An Integrated Development Environment for the Prototype Verification System

Paolo Masci
National Institute of Aerospace
Hampton, VA, USA
paolo.masci@nianet.org

César A. Muñoz
NASA
Hampton, VA, USA
cesar.a.munoz@nasa.gov

The steep learning curve of formal technologies is a well-known barrier to the adoption of formal verification tools in industry. This paper presents VSCode-PVS, a modern integrated development environment for the Prototype Verification System (PVS). This new environment integrates the editing and proof management functionalities of PVS in Visual Studio Code, a popular code editor widely used by software developers. VSCode-PVS provides functionalities that developers expect to find in modern verification tools, but are not available in the standard Emacs front-end of PVS, such as auto-completion, point-and-click navigation of definitions, live diagnostics for errors, and literate programming. The main features and architecture of the environment are presented, along with a comparison with other similar tools.

1 Introduction

Early detection of design anomalies and increased confidence that the system will operate as intended are some of the benefits of the use of formal verification technologies in industry. However, outside safety-critical domains such as avionics, the use of verification tools is still rather limited. One of the reasons is the steep learning curve of verification technologies, which creates an initial cost that is often deemed excessive with respect to the long term benefits.

The work presented in this paper aims to reduce the learning curve of the Prototype Verification System (PVS) [9], a state-of-the-art verification tool for formal modeling and analysis of system designs. The PVS modeling language is based on higher-order logic. It supports basic types (`boolean`, `int`, `real`, etc), as well as datatypes such as `string`, `set`, `list`. The proof engine is based on Gentzen's sequent calculus, and supports the use of proof strategies for automated analysis. The verification system provides an evaluation environment, called PVSio [7], for animation of executable specifications.

PVS is a powerful analysis tool with a long history of success stories in a range of different domains [8]. PVS is widely used at NASA Langley Research Center for the analysis of algorithms and protocols for avionics systems[1]. Research groups have also applied PVS to the analysis of human-machine interfaces in medical systems [5] and for co-simulation of semi-autonomous systems [10].

Becoming fluent with PVS, however, usually requires several months. Common difficulties faced by developers are often rooted in the PVS front-end:

- **Conceptual gap.** The PVS front-end is based on the Emacs editor. Emacs does not provide separate visual components for editing files, executing commands, and browsing the file system. Rather, it provides *buffers*, abstract entities that can be used to interact with any resource. When a buffer is linked to a text file, e.g., a PVS specification, Emacs acts like a code editor. When

[1] `https://shemesh.larc.nasa.gov/fm/fm-main-research.html`

Submitted to:
F-IDE 2019

the buffer is linked to an interactive process, e.g., the PVS theorem prover, Emacs provides a command-line interface for sending commands to the process. Developers need to learn the set of commands for operating with buffers, as well as to recognize the buffers — the visual appearance of different buffers is identical to the untrained eye.

- **Knowledge gap.** The PVS Emacs interface favors the use of the command line integrated in Emacs to point-and-click interactions. Developers need to learn several commands and keyboard shortcuts to be fluent when editing, parsing, and analyzing PVS specifications. Means for efficient navigation of the PVS libraries are not provided in PVS Emacs. This can significantly slow down the development of PVS specifications and proofs. For example, the NASA PVS Library [2] includes over 100,000 theorems and definitions useful for modeling and analysis of different aspects of safety-critical systems. Without appropriate tool support, finding anything within these libraries is a prohibitive task.

The main contribution of this work is VSCode-PVS, a new integrated development environment designed to reduce substantially the conceptual and knowledge gaps faced by PVS users. A preliminary but fully functional version of the new environment is presented, along with a comparison with other environments. The environment is publicly available at GitHub[3] under NASA Open Source Agreement.

The rest of this paper is organized as follows. Section 3 introduces the main features and high-level architecture of VSCode-PVS. Section 5 identifies a set of metrics that are general across the front-end of different modeling and analysis tools, and uses these metrics to compare VSCode-PVS to other environments. Section 6 presents related work. Finally, conclusions and future work are presented in Section 7.

2 Background

This section provides background information on PVS Emacs, the standard front-end of PVS, and Visual Studio Code, the baseline technology used for the development of VSCode-PVS.

2.1 PVS Emacs

PVS Emacs is a text-based environment built on the Emacs editor to support user interaction with PVS. Commands necessary to interact with the PVS system are invoked by typing the command name and its arguments, if any, in the Emacs minibuffer. To enter a command in the minibuffer, users need to pre-fix the command with the key sequence `M-x`, where `M` is usually the Alt key. An example is `M-x typecheck`, which executes the PVS type-checker command on the PVS file opened in the Emacs editor. An extensive list of shortcuts is implemented to facilitate data entry. For example, type-checking can be executed by typing `M-x tc` in the minibuffer, or by performing the key sequence `C-c C-t`, where `C-` is the Ctrl key.

2.2 Visual Studio Code

Visual Studio Code is a cross-platform open-source code editor created in 2015 by Microsoft. The editor provides a rich graphical user interface that integrates the essential components typically used by programmers: a source code editor that supports auto-completion, hovers, embedded mini-editors, and contextual menus; an integrated graphical debugger, which allows the user to set break-points and

[2] `https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library`.
[3] `https://github.com/nasa/vscode-pvs`.

perform step-by-step execution of source code; a tree-based view for browsing files in the file system; an integrated terminal for executing commands; and integration with source code management tools such as Git.

The behavior of all these components can be extended or re-programmed to provide support for a specific programming language. The community has already created extensions for over 80 programming languages, including C++, Java, JavaScript, and Python. Besides programming languages, there is also a growing interest in integrating verification tools in Visual Studio Code. Examples include: Dafny[4], Lean[5], Coq[6] and Isabelle[7]. Some of these extensions will be discussed further below, in Section 5.

3 VSCode-PVS

VSCode-PVS is a new integrated development environment for creating, evaluating and verifying PVS specifications. The environment, shown in Figure 1, redefines the way developers interact with PVS, and better aligns the PVS front-end to the functionalities provided by development environments used by software developers. The main features provided by the environment are as follows:

- **Syntax highlighting.** PVS keywords and library functions are automatically highlighted.
- **Autocompletion and code snippets**. Tooltips suggesting function names and language keywords are automatically presented to the user when is typing a symbol name in the editor. Code snippets are provided for frequent modeling blocks, e.g., `if-then-else`.
- **Hover information for symbol definitions**. Hover boxes providing information about identifiers are automatically displayed when the developer places the cursor over an identifier.
- **Jump-to declaration**. Navigation of symbol declarations can be performed with simple point-and-click actions: the user places the cursor over the name of an identifier, and a click on the name of the identifier while holding the Ctrl key down opens a window with the location where the identifier is declared.
- **Live diagnostics**. Parsing is automatically performed in the background, and errors are reported in-line in the editor. Problematic expressions are underlined with red wavy lines. Tooltips presenting the error details are shown when the user places the cursor over the wavy lines.
- **In-line actionable commands**. Actionable commands are available for PVS theorems. They are rendered in-line in the editor, above the name of the theorem, and can be used to start a new prover session for the theorem with a simple click action.
- **Overview of PVS theories**. The overall structure of a set of PVS theories is rendered using an interactive tree-based view. It shows the set of PVS theories in the active workspace, as well as the name and status (proved, unfinished, etc.) of the theorems defined in each theory. Point-and-click actions can be used to jump to theory definitions and type-check the theories.
- **Interactive proof tree visualizer and editor**. An interactive tree-based view shows the proof associated with a theorem. Point-and-click actions are provided for step-by-step execution of proof commands. Functionality for editing the proof are currently under development.

[4] `https://github.com/DafnyVSCode/Dafny-VSCode`
[5] `https://github.com/leanprover/vscode-lean`
[6] `https://github.com/siegebell/vscoq`
[7] `https://github.com/seL4/isabelle/tree/master/src/Tools/VSCode`

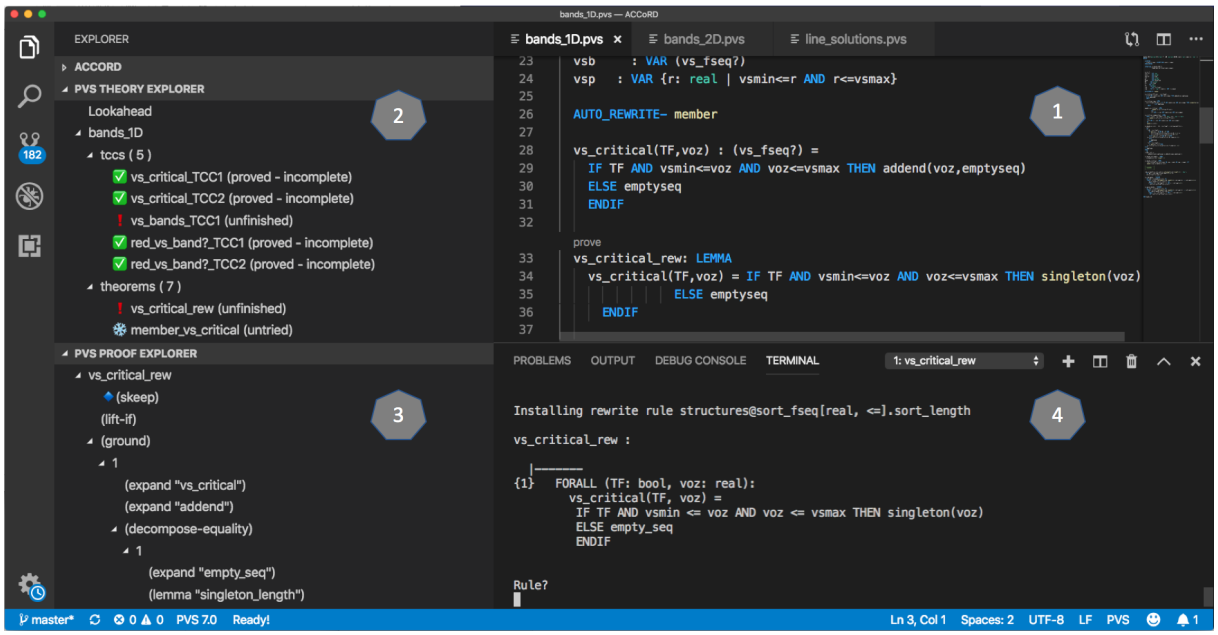


Figure 1: VSCode-PVS: (1) Main Editor; (2) Theory Explorer; (3) Proof Explorer; (4) Integrated PVS Command Line Interface (CLI).

- **Integrated PVS and PVSio Command Line Interfaces**. Integrated command line interfaces allow interaction with the theorem prover and the PVSio evaluator. Auto-completion is provided for prover commands, as well as access to the commands history.

3.1 Architecture

The overall architecture of VSCode-PVS is depicted in Figure 2. It builds on the Language Server Protocol (LSP), a tool-independent communication protocol for exchanging data and events between two architectural elements: an editor front-end and a language server back-end.

The *editor front-end* is responsible for rendering visual feedback to the user, and transforms user interactions with the editor into corresponding LSP events to be dispatched to the language server. The *language server* defines the syntax and semantics of the language (e.g., in the case of PVS, parsing, typechecking, etc.) and continuously listens to LSP events. An example LSP event is `onHover`. This event is triggered by the editor front-end when the user places the cursor over an identifier in the text document. This event is sent to the language server, along with information on the path of the text document and the location of the cursor in the document. The language server acts upon this event, in this case, by sending the identifier's definition back to the editor. The editor, in turn, displays the server response to the user as a hover box. The language server can also generate events. For example, an event `sendDiagnostics`, is used by the server to publish diagnostics information (e.g., parsing error).

The LSP protocol is extensible. It builds on Remote Procedure Calls (RPCs) and the JavaScript Object Notation (JSON) format. It allows the definition of new event types to accommodate language-specific features. In the case of PVS, commands necessary for interactive analysis of PVS specifications, e.g., type-checking, discharging proof obligations, proving theorems, etc.

The LSP-based architecture has been chosen because it promotes reuse of modules and facilitates

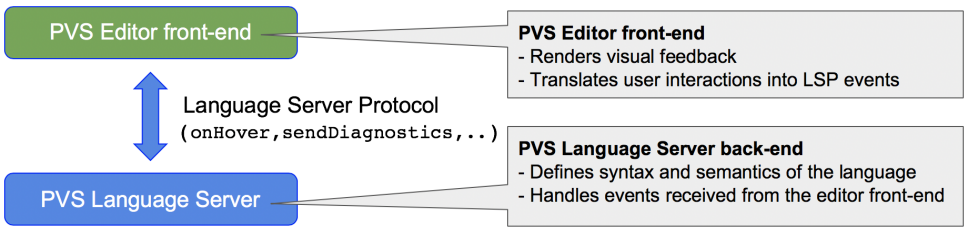


Figure 2: Overall architecture of VSCode-PVS.

sustainability of the overall development effort — any LSP-compliant editor can be connected to any Language Server back-end. All major code editors support the LSP, which makes it relatively simple to connect the PVS language server to a different editor front-end. Features implemented in the language server become automatically available in the connected editor.

3.2 The Editor Front-End

The PVS editor front-end builds on Visual Studio Code[8], an open-source code editor widely used by software developers. The main software modules of the front-end are shown in Figure 3. They are illustrated in the following.

Editor Extensions. These modules are used to tailor the functionalities of the textual editor provided by Visual Studio Code to the PVS language. Functionalities implemented in these modules include: decorators necessary for syntax highlighting; code snippets for rapid creation of PVS code blocks; code folding; key bindings and contextual menus for PVS commands.

Explorer Extensions. These modules customize Explorer View, a graphical tree-based view integrated in Visual Studio Code. A new module, *Theory Explorer*, introduces support for click-and-point navigation of PVS specifications. A second module, *Proof Explorer*, aims to support interactive visualization and editing of proofs using click-and-point operations.

Integrated PVS Terminals. These modules seamlessly link the integrated terminal of Visual Studio Code to the interactive read-eval-print loops of the PVS theorem prover and the PVSio [7] evaluator. That is, in the *prover terminal*, developers can type proof commands for the theorem prover, and watch the proof state returned by the prover directly in the terminal. Similarly, in the *evaluator terminal*, developers can type ground expressions and thus animate executable fragments of a PVS specification.

VSCode APIs. This is a library provided by Visual Studio Code for extending and customizing the functionalities of the editor. It includes communication primitives necessary to support the LSP.

3.3 The Language Server

The PVS Language Server includes the following modules (see Figure 4):

LSP Service Providers. These modules handle LSP events received by the language server. They reconcile the APIs of PVS with the logic of the LSP. For example, *Hover Provider*, which is activated when an `onHover` event is received, uses the APIs of PVS to gather the information to be shown in hover boxes, and then sends the response to the editor front-end using the LSP format. *Definition Provider* defines the logic necessary to support functions such as go-to definition and peek definition, which are

[8] `https://code.visualstudio.com`

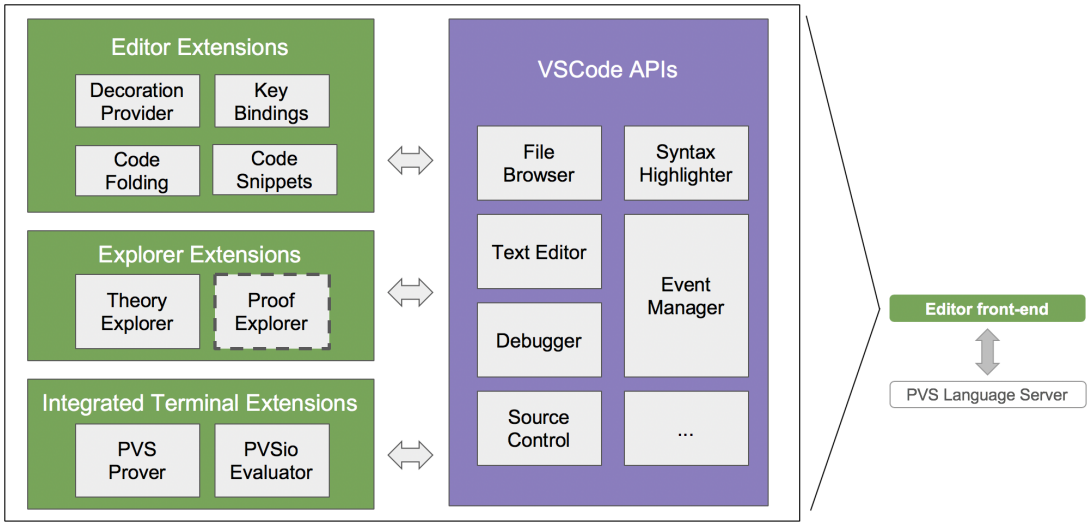


Figure 3: Inner architecture of the PVS editor front-end. Modules are represented as boxes. Communication between modules is indicated with arrows. Dashed line indicates under development.

used in the editor front-end for point-and-click navigation of PVS specifications. *CodeLens Provider* defines the logic behind actionable commands embedded in-line in the PVS specification — this is used to introduce in-line actionable *prove* commands at the location of theorem definitions. *PVS Commands Provider* provides support for language-specific commands used for analysis of PVS specifications, e.g., type-check, show proof obligations, prove theorem. *Diagnostics Provider* is a background process that continuously sends diagnostics information to the editor front-end to report syntax / type-checking errors.

LSP Connection Manager. This is a routing module for managing the exchange of events and data with the client front-end. The module listens for client connections, receives LSP events from connected clients, and dispatches the events to the appropriate service provider.

PVS Process Workers. These modules embed the functionalities of the PVS verification system in the language server. Each worker executes a PVS instance in a self-contained execution environment. A pool of workers supports parallel execution of multiple PVS instances — this is used for running background services like parsing and type-checking. Furthermore, in contrast to PVS Emacs, VSCode-PVS supports running simultaneously proofs of different formulas.

3.4 Implementation

VSCode-PVS is implemented in TypeScript[9], an extension of JavaScript that supports type annotations. Typescript programs can be statically checked for type correctness. A transpiler translates TypeScript code into plain JavaScript, which allows the execution of TypeScript programs in standard JavaScript engines. The editor front-end builds on the APIs of Visual Studio Code. The language server builds on NodeJS[10], a JavaScript environment that provides libraries necessary for creating web-services, including functions for spawning processes and performing operations on file systems. Process workers use the native Lisp interface of PVS to exchange commands and data with the PVS system. TSLint[11], a static

[9] `https://www.typescriptlang.org`
[10] `https://nodejs.org`
[11] `https://palantir.github.io/tslint`

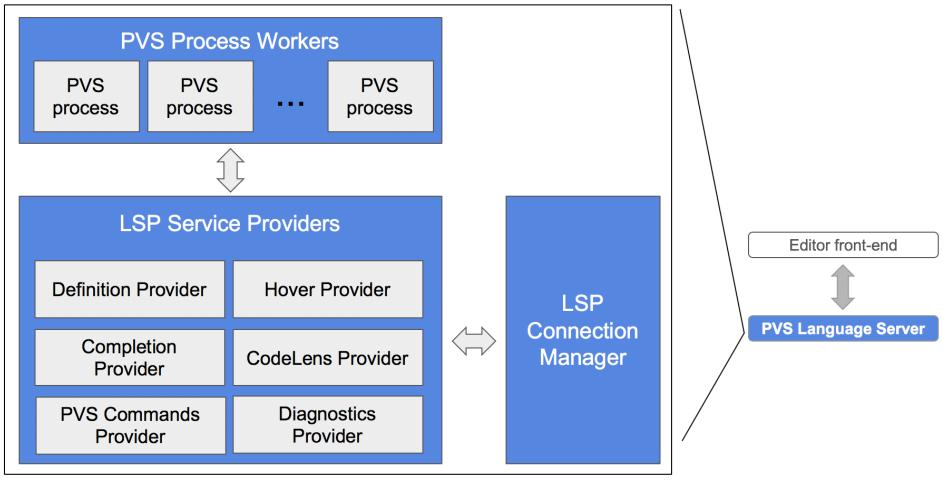


Figure 4: Inner architecture of the PVS Language Server.

analyzer for TypeScript, is routinely used for checking compliance with established coding conventions. Jasmine[12] is used for testing the APIs of the developed modules.

The implementation effort to date amounts to 7K LoC (3K LoC for the editor front-end, and 4K LoC for the language server). Most of the code developed for the editor front-end is associated with the interactive tree-based views for visualizing theories and proofs. The server back-end required the development of a Lisp interface to dialogue with PVS, as well as additional logic for LSP events that are not directly supported by PVS, e.g., auto-completion for identifiers, hover information, live diagnostics.

4 Example use of VSCode-PVS

This section showcases the main features of VSCode-PVS for two representative tasks typically carried out by PVS users. Tasks are provided in the form of short descriptions presenting the overall *goal* of the task and the *context* within which the task is carried out. A comparison with the standard PVS Emacs front-end is included in each task to better appreciate the improvements introduced by VSCode-PVS.

4.1 Task 1: Navigation of symbol declarations

Goal: Inspect function and type declarations from imported PVS files.

Context: When developing a new PVS specification or a new proof, users typically need to navigate symbol declarations imported from other PVS theories. This is necessary, e.g., to inspect the structure of complex datatypes defined in the PVS libraries.

Workflow with VSCode-PVS. The standard workflow for inspecting a symbol declaration involves using the hover functionality (see Figure 5b). The user can place the cursor over the symbol definition, and a tooltip will be automatically shown. The tooltip includes three main elements: a brief description of the symbol (e.g., built-in type); a clickable link for jumping to the location of the declaration; a preview of the symbol declaration. This standard workflow can always be adopted when a PVS specification type-checks correctly. An alternative workflow is also available for theories that are still not type-checked,

[12] `https://jasmine.github.io`

e.g., because the user has not finished yet typing the content of the specification. In these cases, the resolution of symbol declarations can still be performed, but can be less accurate when symbol names are overloaded. That is, when the symbol to be resolved is overloaded and the theory is not type-checked, an array of candidate declarations is presented. In these cases, the *peek declaration* functionality is employed (see Figure 5c). It opens a mini-editor and a file browser in the current editor window that can be used to inspect the candidate declarations.

Workflow with Emacs. Symbol declarations can be inspected with the command `show-declaration`. The command takes the name of the symbol as argument. The current location of the cursor can be used to auto-complete the symbol name. The command opens a new Emacs buffer with a preview of the declaration. The command `goto-declaration` can then be used to jump to the location of the declaration. The command takes the name of the symbol as argument. When the theory is not type-checked, a command `find-declaration` can be used to inspect a list of possible candidates. The jump-to functionality is however not available in this case, and the user needs to manually open the file (command `C-c C-f` followed by the filename) and scroll the text to the position of the declaration.

4.2 Task 2: Proving a theorem

Goal: Verify that a PVS specification satisfies given formal (mathematical) properties.

Context: For complex systems, it is important to analyze a system design before the actual system is built. This helps developers gain confidence that the system design complies to given specifications, and identify and fix potential design issues early in the development process, when the cost of design changes is still relatively low. In safety-critical application domains, such as avionics and healthcare, such design analysis is usually mandated by regulatory frameworks. Proving mathematical theorems that capture properties of the intended characteristics and functionalities of the system is the core approach used in formal methods. It provides means to check properties of a system design for all possible inputs in all possible system states.

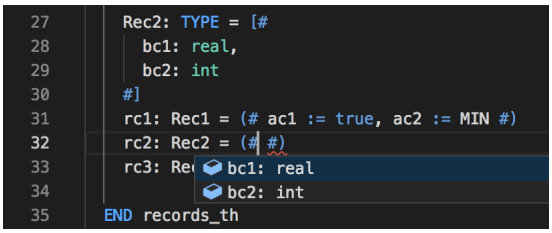
Workflow with VSCode-PVS. In-line actionable commands are provided next to each theorem. For example, in Figure 1, an actionable command `prove` is shown in the main editor window, above the theorem name at line 33). A click on the actionable command, triggers type-checking, and launches a new prover sessions in the integrated PVS Command Line Interface (see Figure 1, lower-right panel). Proof commands can be typed in the command line interface. The current proof is shown on the side, using an interactive tree-based view, called Proof Explorer (see Figure 1, lower-left panel). Nodes in the tree view can be collapsed/expanded to facilitate inspection of large proofs. Editing of the proof tree from Proof Explorer is under development.

Workflow with Emacs. The user needs to type-check the file (`M-x tc`) and then start a theorem prover session with the command `M-x prove`. The command opens a new Emacs buffer that can be used to interact with the theorem prover. Proof commands are typed in this new buffer. A command `M-x x-show-current-proof` can be used to start open a window showing the proof tree (see Figure 6d). The proof tree cannot be edited, and it does not allow collapsing/expanding of proof branches.

5 Comparing VSCode-PVS to Other Analysis Tools

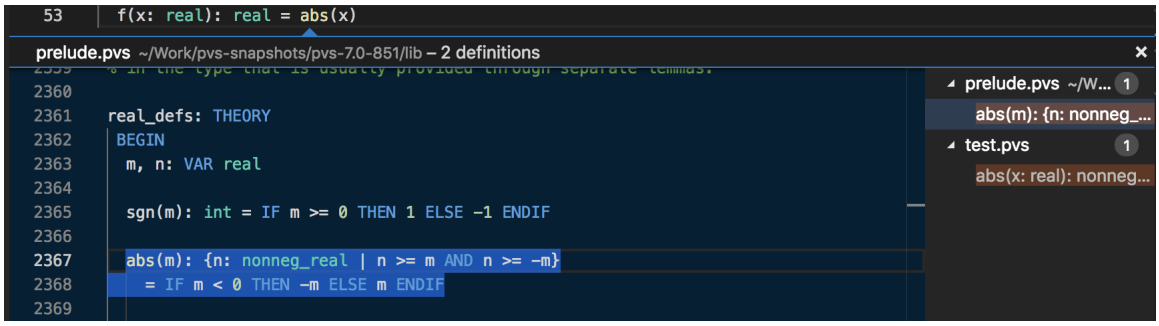
This section presents a comparison between VSCode-PVS and other similar environments. The following verification environments are considered:

- PVS Emacs, the standard front-end of PVS;



(a) Autocompletion.

(b) Hover.



(c) Peek definitions.

Figure 5: Example functionalities of VSCode-PVS.

- Isabelle/jEdit [14], the standard front-end of the Isabelle/HOL theorem proving system;
- SublimeHOL, the standard front-end to the HOL theorem prover;
- CoqIDE, the standard front-end of the Coq theorem proving systems;
- Proof General [1], a generic front-end for theorem provers;
- Lean [6], a new open-source theorem proving system from Microsoft;
- KeYmaera-X [2], an interactive theorem prover for hybrid systems.

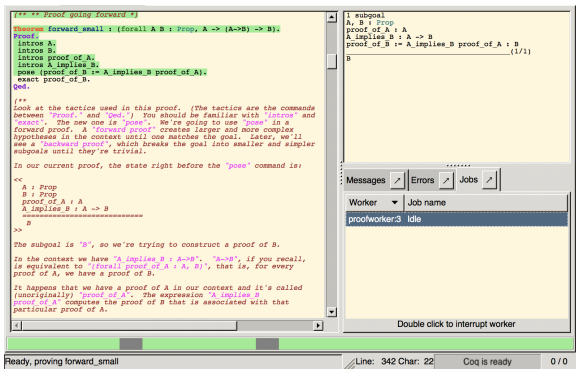
5.1 Metrics

A set of metrics has been defined to guide the comparison. The set of metrics is not exhaustive. Rather, they capture core functionalities necessary to support common modeling and analysis tasks.

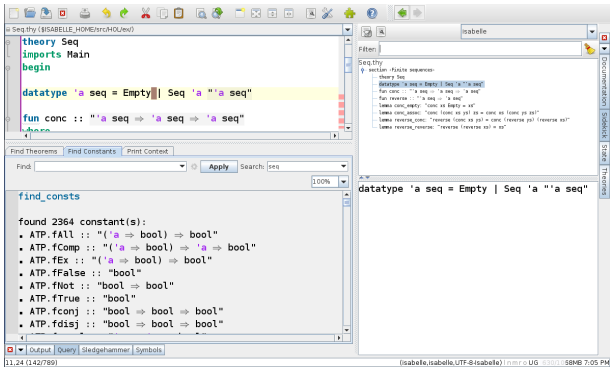
1. **Autocompletion**: ability to suggest keywords and identifiers while typing;
2. **Hover information**: ability to show informative pop-ups for keywords and identifiers;
3. **Jump-to-definition**: ability to open a file at the location of an identifier's definition;
4. **Refactoring**: ability to rename identifiers;
5. **Live diagnostics**: ability to show diagnostics information while typing;
6. **Animation**: ability to test executable specifications;
7. **Proof visualizer**: ability to visualize a proof tree;

5.2 Assessment

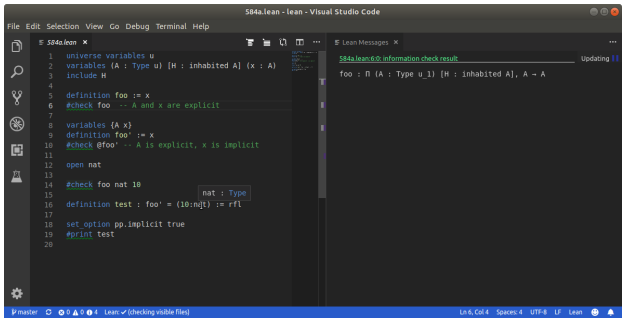
VSCode-PVS. Autocompletion is provided for language keywords, as well as for types and functions defined in the standard PVS library (the `prelude`). Context-sensitive autocompletion is supported for



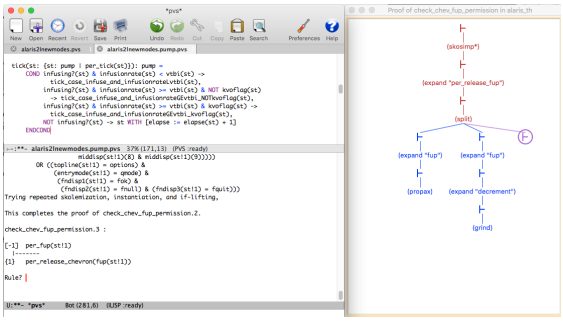
(a) CoqIDE: editor and side view for proof inspection.



(b) Isabelle/jEdit.



(c) Lean.



(d) PVS Emacs: editor and proof window.

Figure 6: Screenshots of other analysis tools.

record types: items suggested by the editor range over record accessors (see Figure 5a). Hover information is provided for identifiers. It shows a preview of the definition of the identifier, as well as an hyperlink that can be used to jump-to the location of the definition (see Figure 5b). Live diagnostics for syntax errors are automatically provided after a period of inactivity in the editor — syntax errors are underlined with red wavy lines, and a tooltip with details on the error is provided when placing the cursor at the error position. Integrated terminals can be used to animate specifications and prove theorems. An interactive tree-based view allows for the visualization of proof trees and proof commands step-by-step.

PVS Emacs. The environment, shown in Figure 6d, provides syntax highlighting and a basic form of jump-to-definition through commands and key bindings. A basic form of autocompletion is provided for proof commands. The logic for hover information is not implemented. Diagnostics are obtained on-demand, when the developer decides to parse or type-check the specification. Animation is provided through an interactive read-eval-print loop. Proofs can be visualized in a tree-based view. Nodes in the tree represent either proof sequents or proof commands. An interactive text-based view is used to show proof tactics and allows the user to perform step-by-step execution of proof commands.

CoqIDE. The environment is shown in Figure 6a. It provides syntax highlighting. Autocompletion, hover information, and jump-to-definition are not provided. Diagnostics are provided on-demand, when the developer decides to attempt verification. A linear workflow is enforced: verification always starts from the beginning of the file, and a marker is advanced in the editor to indicate what has been verified. Everything above the marker is locked and cannot be modified. A side panel shows the proof state, including current goal, tactic state, and error messages. Point-and-click operations allow step-by-step

execution of proof commands. An alternative front-end, vscoq[13], based on Visual Studio Code is currently being developed for Coq. However, the implementation is at the very early stages and does not provide significant new functionalities with respect to the standard CoqIDE.

Proof General. The environment builds on Emacs, and its visual appearance is similar to PVS Emacs. The main design goal of this environment is to facilitate the execution of proof commands and the navigation of proof strategies. A plug-in based architecture allows the introduction of language-specific extensions. Plug-ins for Coq and Isabelle/HOL are available. They provide syntax highlighting and basic forms of autocompletion. Proof visualization builds on a tree-based view similar to that used in PVS Emacs. Commands are provided for step-by-step execution of proof tactics.

SublimeHOL. The environment builds on the Sublime[14] editor. It provides syntax highlighting for the HOL specification language. A basic form of autocompletion is provided for keywords and mathematical symbols. Hover information, jump-to-definition and visualization of proof trees are not provided. Interactive panels allow the exchange of commands with HOL and edit/inspect the proof state using a read-eval-print loop.

Isabelle/jEdit. The environment, shown in Figure 6b builds on jEdit. Context-sensitive autocompletion is provided based on the syntax of the language, as well as on name-space information provided by the prover engine. For comments and other sections in the specification that contain sentences in natural language, a dictionary-based spell-checker is used to suggest completion items. Hover information shows the type of language symbols. Hover boxes can be detached from the current editor and turned into separate windows to facilitate navigation of file content. Live diagnostics are provided for errors in the form of red wavy lines, along with hover information at the error location. A panel shows proof information and allows the user step-by-step evaluation of a proof tactic. A query panel enables the filtering of information displayed for the proof state.

Lean. The environment is shown in Figure 6c. It builds on Visual Studio Code. Because of this, its overall visual appearance is similar to VSCode-PVS. Context-sensitive autocompletion is provided for language symbols. A reasoning engine is always active in the background and tries to autocomplete expressions based on context information, in a manner similar to the *hole* functionality provided in Haskell. Hover information shows the definition of identifiers. Navigation of definitions is supported by the jump-to functionality and mini-editors rendered in line in the current editor. Live diagnostics are provided for errors, in the usual form of red wavy lines at the location of the error, and hover information reporting information about the error. Proof commands are embedded in the specification and can be activated with point-and-click operations. Similarly to Coq and jEdit, the proof state is shown in a side panel.

KeYmaera-X. The environment uses a web-based front-end. Basic functionalities are provided to support modeling tasks: syntax highlighting is provided only for language keywords; hover information, jump-to definition and refactoring are not available. The prover interface renders proofs in sequent form, with horizontal lines as in the Gentzen-style layout. An extensive set of menus provides access to all proof commands. Heuristics are used to suggest proof tactics during proof attempts.

5.3 Results

An overview of the comparison is in Table 1. It can be seen that VSCode-PVS already provides several features that other similar environment are still missing. Most of the environments are mainly designed

[13] `https://github.com/siegebell/vscoq`
[14] `https://www.sublimetext.com`

	Version	Base editor	Autocompletion	Hover information	Jump-to-definition	Refactoring	Live diagnostics	Animation	Proof visualizer
VSCode-PVS	1.0.12	VSCode	●	●	●	○	●	●	●
PVS-Emacs	6.0	Emacs	◒	⊗	◒	○	⊗	●	●
CoqIDE	8.9.0	N/A	⊗	⊗	⊗	⊗	⊗	⊗	◒
SublimeHOL	2018	Sublime	◒	⊗	⊗	⊗	⊗	⊗	◒
Proof General	4.5	Emacs, Eclipse	◒	⊗	⊗	⊗	⊗	⊗	◒
Isabelle/jEdit	2018	jEdit	●	●	●	○	●	⊗	●
Lean	0.14.1	VSCode	●	●	●	⊗	●	⊗	●
KeYmaeraX	4.6.3	N/A	⊗	⊗	⊗	⊗	⊗	⊗	●

Table 1: Overview of the comparison results. The following symbols summarize the characteristics of a feature: fully implemented (●); basic implementation (◒); planned feature (○); not available (⊗).

to provide an interface for exchanging proof commands with the theorem prover. Few environments provide adequate support for modeling activities. For example, all environments currently lack *refactoring*, and developers need to rely on search-and-replace functionalities of the editor when renaming identifiers. However, this solution is not robust, as careful inspection is necessary for overloaded identifiers. Two other important features commonly used during modeling activities are also missing in most of the environments: *animation* of executable fragments and *live diagnostics*. Animation provides a means to developers to test a specification, e.g., to check whether it correctly captures what the developer wants to model. This functionality can be especially useful to software engineers that approach formal verification, as it is congruent with the testing methods they routinely use for software. It also provides a form a lightweight formal verification — properties can be checked for specific execution traces prior to running the full formal proof. Live diagnostics promote immediate identification of specification errors. This may facilitate understanding and resolution of errors, as a developer's focus of attention is already at the location of the error.

6 Related Work

VSCode-PVS aims to align the functionalities of the PVS front-end to those of program analyzers such as Dafny [3], or reasoning engines like Imandra [11]. The front-end of these tools provides all functionalities typically available in modern IDEs for programming languages, including context-aware help and an integrated debugger. Verification is carried out in the background, by continuously querying a pool of solvers while the user types the code. Design solutions are adopted to keep the interface responsive and provide an overall smooth programming experience to the user.

The verification technology used by VSCode-PVS is not automatic as in the case of Dafny and Imandra. However, when the PVS analysis targets routine tasks such as discharging proof obligations necessary to prove type correctness, automatic analysis is usually feasible thanks to the powerful proof strategies provided by PVS. This opportunity needs to be exploited, as it would allow the completion of simple but time-consuming activities that developers need to carry out while developing a formal specification. Appropriate mechanisms need to be developed to limit the use of CPU time and memory

resources that could make the interface not responsive. The possibility of creating an integrated debugger for executable fragments of PVS specifications is another interesting option that needs to be explored for VSCode-PVS. The Visual Studio Code editor provides already the graphical elements necessary for interacting with the logic of the debugger, including break-points, an interactive panel with the usual run/step-into/step-over commands, as well as an interactive view for inspecting the value of variables, call stack, etc. These elements need to be customized for the PVS language, and appropriate hooks need to be implemented in the back-end to provide the logic necessary for debugging.

In [12], a Proof General plugin is developed that introduces syntax highlighting and autocompletion for the Coq specification language. In [13], a generic user interface for theorem proving systems is introduced. The editor front-end builds on jEdit, and a prototypical specification language for declaring formal terms such as theories, terms, and context. This approach proves useful to implementing a generic version of basic front-end features such as autocompletion, abstract syntax display, error highlighting, and tooltips. These and other similar efforts are certainly worth exploring. However, it is unclear if in the long run they will stand against the rapid evolution of editors such as Visual Studio Code and Atom.

7 Conclusion and Future Directions

A new development environment for the PVS verification system has been presented that aims to align the PVS front-end to that of main stream tools used by software developers. A split architecture is adopted, where an editor front-end communicates with a server back-end. The back-end uses process wrappers to adapt the APIs of PVS to the Language Server Protocol, a de-facto standard communication protocol for code editors and analyzers. The editor front-end builds on the the features of Visual Studio Code, a modern open-source code editor.

VSCode-PVS is under active development. The environment is still in its infancy, but it already advances the standard Emacs front-end of PVS in many respects — live diagnostics, context-sensitive auto-completion, point-and-click navigation, interactive tree-based view for proof exploration.

Previous attempts carried out by others to develop a new front-end for PVS had little success. One attempt aimed to integrate PVS in Eclipse. Difficulties were encountered to align the APIs provided by PVS to the functionalities required for Eclipse, and the development was ultimately abandoned. Another attempt involved the development of a Python front-end for PVS, using the wxPython[15] graphic library. A simple interface was created to exchange commands with PVS. Fragments of these implementations can be found in the GitHub repository of PVS[16].

Current work on the VSCode-PVS front-end focuses on the integration with the next release of PVS, which provides a new XMLRPC interface that will improve performance and robustness of the language server. The creation of an integrated debugger is also planned. It will align the functionalities of the PVSio evaluator to those of debuggers used in programming languages. Integration with the PVSio-web [4] prototyping environment is another future direction. PVSio-web enables the creation of interactive prototypes based on formal models. The prototypes resemble the visual appearance of a final system. They can be used to create scenario-based simulations that facilitate engagement between PVS experts and developers that are not familiar with PVS or formal methods.

[15] `https://wxpython.org`
[16] `https://github.com/SRI-CSL/PVS`

References

[1] David Aspinall (2000): *Proof General: A generic tool for proof development*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 38–43, doi:10.1007/3-540-46419-0_3.

[2] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp & André Platzer (2015): *KeYmaera X: An axiomatic tactical theorem prover for hybrid systems*. In: *International Conference on Automated Deduction*, Springer, pp. 527–538.

[3] K Rustan M Leino (2010): *Dafny: An automatic program verifier for functional correctness*. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Springer, pp. 348–370, doi:10.1007/978-3-642-17511-4_20.

[4] Paolo Masci, Patrick Oladimeji, Yi Zhang, Paul Jones, Paul Curzon & Harold Thimbleby (2015): *PVSio-web 2.0: Joining PVS to HCI*. In: *International Conference on Computer Aided Verification*, Springer, pp. 470–478, doi:10.1007/978-3-319-21690-4_30.

[5] Paolo Masci, Yi Zhang, Paul Jones, Paul Curzon & Harold Thimbleby (2014): *Formal verification of medical device user interfaces using PVS*. In: *International Conference on Fundamental Approaches to Software Engineering*, Springer, pp. 200–214, doi:10.1007/978-3-642-54804-8_14.

[6] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn & Jakob von Raumer (2015): *The Lean theorem prover (System Description)*. In: *International Conference on Automated Deduction*, Springer, pp. 378–388, doi:10.1007/978-3-319-21401-6_26.

[7] César A Muñoz & Ricky Butler (2003): *Rapid prototyping in PVS*. Technical Report, NASA/NIA. NASA/CR-2003-212418, NIA Report No. 2003-03.

[8] César A. Muñoz & Ramiro A. Demasi (2012): *Advanced Theorem Proving Techniques in PVS and Applications*. In Bertrand Meyer & Martin Nordio, editors: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 96–132, doi:10.1007/978-3-642-35746-6_4. Available at `https://doi.org/10.1007/978-3-642-35746-6_4`.

[9] Sam Owre, John M Rushby & Natarajan Shankar (1992): *PVS: A Prototype Verification System*. In: *International Conference on Automated Deduction*, Springer, pp. 748–752, doi:10.1007/3-540-55602-8_217.

[10] Maurizio Palmieri, Cinzia Bernardeschi & Paolo Masci (2018): *Co-simulation of Semi-autonomous Systems: TheLine Follower Robot Case Study*. In Antonio Cerone & Marco Roveri, editors: *Software Engineering and Formal Methods*, Springer International Publishing, Cham, pp. 423–437, doi:10.1007/978-3-319-74781-1_-29.

[11] Grant Olney Passmore & Denis Ignatovich (2017): *Formal verification of financial algorithms*. In: *International Conference on Automated Deduction*, Springer, pp. 26–41.

[12] Clément Pit-Claudel & Pierre Courtieu (2016): *Company-Coq: Taking Proof General one step closer to a real IDE*. In: *CoqPL'16: The Second International Workshop on Coq for PL*, Zenodo, doi:10.5281/zenodo.44331. Available at `https://doi.org/10.5281/zenodo.44331`.

[13] Florian Rabe (2014): *A Logic-Independent IDE*. *Electronic Proceedings in Theoretical Computer Science* 167, p. 4860, doi:10.4204/eptcs.167.7. Available at `http://dx.doi.org/10.4204/EPTCS.167.7`.

[14] Makarius Wenzel (2012): *Isabelle/jEdit – A Prover IDE within the PIDE framework*. In: *International Conference on Intelligent Computer Mathematics*, Springer, pp. 468–471, doi:10.1007/978-3-642-31374-5_-38.