Proof General / Eclipse: A Generic Interface for Interactive Proof

Daniel Winterstein[1], David Aspinall[1], and Christoph Lüth[2]

[1] LFCS, School of Informatics, The University of Edinburgh, U.K.
[2] Department of Mathematics and Computer Science, Universität Bremen, Germany

Abstract. This paper introduces PG/Eclipse; a sophisticated new interface for interactive theorem provers, offering users a rich set of proof development tools. It is based upon two complementary frameworks. The first is PG/Kit, a generic communication framework for connecting theorem provers and interfaces. PG/Kit should allow straightforward adaptation to most interactive theorem provers. Moreover, by separating interface development from proof engine development, this framework should facilitate the development of both. The second is Eclipse, a sophisticated open source framework for building IDEs. Eclipse is highly modular and extensible, making it a good platform for interface research. Using it has allowed us to provide a rich range of interface features. These frameworks correspond to the twin goals of this project: to define a clear separation between provers and interfaces, and to translate programming development tools to a theorem proving environment.

1 Introduction

Developing formal proofs is an arduous and difficult task. Nevertheless, larger and more complex formalisations are being undertaken in numerous interactive theorem provers. However inspite of the considerable achievements of the formal proof programme, take-up of these systems by mathematicians and programmers remains poor. At least one reason for this is the lack of good development tools.

Proof scripts can be very large (e.g. work on formalising Java in `Isabelle` ran to 30k lines with 1400 lemmas [23]), with complex dependencies. Yet the facilities for developing and maintaining formal proofs are in general rather rudimentary, especially when compared with the sophisticated IDEs available to the modern programmer. One reason for this is the heterogenous nature of the formal proof community – which has a wide variety of different systems.

1.1 PG/Emacs

The *Proof General* (PG) project is an ongoing attempt to redress this issue [3, 4]. The previous PG interface was built on the Emacs text editor (we will refer to it here as PG/Emacs to distinguish it from the new interface). PG/Emacs has been successfully used for several years, with an estimated *sim*300 active users. Its success is due to its genericity, allowing easy adaption to a variety of provers

(primarily, `Coq`, `PhoX`, and `Isabelle`, for which it is the 'official' interface), and its design strategy, which targets both experts and novice users.

Although successful, the limitations of the PG/Emacs system are becoming increasingly clear. From the users' point of view, it requires learning Emacs and putting up with its idiosyncratic and at times unintuitive UI. From the developers' point of view, it is rather too closely tied with the Emacs Lisp API which is restricted, somewhat unreliable, often changing, and differs between different flavours of Emacs.

Another engineering disadvantage of PG/Emacs arose from its construction by successively extending a generic basis to handle more provers. This strategy meant that little or no specific adjustment of the provers was required, but it resulted in overcomplicated configuration and internal mechanisms.

1.2 PG/Eclipse

In this paper we present PG/Eclipse, an interface/development environment that marks a new phase in the PG project. It not only provides a more sophisticated suite of editing, browsing and debugging tools – it does so using a generic framework that should allow it to be used by a wide range of systems. We first describe the principles underlying its design, then present the features it offers. We then describe the framework it operates in (called the PG/Kit), the framework it is built on (Eclipse), and give an overview of the implementation.

2 Design principles

The design of PG/Eclipse and the PG/Kit is influenced by a wide range of UI design principles. For example, the UI principle of supporting novices and experts – and providing a pathway to convert one into the other[3] – has always been an important consideration in PG. This has led us to use a *hierarchical* (a.k.a. *level-structured* [21]) interface design[4] Here we describe two principles that are specific to theorem proving.

2.1 Proof scripting is a form of *programming*

It has often been noted that theorem proving has a great many similarities with programming. However this insight has not yet been extensively used in prover interface design. Since programming interfaces serve a much larger community, and consequently have been much more actively developed, it is not surprising that they are more advanced. Hence it is sensible to borrow ideas from programming interfaces where possible. This principle could also be used the other way, and we will mention a couple of areas where TP interface design could potentially benefit programming IDE design.

[3] Novices into experts that is.

[4] In practice this means providing multiple ways of accessing tools: buttons for novices, menus for intermediate users and key bindings for experts, plus focused help to encourage users to move from 'novice' use to more sophisticated use.

2.2 Separation of proof engine and interface

Modularisation brings clear benefits to the development of software. Separating provers from interfaces is a specific instance of this. There are lots of different TPs, and more are created all the time as new ideas and logics are developed. Each of these needs a good interface if it is to be successful except as a demonstration of a concept. Hence separating prover and interface should facilitate the development of TPs, by giving TP developers easy access to sophisticated interfaces. It would also benefit interface designers, who need not be tied to one proof system. Note that whilst we have designed PG/Kit to implement this separation, whether or not it is successful will in the end depend upon the interactive prover community.

3 The PG/Eclipse interface: a functional description

This section presents the main features of PG/Eclipse from a user-perspective. Many of these features will be familiar to users of modern programming IDEs, where they are now increasingly standard.

3.1 Eclipse

The restrictions of Emacs have led us to adopt Eclipse as the new platform for developing Proof General. Eclipse is a modern IDE, originally intended for Java programming. As such it fits with this project's 'proving as programming' design principle. Amongst the features Eclipse offers are:

- A state-of-the-art user-friendly GUI. This is based around a collection of editor and view windows. Editors are (typically) enhanced text-editors, whilst views provide focused information, navigation tools and access to task-specific features (e.g. CVS actions). It is a flexible system which can be easily reconfigured by the user. Views and editors can be opened, closed resized, grouped together (with tabs), and the last configuration is automatically remembered. These learnt configurations are task specific (in Eclipse jargon, UI configurations are linked to *perspectives*) - which allows different configurations for, say, browsing and editing. Configurations can also be saved and loaded, but the option of task-specific perspectives means that this is rarely necessary.
- Integrated CVS support, including a CVS client with a GUI and a comparison editor for accepting/rejecting differences.
- Good platform independence. PG/Eclipse can be installed on Linux, Windows or Mac boxes without any special configuration. Of course, the underlying theorem prover may be platform dependent, but a socket layer allows the editor to be run on a different machine from the prover.

3.2 Script management

The central feature of PG/Eclipse is an advanced version of *script management* (c.f. [8]), which is a form of step debugging tailored to linear scripts. To interactively 'run' a proof script, we sequentially send each line to the prover. Script management says that each script can be divided into a part which has already been processed, a part which is currently being processed, and a part which has not been processed (yet). Proof General supports this by colouring the parts of the text being processed or already processed, and preventing editing in those regions.[5] This provides clear feedback to the user on processing, and protects against edits that might invalidate the current proof state. A toolbar provides buttons for navigating (moving the prover's position) within the proof.

A screenshot in Fig. 1 shows this in action. The main editor window shows the proof script; view windows below show the prover output. An Eclipse Problems view (not shown here) lists outstanding problems, such as syntax errors or unfinished proof-goals. To the left of the editor window is an Outline view of the proof script showing its structure. Above the editor, the dedicated toolbar triggers proof or undo steps by sending instructions to the prover (left-to-right, the buttons are: *undo all*, *undo*, *interrupt*, *step forward*, *process all*, and – most useful of all – *go to cursor location*).

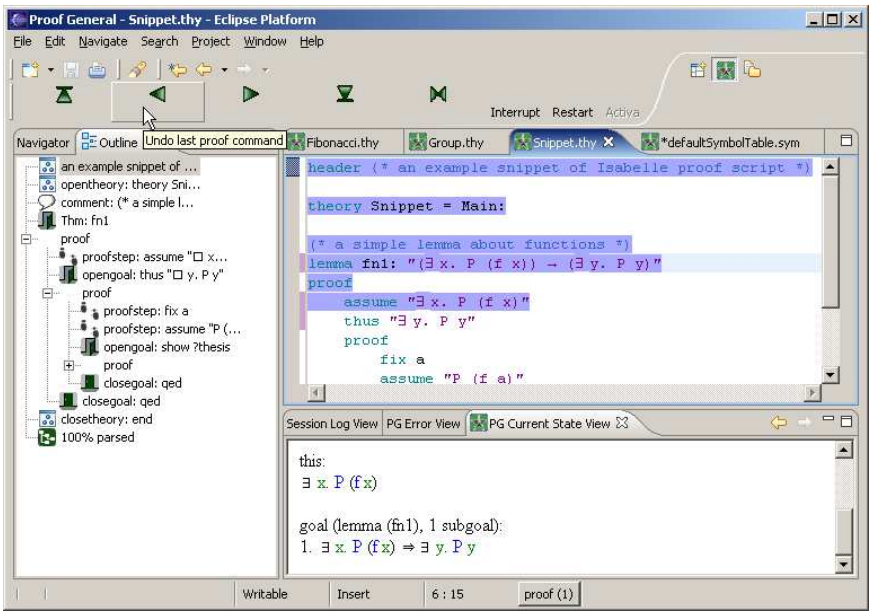

Fig. 1. Eclipse Proof General Display

[5] The colouring of script management is demonstrated in the screenshot in Fig. 1.

3.3 Symbol support

Using mathematical symbols can make a huge difference to how readable a proof script is. This is possibly an area where TP interfaces have something to offer to programming interfaces.

PG/Eclipse provides symbol support similar to the Emacs X-Symbol package [24]. Like X-Symbol, it supports the use of typing shortcuts to enter symbols (e.g. typing "--->" for "\<longrightarrow>"). At present, it is less powerful than X-Symbol in that it does not properly support subscripts and superscripts. PG/Eclipse also provides a symbol table editor (shown in Fig. 2), allowing users to adapt and extend the use of symbols to fit their own needs. Symbol support is based on Unicode, which has the advantage that the user can cut and paste text between modern applications, preserving symbols. The disadvantage is that the symbol support available depends on access to a suitably rich unicode font.[6]

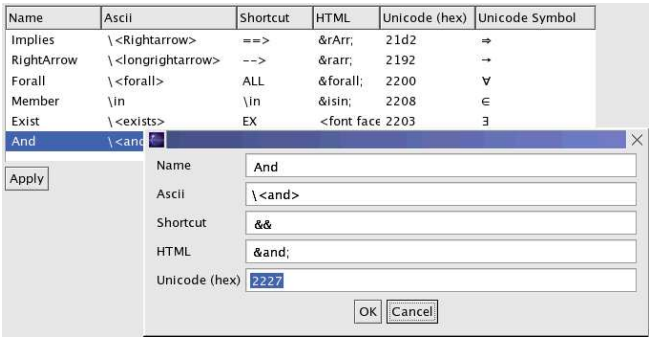

Fig. 2. The PG/Eclipse Symbol-Table Editor

3.4 Theory navigation

As a theory grows, finding specific definitions and proofs can become increasingly difficult and time consuming. This alone can make a large difference to the usefulness of a theory. PG/Eclipse provides support for theory navigation. As proof script files are read, their structure is analysed and the theories, theorems and lemmas they contain are indexed. This gives the user several ways to navigate a project:

– Via a 'Theory Index View' (which supports user-specified filters). This is a list of previously processed theories, theorems and definitions. The list

[6] Standard Linux and Windows installations should be fine; we have not tested on Macs.

entries are hyperlinked to the relevant proof-script section, allowing quick access.
- Via a similar 'Problems View' shows a (hyperlinked) list of unsolved problems, such as unproved goals or syntax errors.
- Via a 'jump to definition' command, that allows users to look-up the source for a rule.
- Within a file, a zoomable/collapsible outline view shows the file structure at a glance and allows quick navigation.

There are also Eclipse's in-built indexing and browsing features, such as 'edit browsing'(which takes the user back through the locations of recent edits in a file), book-marks and To-Do lists.

3.5 Content Assistant

The Content Assistant suggests completions for keywords/phrases. It is like the Auto-Completion feature found in word-processors, but activated by a hot-key combination rather than by typing alone (making it less intrusive). At present, it uses a list of theorems plus a static list of keywords. However there is support in PGIP for the prover to amend this list and provide more context sensitive suggestions.

3.6 Integrated help

As with programming, good documentation and help is key to both the uptake of theorem provers and the reuse of proof scripts. PG/Eclipse provides tools for documenting prover commands, prover settings and proof scripts. Help is then presented to the user via tooltips, shown in response to a pause or 'hover' by the user (see Fig. 3).

Help for prover commands can be defined via a simple XML file. Help for prover settings (e.g. heap size, start-up directory, etc.) is handled similarly. PG/Eclipse provides a GUI front-end for viewing and changing prover settings. This is configured via another simple XML file, and can include help on these settings. Help for theory elements (e.g. theorems) is created simply by writing a preceding comment. This idea is taken from the Java approach to documenting code (c.f. [13]). The top half of Fig. 3 shows an example of this: the comment creates a help entry for "Cantor's Theorem"; the bottom half of Fig. 3 shows this entry being displayed when the cursor is held over a matching term. Using comments gives a method for documenting proof scripts whilst writing them – which aids both creation and maintenance of documentation ([13]). It is straightforward (and will already be familiar to many users), and not overly time-consuming. Also it means that help entries can be extracted from old files, even though the comments were not originally intended to be used in this way. Sometimes this will result in unhelpful tips. However the cost associated with such misses is very low ($\sim$2 seconds of the user's time).

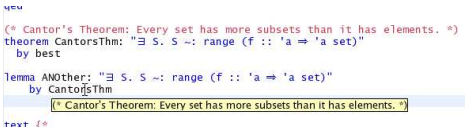

Fig. 3. Defining and displaying focused help via tooltips.

Help on the PG/Eclipse system itself is provided via links to a Wiki[7]. This is, we hope, a good solution to the problem of documenting systems that serve a small community (i.e. systems where the developers cannot afford to provide professional level support and documentation). The PG Wiki includes a mechanism for requesting help from the developers (text of the form "QUESTION: blah blah blah" is interpreted as a request for help). This has the benefits of a Q&A user-group, whilst simultaneously creating structured documentation. It also allows users to easily share information, thus alleviating some of the documentation burden on the developers. Although PG/Eclipse's current documentation is slight, using a Wiki means that it can grow as necessary in response to user demands.

3.7 Teaching tool

One of the great potentials for provers, and as yet largely untapped, is in mathematical/scientific education. Although there are excellent computer algebra and graphing systems used in education (e.g. Mathematica [26]), these neither perform nor teach proofs, which is central to mathematical work.

PG/Eclipse introduces a teaching tool, designed for delivering teaching material that interacts with a prover.[8] The tool uses an embedded browser to display web-pages. Links to proof scripts (recognised by file type) cause the script to be opened in the PG editor. PG also defines some Javascript commands for interacting with the script editor (e.g. setting preferences). The use of a web-browser means this tool is very flexible, and should be easy to integrate with existing web-based teaching material.

3.8 Interface scripting

Often a script will run better or view better under certain settings. PG/Eclipse allows a theory developer to encode these settings in the proof script file. It defines a small set of commands that can be used to script the interface itself. Currently, there are commands for setting preferences, both of the interface and of the underlying prover, for extending symbol support (e.g a script can specify that `myRelation` should display as $\star$), and for running other such scripts (allowing common settings to be factored out). These commands can be embedded

[7] A *Wiki* is a website that can be edited by any visitor.
[8] This tool could also be used to deliver documentation linked to 'live' examples.

inside comments, so that the proof scripts will still run outside of PG/Eclipse.[9] This feature could also be useful when using PG/Eclipse as an educational tool (where, for example, some automatic proof features might want to be temporarily disabled).

```
(* <proofgeneral>
   <loadscript src="defaultThySettings.pgip"/>
   <setpref name="Isabelle:full-proofs" value="true"/>
   <addsymbol ascii="myOperation" unicode="066D"/>
</proofgeneral> *)
```

Fig. 4. An example interface script 'hidden' inside an Isabelle/Isar comment.

4 Proof General Kit architecture

The PG/Kit framework is an attempt to achieve the design principle of separating the prover from the interface. It was developed using experience gained from the PG/Emacs project. We give an overview of its design here; for more details see [10].

It is unrealistic to expect that a prover should not need modification to support a sophisticated interface. So instead of trying to match a range of different behaviours, we propose a uniform API which captures the behaviour common to most provers at an abstract level, and ask that each proof system implements that. If done correctly, this will not place a great burden on the prover developer, and by defining a clear separation between the interface and the prover, it should greatly facilitate interface development.

The PG/Kit API consists of two linked parts: A model for prover behaviour, and a protocol for communication within proof sessions. This design reflects a compromise between creating a flexible system, and working with existing systems. An alternative would be to use a more communication based approach, where the prover informs the interface about the proof-script relevant effects of executing commands. For example, consider the `undo` command. the PG/Kit model specifies how undo behaviour should change depending on context; the communication-based approach would be to leave undo behaviour unspecified, and for the response to an undo command to specify precisely what has been undone. This would be more generic, but would require a greater change from existing systems.

[9] This is similar to the way that Javascript used to be hidden inside html comments, so that the document would render correctly on old browsers.

4.1 Modelling the prover state

PG/Kit assumes an abstract model of how interactive provers behave, where we suppose there are four fundamental states occupied by the prover. Transitions between the states are triggered by commands issued via the interface. Fig. 5 shows the states, and the commands to change between them.

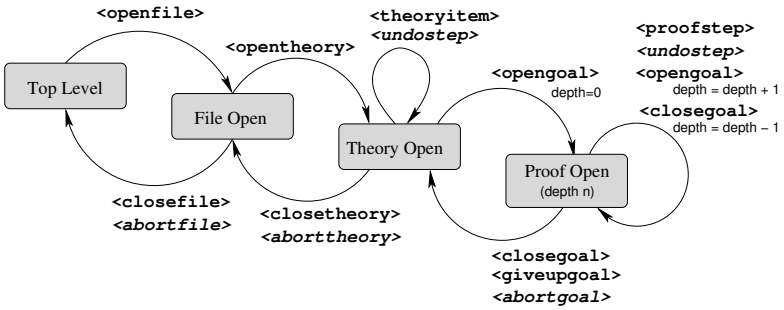

Fig. 5. Proof states during development.

The reason for distinguishing the states is that different commands are available in each state, and the prover's undo behaviour in each state can be different. This model is based on abstracting the common behaviour of many interactive proof systems, but it is not intended to capture precisely the way every proof system works. Rather it acts as a clearly specified "virtual layer" that must be emulated in each prover to cooperate properly with the broker.

4.2 PGIP: A protocol for interactive proof

The protocol for directing proof used by PG/Kit is called *PGIP*, for *Proof General Interactive Proof* [5]. It was designed by examining the communications used in PG/Emacs, and covers a wide range of prover-display interactions. The format of the messages is defined by an XML schema[10] Messages are sent over channels (both sockets and Unix pipes are supported). Note that it is not necessary for a prover to support all of PGIP in order to use the PG/Eclipse interface (and indeed, at present PG/Eclipse does not implement all of PGIP). The main types of command in PGIP are:

Proof script commands, corresponding to the commands in a conventional proof script. These would typically be created from such a script. They affect the internal (proof-relevant) state of the prover.

Improper commands are those which should not appear in a proof script. They are used for controlling the proof session; examples are the *undo* and *abort* commands appearing in Fig. 5.

[10] Written in RELAX NG [18], but also available as an XML DTD).

Display messages are sent from the prover, and contain output directed to the user, such as the current proof state or error messages.

Parsing commands define a mechanism whereby PGIP mark-up is added to sections of 'raw' proof scripts (i.e. scripts in the language of the prover). We expect that this task will normally be implemented as part of the theorem prover, which should know how to parse its own language.

Other message kinds include *configuration messages* (allowing dynamic setup of components), descriptions of the current proof environment (e.g., the theorems that are currently available) and a meta-data category for miscellaneous prover-specific messages.

4.3 Proof scripts in PGIP

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 by an interface. This mark-up is only used internally; it is not seen or edited by the user.For example, Fig. 6 shows the PGIP representation of the a short example proof script. Notice the named and unnamed <opengoal> elements, and the indentation structure introduced by <opengoal> and <closegoal>.[11]

```
lemma fn1: "(EX x. P (f x)) --> (EX y. P y)"
proof
  assume "EX x. P (f x)"
  thus "EX y. P y" by simp
qed
```

With PGIP mark up:

```
<opengoal name="fn1">
    lemma fn1: &quot;(EX x. P (f x)) --&gt; (EX y. P y)&quot;
</opengoal>
<proofstep>proof</proofstep>
  <proofstep>assume &quot;EX x. P (f x)&quot;</proofstep>
  <opengoal>thus &quot;EX y. P y&quot;</opengoal>
    <closegoal>by simp</closegoal>
<closegoal>qed</closegoal>
```

Fig. 6. An example proof script in Isabelle/Isar, with PGIP marked-up version.

The design decision to build PGIP as a wrapper for native languages has two main consequences:

[11] One may wonder why <opengoal> and <closegoal> are separate and distinct elements; we do not use a single <goal> element to enclose the block structure because we need to be able to incrementally parse and evaluate text, which means handling ill-structured fragments of a block.

Firstly, the user employs the prover's native language (such as Isar in Fig. 6) to write the proof scripts instead of one generic language. This is necessary, since the wide variety of logics and proof styles supported by modern provers make it very hard to come up with one language which efficiently supports all of these. It is also pragmatic, as users can continue to work in the proof language they are familiar with and old proof scripts can still be used. Moreover, the interface does not lock the user into always using it. Proof scripts developed with PG/Eclipse can be run independently from the interface.

Secondly, the parsing of the proof script, and thus all user input, has to be done by the prover (or a component coming with the prover), and not by the interface, which knows nothing about the prover's language. Note that this parser only needs to understand *some* aspects of the prover's language – principally, it must be able to distinguish between the command types that appear as labels on the state-transition arcs in Fig. 5. Hopefully, it should not be arduous to develop such parsers (although c.f. §5.4 for a case study).

5 Implementation

5.1 Using Eclipse

Eclipse is an open-source IDE and tool integration platform written in Java and SWT (SWT is a widget set created by IBM, which exploits native operating system widgets to provide an interface that is both faster and more familiar than most cross-platform GUIs) [20, 12]. Most prominently, Eclipse provides a powerful and attractive IDE for Java, but it also has a modular design based on *plugins* and *extension points* that allows almost any aspect of the platform to be customised and extended to new domains. Many plugins are available, supporting other programming languages, profiling and testing tools, graphical modelling facilities, etc.

Extending Eclipse is not as straightforward as one might hope. To provide functionality similar to that offered for Java, it is necessary to re-implement much of the functionality. The learning curve for using Eclipse is quite steep. One of Eclipse's aims is to provide an extensible API that is cleanly separated from the code-base, with custom extensions created using XML schemas rather than via Java code. In practice, this has only partially been achieved. Most extensions require Java code to complete their setup, which means engaging with a complex and inter-linked code-base. For example, to set up basic syntax highlighting (e.g. keyword and comment colouring – something that can be done with one straightforward file in many editors), requires an XML configuration file plus properly sub-classing and setting up six different Java classes.

However, Eclipse does provide a powerful and wide-ranging set of tools, plus considerable support for creating new tools that work within a development environment. These tools are also frequently interlinked, so that implementing one aspect of a tool can unlock a range of extra functionality 'for free'. Also, Eclipse imposes relatively few limits on what can be implemented within its framework.

5.2 PG/Eclipse Architecture

The PG/Eclipse plugin is implemented with around 10 Java packages containing about 100 classes, most of which contain small pieces of code to interface to the Eclipse platform. It represents roughly one year's work. The architecture of the main editing loop of PG/Eclipse is shown in Fig. 7.

PG/Eclipse is implemented along Model-View-Controller lines. On loading, proof scripts are converted into tree structured 'document objects' – this is the model. Efficiently maintaining these document objects whilst the user edits the text is a difficult problem, which we discuss in §5.3. This is presented to the user via a specialised text-editor and associated viewers (e.g. the document outline view). These 'views' support various actions (the 1controller' aspect), most importantly text editing and script management. Communication with the prover is handled via the PGIP Gateway module, which converts between internal command and event objects, and PGIP messages. This module also handles establishing proof sessions, and maintains a model of the prover's state, which is necessary to properly implement actions such as `undo`.

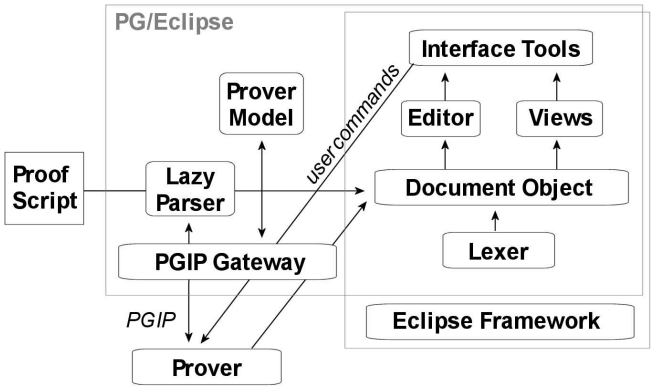

Fig. 7. PG/Eclipse system architecture.

There are also several parts of PG/Eclipse that sit outside the editing loop, and are not shown in Fig. 7. Most important of these are the views giving feedback on the session: the Current State View, an error log view, and a session log view. These views simply listen to the event traffic generated by the PGIP Gateway, and display appropriate portions of it. Other aspects of PG/Eclipse not illustrated here are the symbol table editor, the teaching tool, and the preference system.

5.3 Parsing

PG/Eclipse provides tools that use information gleaned from the structure of the script (and stored in the Document Objects). However it also supports free form text edits – which can wreak arbitrary changes to a script's structure. This requires re-parsing following all user edits – something which could seriously interfere with editing unless it can be done very fast. We solve this problem by dividing parsing into two phases.

In the first phase, a fast lexer is used to perform syntax highlighting and to break scripts into smaller partitions as the user is typing. This uses a simple scanning algorithm, and is based on Eclipse's standard classes.

In the second phase, the PGIP mark-up structure is obtained using a 'lazy' parser. Typically, the Lazy Parser module will act as a go-between, and the actual parsing will be done by the prover or another outside process (c.f. Fig. 7).[12] The lazy parser is not fast enough to be run as the user types. Instead it is run as needed in response to specific user commands (such as evaluating a script).

5.4 Converting Isabelle to use PGIP

Adapting a prover to use PGIP (and hence the PG/Eclipse interface) can be done either within the prover or via a PGIP wrapper. The main work (besides XML wrapping and unwrapping, which can be done using standard libraries) is building a parser for the prover's scripting language. For many proof languages, this should be a straightforward task. Converting Isabelle presented more of a challenge due to the complex and extensible nature of it's syntax.

It was decided to build the parser component within Isabelle. While straightforward in principle, this turned out to be harder than expected. Isabelle/Isar interprets scripts using a mix of abstract parsing and operational semantics, and there were difficulties with parsing proof scripts independently of their execution: the Isabelle code uses functional combinators to build combined parse-execute functions that were hard to unravel.

We expect that this will usually be easier to do in other systems (and PG/Kit also supports standalone parsing components).

5.5 Displaying prover output

There are two facilities available for attractively displaying output from the prover. Firstly, the output uses the same symbol support as the text-editor, allowing it to contain mathematical symbols. Secondly, the output is displayed using an embedded html display widget, and can be formatted in arbitrary ways via an XSL style-sheet. The default style-sheet provides pretty-formatting (e.g. colouring of free vs. bound variables) for the PG *Markup Language* (PGML).

[12] Although our implementation also allows for a Java-based parser to be used, if one is available, which would be more efficient.

This is a sub-protocol of PGIP, which can be used to mark-up display messages. PGML combines lightweight mark-up for terms with meta-information on how terms should be displayed.

6 Related work

A notable exception to the rule that theorem provers have poor front-ends is Theorema, a theorem prover built on top of Mathematica [7]. This provides it with an excellent user-interface for entering and viewing mathematical expressions. However this is also a weakness, as Mathematica is proprietary software with closely-guarded source code. This means its GUI cannot really be extended, and the Theorema system cannot be made freely available. The TeXMacs editor has a similar GUI [17], and is both more extensible and an open source project. Although originally intended as a WYSIWYG latex editor, it can perform scripting (and there are projects to develop TexMacs-based interfaces for Omega and Coq). An added advantage of TexMacs is that it supports converting a proof into a Latex paper.

Various systems already exist for script management (e.g. PG/Emacs, or the `PCoq` interface for `Coq` [1]). However, these offer less functionality than PG/Eclipse, and do not provide a satisfactory solution to the problem of handling different provers.

There are several efforts to publish formalised mathematical content, including Mizar [22], HELM [2], MoWGLI [16], Logosphere [19], and the XML format OMDoc [15]. OMDoc explains the semantical content of logical terms, which goes beyond the PG/Kit. It would be interesting to consider an extension of our protocols to support OMDoc exchange, although we would not want to force the underlying provers to implement OMDoc. The MathWeb project builds on OMDoc, providing a standardised XML-RPC interface to a range of automated provers (Otter, Spas, etc.) [27]. It does not have support for interactive systems though (partly because it has no inherent concept of a proof state or a proof session). This makes it a valuable possible partner to Proof General.

The most closely related work is that done within the PG/Kit framework. The 2nd author has developed a PGIP-based version of PG/Emacs, whilst the 3rd author is working on a PGIP-based 'proof desktop', where theories and proofs are built up using graphical actions such as drag-and-drop. There is also a *broker* component in that can act as a middle-man between a collection of PGIP-equipped provers and interfaces [10]. This should lead to increasingly flexible ways to develop proofs.

7 Future work

There are many possible lines for future development.

Firstly, we want to use the Eclipse framework to further explore the analogy between theory development and software engineering. There are several ideas – such as *code folding* (which makes large files more navigable by allowing blocks of

a proof script to be hidden (folded away)), *refactoring* (i.e. support for renaming and reorganising sections of code), and pre-emptive type checking – which could usefully be applied to proof development tools.

7.1 Proof Planning support

We can also go beyond adapting program development tools. One promising line of work is on using interactive proof planning to construct proof scripts.Proof planning is a powerful backwards-reasoning technique based on capturing expert knowledge of proof structures. Moreover, proof-planning systems can include automatic search and automatic analysis of failed proof attempts [9]. We are currently working with Lucas Dixon on integrating IsaPlanner (an Isabelle based proof-planner) with PG/Eclipse [11]. This should lead to a PGIP-based API for generic proof-planner support. The proposed UI for this is:

1. The user asks for help on how to tackle an unsolved goal. This can be the current goal, but it could also be any goal in the script which is marked as unsolved. For example, in the script shown in the top half of Fig. 7.1, the user selects to apply IsaPlanner to solve the 'sorry' (which marks an unsolved goal in Isabelle) via a right-click activated context menu.
2. PG then uses the 'goto' command to process or undo the script as necessary (so that the prover is at the appropriate point in the script), and then calls the planner.
3. The planner responds with a list of suggestions, comprising possible changes to the script. At the simplest level this could be a list of steps to try, but it could also include advice on lemma-speculation, or generalisations that might be easier to prove ([9]).
4. The user can then select one of these suggestions, or ignore them. If a suggestion is selected, then the proof script text is edited accordingly (leading in our example to the script shown in the bottom half of Fig. 7.1).

Often a proof plan will include unsolved sub-goals, as happens in the example shown in Fig. 7.1. In this case, the process can be repeated if desired. This modus operandi allows the user to switch seamlessly back and forth between free-form text-editing and using a proof planner.

Note that this could be a valuable idea to transfer to IDE based programming. The closest existing analogue is the idea of 'programming templates'; fragments of code which can be used to help build up programs (e.g. a `for` loop, or a standard widget initialisation method). Proof planning is considerably more powerful.

7.2 Longer term aims

The PG/Kit framework should provide a good platform for developing a variety of ideas to enhance and improve interactive theorem provers. Ideas we are considering include:

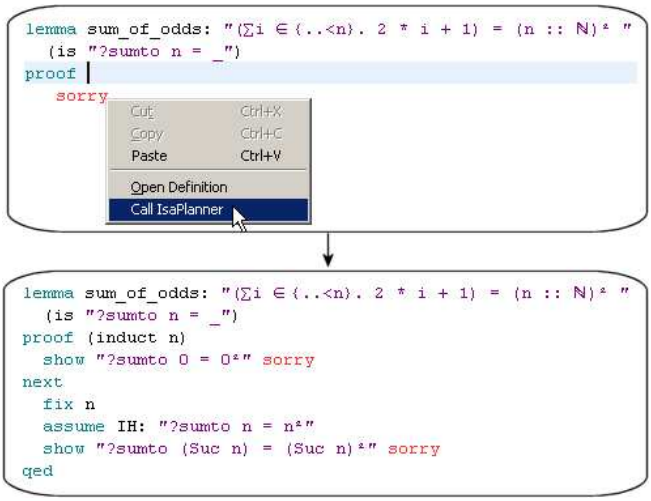

Fig. 8. A partial Isabelle proof is extended using a proof-planner.

- Greater support for mathematical layouts (e.g. fractions).
- Diagrams provide intuitive ways of representing and reasoning about many domains. They are ubiquitous in mathematics texts, but hardly used in theorem provers.[13] Those provers that do use diagrams (e.g. HyperProof [6] or Dr.Doodle [25]), are currently very domain specific, and not suitable for serious mathematical research. It would be a major improvement to develop support for diagrammatic editors and views that would allow them to be used as part of proof texts.
- Supporting formal proofs that utilise multiple provers. This requires translation mechanisms, which are inherently logic-specific (or worse, specific to a 2-prover combination). It also requires great care if the guarantee of soundness is to be preserved. Nevertheless, there are some standards which might make this feasible (e.g. `Otter` syntax is understood by many 1st order theorem provers). This goal would probably be realised by a link-up with the MathWeb project.
- User-friendly enhancements to proof languages. The Proof General framework already partially separates the proof language seen and edited by the user (the 'user level language') from that seen by the broker/prover (which uses PGIP). Thus this framework is well suited to allowing user level proof languages to be modified without having to re-program the underlying theorem prover. This could be used to support productivity-enhancing features

[13] Apart from *proof trees*, which – whilst valuable as an interactive representation – are notable for their complete absence from textbooks.

such as Latex-style macros.[14] More ambitiously, we could look at allowing some use of natural language in proof scripts.

8 Conclusion

The lack of good interface and development tools is one factor holding back the use of interactive provers. PG/Eclipse attempts to redress this, drawing heavily on the analogy between formal proof and programming. We have outlined the main features of the system, and given details on how these were implemented.

An important aspect of the system is that it is based on the PG/Kit framework. This framework is designed to aid the development of both provers and proof-interfaces by providing a clear separation between them. Modularisation is especially important in this domain, where – inspite of the relatively small size of the community – there are a range of target applications and a diverse wealth of systems. Eventually, we hope that implementers of both new and existing proof systems will have a compelling reason to add PGIP support to their systems to access powerful front-ends. On the other end, we hope that the PG/Kit will stimulate development of a wider range of interface tools. Such tools could be developed as Eclipse plugins – extending PG/Eclipse– or as stand-alone components.

Although the benefits of separating provers from interfaces are large, ultimately this separation can only be achieved by the community acting together. We hope that PG/Eclipse will provide a good starting point for this.

8.1 State of the project

An alpha-release of PG/Eclipse is now available. It can be downloaded from http://proofgeneral.inf.ed.ac.uk/kit/wiki. The latest version of Isabelle now supports PGIP (due to work by the 2nd author), and is available from [14]. Developers interested in using Proof General for other provers should contact the authors. We also welcome contact from researchers interested in working with us on future directions.

Acknowledgments: D.Winterstein was supported by a 2004 Eclipse Innovation Grant awarded by IBM. D.Aspinall benefited from support provided by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing. C.Lüth does it for love.

References

1. A. Amerkad, 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)*, 2001.

[14] Macros would be easy to develop within the PG/Eclipse framework. However, since there is not a well-defined usage for macros in proof scripts, we do not want to commit to a macro-language without further investigation.

2. A. Asperti, L. Padovani, C. S. Coen, and I. Schena. HELM and the semantic math-web. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2001.
3. D. Aspinall. Proof General: A generic tool for proof development. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1785, pages 38–42. Springer, 2000.
4. D. Aspinall, H. Goguen, T. Kleymann, and D. Sequeira. Proof General system documentation, 1999-2004. Available at http://proofgeneral.inf.ed.ac.uk/doc.
5. D. Aspinall and C. Lüth. Commentary on PGIP. Available from http://proofgeneral.inf.ed.ac.uk/kit/, September 2003.
6. J. Barwise, J. Etchemendy, and G. Allwein. *Hyperproof*. CSLI Lecture Notes. University of Chicago Press, 1994.
7. B.Buchberger, T.Jebelean, F.Kriftner, M.Marin, E.Tomuta, and D.Vasaru. A survey on the theorema project. In W.Kuechlin, editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation)*. ACM Press, 1997.
8. 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.
9. A. Bundy. Planning and patching proof. In *Artificial Intelligence and Symbolic Computation (AISC 2004)*. Springer, 2004.
10. C. L. David Aspinall and D. Winterstein. A framework for interactive proof, 2004.
11. L. Dixon and J. Fleuriot. Isaplanner: A prototype proof planner in isabelle. In *19th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence. Springer, 2003.
12. Eclipse homepage: eclipse.org. See http://www.eclipse.org.
13. L. Friendly. The design of distributed hyperlinked programming documentation. In *International Workshop on Hypermedia Design '95*, 1995.
14. Isabelle home page. See http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.
15. M. Kohlhase. OMdoc: Towards an OpenMath representation of mathematical documents. Available from http://www.mathweb.org/omdoc/.
16. MoWGLI. mathematics on the web: Get it right by logics and interfaces. http://www.mowgli.cs.unibo.it/.
17. L. R. Philippe Audebaud. Texmacs as authoring tool for formal developments. In D. Aspinall and C. Lüth, editors, *User Interfaces for Theorem Provers UITP'03*, volume 103 of *Electronic Notes in Theoretical Computer Science*, 2003.
18. RELAX NG xml schema language, 2003. Home page at http://www.relaxng.org/.
19. C. Schürmann, F. Pfenning, M. Kohlhase, N. Shankar, and S. Owre. Logosphere. a formal digital library. http://www.logosphere.org/, 2003.
20. S. Shavor, J. D'Anjou, S. Fairborther, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2003.
21. B. Shneiderman. *Designing the User Interface*. Addison-Wesley, 1997.
22. A. Trybulec et al. The mizar project, 1973. See web page hosted at http://mizar.org, University of Bialystok, Poland.
23. D. von Oheimb and T. Nipkow. Machine-checking the java specification: Proving type-safety. In *Formal Syntax and Semantics of Java*, pages 119–156, 1999.
24. C. Wedler. Emacs package X-Symbol. Available from http://x-symbol.sourceforge.net, 2003.
25. D. Winterstein. Dr.doodle: A diagrammatic theorem prover. In *Automated Reasoning: Second International Joint Conference (IJCAR 2004)*. Springer, 2004.

26. S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, 1991.
27. J. Zimmer and M. Kohlhase. System description: The mathweb software bus for distributed mathematical reasoning. In *18th International Conference on Automated Deduction (CADE 18)*. Springer, 2002.