module includes approximately 2K lines of TypeScript code. Only minor additions were necessary in the other modules to correctly integrate Proof Mate in the VSCode-PVS front-end.

5 Related Work

Pumpkin [8, 9] is a proof repair tool for the Coq proof assistant. The tool provides a semi-automatic *repair-by-example* approach to proof repair. The basic intuition is that a same breaking change may cause similar problems in different proofs. When a proof breaks, the proof engineer can therefore develop an example patched proof, and then use automatic differencing techniques and proof term transformations to synthesize a template patch candidate that can potentially fix other proofs that were broken in a similar way. While this approach is specifically designed for Coq, the concept appears to be generally applicable to other theorem proving systems, including PVS, and will be explored to automate some of the functionalities of Proof Mate.

Tactician [1] and TacticToe [3] are interactive proof helpers for Coq and HOL4, respectively. Both tools are designed to suggest proof tactics than can be used to complete a proof. Patterns are learned from existing proofs using machine learning techniques. In Proof Mate, a different approach is taken, based on direct encoding of expert knowledge into heuristics rules. An attempt is currently underway to extend the capabilities of Proof Mate with machine learning, targeted at suggesting lemmas.

PeaCoq [10] is an experimental front-end designed to help novice users develop a proof. The tool uses a *visual diff view* to highlight the effects of the refactoring changes on the proof tree. Color-coded text is used to highlight differences between the old and the new version of the proof script. A similar kind of visualization was considered for VSCode-PVS, where changes in the proof tree were directly visualized in Proof Explorer using strikethrough text for the highlighting deleted fragments. This possible solution was discarded because of usability issues — the window quickly became cluttered and hard to navigate.

6 Conclusion and Future Work

Proof Mate brings a collection of new capabilities to the users of VSCode-PVS, by suggesting relevant proof commands, providing a sketchpad for proofs or proof sections to be assembled outside the interactive prover, and assisting in repairing broken proofs in a number of ways. In contrast to most other interactive theorem provers, PVS does not support editing of proofs outside of the sequential interactive console in a simple way. Proof Mate fills this role and others, providing a playground for copying, editing, writing, and even suggestion of proof sections without restriction, and during a live proof session.

Each of the functionalities that Proof Mate provides (suggestion, sketching, repairing) are ripe for modification and improvement. The current command suggester is based on pattern matching of particular statements in the sequent. While this is certain to find *a* command that will apply, there are more sophisticated methods for finding relevant commands. Future efforts will incorporate machine learning techniques to find commands that may be more relevant to the user, such as suggesting appropriate lemmas to be used.

The proof sketching functionality in Proof Mate can also be extended in several ways. Currently, Proof Mate allows for a block of commands to be selected and used in the interactive prover console. A small extension would be to facilitate a user creating a custom *local strategy* from these commands, including variables that could be replaced on use. This is a step toward the larger goal of making the PVS strategy language more user-friendly and applicable. A much more ambitious goal is the translation of a natural language proof of a theorem or statement into a proof inside of PVS. While a full proof is unreasonable to expect, a system that could identify and sketch the main skeleton of a proof from a natural language description could aid in the formal verification.

The proof repair function of Proof Mate takes the pruned branches of a previous proof attempt and copies it to the sketchpad. While this catches a large number of broken proofs, there are situations where the sequent diverges prior to where Proof Mate catches the change, and so repair is more difficult. For example, if a change in a specification adds a statement to the antecedent, there can be a long sequence of successful commands (hiding formulas, calling lemmas, etc.) before the first true "break" in the proof. Adding functionality to find this divergence point is more difficult, since the proof is stored as a sequence of commands and does not carry information about the sequent(s) resulting from a command. Another possible enhancement would be for Proof Mate to save not just the pruned sequent, but store the actual repair that was used, since a repair in one proof is often needed in other similar repairs.

References

1. Blaauwbroek, L., Urban, J., Geuvers, H.: The Tactician. In: International Conference on Intelligent Computer Mathematics. pp. 271–277. Springer (2020)
2. Dutle, A., Moscato, M., Titolo, L., Muñoz, C., Anderson, G., Bobot, F.: Formal analysis of the Compact Position Reporting algorithm. Formal Aspects of Computing (2020). https://doi.org/10.1007/s00165-019-00504-0, http://link.springer.com/article/10.1007/s00165-019-00504-0
3. Gauthier, T., Kaliszyk, C., Urban, J.: Learning to reason with HOL4 tactics. CoRR **abs/1804.00595** (2018), http://arxiv.org/abs/1804.00595
4. Masci, P., Muñoz, C.A.: An Integrated Development Environment for the Prototype Verification System. In: Monahan, R., Prevosto, V., Proença, J. (eds.) Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019. EPTCS, vol. 310, pp. 35–49 (2019). https://doi.org/10.4204/EPTCS.310.5, https://doi.org/10.4204/EPTCS.310.5
5. Munoz, C.: Batch proving and proof scripting in PVS. NIA/NASA Langley, NASA/CR-2007-214546, NIA Report No. 2007-03 (2007)
6. Muñoz, C., Narkawicz, A.: Formal analysis of extended well-clear boundaries for unmanned aircraft. In: Rayadurgam, S., Tkachuk, O. (eds.) Proceedings of the 8th NASA Formal Methods Symposium (NFM 2016). Lecture Notes in Computer Science, vol. 9690, pp. 221–226. Springer, Minneapolis, MN (June 2016). https://doi.org/10.1007/978-3-319-40648-0$_1$7
7. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: International Conference on Automated Deduction. pp. 748–752. Springer (1992)
8. Ringer, T.: Proof Repair. Ph.D. thesis, University of Washington (2021)
9. Ringer, T., Porter, R., Yazdani, N., Leo, J., Grossman, D.: Proof repair across type equivalences. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 112–127 (2021)
10. Robert, V.: Front-end tooling for building and maintaining dependently-typed functional programs. Ph.D. thesis (2018)