A Framework for Interactive Proof

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

[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 a software framework for conducting interactive proof, dubbed the *Proof General Kit*. It defines a component infrastructure, the syntax of messages exchanged between components, and the protocol governing message exchanges. The infrastructure connects *provers* to one or more *display* components for interacting with the user, such as the Emacs editor or a plugin for the Eclipse IDE, via a *broker* middleware component.
The aims of the framework are to formulate a generic notion of conducting interactive proof, to reduce effort in building proof engines by providing reusable user-level interfaces, and to reduce effort in building interfaces. The first aim helps achieve the second: by designing a common interface protocol we get one interface usable by many provers. By putting most of the interface functionality into a common core and supporting high-level operations in the broker (e.g., history and dependency management), we achieve the third aim, supporting multiple front-ends with a lightweight protocol.

1 Introducing Proof General Kit

The use of machine proof is becoming more widespread, and larger and more complex formalizations are being undertaken in numerous interactive theorem proving systems ("provers" for short). There are both general purpose provers (for example, HOL, Isabelle, PVS, Coq, ACL2, NuPrl), and provers which are more specialized to particular domains (for example, Atelier B or the B Tool, the KeY tool). For most of these systems, the record of instructions of how to create the proof, or a representation of the proof itself, is kept in a text file with a programming language style syntax. We call these files *proof scripts*. Each system uses its own proof script language, and while there are similarities across languages, there are crucial differences as well, particularly concerning the underlying logic.

For large examples, the proof scripts are also large: some notable cases include work on formalizing type safety and a programming logic for Java in Isabelle (30k lines, 1400 lemmas [27]); the Fundamental Theorem of Algebra in Coq (80k lines, almost 3000 lemmas [12]), and work on specification and verification of pipelined processors in PVS (examples ≈10k lines, 500 lemmas [15]). Each of these cases represents several person-years of work. Yet compared with the facilities available to the modern programmer, such as sophisticated IDEs

with browsing and refactoring facilities or visual tools for generating code fragments, the facilities for developing and maintaining formal proofs are in general rather poor. There are two apparent reasons for this. First, more research is needed into the foundations of such tools (early work includes [18, 19, 13]). Second, and equally important, the fragmentation of the community of formal proof developers across different systems means that the users of each language do not have the resource to provide elaborate tools individually.

In this paper we introduce a new software framework, the *Proof General Kit* (or PG/Kit for short). We believe that this framework will help provide sophisticated and useful development tools for a whole class of interactive provers, and also form a vehicle for research into the foundations of such development tools.

Outline Sect. 2 motivates the PG/Kit framework, describing the contribution of the current Proof General system and the component architecture for the new framework. Sect. 3 introduces the PGIP protocol, describing the syntax of messages and the protocol for their exchange between components. Sect. 4 describes the central role of the broker component. Sect. 5 describes several display components which provide user interface elements. In Sect. 6 we conclude, mentioning future and related work.

2 Proof General Kit architecture

The claim that we can provide a uniform framework for interactive proof seems bold. Why can we expect to provide useful tools at a generic level for a dozen different provers? Especially when those provers do not just differ in their underlying languages, but also in their existing interaction mechanisms as well.

2.1 Proof General and script management

The *Proof General* project [2, 5, 20] demonstrates that our aims are feasible. Proof General is a successful generic interface for interactive proof assistants, built on the Emacs text editor. Its success is due to its genericity, allowing easy adaption to a variety of provers (primarily, Isabelle, Coq, and PhoX), and its design strategy, which targets experts as well as novice users. Its central feature is an advanced version of *script management*, closely integrated with the file handling of the proof assistant.

To explain script management, consider the short example proof script in Fig. 1. To interactively "run" this script, we send each line to the prover; thus, each line corresponds to a prover state, and the prover's current state always corresponds to one particular line of the script (i.e. the prover's *position* in the proof). Script management says that for 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[3]. A toolbar provides buttons for navigating (moving

[3] The colouring of script management is demonstrated in the screenshot in Fig. 8.

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

Fig. 1. An example proof script in Isabelle/Isar.

the prover's position) within the proof. A particularly attractive feature of the interface is that these navigation buttons behave identically across numerous different systems, despite the behind-the-scenes use of rather different control commands to jump to different positions in the proof.

Although successful, there are drawbacks to the present Proof General. 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 the present Proof General arose from its construction by successively generalising 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 an overcomplicated instantiation mechanism.

2.2 Introducing the framework

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 a better idea is to propose a uniform protocol (and API or message language) which captures the behaviour common to all 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. We also want to generalise away from Emacs and allow other front-ends, and possibly several at once, so that the proof progress can be displayed in different ways, or to other users. It would also be useful to run several provers at once (which the present interface forbids). In the end, what we need is a software framework; a way of connecting together large interacting components customized to the domain.

The PG/Kit framework has three main component types: interactive *prover* engines, front-end *display* components, and a central *broker* component which orchestrates proofs-in-progress. The architecture is pictured in Fig. 2.

The components communicate using messages in the PGIP protocol, described in the next section. The general control flow is that a user's action

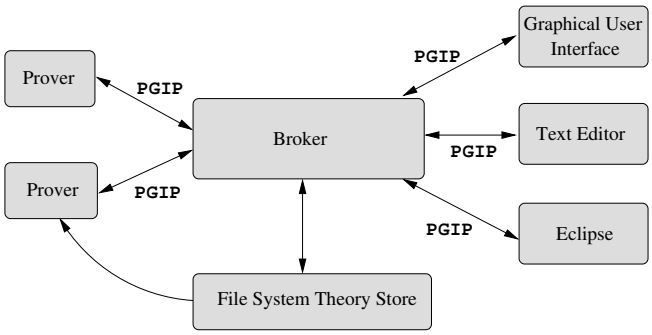

Fig. 2. PG/Kit Framework architecture

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. Messages are sent over channels, typically sockets or Unix pipes.

3 A protocol for interactive proof

The protocol for directing proof used by PG/Kit is known as **PGIP**, for *Proof General Interactive Proof* [6]. It arose by examining and clarifying the communications used in the existing Proof General system, and early ideas were described in unpublished notes a few years ago [3, 4]. Since then, as we developed prototype systems following the ideas, the protocol has been revised to encompass graphical front-ends and a transparent markup scheme for proof scripts [6, 7].

The syntax of PGIP messages is defined by an XML schema written in RELAX NG [21]. 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.[4] Every message is wrapped in a <pgip> packet which uniquely identifies its origin and contains a sequence number and possibly a referent identifier and sequence number. In order to define the message exchange protocol, we distinguish several *kinds* of messages. The most important ones are:

- *Display commands* are sent from the display to the broker, and correspond to user interaction, such as start a prover, load a file <loadparsefile>, edit this command <editcmd>, or others (<setcmdstatus>).
- *Prover commands* are sent to the prover, and may affect the internal (proof-relevant) state of the prover. The broker has an abstract view of the internal state of the prover which behaves according to a model described in Sect. 3.2.

[4] Another possibility would be to use the tags of MathML [28], but PGML is designed to be easier to support for existing systems.

– *Display messages* are sent from the prover or broker, and contain output directed to the user, such as <`normalresponse`>, <`errorresponse`> or <`ready`>. A display model gives hints where and how the messages are displayed: in a status line, a window of their own, or a modal dialog box.
– *Configuration messages*, used for initially setting up components. For example, a prover component sends a configuration message which describes some elements of its concrete syntax, and preference settings available to the user; or it specifies which icons to use in a graphical interface.

Other message kinds include system inspection and control commands, and meta data sent from the prover, for example, dependency information between loaded files and proven theorems.

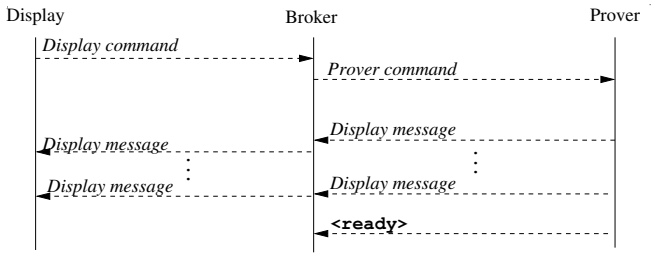

Fig. 3. Message exchange in the PGIP protocol.

Fig. 3 shows a schematic message exchange. 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 the display 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). In the default asynchronous message exchange between prover and broker (corresponding to a command that may cause a proof attempt), the prover will send several responses, eventually followed by a <`ready`> message, which signals availability of the prover to the broker.

On top of this exchange mechanism, interactive proof proceeds in a cycle of edit-parse-prove messages. The user enters a command via the display; it gets parsed and evaluated, possibly giving a new prover state. Repeating this builds up a sequence of prover commands called a *proof script* inside the broker.

3.1 Proof scripts in PGIP

Proof scripts are the central artefact of the system. Provers usually just check proof scripts to guarantee their correctness, but do not construct them, relying on external tools.

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 structure needed for the interface. For example, Fig. 4 shows the PGIP representation of the example proof script from Fig. 1 with the structural markup, and including a PGML <sym> symbol element (we omit the symbol markup on EX for brevity). Notice the named and unnamed <opengoal> elements, and the indentation structure introduced by <openblock> and <closeblock>.[5]

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

Fig. 4. A proof script in Isabelle/Isar, marked up in PGIP.

Proof scripts consist of prover commands, but not all prover commands appear in a proof script; we distinguish between *proper* commands which can appear and *improper* commands which should not.

Proper commands are sent to the prover in plain text, so the prover can interpret them as it would do ordinarily when reading a file. Although the broker does not know how to generate the specific concrete syntax to build up proper commands, it is possible to give an <operationsconfig> configuration message which provides a prover-specific set of *prover types* and *prover operations* that may be used to build up commands. The operations are defined in terms of textual substitution. A simple example for Isar is the operation which takes an identifier *id* and a string *tm* standing for a term, and produces the open-goal command **lemma** *id* `: "`*tm*`"`. It is optional for a prover to give an operations configuration, but if it does, additional interface features can be supplied (especially for graphical interfaces, see Sect. 5.3).

[5] One may wonder why <openblock> and <closeblock> are separate and distinct elements; we do not use a single <block> 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.

The improper commands are only used for controlling the prover's state, and should not appear in the proof script being developed; examples are the three italicised undo commands appearing in Fig. 5 below. Improper commands are not treated as markup, so the prover must interpret these directly.

3.2 Modelling the prover state

PG/Kit assumes an abstract model of incremental interactive proof development, where we suppose there are four fundamental states occupied by the prover, with transitions between the states triggered by both proper and improper prover commands. Fig. 5 shows the states, and the commands to change between them. The four states illustrated are:

1. the *top level* state where nothing is open yet;
2. the *file open* state where a file is currently being processed;
3. the *theory open* state where a theory is being built;
4. the *proof open* state where a proof is currently in progress.

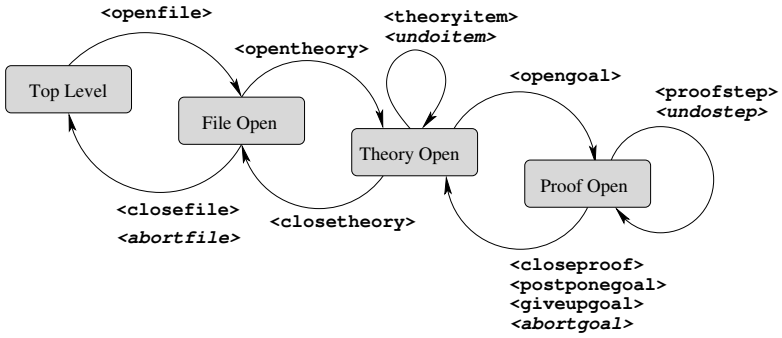

Fig. 5. Proof states during development.

These fundamental states give rise to a hierarchy of named items: The top level may contain a number of files. A file contains a proof script, structured into theories (roughly equivalent to modules), and theories in turn may contain theory items (declarations etc.) and proofs consisting of proofsteps.

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. In the theory state, for example, we may issue *theory steps* which add declarations or definitions to the theory, or we may undo the additions. In the proof state, we can issue *proof steps* and undo these steps, or complete the current proof attempt in a number of ways.

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.

3.3 The edit-parse-prove cycle

The markup on a proof script makes the structure of the proof script explicit, and splits the source code into several text spans containing a prover command each (see Fig. 4). Each of these commands has a status which ranges over five possible values (see Fig. 6). A text 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. Displays can either make the outdate step explicit, requiring the user first to outdate the text range manually, or they can perform the outdate behind the scenes; in any case, all commands depending on the edited command will be outdated as well.

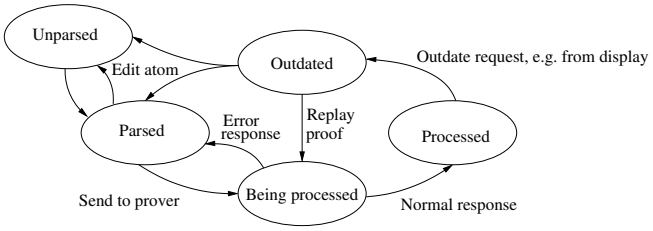

Fig. 6. Command state transitions.

The different transitions between the commands are a refinement of the script management as implemented by Proof General, which is based on a simple linear dependency model: every line potentially depends on all lines that come before. By splitting the text into commands, we can have a more fine-grained dependency analysis (if the prover reports the necessary dependency information), where to process a command we only need to process those commands which are really needed. If the prover does not provide the necessary dependency information, the broker automatically assumes linear dependency.

To demonstrate the edit-parse-prove cycle in action, we consider the message exchange in a typical situation: the user requests a file to be loaded, then edits a part of the text, and finally runs the proof. Fig. 7 shows the resulting messages being sent between display, broker and prover. Note that the proof is "run" by

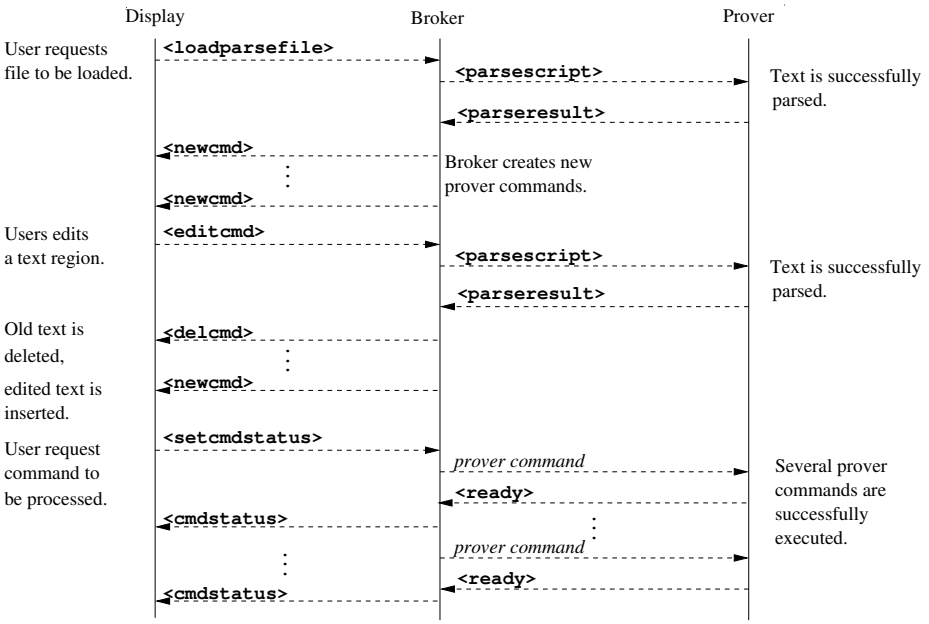

Fig. 7. The edit-parse-prove cycle in a typical situation.

requesting a command be processed, which can cause a lot of other commands to be processed first. If an error occurs at some point in this scenario, the prover sends an <errorresponse> and the broker flushes all outstanding requests. If the error occurs during the parsing, it will insert the corresponding text as an <unparsed> element into the proof script, to allow the user to edit it later.

4 Brokering electronic proof

The broker is the central middleware component of the PG/Kit framework. In general, the broker gathers input from the displays, sends prover commands to the provers, handles the responses and does the house-keeping, i.e. keeps track of the files and the commands, their respective status and the dependencies between them, as provided by the prover. Using this dependency information, it can translate abstract display commands such as <setcmdstatus> into a series of prover commands.

Provers and displays are handled uniformly on the lower level as *components*, but they differ in their communication pattern: prover commands are sent to one specific prover, whereas display messages are broadcast to all connected displays. The displays are stateless and straightforward to handle, but for each prover the broker needs to model the state of the prover according to the abstract state model from Sect. 3.2. For each prover, the broker keeps a queue of all pending

prover commands, sending the next one only once it has received a <ready> message from the prover. If the prover has sent an <errorreponse> before, the queue of pending messages is cleared, as it makes little sense to continue.

The broker also handles the parsing: it sends the parsing request to the prover, and extracts the new commands from the answer. While doing so it checks that the parsing result returned by the prover satisfies the invariant that when we strip the markup, we get back the original proof script; if it fails this invariant, it inserts the dropped text.[6]

The broker is implemented in Haskell. It is multi-threaded: there is one main thread keeping the broker state, and one reader and writer thread for each connected component. To model PGIP messages faithfully and in a type-safe way, we use HaXML [29]. From a given RELAX NG schema, HaXML generates a series of Haskell datatypes, one for each element, along with functions to read and write XML. The advantage of this approach is that the type security given by the schema extends right into the broker, making it impossible to send messages containing invalid XML, and detecting the reception of invalid XML immediately; a very valuable property for such a central piece of middleware.[7]

Components can either be started as a child process and communicate via their standard input and output, they can start the broker and communicate via the broker's standard IO, or they can connect to the broker on a socket. Particular attention needs be paid to the ability to *interrupt* a running prover: when running a prover as subprocess, we do not send the PGIP <interruptprover> message, but instead send a Posix signal. Over a socket, we either run a wrapper on the other side, which starts the actual prover as a subprocess, and translates <interruptprover> messages into signals, or we transmit <interruptprover> messages over a separate, out-of-band channel.

5 Display components

The display components provide the front-ends with which the user interacts. These may vary in complexity from simple web-based displays which offer restricted user interaction, up to fully-fledged environments which have sophisticated support for proof text editing or graphical manipulation of theorem prover objects. We briefly introduce three displays currently available: an Emacs display, an Eclipse plugin, and a graphical interface.

5.1 Emacs Proof General revisited

The Emacs display for PG/Kit will eventually replace the present Proof General system. By moving complex functionality into the broker, the Elisp in Emacs can

[6] This simplifies parsing on the prover side, e.g. if whitespace is collapsed during lexing.

[7] However, we did hit a spot of bother with handling of white space. The deceptively simple rule of "If it is not markup, it's data" [8, Sect. 2.10] means that when parsing a document, white space is treated differently depending on if it occurs in element content or mixed content.

be greatly simplified, greatly increasing maintainability. The Emacs display may be somewhat limited in facilities, but it has the advantage of greater portability, including functioning in a plain terminal, and could also serve as a test bed for the broker initially.

Emacs has a built-in notion of text region which can have special properties attached, called "spans". Spans are used to directly capture the commands described by the broker. Emacs keeps a record of which spans have been altered, and automatically sends requests to the broker to reparse them, either when the file is saved, or during editor idle time. Additionally each span provides a context sensitive menu to adjust its state according to the diagram in Fig. 6. Spans which are in the "being processed" state cannot be edited, and there is customizable protection against editing those which are in the "processed" state. Compared with the present Emacs interface, this generalises to allow non-sequential dependencies within proof scripts, under control of the broker. However, the same toolbar and navigation metaphor for processing the next step is still possible.

5.2 Eclipse Proof General

Eclipse [23, 11] is an open-source IDE and tool integration platform written in Java. Most prominently it provides a powerful and attractive IDE for Java, but it has a highly open and modular design based on *plugins* and *extension points* that allows almost any aspect of the platform to be customized and extended to new domains. Many plugins are available, supporting other programming languages, profiling and testing tools, graphical modelling facilities, etc.

Eclipse provides the chance for a robust and reliable IDE for formal proof. The primary mode of working remains the editing of proof script files, supported with script management. A screenshot in Fig. 8 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 button triggers proof or undo steps by sending PGIP instructions to the broker.

Exploiting PGIP configuration and script markup, the Eclipse plugin offers further useful features, including hyperlinked indexes of previously processed files, allowing quick access to theorems, definitions, unfinished proofs, and integrated help: by associating named commands such as theorems with a preceding comment, the interface gives tooltip-help for later references to that item. Completion of identifiers based on PGIP markup and messages from the prover containing identifier tables is another useful feature that will be supported.

Like the revised Emacs interface described above, the Eclipse editor window must deal with managing information gleaned from the structure of the script, while allowing free form text edits – which can wreak arbitrary changes to the structure. This is solved in the Eclipse plugin 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. In the second phase, the PGIP

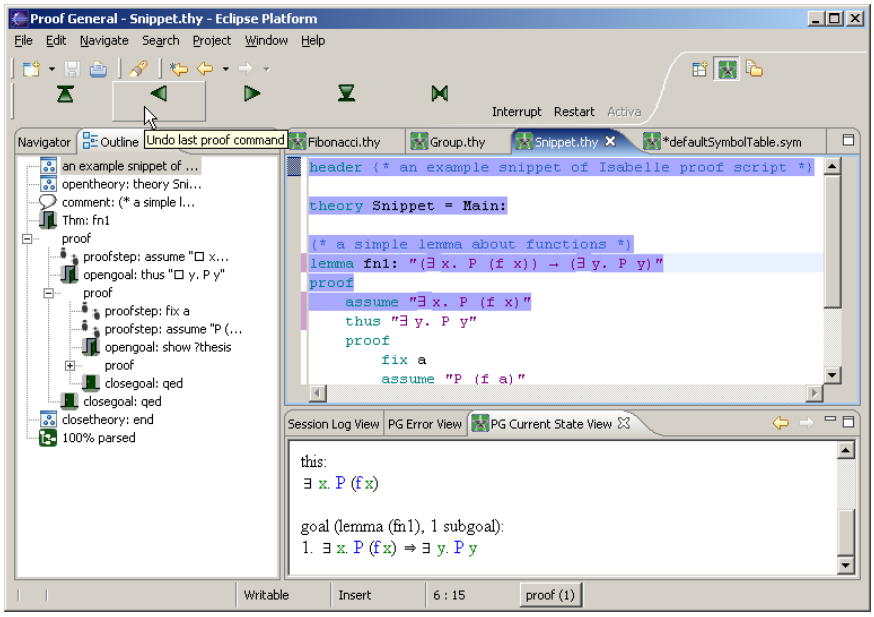

Fig. 8. Eclipse Proof General Display

mark-up structure is obtained by calling the broker with <`editcmd`> messages. This can either happen in a low-priority background thread, or with specific user commands (such as evaluating a script).

The fast lexer is configured for each prover by a PGIP configuration command called <`proverinfo`>. This configuration command informs the display about the keywords in the prover's language, and can also provides tool-tip help for commands (for example, to remind the user of the command syntax).

The Eclipse PG 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. At the moment, providing support for a new language in Eclipse is not as straightforward as one might hope; to provide functionality similar to that offered for Java it is necessary to copy pieces of code from there. Supporting mathematical symbols and rich markup on source text is also a challenge, because the standard editing widget allows only a single font to be used in the editor panes. Thus, symbol support depends on access to a suitably rich unicode font.

5.3 A Theorem Proving Desktop

A third display is a theorem proving desktop built in the spirit of IsaWin [16]. IsaWin provides a more abstract, less syntax-oriented interface to Isabelle (and related provers), based on direct manipulation and supported by the visual

metaphor of a *notepad* [16]. All objects of interest, such as proofs, theorems, tactics, sets of rewriting rules etc. are visualised by icons on the notepad, and manipulated using mouse gestures. The icon is given by the type of the object, which determines the available operations. Complex objects such as proofs can be manipulated by opening them in a separate window.

PGIP supports this style of GUI with the <operationsconfig> specification, which describes types and operations as mentioned in Sect. 3.1, and can also include icons and hints for selecting operations. For example, if the operations configuration specifies types for theorems and rewriting sets, it can specify that dropping a theorem onto a rewriting set adds it to the set, whereas dropping a rewriting set onto the current proof performs a rewrite operation on the proof state. The operations scheme also allows for user input, so the display can ask for information which then gets fed into the operation (e.g., the display may ask which subgoal to rewrite). Moreover, it allows for context-sensitive generation of menus by interacting with the prover to pass term position information.

We have implemented a prototypical graphical display engine called PGWin for an earlier version of PGIP [7], where display commands and messages were not represented in XML. It is currently adapted to the new version of PGIP, and made into a separate PGIP component.

6 Conclusions

The Proof General Kit is a framework for connecting interactive proof systems to interface tools. This paper has provided an overview; elsewhere we provide full details including the XML schemas and protocol descriptions [6]. Ultimately, we hope that implementers of existing proof systems will have a compelling reason to add PGIP support to their systems to access powerful front-ends, and we hope that implementers of new systems will now have a clear model to follow to gain interface support with minimal effort.

At the time of writing, the broker component, Emacs display and Eclipse plugin are near to beta release. These have been developed for the upcoming 2005 version of Isabelle, to which support for PGIP has been added by the first author. While straightforward in principle, supporting PGIP in Isabelle/Isar turned out to be harder than expected because of 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).

PG/Kit is unique in proposing a specific framework customised for interactive proof, although there is related work in different settings. Perhaps most related is the MathWeb project, which provides a standardised XML-RPC interface to a range of automated provers, using the XML format OMDoc as an exchange language [14]. OMDoc explains the semantical content of logical terms, which goes beyond the PG/Kit. It would be intriguing to consider an extension of our protocols to allow OMDoc exchange, although of course this would entail

adding OMDoc support for each of the underlying provers. In addition to MathWeb, there are several other efforts to publish formalised mathematical content, including Mizar [26], HELM [1], MoWGLI [17] and Logosphere [22]. Other frameworks in theorem proving include Prosper [9], which connects several automatic provers with an LCF prover ensuring logical consistency, and ETI [24], which allows tools to be combined in one platform, but they are both more ambitious, wider in scope and hence less specific than PG/Kit.

There are many possible lines for future development. First, we want to use the framework to investigate foundations for *proof engineering*, exploring the analogy with software engineering to support notions of refactoring, code browsing, etc. This would all ideally be supported within the Eclipse plugin. We can also go beyond program development, exploiting the interactivity of the framework to allow e.g. interactive proof planning to construct proof scripts [10].

Another promising direction lies in providing extra language layers or enhancements in a generic way. For example, we could provide literate style markup or a document-driven development methodology [25]. We can also use the broker itself to control proof construction and search: PGIP contains almost enough functionality to support a tactic language at a generic level, in fact.

We welcome contact from researchers interested in working with us on future directions or in connecting their systems to PG/Kit.

References

1. 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.
2. 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.
3. D. Aspinall. Protocols for interactive e-proof, 2000. Short talk at TPHOLS 2000.
4. D. Aspinall. Proof General Kit. White paper, 2002.
5. D. Aspinall, H. Goguen, T. Kleymann, and D. Sequeira. Proof General system documentation, 1999-2004. Available at http://proofgeneral.inf.ed.ac.uk/doc.
6. D. Aspinall and C. Lüth. Commentary on PGIP. Available from http://proofgeneral.inf.ed.ac.uk/kit/, September 2003.
7. D. Aspinall and C. Lüth. Proof General meets IsaWin. In D. Aspinall and C. Lüth, editors, *User Interfaces for Theorem Provers UITP'03*, volume 103 of *Electronic Notes in Theoretical Computer Science*, 2003.
8. T. W. Consortium. Extensible markup language (XML). Technical report, W3C Recommendation, 2004.
9. L. A. Dennis, G. Collins, M. Norrish, R. J. Boulton, K. Slind, and T. F. Melham. The PROSPER toolkit. *International Journal on Software Tools for Technology Transfer*, 4(2):189–210, 2003.
10. L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics 2004 (TPHOLs'04)*, volume 3223 of *Lecture Notes in Computer Science*. Springer, 2004.

11. Eclipse homepage: eclipse.org. See http://www.eclipse.org.
12. H. Geuvers, F. Wiedijk, and J. Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In *TYPES*, pages 96–111, 2000.
13. E. B. Johnsen and C. Lüth. Theorem reuse by proof term transformation. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *International Conference on Theorem Proving in Higher-Order Logics TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 152–167. Springer, Sept. 2004.
14. M. Kohlhase. OMdoc: Towards an OpenMath representation of mathematical documents. Available from http://www.mathweb.org/omdoc/.
15. D. Kroening. *Formal Verification of Pipelined Microprocessors*, pages 71–80. Gesellschaft fuer Informatik, 2002.
16. C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2):167– 189, Mar. 1999.
17. MoWGLI. mathematics on the web: Get it right by logics and interfaces. http://www.mowgli.cs.unibo.it/.
18. O. Pons. *Conception et réalisation d'outils d'aider au d'eveloppement de grosses théories dans les systeèmes de preuve interactifs*. PhD thesis, Conservatoir National des Arts et Métiers, 1999.
19. O. Pons, Y. Bertot, and L. Rideau. Notions of dependency in proof assistants. In *Proc. User Interfaces for Theorem Provers, UITP'98*, 1998.
20. Proof general home page. See http://proofgeneral.inf.ed.ac.uk/.
21. RELAX NG xml schema language, 2003. Home page at http://www.relaxng.org/.
22. C. Schürmann, F. Pfenning, M. Kohlhase, N. Shankar, and S. Owre. Logosphere. a formal digital library. http://www.logosphere.org/, 2003.
23. S. Shavor, J. D'Anjou, S. Fairborther, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2003.
24. B. Steffen, T. Margaria, and V. Braun. The Electronic Tool Integration platform: Concepts and design. *International Journal on Software Tools for Technology Transfer*, 1:9– 30, 1997.
25. L. Théry. Colouring proofs: a lightweight approach to adding formal structure to proofs. In D. Aspinall and C. Lüth, editors, *User Interfaces for Theorem Provers UITP'03*, volume 103 of *Electronic Notes in Theoretical Computer Science*, 2003.
26. A. Trybulec et al. The mizar project, 1973. See web page hosted at http://mizar.org, University of Bialystok, Poland.
27. 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.
28. Mathematical markup language (MathML). W3C Recommendation, 1999.
29. M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming ICFP'99*, pages 148– 159. ACM Press, 1999.