Parsing, Editing, Proving: The PGIP Display Protocol

David Aspinall [1]

LFCS, School of Informatics, University of Edinburgh, U.K.

Christoph Lüth [2]

Department of Mathematics and Computer Science, Universität Bremen

Daniel Winterstein [3]

LFCS, School of Informatics, University of Edinburgh, U.K.

Abstract

This paper describes how proof texts are constructed and edited in the Proof General Kit framework. Proof texts are the central object of development within our framework and we want to allow flexible ways to construct them, both explicitly via text editing and implicitly by graphical manipulation or meta-manipulation. To this end, the framework allows for user-oriented *display* components, connected to provers via a central broker component. The display components and the broker exchange messages in a format specified by the *PGIP display protocol*, which facilitates parsing, editing and proving of proof texts.

The design of this part of the framework is new; the remainder of the framework, which connects the *prover* components to the *broker*, is based more closely on refining work of the previous Proof General project, and was described in [4].

1 Introduction

The *Proof General Kit* (PG Kit) is a software framework for conducting interactive proof. Its design grew out of the predecessor *Proof General* project, which constructed a generic interface to numerous interactive theorem provers

[1] WWW: `http://homepages.inf.ed.ac.uk/da`
[2] WWW: `http://www.informatik.uni-bremen.de/~cxl`
[3] daniel.winterstein@gmail.com

in a somewhat piecemeal, ad hoc (but nonetheless successful) approach. The principle disadvantage of this system is that it can be confusing to connect to new provers and difficult to modify for existing provers; moreover the implementation, principally in Emacs Lisp, became unwieldy to maintain.

The basic idea of PG Kit is that the user interface can be greatly improved by removing as much knowledge about the theorem prover from it as possible, and the implementation of the theorem prover can be greatly simplified by not worrying about a user interface. To connect the two, PG Kit defines a framework architecture together with a communication protocol dubbed PGIP. In this framework, provers communicate with visualisation and editing components called *displays* via a central co-ordinating component called the *broker*. The broker encapsulates an abstraction of the prover's state and can be implemented robustly in a strongly typed high-level language, while the front ends can remain largely simple, and be implemented in a variety of appropriate languages. The user interacts with the front ends, allowing the creation and execution of proofs in a controlled way, perhaps directly as textual scripts (e.g. in Emacs) or indirectly via a graphical desktop metaphor (e.g. as in IsaWin [8,10]).

Initial versions of PGIP focused on the interaction between the prover and an interface, building on the patterns used in the predecessor. In this paper, we also consider the displays as separate components within the architecture, extending PGIP to cover the interaction between the broker and one or more displays. This allows a more modular framework and explains the abstractions that must be implemented within displays, whether they be textual or graphical. To do this, we must first define a model for the target proof text (which is constructed internally by the broker), and how it is parsed, edited and executed. Then we consider how displays may interact in this process.

This paper is structured as follows. We start with a brief run-down of the PG Kit system architecture to set the general picture, then describe the proof text model and display abstractions in some detail. We then see what the broker has to do to implement the display model, and we introduce the displays already implemented or currently in development. We conclude, mentioning related and future work.

2 The PG Kit System Architecture

PG Kit defines a component infrastructure, based around PGIP, which specifies the syntax of messages exchanged between components, as well as the protocol: the permitted sequences and effect of message exchanges. The syntax of messages is given by an XML schema. The protocol for message exchanges is given by an informal specification [3] and enforced dynamically by our canonical implementation of the central broker component (see Section 4).

2.1 The PG Kit framework architecture

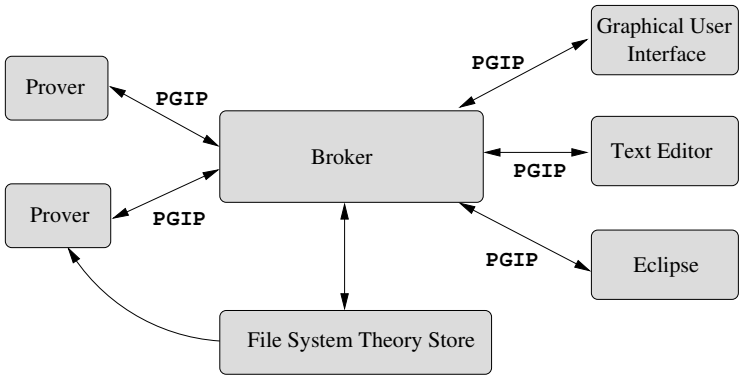

Fig. 1. PG Kit Framework architecture

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. 1.

The components communicate using messages in the PGIP protocol, outlined in the next section. The general control flow is that a user's action causes a message 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. Messages are sent over channels, typically sockets or Unix pipes.

2.2 The PGIP protocol

The protocol for directing proof used by PG Kit is known as *PGIP*, for *Proof General Interactive Proof* [3]. 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 [1,2]. 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 [3,4].

The syntax of PGIP messages is defined by an XML schema written in RELAX NG [12]. 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 between the display and 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 maintains a simple abstract view of the internal state of the prover which characterises the native history and undo mechanisms available.
- *Output messages* are sent from the prover or broker, and contain output directed to the user, such as <`normalresponse`> and <`errorresponse`>. Output messages can be supplied with hints about where and how the contents should be 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 can send a configuration message which describes some elements of its concrete syntax, and preference settings available to the user; it can also specify which icons to use in a graphical interface.

Other message kinds include system inspection and control commands, and metadata sent from the prover, for example, names of available items (defined types, constants, theorems, etc) and dependency information between them. Displays may render the metadata to give additional information to the user, or may use it to assist the user in constructing proof scripts.

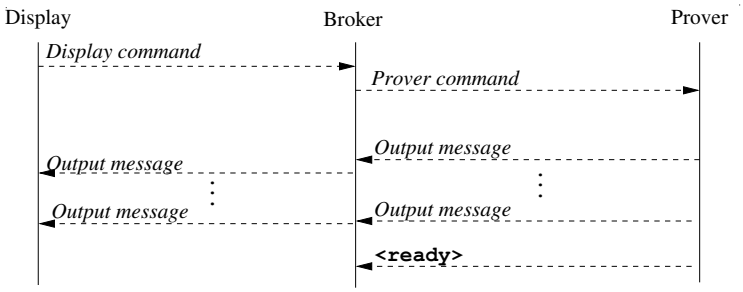

Fig. 2. Message exchange in the PGIP protocol.

Fig. 2 shows a schematic message exchange. A command sent from the display causes the broker to issue a command to the prover; the prover responds with several output messages which are relayed to the display. The output messages are terminated with an <`ready`> response from the prover.

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 then evaluated, possibly giving a new prover state. Repeating this builds up a sequence of prover commands called a *proof script* inside the broker. Proof scripts are text files in a format native to a particular theorem prover; it is a central design principle of the PG Kit framework to construct scripts in the prover's native language, rather than to prescribe a uniform language for all provers.

In the rest of this paper, we focus on the message exchange between bro-

ker and display components, the *display commands*. The message exchange between the broker and the prover, including the *prover commands* and the internal modelling of the prover's state, has been described in an earlier paper [4]. Finally, *output messages* are sent from the prover or broker and relayed directly to interested displays. The communication between broker and displays is always asynchronous (single request, non-waiting multiple response): the display may send a command, and the broker may send several responses later.

3 The PGIP Script and Display Model

The display part of PGIP has been designed with the aim of keeping the implementation of a display as simple as possible, but still allow the construction of a diverse range of displays, ranging from simple text editors to sophisticated graphical theorem proving desktops. This means that the broker sends rather a lot of information to displays, leaving it to the display to filter out information it cannot process. The display should not have to keep an extended internal state, instead processing each message as it arrives, although some bookkeeping is inevitable as we will see below.

3.1 Proof scripts in PGIP

Proof scripts (stored in text files) are the central artefact of the system. Theorem provers check proof scripts to guarantee their correctness, but usually do not assist in constructing them, relying on external tools — unfortunately often, users armed with a nothing more than a primitive text editor.

The basic principle for representing proof scripts in PGIP is to use the prover's native language and *mark up* the content with PGIP *prover commands* which expose some structure of the proof script which is needed for the interface. For example, Fig. 3 shows the PGIP markup on an example Isabelle/Isar proof text with the structural PGIP markup.

The text starts with an <`opengoal`> proof command, which has an attribute to name the lemma being proved. The text ends with a <`closegoal`> proof command. Other proof commands are not further interpreted, and decorated simply with a <`proofstep`> element. A block structure on the script (reflecting the indentation) is introduced by <`openblock`> and <`closeblock`> elements in between the proof commands. One may wonder why <`openblock`> and <`closeblock`> are separate and distinct elements; the reason not to use a single <`block`> element to enclose the block structure is that we need to be able to incrementally parse and evaluate text, which necessitates handling ill-structured fragments of a block (with unmatched opening or closing elements).

To allow for symbol characters, PGIP has a simple sub-schema called

```
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
```

```
<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. 3. A proof script in Isabelle/Isar, and its marked-up form in PGIP.

PGML (for *Proof General Markup Language*). The use of PGML is demonstrated in Fig. 3 with the <sym> symbol element which names a symbol alternative for an ASCII sequence used in the proof script. (The EX symbol could be marked up similarly, we omit this for brevity).

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. The improper commands are used for controlling the prover's state (for example, issuing undo steps; see [4] for further details).

Proof scripts enter the framework in the following three ways:

(i) they may be read verbatim from a file,

(ii) they can be entered piecemeal (as text) by the user, or

(iii) they can be generated with configurable (prover-specific) *proof building operations*.

The first two cases are similar, as they just enter plain text into the system from different sources. The third is a schematic way to produce text, inter-

acting with the use of the PGIP structural markup. We will look at that in detail in Sect. 3.4.

3.2 The edit-parse-prove cycle

The PGIP markup on a proof script reveals its structure of the proof script explicitly, and moreover splits the source code into several non-overlapping text spans containing a prover command each. In Fig. 3, there are 11 commands. When the proof script is loaded into the system, each command may be in one of five possible states:

- *unparsed*
- *parsed*
- *being processed*
- *processed*
- *outdated*

The transitions between these states are shown in Fig. 4.

A span of text starts off as *unparsed*, and after parsing becomes one or more freshly *parsed* prover commands. Thus, an unparsed text may contain more than one actual command.

The broker does not lock the command while it is being parsed, under the assumption that parsing is sufficiently quick. Thus, the user could actually edit a piece of text while it is being parsed; in that case, the broker would discard the result of the parse along with the old, edited command. This means that the display should not attempt to send each character as it is being typed, but rather group together changes, and send edit messages in larger chunks. Two possible strategies here are either to send edit messages only on user request, or in idle time, when the user is has not been editing text for a while.

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 *parsed*. To successfully process a command we will need to process all commands it depends on, which causes them to change state too. Typically there may be a queue of commands in the *being processed* state.

The display model does not require users to explicitly undo commands. Instead they just require commands to become outdated. This is typically to be able to change a command, because 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

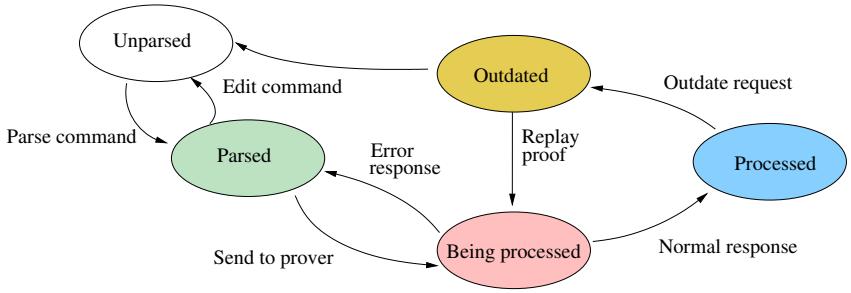

Fig. 4. Command state transitions.

edited command will be outdated as well.

Other changes in the prover state may cause commands to be outdated without the user directly requesting it (for example, restarting the theorem prover engine); similarly this may happen for processing commands (for example, by asking the theorem prover to directly read a file). The display has to be careful to revert the contents of command text in case user editing collides with state changes like this.

Note that the dependencies are completely hidden from the display, so the user can request to outdate or process any command; it is up to the broker to calculate all other necessary state changes. This entails that requesting to outdate or process one particular command can result in a long series state change messages from the broker. However, the broker makes sure that outdates are performed as quickly as possible so as to conform with users' expectations.

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.

3.3 An example interaction.

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. 5 shows the resulting messages being sent between display, broker and prover. Note that the proof is "run" by 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.

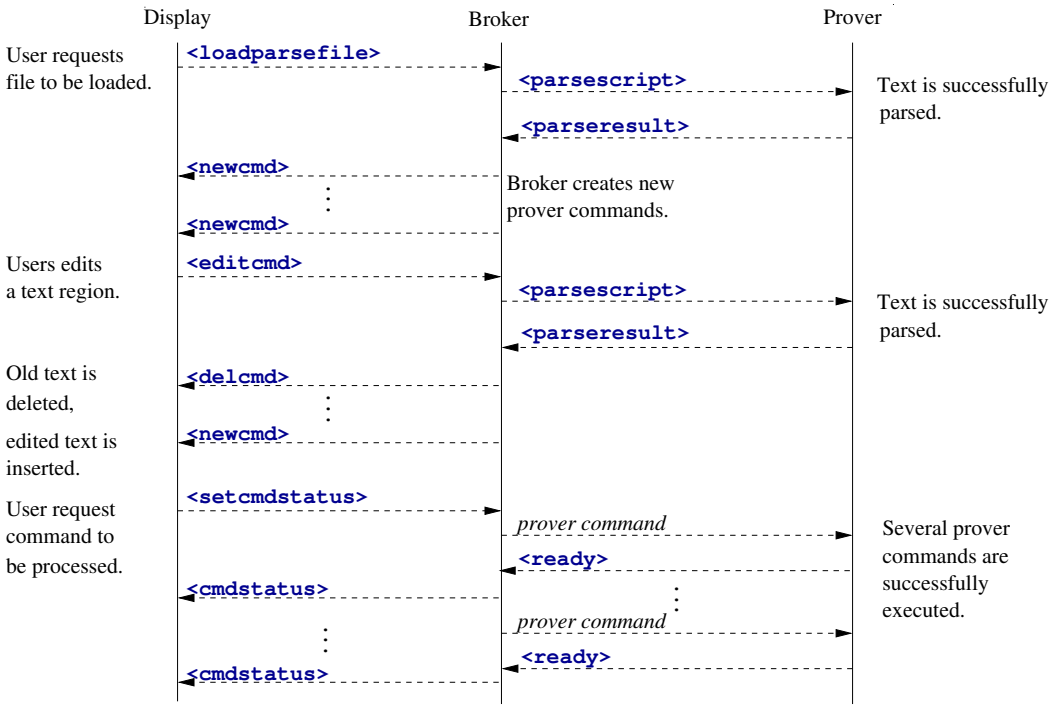

Fig. 5. The edit-parse-prove cycle in a typical situation.

Fig. 5 also shows that in general it is the prover's responsibility to parse the text into commands. [4] The framework does not include a state for commands which are being parsed: we expect parsing to be fast, and the parsing service always available. This can either be implemented in a separate parsing thread inside the theorem prover, or one may use a standalone parsing component which filters out PGIP parse requests and answers them, passing on all other commands to the prover. This flexibility is one of the advantages of the loosely coupled system architecture.

[4] Indeed, the result of parsing may in principle depend on the state of the proof engine, for example, for proof languages whose syntax is dynamically extensible. For Proof General we would like to discourage this language feature (or at least, restrict the points at which it can occur), to retain the robustness of being able to parse fragments of proof text at any point. This remark applies to the syntax of commands (in Isabelle/Isar, the so-called *outer* syntax), it does not apply to logical expressions which PG Kit does not attempt to interpret at this point.

Proper commands from proof scripts 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 *proof building operations* that may be used to build up commands.

Proof building 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 command **lemma** *id* : "*tm*". This

It is optional (but encouraged) for a prover to give an operations configuration. It does so via operation configuration messages, which specify a set of types for the prover, and proof operations mapping zero or more source types to a target type or a proof command. Certain types are fixed and assumed by the framework, including types for theories, theorems, comments, and files. Types can have a hierarchical structure, so for example, theories can contain theorems, and theorems might contain attributes (e.g. to flag some theorems as introduction rules or for use in simplification). Types can be provided with graphical icons for the displays to show.

In a textual interface, proof building operations can provide a facility for input using templates, a loose form of structured editing. In a graphical interface they allow for fragments of text to be represented graphically, and manipulated using interface elements such as menus or gestures such as drag&drop (see Sect. 5.2).

4 The Broker

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.

The broker must keep track of the state of the prover, and the state of the proof script. (Note that we can have more than one display, but each proof script has one global state. State changes made on one display propagate to the other displays.)

Within the broker, each prover has a *focus*, an abstraction of its current state; this is essentially the last command it has processed (which leads to the current state). In a linear view of script management, the focus separates the

beginning part of a proof script which has been processed from the remaining part which is still to be processed. If a prover supports non-linear dependencies, processed and outdated regions in the text may be interleaved. In both cases, we have that:

(i) if the focus is set, then the corresponding command has been processed;

(ii) everything depending on the focus is outdated or freshly parsed (but see below);

(iii) everything which the focus depends on is processed;

There is a transitional situation right after a command has been sent to the prover which is an exception to rule (ii) above: the state of the command will be set to *being processed*, but the focus has not been moved yet. Once the prover returns successful completion of this command, the focus moves forward; if it returns an error, the focus stays put. This is part of the protocol specification: errors must not change the prover state.

When the broker receives an outdate request for a particular command, it just outdates the required command, and everything depending on it. It does not send anything to the prover just yet, as users would expect an outdate operation to be performed quickly. (In fact, the user may not even do the outdate; it may be that the user just starts editing, and the display sends the outdate request on behalf of the user to allow him to start editing.) Only when the broker receives a request to process a command will it start to move the focus (if necessary) by means of the <`undostep`> improper proof command.

The broker handles parsing of text. 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 the markup is stripped, we get back the original proof script; if it fails this invariant, it inserts the dropped text. [5]

As well as the focus within the current proof script and a record of the status of individual commands in that script, the broker has to keep track of (other) files containing other scripts. There are two possibilities for file handling: either a file is processed incrementally via the interface, or it is processed directly by the theorem prover. In the former case it may be possible to undo into an arbitrary position within the file, but in the latter case it is only possible to undo the whole file. Similarly to commands, any dependent files are also undone. The broker allows the display to reflect the status of a file to the interface in case the user wishes to examine it.

[5] This can simplify parsing on the prover side, e.g. in case whitespace is collapsed during lexing.

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 proof development environments which have sophisticated support for proof text editing or graphical manipulation of theorem prover objects.

In this section we describe two displays for PG Kit: an Emacs-based display and a graphical interface. This demonstrates how the script and display model can cover displays with quite different mode of user interaction — one is textual and can be controlled via craftily cryptic key sequences, the other is all mouse gestures and groovy graphics. A third, and the most substantial display component, is the Proof General plugin for the Eclipse IDE, which is described elsewhere in this volume [17].

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 be 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 also serves as an initial test bed for the broker development.

Emacs has a built-in notion of text region which can have special properties attached, which we call "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. 4. Spans which are in the "being processed" state cannot be edited, and there is customisable 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, so the interface will elicit a very similar user experience.

5.2 A Theorem Proving Desktop

An alternative display is a theorem proving desktop built in the spirit of IsaWin [9]. 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*. 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 *PG-Win* for an earlier version of PGIP [4], 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, concentrating on the mechanisms used for constructing and editing proof scripts within the broker. Elsewhere we provide full details including the XML schemas and protocol descriptions [3]. 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. Alternatively, a standalone parsing PGIP parsing component is currently being developed which can be configured via regular expressions (just like it used to be in Proof General).

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 [7]. 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 [16], MoWGLI [11] and Logosphere [13]. Other frameworks include Prosper [5], which connects several automatic provers with an LCF prover ensuring logical consistency, and ETI [14], 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 ideally be supported within the Eclipse plugin, taking advantage of its existing facilities. We can also go beyond program development, exploiting the interactivity of the framework to allow e.g. interactive proof planning to construct proof scripts [6].

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 [15]. 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, and we are investigating making this extension.

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

References

[1] D. Aspinall. Protocols for interactive e-proof, 2000. Short talk at TPHOLS 2000.

[2] D. Aspinall. Proof General Kit. White paper, 2002.

[3] D. Aspinall and C. Lüth. Commentary on PGIP. Available from `http://proofgeneral.inf.ed.ac.uk/kit/`, September 2003.

[4] 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.

[5] 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.

[6] 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.

[7] M. Kohlhase. OMdoc: Towards an OpenMath representation of mathematical documents. Available from http://www.mathweb.org/omdoc/.

[8] C. Lüth, H. Tej, Kolyang, and B. Krieg-Brückner. TAS and IsaWin: Tools for transformational program developkment and theorem proving. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering FASE'99*, number 1577 in Lecture Notes in Computer Science, pages 239– 243. Springer Verlag, 1999.

[9] 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.

[10] C. Lüth and B. Wolff. More about TAS and IsaWin: Tools for formal program development. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering FASE 2000*, number 1783 in Lecture Notes in Computer Science, pages 367– 370. Springer Verlag, 2000.

[11] MoWGLI. Mathematics on the web: Get it right by logics and interfaces. http://www.mowgli.cs.unibo.it/.

[12] RELAX NG XML schema language, 2003. Home page at http://www.relaxng.org/.

[13] C. Schürmann, F. Pfenning, M. Kohlhase, N. Shankar, and S. Owre. Logosphere. a formal digital library. http://www.logosphere.org/, 2003.

[14] 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.

[15] 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.

[16] A. Trybulec et al. The mizar project, 1973. See web page hosted at http://mizar.org, University of Bialystok, Poland.

[17] D. Winterstein, D. Aspinall, and C. Lüth. Proof General/Eclipse: A generic interface for interactive proof. In *User Interfaces for Theorem Provers UITP'05*, 2005.