Paral-ITP Front-End Technologies second interim report

Makarius Wenzel
Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
CNRS, Orsay, F-91405, France

with contributions by Carst Tankink

August 20, 2014

Abstract

This is an overview of front-end technologies for pervasive parallel ITP systems, during the main phase of the project Paral-ITP (ANR-11-INSE-001). It corresponds roughly to milestone M2.

Contents

1 Introduction

The PIDE (Prover IDE) front-end technology integrates parallel proof checking with asynchronous user interaction, based on a document-oriented approach of continuous proof processing. This enables the user to edit whole libraries of formalized mathematics directly in the editor, with real-time visualization of feedback produced by the prover. Today, Isabelle/jEdit is the default user-interface for Isabelle, but this has required many years of developing the PIDE concepts and getting the underlying Isabelle/Scala infrastructure into a robust and scalable state.

An important objective of the Paral-ITP project is to let Coq participate in this upgrade of ITP user-interfaces as far as possible, catching up approximately 5 years of reforms on prover architecture. This will eventually allow to re-use most of the existing front-end technology for that particular back-end.

The Isabelle/jEdit implementation of PIDE [10, 4] is a typical example for a *rich-client application* that is run on the local machine, with non-trivial resource requirements: 2–4 CPU cores and 2–4 GB memory minimum. An interesting alternative is the collaborative *client-server application* Clide [2], which combines Isabelle/Scala/PIDE with a recent Web framework on the JVM, and supports collaborative interactive theorem proving in particular.

Here is a brief historical overview of PIDE so far:

- In 2005 all major CPU manufacturers started to ship multicore systems for the consumer market. Ever since the burden of explicit parallelism has been imposed on application developers, in order to keep up with the changed side-conditions of *Moore's Law*, and participate in continued performance improvements of computing hardware.
- In 2006–2008 Isabelle and its underlying Poly/ML compiler and runtime system have managed to follow the multicore trend. Isabelle2008 (June 2008) was the first official release to support parallel proof processing in *batch mode* by default. At the same time it became apparent that user-interfaces for parallel proof assistants require significant reworking of the interaction model.
- In 2010 the new Isabelle/PIDE concepts, with the underlying Isabelle/Scala infrastructure, and Isabelle/jEdit as experimental application, were presented in public at UITP 2010. That was 2 years after the first concrete ideas for it had emerged, but still more than 1 year to go before the first "stable release" of Isabelle/jEdit with Isabelle2011-1 (October 2011).

This also defines the starting point for the Paral-ITP project in November 2011.

- In 2012 that initial release of Isabelle/jEdit was presented as system description and tool demonstration at CICM 2012 [6] and some of its concepts were explained at the co-located UITP 2012 [8]. Moreover there were some early experiments to connect Coq as alternative PIDE backend [7].

- The subsequent releases of Isabelle/jEdit in May 2012, February 2013, and November/December 2013 have consolidated the PIDE concepts and its implementation. So many new things were introduced each time, that users have occasionally complained about having to learn a new Prover IDE with each release.

It is important to understand that parallelism is a feature of last resort in hardware design: it makes it more difficult for application software to use the available CPU cycles. On the other hand, an application that works with parallel execution routinely opens new possibilities in interaction design.

2 Activities and achievements

2.1 Official system releases

According to the original plan of the project, roughly one official release per year of Isabelle and Coq were anticipated, with increasing support for parallel prover architecture and PIDE front-end technology. Release cycles of Isabelle are much shorter than for Coq, though.

Following the schedule, a second Isabelle release in 2013 was staged, but a technical incident in the PIDE implementation of Isabelle2013-1 (November 2013) forced a relaunch the same with minor modification as Isabelle2013-2 (December 2013). The final Isabelle release for the sake of the Paral-ITP project is Isabelle2014 (August 2014).

Coq version 8.4 was released in August 2012, with some minor cleanup and reforms for external system connectivity, notably for CoqIDE, using an XML-based protocol instead of former stdin/stdout text channels.

Coq version 8.5 was originally planned for the start of 2014, but got delayed due to its immense amount of contributions. When released at the end of 2014 or the start of 2015, it will also contain important add-ons for parallel and asynchronous interaction, as sketched in [3].

2.2 Practical feedback via tutorials

Isabelle/jEdit is already the default user-interface of Isabelle. We have continued to explore its usability by occasional hands-on tutorials, especially for new users or users of other proof assistants. As before, this was done informally and without systematic evaluation, since the project is not primarily about Human-Computer-Interaction.

23-May-2014 Isabelle tutorial at ENS Paris.

Participants: mostly undergraduate students from ENS Paris, two professors from ENS, three guest researchers from INRIA Rocquencourt (the latter had some prior experience with Coq and/or Agda).

13-Jul-2014 Isabelle tutorial at the Vienna Summer of Logic (VSL 2014) multi-conference.

Participants: more than 15 attendants, approximately half of them without prior knowledge of Isabelle nor any other proof assistant; various colleagues from the ATP community, without ITP experience.

As already experienced before, proficient users of Coq and/or Agda have more problems picking up Isabelle/PIDE interaction than fresh users.

The tutorial at Vienna was notable due to a minor technical incident with the official website: the workshop pages remained read-only for several weeks before the event, so our participants were missing the latest URL for the required Isabelle2014-RC0 snapshot. Consequently, the majority of attendants had to make a quick installation of the system on their own machines on the spot, which worked smoothly within a few minutes.

In such tutorials we now present the prover, the IDE front-end, and its asynchronous and parallel add-on tools as one big system that incorporates interactive and automated theorem proving tools. End-users actually cannot tell the difference easily: a tool like Sledgehammer might refer to local or remote ATPs (E prover, SPASS, Vampire, Z3), but it now participates in the PIDE document-model as *asynchronous print function* as explained below.

3 Conceptual advances of the Prover IDE

3.1 Asynchronous print functions

The concept of *asynchronous print functions* in the PIDE document-model combines *user interaction* and *tool integration* as explained in [9], which was presented ITP 2014 in July 2014. The general approach is to continue the reforms of READ-EVAL-PRINT [8] as follows [9, §5]:

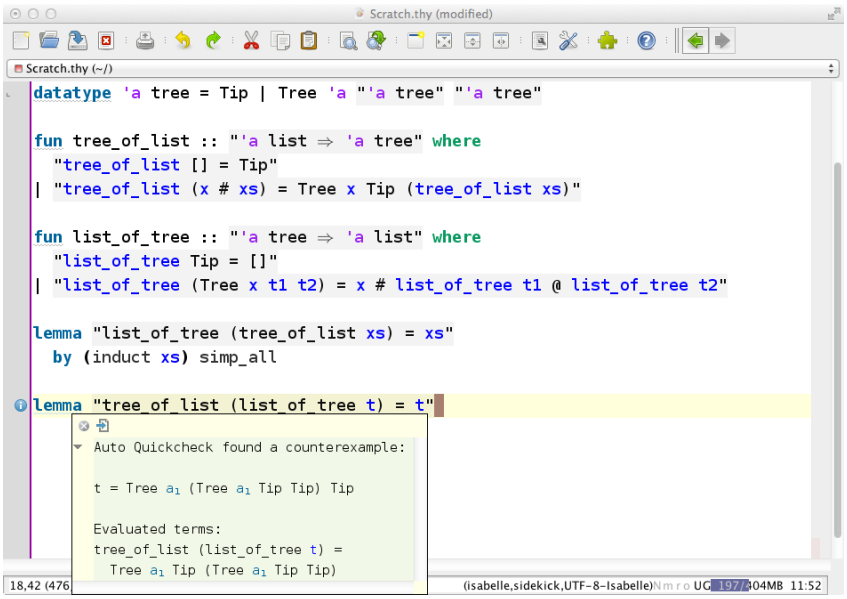

Figure 1: Results of automatically tried tools

- Edits may add or remove PRINT operations, without disturbing the corresponding EVAL tasks. This principle of *monotonicity* avoids interruption of tasks that are still active in the document model, and allows to use long-running or potentially non-terminating tools as print functions. Typically these are automated provers (via Sledgehammer) or disprovers (Quickcheck, Nitpick).

- Activation or deactivation of PRINT tasks is subject to the *document perspective*. The whole theory library that is edited might be big, but only small parts are visible in the editor. PIDE document processing takes the open text windows as indication where to invest resources for continuous processing. Various declarative parameters control print functions that are implemented in user-space of Isabelle/ML: startup delay, time limit, task priority, persistence of results within the document model.

- Support for explicit *document overlays*, which are print functions with arguments provided by some GUI components. This recovers the appearance of direct access to command execution in the prover, despite the thick layers of asynchronous PIDE protocol between the stateless/timeless prover and the physical editor.

Figure 1 shows various *automatically tried tools* that operate on outermost goal statements (e.g. **lemma**, **theorem**), independently of the state of the

current proof attempt. They work implicitly without arguments, but there are global options in *Plugin Options / Isabelle / General / Automatically tried tools*. Results are output as *information messages*, which are indicated in the text area by blue squiggles and a blue information sign in the gutter of the text window. The message content may be shown as for other prover output in a separate window. Some tools produce output with *sendback* markup, which means that clicking on certain parts of the text, inserts that into the source in the proper place.

Figure 2 shows the *Sledgehammer* panel, which provides a view on some independent execution of the Isar command **sledgehammer**, with progress indicator (spinning wheel) and GUI elements for important Sledgehammer arguments and options. Any number of Sledgehammer panels may be active, according to the standard policies of jEdit window management. Closing such a dockable window also cancels the corresponding prover tasks.

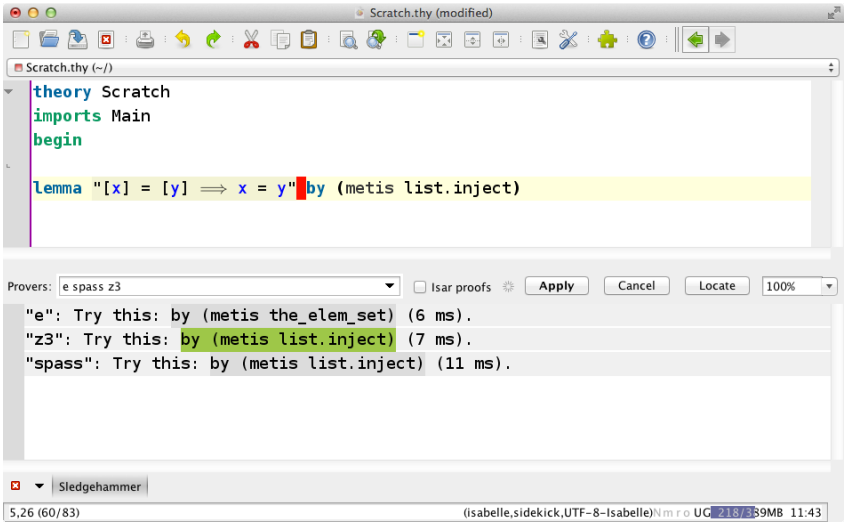

Figure 2: An instance of the Sledgehammer panel

Technically, the Sledgehammer panel is a conventional GUI component on the surface, but it is connected to the PIDE document model by producing some document overlay when the user pushes the *Apply* button. This leads to some document edit that attaches a suitable asynchronous print function (with arguments taken from the GUI panel), and eventually forks some print task on the prover side. Any output from that task is incrementally shown in the GUI panel. The *Cancel* button uses the execution id of the running print operation to interrupt it on demand.

The overall interaction of the PIDE front-end with the prover back-end does not prevent the user from editing the text nor the prover from checking

proofs in parallel. The only impact is some loss of performance to other tools in the background, but this can be balanced via global system options to adjust to the available number of cores.

3.2 Syntactic and semantic completion

Semantic completion, based on authentic information from the proof context, has been a "nice to have" features over several years. It was not immediately obvious to teach that trick to a traditional LCF-style proof assistant like Isabelle, which was not made for that 25 years ago.

Even just the editor GUI part of auto completion has turned out much less trivial than anticipated in 2009/2010 [5], where the (naive) idea was to connect to an existing completion plugin of jEdit. Over the last 5 years the completion mechanism in Isabelle/jEdit has changed several times, but problems with the timing of GUI events could still happen in Isabelle2013-2 (December 2013).

Completion intercepts the regular key event handling of the main text area, and needs to work smoothly as the user is typing slowly or quickly. The completion popup changes the keyboard focus to a different component, which can lead to odd effects of loosing key events in a situation where the user is typing fast, but the graphics display is too slow to catch-up (e.g. due to bad X11 rendering performance, which can happen both for local and remote displays).

Both the GUI event handling and the semantic aspects of completion in Isabelle/jEdit have been significantly reworked in 2014. The following general principles are presently applied:

- **Syntactic completion** is based on information that is *immediately* available in the editor, e.g. keyword tables for certain sub-languages of Isabelle, like the so-called "outer syntax" of Isabelle/Isar, or Isabelle/ML. Completion for Isabelle symbols is an important special case of this: when the user types "`==>`" he normally expects to get "$\Longrightarrow$" within formal text.

- **Semantic completion** is produced by the prover *eventually*, after a full round-trip through the asynchronous PIDE protocol. This information usually arrives with a delay of 100–500 ms and is then merged with the available syntactic completion, before it is used for GUI rendering (e.g. for emphasis of text or a popup).

- **Completion markup** may be produced by the prover in any of the following forms:

- **Language context** guides the syntactic completion. Isabelle is a framework of many sub-languages, which have different requirements for completion. The language context for some text range informs the editor about the language name (e.g. to use a different keyword table), and some common flags like use of Isabelle symbols and antiquotations.

  For example, the term language in Isabelle supports symbols, but no antiquotations. In contrast, the document language (a semi-formal version of LaTeX) supports antiquotations, but no symbols. An antiquotation that puts a term inside some document source needs to switch the language context accordingly, and several such changes of language context can happen in a small piece of theory source.

- **Completion items** result from failed name space lookups of formal entities (type names, term constants, fact names etc). Luckily the prover already has a uniform concept of name spaces, in order to intern names given by the user to the actual formal entities from the context. The error situation has been slightly modified to include a list of alternative names into the error message, as PIDE markup that is not immediately visible, but available to the completion mechanism.

  For performance reasons, it is important to produce completion items only for failed name-space lookups, which are relatively rare, and not for the majority of successful ones. There is nonetheless a simple way for the user to request more information: adding a suffix of underscores to a partial name provokes an error with extra completion information. A double underscore on its own serves as wildcard to query the whole name space, but output is always truncated to a reasonable limit for display. Explicit completion requests via underscores are particularly important for the term language, because undeclared constants alone are accepted as free variables, without any error nor completion information.

- **No-completion zones** enable the prover to *negate* already discovered syntactic completions of the editor. Such non-monotonic change of the meaning of incremental document content is always critical, and can lead to erratic behaviour. Here it should be seen as a feature of last resort.

Spell-checking is another application of the same PIDE infrastructure, which is somewhere in between syntactic and semantic completion. Based on prover markup for the language context, e.g. to determine ranges of prose text inside document sources or comments, the editor uses a conventional dictionary-based spell-checker to propose alternatives to words spotted in

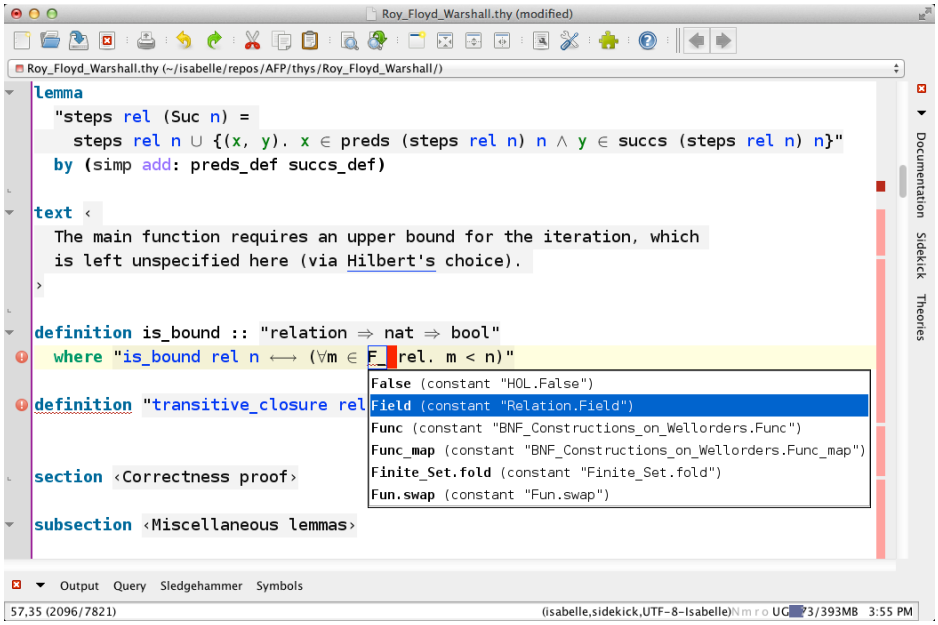

Figure 3: Spell-checking within informal text and semantic completion within terms.

the text.

Figure 3 illustrates spell-checking within informal text: the default dictionary does not know about `Hilbert's`, but this is not an error, merely highlighting. Moreover there is semantic completion within the term language, using an extra underscore to let the prover expose constants from the theory context.

In a sense, our *continuous document processing* for parallel ITP systems is like a very sophisticated "spell-checker" for formal proofs. Here we close the circle and incorporate traditional spell-checking into the grand-unified interaction model.

3.3 Auxiliary files within the PIDE document-model

Ultimately, the main job of an IDE is to manage a collection of sources and the results of processing them seamlessly, taking implicit and explicit structural dependencies into account. So far the PIDE document model was based on two levels in the structural hierarchy: an acyclic graph of *document nodes* (theories), where each node consists of a list of *command spans* (like in Proof General [1]).

Apart from that, it was always possible to refer to auxiliary files as a semi-

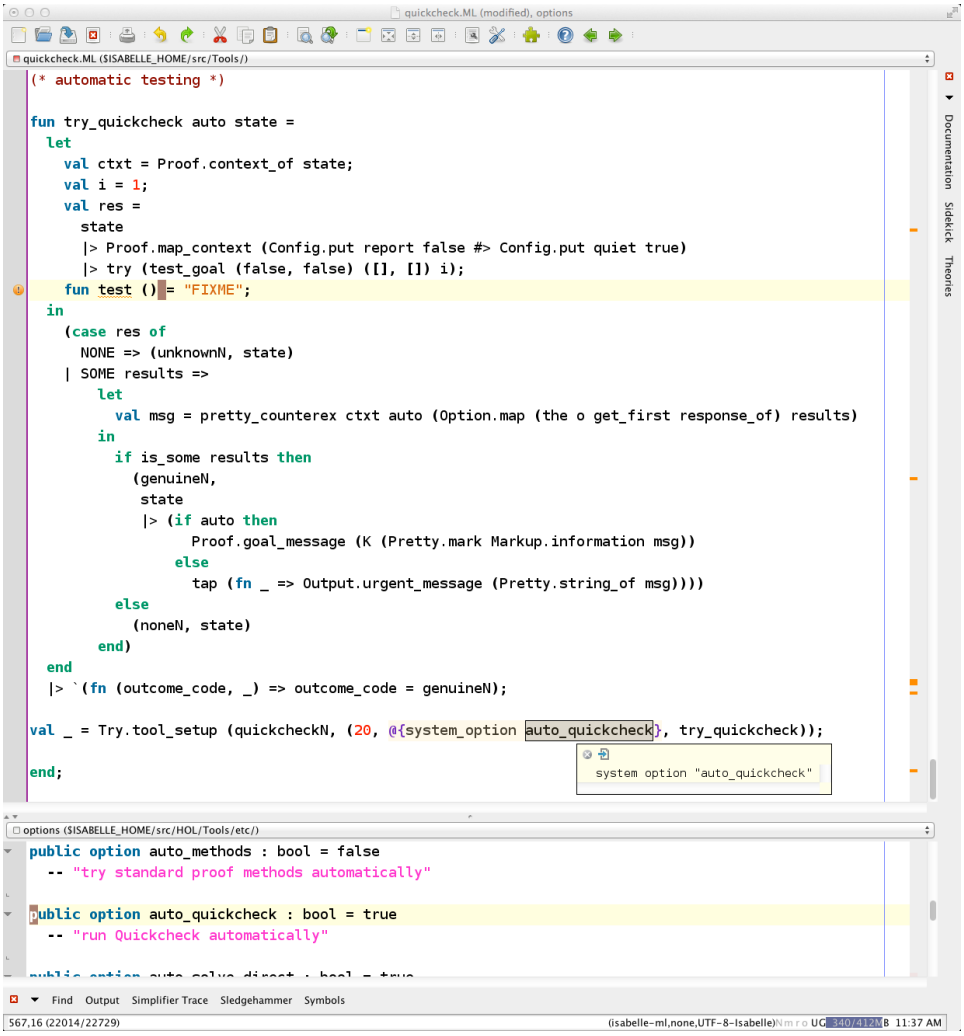

Figure 4: Live editing and browsing of Isabelle/HOL ML files

official feature, but with limited management in the IDE. That unsatisfactory situation has ended, and there is now first-class support for auxiliary files that appear as arguments to special *load commands* inside document nodes. Thus the source text is conceptually extended by so-called *text chunks* that are stored elsewhere, and may be edited / loaded / saved independently of the theory itself. The Prover IDE takes care to forward the correct version of auxiliary file content to the prover as a *blob*, but without using the global file-system.

This extra file management is particularly relevant for development of Isabelle/HOL itself within the Prover IDE. According to usual practice of

LCF-style proof assistants, the main logical environment emerges by alternating theory specifications with ML modules. It is now possible to use Isabelle/jEdit to explore the inner workings of Isabelle/HOL modules and their dependencies on theory content, with additional navigation support like regular Web browsers.

The user may edit Isabelle/ML source files, without ever saving the content, while the Poly/ML compiler provides continuous feedback on warnings, errors, name references, inferred types etc. as part of the PIDE document model (see figure 4). This works reasonably well for source files up to 100 KB each. The total volume of ML sources contributing to Isabelle/HOL is actually so high that its cumulative PIDE markup requires more than 2 GB Java heap space. This performance bottle-neck is addressed by some special tricks with asynchronous print functions (§3.1) which were introduced in 2013 for quite different applications. Here the mechanism is re-used as follows: Poly/ML compiler markup is stored in compact form within the ML process, and only reported to the editor when the corresponding ML file becomes part of the visible perspective. The document markup is removed from the editor process when visibility gets lost, and thus becomes subject to garbage collection on the JVM.

As a corollary to this scalable approach to continuous editing and compilation of Isabelle/ML files, there is also support for official Standard ML via the **SML_file** command. Thus Isabelle/jEdit can be used as IDE for SML'97, without any connection to theory or proof development. The two ML environments are managed independently within the same runtime system, but there are simple means to exchange toplevel ML bindings, e.g. to re-use the parallel functional programming library of Isabelle/ML in Standard ML, or to print messages in Standard ML that are recognized by the Prover IDE for its *Output* panel. The Prover IDE provides some simple examples for that in its *Documentation* panel.

4 More realistic integration with Coq

Carst Tankink, who joined the Paral-ITP project in March 2014, has continued earlier experiments on Coq-PIDE integration [7] to provide a more realistic prototype within a few months [3]. A significant part of the work was a reform of the programming interface for the Coq toplevel, to allow alternative plugins to replace the traditional command loop, see also figure 5.

Using the `PIDEtop` module inside Coq, the prover understands a version of the PIDE document update and feedback protocol directly, similar to Isabelle. Further details of processing are up to the prover: the State Transaction Machine of forthcoming Coq 8.5 accommodates the programming

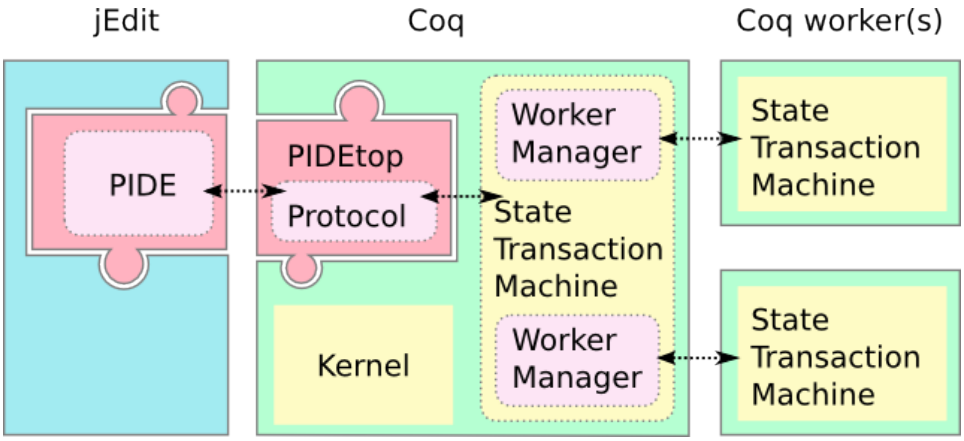

Figure 5: The overall architecture to integrate Coq into PIDE

model of separate processes with distributed mutable state, which is quite different from Isabelle with its emphasis of shared-memory multiprocessing and immutable data.

The actual connection to the PIDE front-end turned out quite simple, with minimal modifications of the existing Scala modules, and a few add-ons for prover syntax and process management specifically for Coq. The integrated application is shown in figure 6 and is already usable for small theory modules. It was presented at VSL 2014 at the UITP and Coq workshops.

Note that the lack of "eye candy" in the visualization of the PIDE document is not a problem of the front-end, but the back-end: the prover needs to provide more markup information, e.g. about the syntax of the formal language. It is an inherent advantage of the PIDE architecture that the prover can improve the coverage of the sources via markup *peu à peu*, without requiring significant changes on the front-end side.

References

[1] D. Aspinall. Proof General: A generic tool for proof development. In S. Graf and M. Schwartzbach, editors, *European Joint Conferences on Theory and Practice of Software (ETAPS)*, volume 1785, 2000.

[2] M. Ring and C. Lüth. Collaborative interactive theorem proving with clide. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving – 5th International Conference, ITP 2014, Vienna, Austria*, volume 8558 of *Lecture Notes in Computer Science*. Springer, 2014.

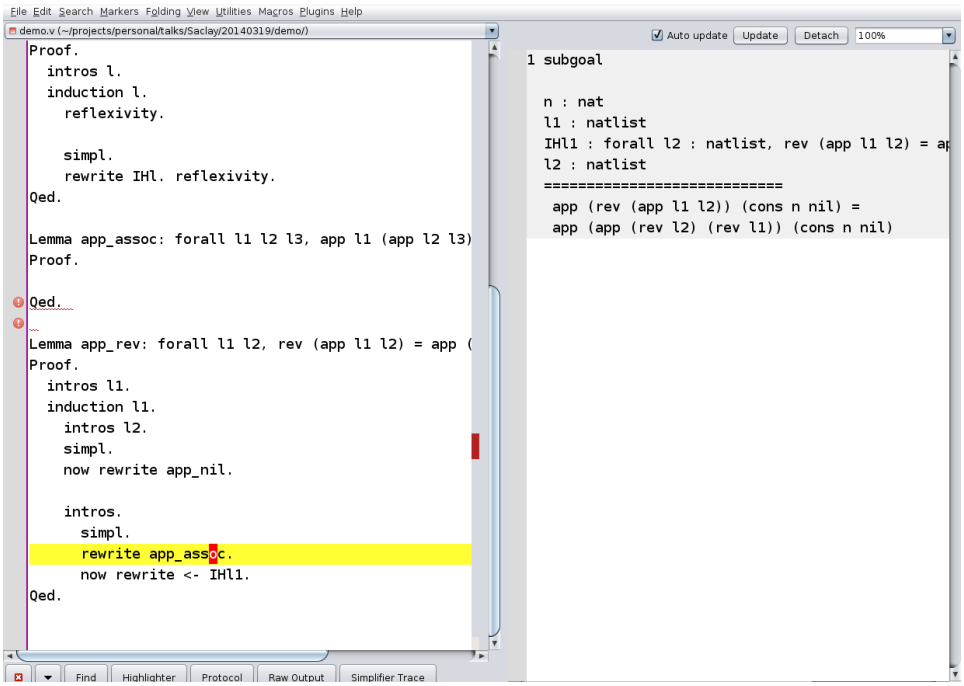

Figure 6: The jEdit PIDE plugin working with Coq

[3] C. Tankink. PIDE for asynchronous interaction with Coq. In *Presented at User Interfaces for Theorem Provers (UITP Workshop)*, July 2014. http://vsl2014.at/uitp.

[4] M. Wenzel. *Isabelle/jEdit.* Part of the Isabelle distribution http://isabelle.in.tum.de/doc/jedit.pdf.

[5] M. Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In C. S. Coen and D. Aspinall, editors, *User Interfaces for Theorem Provers (UITP 2010), FLOC 2010 Satellite Workshop*, ENTCS. Elsevier, July 2010.

[6] M. Wenzel. Isabelle/jEdit — a Prover IDE within the PIDE framework. In J. Jeuring et al., editors, *Conference on Intelligent Computer Mathematics (CICM 2012)*, volume 7362, 2012.

[7] M. Wenzel. PIDE as front-end technology for Coq, February 2013. http://arxiv.org/abs/1304.6626.

[8] M. Wenzel. READ-EVAL-PRINT in parallel and asynchronous proof-checking. In C. Kaliszyk and C. Lüth, editors, *User Interfaces for Theorem Provers (UITP 2012)*, volume 118, 2013.

[9] M. Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Vienna, Austria*, volume 8558 of *Lecture Notes in Computer Science*. Springer, 2014.

[10] M. Wenzel. System description: Isabelle/jEdit in 2014. In *Presented at User Interfaces for Theorem Provers (UITP Workshop)*, July 2014. http://vsl2014.at/uitp.