A Lightweight Theorem Prover Interface for Eclipse

Julien Charles [1]

Everest Group
INRIA Sophia Antipolis
2004 Route des Lucioles - BP 93
FR-06902 Sophia Antipolis, France

Joseph R. Kiniry [2]

Systems Research Group
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland

Abstract

A major deliverable of the EU FP6 FET program MOBIUS project is the development of an Integrated Verification Environment (IVE)—the synthesis of a programming-centric Integrated Development Environment (IDE) with a proving-centric Interactive Theorem Prover (ITP). This IVE focuses on Java verification. Therefore, Eclipse was chosen as the IDE in which to integrate the system.

In this paper we present ProverEditor, a system used to interact with theorem provers from within Eclipse. It is similar to the Proof General Toolkit for Eclipse, except that it has a much more lightweight architecture, and consequently less features and more flexibility. In this paper we summarize its main functionality, as well as the plugin for the initial and primary prover that is well-supported, Coq. We also summarize the system's architecture and discuss our work on integrating other ITPs, PVS in particular.

Key words: interactive theorem prover, Eclipse, IDE, higher-order, interface, editor

[1] Email: julien.charles@sophia.inria.fr
[2] Email: joseph.kiniry@ucd.ie

1 Introduction

Developing proofs using proof assistants can be a peculiarly difficult task. Even using current modern tools, formulating a moderately complex proof is sometime not easy. This difficulty is particularly noticeable in modern software verification efforts that regularly use theories with hundreds of, and sometimes (many) thousands of, definitions and theorems.

For instance, the Coq proof assistant has three main user interfaces [3] :

- CoqTop, which is just a LISP-like command-line top-level. While somewhat useful, it is a bit awkward to use and very limited in functionality.
- CoqIDE, which supports editing and evaluating a specific Coq proof script. While this interface offers some facilities for automatically constructing proof scripts and finding help, it is still quite difficult to manage the aforementioned complex theories and, e.g., navigate through hierarchies of proof files.
- Proof General, a major mode for Emacs, which has many features and is a very rich interface, leveraging the power and flexibility of Emacs. Ironically, its dependence on Emacs is one of its main drawbacks, as, in recent years, IDEs like Eclipse are replacing older tools like Emacs as the standard environments for developing software.

Ideally, one would just make Proof General a part of Eclipse. The Proof General Toolkit represents one such effort. Unfortunately, at this time Proof General Toolkit only supports the Isabelle theorem prover. Thus far, no other prover is supported by Proof General Toolkit within Eclipse, mainly because the inclusion of a new proof assistant is difficult due to the complexity of the Proof General Toolkit architecture and its communication protocol(s). (An attempt was made with Coq, but it seems to be discontinued.)

In this paper we present ProverEditor, a multi-prover interface for Eclipse. The architectural approach chosen here turns the Proof General Toolkit architecture on its head—our architecture is very *lightweight* and only a *simple interface* must be implemented to support the integration of a new proof assistant. Not only is proof script creation and editing supported, but interactions with the prover are enabled via a formally specified API that support communication with integrated and automatic provers via Java. ProverEditor also has some more advanced features like the Eclipse's outline view and a completion system.

The paper is divided as follows: first we provide a detailed overview of some existing tools and environments. Second, we review the design and architecture of our system. Third, we describe the ProverEditor plugins and features currently available, and summarize how they are similar to, and different from, the interfaces familiar to those that use theorem provers daily. Finally, we conclude with reflections on ongoing development work and next steps in integrating IDEs and ITPs.

[3] There is actually two others, PCoq [3], that is written in Java, and CtCoq, but they are not maintained anymore.

2 User Interfaces for Theorem Proving

There are several different canonical interfaces to interactive theorem provers. We first summarize these interfaces to (a) put ProverEditor in context, (b) compare and contract it with other interfaces, and (c) identify the interfaces and feature interactive provers have in common so as to drive our own architecture design.

2.1 Command-line Interfaces

The provers we are targeting (e.g., Coq, PVS, and Isabelle) have command-line interfaces as some top-level. These top-levels have many similarities. Each allows one to send commands to the prover and receive its answers using ASCII byte sequences and simple syntaxes. Usually the standard output file descriptor is used for the dialog and the standard error file descriptor is used to show the prompt.

A typical "raw" interaction at a top-level is to open a text file (e.g., in a text editor) where one stores all the command steps involved in a given interaction and cut-and-paste its content to the top-level and await results from the prover. This "user-active" kind of interaction is quite unnatural and ungainly. Therefore these low-level interactions are often wrapped in a richer environment like Emacs, or in the case of Coq, its own IDE, CoqIDE.

2.2 Web interface

ProofWeb [9] is a multi-prover web interface. It handles several provers notably Coq, Isabelle and Matita. It is mainly targeted towards students, hence it permits to interact with it without any local installation. It proposes several views to view proofs. It has also a system to access courses and predefined Coq files containing exercises.

Its architecture is similar in some ways to the one of ProverEditor as well as Proof General. The way it handles provers is plugin based, and the interaction is mostly handled plugin side.

This tool is really good for small sized projects and for teaching-oriented use of theorem provers, but it is not the best tool for managing large projects and for library forging. It only permits the edition of one file at a time, it does not handle directory, and it does not allow rich client type interaction. For instance, if a user wants to develop a proof of a program, he will not be able to edit the program and edit the proof obligation in the same environment. He will have to switch context back and forth from the code editor to the proof script editor. It will be worst if the user want to modify the program, and re-generate the proof obligation. The user will have to load the file by hand afterward through the interface. These kinds of problems are partially solved using Emacs as the host for the multi-prover interface, and they are totally solved if using Eclipse [4].

[4] An exemple of rich client type interaction can be found in the tool MOBIUS' DirectVCGen on the MOBIUS Trac server [15]

2.3 Emacs

Emacs a relatively popular tool for computer scientists and programmers. It is now a bit overshadowed by more recent tools like Eclipse, and it is, in part, because of this "popularity with the masses," that we are currently targeting Eclipse as an integration platform.

The core features of modern IDEs that we retain for ProverEditor are the outline/summary view (which is not built-in to Emacs, though is available via the use of the Speedbar and/or CEDET tools), as well as the quite useful "tagging" (completion) system. Especially for development in C/C++ or Java, tagging is a very valuable feature. Eclipse replaces tagging by more semantically-aware form of definition searching and completion akin to Microsoft's "Intellisense" (Microsoft's implementation of autocompletion [5]).

2.4 Eclipse

The Eclipse platform is used like Emacs as a front-end for an Integrated Development Environment or, more generally, as a Rich Client Platform [13]. At first Eclipse focused on Java development. Now, it is used for C/C++ and Python development, as well as a front-end for revision control tools like CVS and Subversion and writing papers in LaTeX.

As Eclipse represents a "modern" development tool, several standard concepts are used: files are grouped in projects, it has an outline/summary view, and navigating source code is simplified via implicit definition retrieving. We believe that, to gain mind-share with today's programmers, it is good idea to be hosted in Eclipse, rather than in Emacs, both for these key features, as well as more social reasons.

2.5 Proof General

Proof General [4] is the *de facto* multi-prover environment. It uses Emacs as its graphical interface. It is multi-prover in the sense that it allows one to interact with Isabelle and Coq as well as other provers. It does not have a lightweight approach: each theorem prover-specific part uses its own communication engine, and each prover language has some of its semantic aspects encoded into this engine.

The new version of Proof General is called Proof General Toolkit [8], and it is an Eclipse feature/plugin. It bases its interactions with provers on the interaction it already supports with Isabelle. This interaction protocol is called PGIP, which uses an XML-based language called PGML.

By using XML, command streams are well-delimited and easy to parse in the Java context using a standard XML parser. This permits to add some of the prover language semantics inside the tags. A typical example of such an addition is the use of a vernacular-like `Show` command within a proof script—the XML around

[5] http://msdn2.microsoft.com/en-us/library/hcw1s69b(vs.71).aspx

this command is independent of the command and the proof script, which makes for a more script-independent protocol.

An interesting application of this PGML might be to support a new feature, like prover-centric refactoring, e.g., scope-aware variable renaming. A problem with realizing such a new feature is that the PGIP-based approach is, in some sense, too heavyweight to be generic. In order to communicate with the prover, the interface must maintain some semantics, in particular, it must have some theory and proof context information.

Since Proof General Eclipse uses the PGIP protocol to communicate with provers, supporting a new prover means realizing this protocol for that prover. Unfortunately, implementing this protocol seems to be not such an easy task for anyone other than the Proof General Toolkit developers, especially for system not written with these kind of interactions in mind. To have the full output and a good view of the state of Coq one must gather two informations: the ones collected from the standard output and the ones from the error output. In common cases these outputs do not interleave but in some specific situations the interleaving is unexpected. Hence it is difficult to gather to a single output and reordering informations for the PGIP protocol. In the same vein, PGIP does the hypothesis that the undo stack is infinite, which is not the case for Coq. One could argue to use instead the PCoq output of Coq which is supposed to offer all the facilities missing, but this output is not properly maintained, so using Coq standard interaction output is safer.

All these problems for adapting Coq to the PGIP protocol have led us to use a simpler protocol much similar to what has been done in Proof General for Emacs but in a lighter and more generic fashion.

3 Analysis and Design

ProverEditor is part of the MOBIUS Program Verification Environment (PVE). The MOBIUS PVE is an integrated environment that supports the specification, implementation, and verification of Java programs. Because the Mobius PVE focuses on Java, the Eclipse platform was a natural choice for a programming environment in which to host the Mobius PVE.

This integration has several aspects:

- Eclipse has a well-defined plugin architecture, so extending the system with rich functionality and interfaces is relatively straightforward, and
- Mobius PVE subsystems can extend and interact easily with Eclipse's Java programming components.

In this section, we present the integration of ProverEditor inside of Eclipse and then summarize its main features.

3.1 Plugin Architecture

ProverEditor's architecture is based on the Eclipse plugin architecture. Fundamentally, ProverEditor is an Eclipse plugin that hosts several other plugins which are specific to the provers with which it interacts.

3.1.1 Plugins in Eclipse

Eclipse's plugin architecture's power and flexibility principally lies in its *extension points* concept. An *extension point* is a facet of a component's interface, thus a plugin can either provide an extension point or extend an extension point of another plugin. Extension points are often used to implement a specific behaviour or, as is often the case in Eclipse, to support the proper integration of a plugin into Eclipse's graphical interface. For instance, if one wants a plugin to be integrated to the preferences menu, an extension point must be used to add a the plugin's preferences page to Eclipse's preferences menu.

This compositional modularity is very useful in combining several facilities provided by other plugins, like synthesizing a generic interface to two similar systems, like the CVS and Subversion revision control systems. In our case, it simplifies the adoption and adaptation of modern IDE features in our interface, as initially advocated by Kiniry [11]. While such adoption and adaption is the hallmark of extensions to Emacs, due to the richer foundations (and consequent greater use of resources) of Eclipse and Java, Eclipse plugins can be more powerful and attractive than their Emacs-based counterparts.

Extension points are defined by a unique identifier that links to a definition of an XML tag. Programmers that wish to extend this extension point must write an XML file called `plugin.xml` (typically with the assistance of a specialized plugin editor that comes with Eclipse) as well as provide the correct parameters to the tag in question. The plugin providing the extension point inspects, at runtime, what was specified with the XML tag. The name of a class to classload is typically provided, and consequently instantiated, otherwise the name of a resource or a simple `String` may be provided.

Eclipse provides many extension points: e.g., editors for specific file types, binding a file type to a specific icon, and syntax highlighting. One can also add specialized widgets to the interface, like the standard "outline" view that so many IDEs provide. A drawback of using extension points is fundamental to their design: they are, in essence, static and global. In addition, as stated above, extension points can only be provided through the addition of a new plugin with a plugin specification file. Thus, in general, Eclipse does not permit any mechanism for (dynamic) extension.

3.1.2 Plugins Used

ProverEditor is based on different plugins which define different subsystems: the editor, the outline, the proof view and the preference page.

The editor is based on Eclipse's integrated editor plugin. The standard Eclipse

file editor is extended to view and edit proof scripts by (a) adding prover-specific syntax highlighting, and (b) adding a means by which subsets of the text can be dynamically highlighted so as to show which parts of a proof script have already been evaluated and may not be modified.

The outline shows the detailed structure of a file being edited via a tree-based view. The basic outline view of ProverEditor is initially empty: it only contains the name of a file opened in an editor and which has the focus. The prover plugins augment this implementation by providing an outline of the proof script files. For example, in Coq, the outline view summarizes the type hierarchy found in the current file. The outline always represents the whole file, much like what is done for the Java outline in Eclipse. This view is especially useful when inspecting a library file, to jump easily to a specific definition.

Another plugin is the *proof view*, which shows the user the result of each interaction with the prover. It is essentially a log of the user interactions with the plugin and needs no input. It is integrated at the same fashion as the outline window in the Eclipse workspace. This interaction can only target a single file at a time. A file has the interaction focus only if the user decides to step through it.

The last extension from Eclipse that is used is the *PreferencePage* extension. It handles the preferences necessary for a particular *proof view*, like whether to expect output to use the Unicode character set.

Currently there are four plugins. The base plugin (ProverEditor) handles all generic (non-prover-specific) interaction. There are plugins to support the Coq and PVS higher-order provers. There is also a plugin that supports interaction with the top-level. It provides a high-level Java API to control a Coq top-level, and is called the CoqSugar plugin.

3.2 ProverEditor

ProverEditor is formed of four parts, as seen in Figure 1: the editor, the top-level view, the outline view, and the toolbar. ProverEditor also understands a number of keyboard shortcuts to trigger toolbar and menubar actions.

The actions associated with the buttons seen in Figure 2 are (from left to right):

- start to evaluate the file from the beginning
- take a step in the current file
- undo a step
- progress to the end of the file
- undo to the beginning of the file
- cancel an action

These actions are fairly standard in proof assistants like Coq, PVS, and Isabelle. Of course, they are also similar to the actions implemented in Proof General and CoqIDE. In fact, the general organization of the view for ProverEditor was inspired by the look-and-feel of those two tools. Still, there are some enhance-

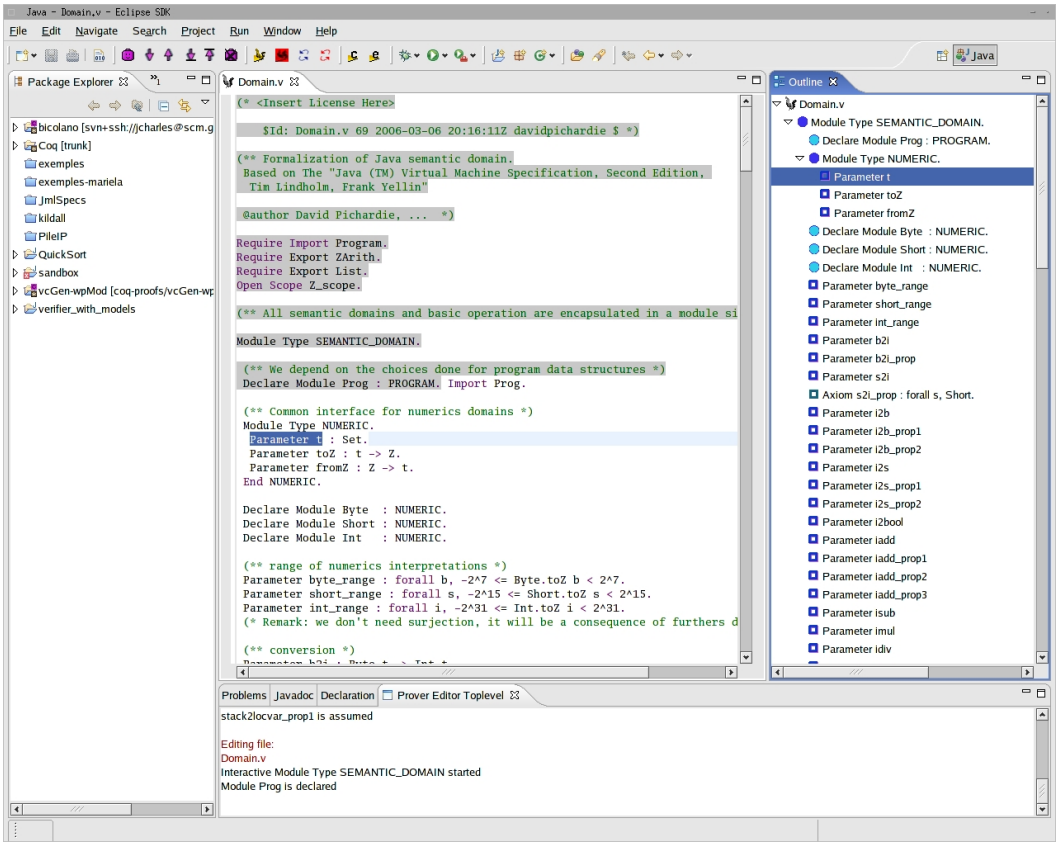

Fig. 1. A Typical ProverEditor Interface

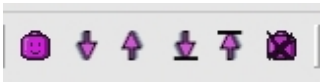

Fig. 2. The ProverEditor Toolbar

ments, as ProverEditor adds syntax highlighting to the top-level output and has a very lightweight interface that reflects its lightweight architecture.

More concretely, extending ProverEditor for new provers is simpler than extending Proof General. In order to provide the base extensions, syntax highlighting, and prover input/output communication for a new prover, less than a hundred lines of Java code is necessary. The base implementation provides library functions to communicate with provers as well, so adding support for further new provers should necessitate even less code.

3.2.1 Core features

First we will discuss the main features provided by the base ProverEditor plugin. This plugin provides some low-level handling of a target prover top-level as well as the basic UI building blocks, to edit a file from the prover, highlight its syntax, show an outline of the file, and step through a proof.

The interaction subsystem, found in the `prover.exec` package, manages in-

teractions like sending a stream to, and receiving data from, the top-level via pipes, typically its standard output and standard error. The main class in this subsystem is the `prover.exec.toplevel.TopLevel` class, which implements the interface `prover.exec.ITopLevel`.

Interaction with the top-level is based on calls to a method called `sendCommand`. In a typical `sendCommand` scenario, the method `sendCommand(String)` of the `TopLevel` class is called. The command passed is supposed to be atomic. The primitive `ITopLevel.sendToProver(String)` is called afterward with a prover specific code.

`sendCommand` is a gateway for other methods to undo commands. For example, the `undo()` method triggers an undo command to the top-level. This method calls a prover-specific command that undos one step from whichever context in which the user is working. In most of the cases, this command reduces to a call to `sendCommand()` with the correct parameter. For instance, in the case of Coq, the undo command is translated into sending one of three Coq top-level commands: `Back` command, the `Undo` command, or the `Abort` command and for this behaviour the top-level state is used.

These interactions are generic and simple, but they permit accurate communication with each prover. They rely on a simple parsing of the inspected file. These base feature can be used in more advanced interactions like the tagging system.

3.2.2 Tagging

A feature that differentiates ProverEditor from other similar systems is its support for tags and tagging. Tags are a standard way of indexing source code contained in libraries for interactive front-ends. One of the main implementations of tagging is found in the program `ctags`. Tagging is used in the vi editor as well as in Emacs. Here we chose to implement a tagging system compatible with the etags [2], which is used with Emacs, as nearly all higher-order theorem provers "native" interfaces are based upon Emacs.

To tag a file, one uses regular expressions to match identifiers with their definitions. Once identifiers and their definitions are all gathered, a system can search through this index at user request. The basic search method is triggered when the user select an identifier (with, say, the mouse pointer), and asks to open its definition, or something that is considered as its definition.

In ProverEditor, definition search is triggered by the standard Eclipse keystroke '`F3`'. This keystroke only works if the current Eclipse project is a ProverEditor project, like a Coq project for instance. This restriction is due to the fact that the tags are built incrementally, and in Eclipse this incremental construction feature is intimately linked to the project nature.

Tags are stored using the standard `etags` format [1]. We find it useful to have a compatible storage format because sometimes one works with Proof General Emacs and ProverEditor at the same time. While this is not the most common situation, it is a valuable feature to those who must use both interfaces, either simultaneously or alternatively.

The tagging system used in our tool is quite efficient for several reasons:

- Eclipse optimizes file search, so definition search is efficient and fast,
- definition search is incremental—each time a file is added, removed, or saved to the disk, its tags are calculated and saved, and
- user interaction identical to Eclipse's Java code browsing functionality. When the user presses '`F3`' over an identifier, the system opens a file containing the corresponding definition and highlights it. This interaction is very familiar to even the beginner Eclipse developer. Since our approach is lightweight and involves no context information, several definitions may be found for a single identifier. Thus, pressing '`F3`' again jumps to the next definition.

To our knowledge, no other tagging system has been implemented for Eclipse. A potential future development is spawning off an independent tagging plugin from this code. Such a plugin would be useful as a separate component for Proof General Toolkit Eclipse, as well as ProverEditor or any other Eclipse plugin.

3.2.3 Extending ProverEditor with New Provers

The key differentiating feature of ProverEditor is the lightweight way in which one integrates new provers. While the programmatic interface is simple, new provers must have certain key properties in order to seamlessly integrate them. In particular, the prover must have a top-level and they must have two modes of interaction: a "definition mode" and a "proof mode". This separation is useful for minimal interaction as presented in this paper, and especially for Coq. To have a richer interaction the way would be to create a plugin that could be extended implementing a specific protocol. This could be useful for better integration of some provers, like PVS.

PVS integration hasn't been an easy task, because interaction with the prover is more complex than in Coq. The main problem is that the two modes tends to interleave in different way than seen Coq or Isabelle. Still, due to ProverEditor's lightweight nature, the current prototype supporting PVS is only around a hundred lines of code as well [6].

The main aim is to keep extensions simple and easy. To add a new prover one must write an Eclipse plugin in the standard way and extend two extension points. Additionally, one must implement at least two classes: one to help handle top-level interactions, and the other, which is more prover-oriented, implements GUI related functionality.

The first extension point to extend is called `org.eclipse.ui.editors`. It is the extension point used to add a new file format handling. The editor to edit the file is provided by ProverEditor is the class `prover.gui.editor.ProverEditor`. This extension point has to be extended, because Eclipse permits to add specific editors only statically, through this extension point. We expect in the future

[6] This plugin is currently experimental, but its source code can be downloaded from the ProverEditor main site [14], as well as the MOBIUS Trac server [15]

developments of Eclipse we will be able to automatically add it like what is done currently for the *PreferencePage*.

The second extension point that must be realized is specific to ProverEditor. Its name is `prover.editor.prover`. This extension point connects the two aforementioned classes that must be implemented to support a prover to the generic plugin. Two classes have to be added, one that extends the abstract class `prover.plugins.AProverTranslator` and the other implementing the interface `prover.plugins.IProverTopLevel`. The latter providing some of the top-level functionalities.

By these two simple steps one can add a prover plugin to ProverEditor. Currently two plugins are using directly these functionalities, the CoqPlugin and the PvsPlugin.

4 Current Plugins

The main motivation to make the ProverEditor was to manage Coq interaction inside of Eclipse. That is why the finished plugins concern Coq. There is also a PVS plugin which is in developpement. ProverEditor plugins are used in 2 tools: JACK [5] (the Java Applet Correctness Kit), and MOBIUS' DirectVCGen [7]. Both tools being Eclipse plugins to do static program verification on Java programs annotated with JML.

Right now the plugins available are the following:

- the core plugin, containing all the basic features which has been described in Subsection 3.2.
- the Coq plugin: it handle the interactions with Coq, basically it send parts of a file to Coq, give an outline of the current edited file and do tagging. This plugin is used with the DirectVCGen and JACK.
- the Coq Sugar plugin, which is an API to interact with the Coq top-level. It is the plugin used in JACK

4.1 The Coq Plugin

This plugin is the genuine plugin for ProverEditor. It contains all the features that were mentionned previously:

This plugin is made to do interaction with Coq. The core features are inside 3 classes. The mandatory plugin class for Eclipse (Eclipse needs a plugin class in order to consider it has a plugin). This class is generated automatically by Eclipse. The two other classes are the one needed to add the extension to ProverEditor: the one used to handle the top-level and the other one containing mainly the highlighting parts.

There is another part which is less mandatory: the one to handle the outline. This adds about 10 classes to represents the leafs and nodes of the tree-view. There are 2 kinds of handling: the leaf kind, containing only constructs like

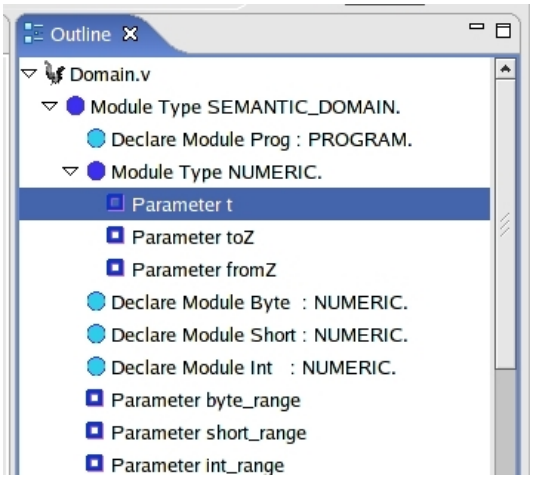

Fig. 3. The outline for Coq

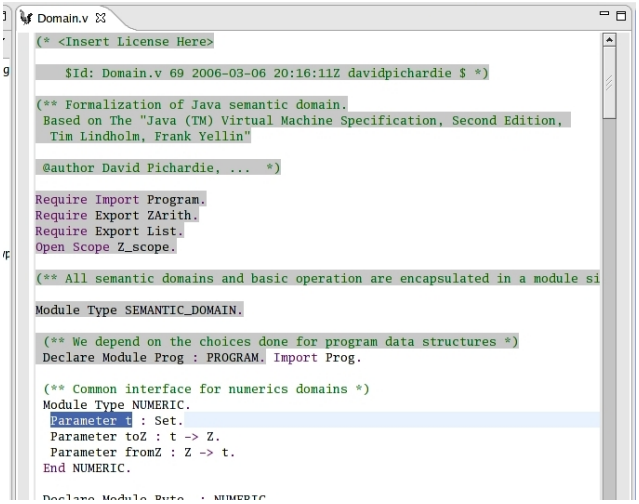

Fig. 4. The editor customized for Coq

`Definition`, `Parameter` and other single non-imbricated definitions. There is another kind of construct, the imbricated ones. In Coq these are the section and the module. These constructs are of a binary kind, with one to begin them (`Module m.` or `Section s.`) and one to end them (`End m.` or `End s.`).

4.2 The Coq Sugar plugin

We have made another plugin which add lots of some superfuous features to the basic plugin. The main purpose of this plugin being to ease the use of Coq within ProverEditor.

We have mainly added a more complete API to handle Coq. It has more high-level 'macros'. It subclasses the prover.exec.toplevel.TopLevel to have a real top-level targeted at Coq. It adds some facilities like methods to declare lemmas, do a

particular standard command or parse the output from Coq to give a parsed result of the command instead of the standard output. For instance, in Coq there is a command `Show Intros` which is used to know which variable name Coq would use with its intros command. Here the method gives an array of String with the different variable names. In Jack we mainly use this API to pretty print the proof obligations with Coq in order to have more user-readable proof obligations.

5 Conclusion

We have implemented a lightweight theorem prover within Eclipse, based on the extensions facilities provided by Eclipse. To have done it lightweight and minimal in its mandatory features has allowed us to extend it quite fast for the theorem provers like Coq or PVS.

What will follow is the extension of these basic features toward more prover specific features. One of the main idea would be to extend it in a component based approach. Instead of having as in the Proof General Toolkit one single mediator communicating through lighweight protocols to other plugins, we have a main plugin, which handle base interaction, that can delegate specific protocol handling to plugins and their extensions.

5.1 Next Steps

The next step will be to have real API plugins for provers. It has already begun through Coq's Sugar plugin which only aim is to provide this kind of API. It will be done later on for PVS.

We plan to include Isabelle/HOL in a near future. The inclusion should be simple as Isabelle can generate simple tagged output.

Other features that should be added as separate plugins are the projects and file wizard to creat new prover specific files or projects dedicated to a single prover.

Tagging has been included in for Coq. This approach should be completed with hints in a similar way as for Java. Like for tagging one of the difficulty is to keep it generic and simple enough. One of the pattern that could be used is to associate each identified tag with the nearest identified documentation. This is the pattern used in Java parser to keep the Javadoc in the bytecode [7] . Now we use the outline to get an idea of the file structures for Coq and PVS. We plan to do an enhanced outline that could give a representation of the proof tree. This extended outline could permit to manipulate the definitions as objects, which would be more akin of the Proof by Pointing [6].

We plan also to integrate it more thoroughly in the Mobius PVE [10]. Especially the look and feel which shall become more uniform with the other plugins part of the Mobius PVE.

[7] Although no real documentation is available, it can be seen in the sourcecode of the OpenJDK[12]

Acknowledgement

Benjamin Grégoire, Gilles Barthe and Yves Bertot. This article has been partially funded by The European Project MOBIUS within the frame of IST 6th Framework.

References

[1] *ETags*, http://www.gnu.org/software/emacs/emacs-lisp-intro/html_node/emacs.html#Tag-Syntax/.

[2] *Exuberant CTags*, http://ctags.sourceforge.net/.

[3] Amerkad, A., Y. Bertot, L. Rideau and L. Pottier, *Mathematics and proof presentation in Pcoq*, in: *Proceedings of Proof Transformation and Presentation and Proof Complexities (PTP'01)*, Sienna, Italy, 2001.

[4] Aspinall, D., *Proof general: A generic tool for proof development*, in: *Tools and Algorithms for the Construction and Analysis of Systems, Proc TACAS 2000*, number 1785 in LNCS, 200.

[5] Barthe, G., L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova and A. Requet, *JACK: a tool for validation of security and behaviour of Java applications*, in: *Formal Methods for Components and Objects*, Lecture Notes in Computer Science **4709** (2007), pp. 152–174.

[6] Bertot, Y., G. Kahn and L. Théry, *Proof by Pointing*, in: M. Hagiya and J. C. Mitchell, editors, *Proceedings of the International Symposium on Theoretical Aspects of Computer Softward* (1994), pp. 141–160, http://citeseer.ist.psu.edu/bertot94proof.html.

[7] MOBIUS Consortium, *Deliverable 4.3: Intermediate report on proof-transforming compiler* (2007), available online from http://mobius.inria.fr.

[8] David Aspinall, C. L. and D. Winterstein, *A Framework for Interactive Proof*, in: *Towards Mechanized Mathematical Assistants* (2007), pp. 161–175.

[9] Kaliszyk, C., F. Wiedijk, M. Hendriks and F. van Raamsdonk, *Teaching logic using a state-of-the-art proof assistant*, in: H. Geuvers and P. Courtieu, editors, *International Workshop on Proof Assistants and Types in Education (PATE'07)*, 2007.

[10] Kiniry, J., *Formalizing the user's context to support user interfaces for integrated mathematical environments*, Electronic Notes in Theoretical Computer Science **103** (2004).

[11] Kiniry, J. and S. Owre, *Improving the PVS user interface*, in: *User Interfaces for Theorem Proving*, 2003.

[12] Sun Microsystems Inc., *OpenJDK* (2007-2008), http://openjdk.java.net/.

[13] The Eclipse Consortium, *Rich Client Platform*, http://wiki.eclipse.org/index.php/Rich_Client_Platform.

[14] The MOBIUS Project, *ProverEditor* (2007-2008), http://provereditor.gforge.inria.fr.

[15] The MOBIUS Project, *The Mobius Trac* (2007-2008), http://mobius.ucd.ie.