Proof General in Eclipse

System and Architecture Overview

David Aspinall
Daniel Winterstein
School of Informatics
University of Edinburgh
U.K.
da@inf.ed.ac.uk

Christoph Lüth
Ahsan Fayyaz
Department of Mathematics and Computer Science
Universität Bremen
Germany
cxl@informatik.uni-bremen.de

ABSTRACT

Interactive theorem proving is the art of constructing electronic proofs. Proof development, based around a proof script, has much in common with program development, based around a program text. Proof developers use rather primitive tools for developing and manipulating proof scripts at present. The Proof General project aims at to change this, by providing powerful generic tools and interfaces. The flagship tool is our Eclipse plugin, which brings the features of a industrial-strength IDE to theorem proving for the first time. In this paper we give an overview of the Eclipse plugin and its underlying architecture.

1. BACKGROUND

An **electronic proof** is the representation of a mathematical proof on machine, usually in a fully formalised way. This means that there is a formal syntax for the *logical language*, which makes statements of properties to be proven, and for the *proof language*, which describes how those properties are proved following a set of rules. Electronic proofs are desirable for at least three reasons, when (i) the properties are too complex for human readability, (ii) a high degree of confidence is desired in a proof, or (iii) the proof itself is too detailed to be managed manually. These reasons motivate the typical application areas:

- software and hardware verification, dealing with many thousands of correctness conditions [8];
- domain specific logics and meta-theory [4];
- deep proofs of mathematical results [6].

With maturing technology, theorem proving in-the-large is becoming more feasible, with significant sized formalisations being attempted in both industry and academia. Recent examples of large efforts include verifying parts of the Pentium micro-architecture [8], formalising type safety and a programming logic for Java [11], a detailed proof of the Four Colour Theorem [6], and the ongoing attempt to construct a formal version of Hales's proof of the Kepler Conjecture [7].

These proofs range in length from around 10,000 lines to 100,000 lines, each proving hundreds or thousands of lemmas and representing some person-years of work. It is encouraging that developments of this size are now possible, but they are far from easy. Formal proof texts are arguably more complex, dense, and interdependent than similarly sized programs. Yet they are being developed with primitive tools, often little more than basic text editors, with no high-level means of rapid construction, easy modification or browsing. Lack of support for in-the-large theorem proving is one reason that theorem proving tools are not used routinely for applied verification and mathematical assistance.

We are interested in *proof engineering*, an emerging field concerned with the construction, maintenance and understanding of large formal proof developments.

The **Proof General** project is building tools for proof engineering which work in a generic setting. Just as there are different programming languages for different applications, so there are many different proof languages and supporting systems presently in use. However, the theorem proving communities for each system are much smaller than they are for popular programming languages, so there are limited resources available for building development environments. It is also a reason that makes implementing generic tools, as far as possible, particularly attractive.

The setting of our work is on *interactive theorem proving* with a class of systems that follow a traditional of goal-directed interaction; this includes popular systems such as Isabelle, HOL, Coq, PhoX, ACL2, Twelf, Agda and PVS. The user writes a proof script in the proof language, which contains proof commands to be checked step-by-step by the theorem prover system. As each step, the system reports on the progress and outstanding goals remaining.

We want to bring modern software engineering tools to bear on this related area, and our main vehicle for doing this is an IDE for proof constructed within Eclipse.

2. PROOF DEVELOPMENT IN ECLIPSE

The central artefacts of proof development are *proof scripts* which are files containing declarations of types and constants, as well as proof goals and proofs themselves. Interactive provers check proof scripts to guarantee their correctness, but rely on users to manually write the input. A simple example proof for Isabelle/Isar [13] appears below:

```
lemma fn1: "(∃ x. P (f x)) ⟶ (∃ y. P y)"
proof
  assume "∃ x. P (f x)"
  thus "∃ y. P y"
  proof
    fix a
    assume "P (f a)"
    show ?thesis ..
  qed
qed
```

This proof language has a *declarative* style, intending to be a readable (if verbose) format. The proof of the simple implication is as follows: suppose that $\exists x.P(fx)$ holds. Then, for some unknown parameter a, we have $P(fa)$. But now we have exhibited a y such that $\exists y.Py$. The logical rules which connect this reasoning are hidden within the language.

Other proof languages have instead a *procedural* style which consist of a series of instructions which describe how to find a proof by explicitly naming logical rules and search procedures (known as *tactics*). This style of proof script describes how the proof is found, but can be difficult to understand later, especially without interacting with the system.

Proof General works with both kinds of proof languages provided they are designed to be *incrementally checked*, so that as we write the proof the theorem prover can check each line. The central idea of the Proof General interface is *script management* which synchronises the state of the theorem prover with the text editor by colouring the background of processed text. At first sight, script management is reminiscent of running a symbolic debugger step-by-step, except that this is the normal interaction mode and we always have the possibility to go back and forth in the history.

The screenshot in Fig. 1 shows this in action: the shaded region is the text that has been processed so far by the system. This region is *locked* (read only) to prevent inconsistencies. The backward and forward buttons allow navigation of the proof script by processing or undoing steps; to change a previous step one has to undo to unlock it first.

As each step is sent to the theorem prover, output is displayed from the system in the output view. The outline displays the structure of the proof and is also annotated to indicate progress. Another view (the PG Teacher View) is provided to display wizard-style tutorials, and finally there is a view on complete log of the interaction with the system. The latter is seldom of interest in normal use.

The interface provides light-weight syntax highlighting (for keywords, strings, etc.) implemented by the standard Eclipse mechanisms, and dynamic parsing of the proof script by the prover. This two-level handling allows both on-the-fly highlighting and structural parsing (see below), as required.

3. THE PGIP ARCHITECTURE

The Eclipse plugin is part of a larger architecture which allows connection to other front ends and offloads complex symbolic manipulation. The architecture connects components together using a custom protocol for message exchange called PGIP (the *Proof General Interaction Protocol*) [1].

PGIP defines:

- A *protocol* for conducting proof with a goal directed prover.
- A generic *system architecture* to connect provers to interfaces.

PGIP gives us a software framework for interactive proof. The reference implementation of this is called **PG Kit**, based around a central **Broker** component; see Fig. 2.

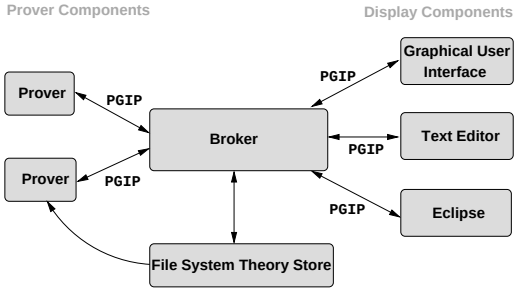

Figure 2: The PGIP system architecture.

Our aim is for the Broker to capture the *extra-logical* functionality of interactive theorem provers and support the core complex proof engineering operations. Although it would be possible to integrate the Broker into Eclipse, at the moment it is implemented as a separate component written in Haskell. This also allows us to connect additional display components, for example, GUIs which may display graphical representations of proofs, or web browser interfaces.

The components communicate using messages in the PGIP. The general control flow is that a user's action causes a command to be sent from the display to the broker, the broker sends commands to the prover, which sends responses back to the broker which relays them to the displays. The format of the messages is defined by an XML schema written in RELAX NG. Messages are sent over channels, typically sockets or Unix pipes. There is a secondary schema called **PGML**, for *Proof General Markup Language*, which is used for annotating concrete syntax within messages (for example, to generate clickable regions) and for representing mathematical symbols.[1]

There are two main sub-protocols within PGIP:

- The *prover protocol* connecting to theorem provers is designed to be simple, small and powerful, to make it easy to implement for existing provers.

[1]Another possibility is MathML [12], but PGML is designed to be easier to support for existing systems.

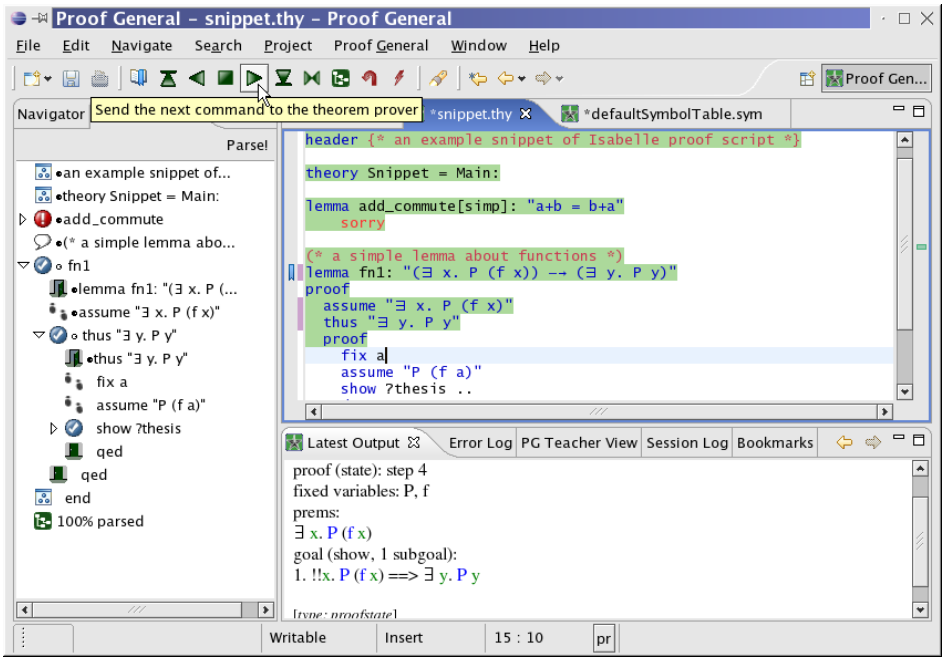

Figure 1: Editing a proof script using Proof General in Eclipse

- The *display protocol* connecting to interfaces is designed to be nearly stateless, very verbose, to make it easy to add displays.

Fig. 3 shows a schematic message exchange which illustrates how the display (here Eclipse) triggers commands to be sent to the prover, and how the Broker relays messages from the display to the prover.

The pattern of exchanges between the components is more permissive than in simple synchronous RPC mechanisms like XML RPC or most web services; this is necessary because interactive provers may send a lot of information while a proof proceeds. Since a proof may diverge (e.g. during proof search), it is essential that this feedback is displayed eagerly so the user can take action as soon as possible. The message exchange between Eclipse and the Broker is always asynchronous (single request, non-waiting multiple response): the display sends a command, and the Broker may send several responses later. The message exchange between the Broker and the prover can be asynchronous or synchronous (single request, waiting single response). The `<ready>` message indicates that the prover is ready for new input.

3.1 Proof script markup

The basic principle for representing proof scripts in PGIP is to use the prover's native language and *mark up* the content with PGIP commands which give the proof script the structure needed for the interface. The markup partitions a file into non overlapping *commands* which can be processed incrementally by the prover.

For example, the PGIP markup on the Isabelle/Isar proof

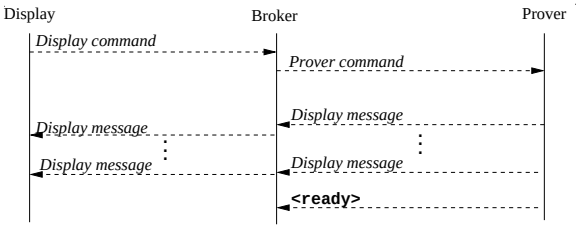

Figure 3: Message exchange in the PGIP protocol.

script shown earlier looks like this:

```
<opengoal name="fn1">lemma fn1: &amp;quot;(EX x. P (f x))
  <sym name="longrightarrow">--&amp;gt;</sym>
                (EX y. P y)&amp;quot;</opengoal>
<openblock/><proofstep>proof</proofstep>
  <proofstep>assume &amp;quot;EX x. P (f x)&amp;quot;</proofstep>
  <opengoal>thus &amp;quot;EX y. P y&amp;quot;</opengoal>
  <openblock/><proofstep>proof</proofstep>
    <proofstep>fix a</proofstep>
    <proofstep>assume &amp;quot;P (f a)&amp;quot;</proofstep>
    <opengoal>show ?thesis</opengoal><openblock/>
      <closegoal>..</closegoal><closeblock/>
  <closegoal>qed</closegoal><closeblock/>
<closegoal>qed</closegoal><closeblock/>
```

This shows the XML markup imposed on the original text. The `<sym>` symbol element is part of PGML (we omit the symbol markup on ∃ for brevity and write it as EX). The named and unnamed `<opengoal>` elements indicate the beginning of a proof or sub-proof, and the indentation structure of the script is reflected by the `<openblock>` and `<closeblock>`.

The theorem prover must provide the markup on arbitrary text for us using its own parser; this allows the interface to be generic without needing to hard-wire syntax and parsers for each different system. It even allows for dynamically extensible command syntax within the same system — a feature which is quite common in proof languages.

Apart from the basic structure, additional meta-information can be added to the markup which indicates the names of definitions made in each command, or the required definitions (dependencies) which are referenced. Additional markup can be added either during parsing or later on, during actual execution (proof checking).

The Broker manages the file contents and communicates it to Eclipse by sending individual text segments and their PGIP markup. Eclipse reassembles this in the text editor view, using the markup to maintain the outline view. User edit actions send commands to the Broker to signal the deletion or insertion of text fragments, which are relayed to the prover for parsing. We describe how this happens next.

3.2 Display protocol

The PGIP display protocol [2] gives an *edit-parse-prove cycle* for commands, which implements script management. Commands appear to the user to be in one of five possible states, shown in Fig. 4.

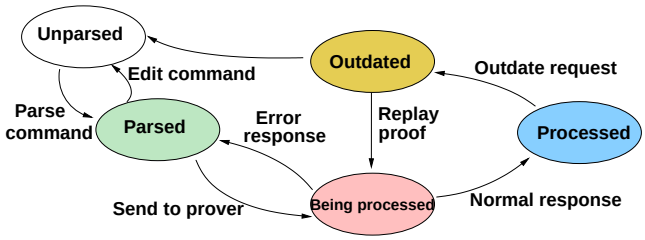

Figure 4: Command states in PGIP displays.

A text segment starts off as *unparsed*, and after parsing becomes one (or more) freshly *parsed* prover commands. Actual proving consists of sending the command to the prover. While waiting for a response from the prover, the command is *being processed*. Once the prover has sent a positive answer, the command becomes *processed*; on the other hand, if the prover sends an error, the command reverts to being parsed. When we *outdate* a command, all commands depending on it are outdated as well. Similarly, to successfully process a command we will need to have processed all commands it is depending on. To edit a processed command, we have to outdate it first.

This model extends the previous model used by Proof General in Emacs and other similar script management systems, which typically provide only *unparsed/parsed*, *being processed* and *processed* states; parsing attempts occur only as a request to process a command is issued. The extra states here are useful for the user. Text may be *unparsed* because a parse attempt has failed, which can be indicated to the user by the familiar wavy red underlines. Text which has been outdated has a different status: we expect that it will be possible to redo such commands without problem.

Eclipse is informed about state changes of command regions by the Broker, and may make change requests triggered by actions in the interface. State change requests are converted into commands that control the prover; the Broker has its own history management which it synchronises with the prover as necessary.

The Eclipse implementation can parse the file as the user types (after an amount of idle time); this updates the outline view and syntax highlighting dynamically. Currently, the plugin assumes parsing to be fast and blocks editing while it happens; this could be improved by adding a sixth command state of *being parsed* behind-the-scenes (to the user this can appear the same as freshly parsed, but the system knows that edited commands must always be reparsed).

3.3 Dependency in proofs

The display protocol also allows an important generalisation of the previous script management dependency model. Classical script management [5] uses *linear dependency*, where every line potentially depends on all lines that come before, because the prover only processes them in a linear fashion. This divides the proof script into a part which has been processed, and a part which still needs to be processed, as shown on the left of Fig. 5.

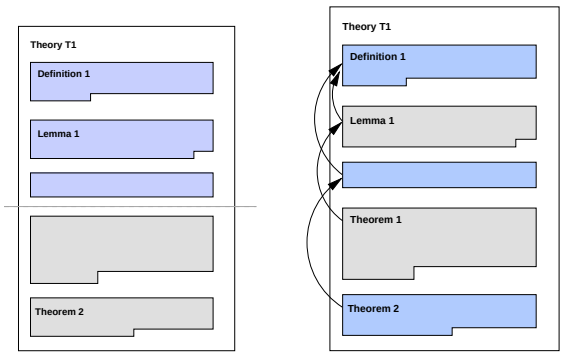

Figure 5: Linear and explicit dependencies in proofs.

By splitting the text into commands, we can have a more fine-grained *explicit dependency*, shown on the right of Fig. 5. Here, to process a command we need only process the prior commands which are really needed. Similarly, to undo a command we only need undo the command and its true dependents. For this to work, the prover must report the necessary dependency information as additional markup.

Similarly, inter-theory dependencies are deduced by Proof General in a linear order by default, or in a graph structure with prover assistance, as shown in Fig. 6.

We want to get a handle on dependencies in proof scripts to enable more sophisticated manipulations, such as smart folding and basic refactorings like renaming and moving theorems between proof scripts. There is much anecdotal evidence from proof developers that support for refactoring is desperately needed, but implementing it correctly for proof scripts (and in a generic way) needs further research.

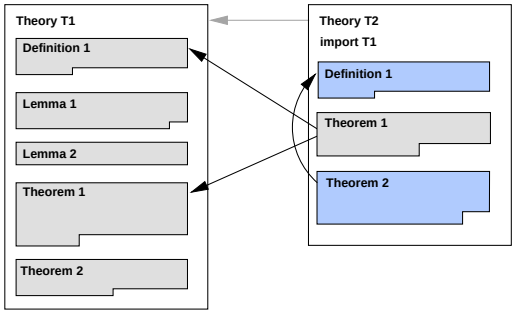

Figure 6: Inter-theory dependencies

4. CONCLUSIONS

We hope that the new Proof General interface based on Eclipse will become the standard working environment for many users of interactive proof systems, as it continues to be developed. Eventually it should replace the existing popular Proof General system based on Emacs. There are still outstanding issues to resolve before we get there.

One issue is that the new PGIP architecture is only so far supported by the Isabelle theorem prover. In principle it would be possible to support others without much effort by using a filtering component which interprets the ordinary prover interaction language (in ASCII), exploiting the existing per-prover customisations taken from our Emacs Lisp code. But we would rather encourage theorem prover implementors to support PGIP directly.

On the Eclipse side, perhaps the most important challenge is in providing the same (or better) flexibility of display and editing of mathematical content which is afforded elsewhere. The *X Symbols* package in Emacs, the MathML markup used by web browsers, and the scientific document editor TeXmacs [10] all go beyond what we could manage with Eclipse. The present symbol support is limited and requires access to a suitably rich unicode font. As a possible improvement, we are experimenting with converting MathML markup from the prover into SVG to be displayed inside Eclipse.

Generalising further, we have designed ways for extending our system to cope with a literate style of development [3]. Using this mechanism, a *central document* stores all content and can be edited by the user in different views. One view reflects the formal proof script, while another reflects a user-oriented view in the style of an informal mathematical paper. Other extensions include the possibility to allow interaction via user gestures such as drag-and-drop; the PGIP protocol already caters for these in a generic way, but this is not supported by the Eclipse display yet.

Proof General Kit is unique in proposing a specific framework customised for interactive proof, although related work exists in other settings. One notable example is the MathWeb project, which provides a standardised XML-RPC interface to a range of automated provers, using the semantic content format OMDoc [9] as an exchange language.

Acknowledgments

Proof General in Eclipse was developed with the assistance of two Eclipse Innovation Grants from IBM, awarded in 2004 and 2005 to Aspinall and Lüth respectively. The actual work on the plugin was undertaken by Winterstein and Fayyaz. Additions and improvements to the system were made by Alex Heneveld. The EPSRC platform grant GR/S01771 has provided some support for the continuing development of Proof General.

For details and downloads of Proof General in Eclipse, please visit `http://proofgeneral.inf.ed.ac.uk/kit/`.

5. REFERENCES

[1] D. Aspinall and C. Lüth. Commentary on PGIP. `http://proofgeneral.inf.ed.ac.uk/kit/`, 2006.

[2] D. Aspinall, C. Lüth, and D. Winterstein. Parsing, editing, proving: The PGIP display protocol. In *User Interfaces for Theorem Provers UITP'05*, Apr. 2005.

[3] D. Aspinall, C. Lüth, and B. Wolff. Assisted proof document authoring. In *Proc. Intl. conf. Mathematical Knowledge Management 2005*, LNAI 3863. Springer, 2005.

[4] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge, 2005.

[5] Y. Bertot and L. Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(7):161–194, Feb. 1998.

[6] G. Gonthier. A computer-checked proof of the four colour theorem. Technical report, Microsoft Research Cambridge, 2004. `http://research.microsoft.com/~gonthier/4colproof.pdf`.

[7] T. C. Hales. The Flyspeck project page. `http://www.math.pitt.edu/~thales/flyspeck/index.html`.

[8] R. Kaivola and K. R. Kohatsu. Proof engineering in the large: formal verification of Pentium 4 floating-point divider. *STTT*, 4(3):323–334, 2003.

[9] M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents*. LNAI 4180. Springer, 2006.

[10] TeXmacs. Web page, 2006. `http://www.texmacs.org/`.

[11] D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[12] Mathematical markup language (MathML). W3C Recommendation, 1999.

[13] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, 2001.