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] Deutsches Forschungszentrum für künstliche Intelligenz (DFKI), Bremen, Germany

Abstract. This paper introduces *Proof General Kit*, a framework for software components tailored to interactive proof development. The goal of the framework is to enable flexible environments for managing formal proofs across their life-cycle: creation, maintenance and exploitation. The framework connects together different kinds of component, exchanging messages using a common communication infrastructure and protocol called *PGIP*. The main channel connects *provers* to *displays*. Provers are the back-end interactive proof engines and displays are components for interacting with the user, allowing browsing or editing of proofs. At the core of the framework is a *broker* middleware component which manages proof-in-progress and mediates between components.

1 Introducing Proof General Kit

The use of interactive machine proof is becoming more widespread, and larger and more complex formalisations are being undertaken in application areas such as hardware or software verification, and formalisation of mathematics, even up to formalising deep proofs of recently established results. Examples of interactive provers include general purpose provers such as Mizar, HOL, Isabelle, PVS, Coq, ACL2, or NuPrl, and domain-specific provers such as the Forte system [19]. Of course, this is to name just a few systems: Freek Wiedijk's database [26] currently lists almost 300 systems for doing mathematics on computer! Although many of these may be classed as small-scale experiments or obsolete, it is natural to expect researchers to continue investigating new logical foundations, and to build domain-specific provers for new application areas.

For interactive provers such as those mentioned, 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*. About 100 systems on Wiedijk's list are based on textual proof script input. 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 proofs, the proof scripts are themselves large: by now there are individual developments and mathematical libraries which reach hundreds of thousands of lines of code and represent many person-years of work.

Yet, compared with the facilities available to the modern programmer, the facilities for developing and maintaining formal proofs are lamentably poor, in general.[3] Modern software development uses sophisticated Integrated Develop-

[3] We note a few exceptions in a related work section in the conclusions.

ment Environments (IDEs), which support features such as automatic documentation lookup, completion of identifiers, and integration with version control and the build process. Modern knowledge management facilities help further: context-aware search finds related definitions; content assistance mechanisms insert declarations and instantiations; advanced software engineering methods like refactoring help improve design, making large-scale structural changes easy.

One reason why these facilities have not yet been provided for theorem proving is the fragmentation of the community across so many different systems, which dilutes the effort available. We believe the community should invest in shared tools as much as possible, and keep only the underlying logical proof engines as separate, distinct implementations. Thus, we are arguing not just for exchanging and relating mathematical knowledge between systems, provided by formats such as OMDoc, but also for the component-based construction of proof management environments themselves, using a uniform protocol.[4]

In this paper we introduce the *Proof General Kit* (PG Kit for short). This is a framework for proof management, based on the *PGIP* protocol. We believe that PG Kit will provide sophisticated and useful development environments for a whole class of interactive provers, and also be a vehicle for research into the foundations of such environments.

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, Sect. 4 describes the central role of the broker component, and Sect. 5 describes several display components. 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, especially considering that 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] provides evidence that at least some of our aims are feasible. Proof General is a successful generic interface for interactive proof assistants, where a proof script can be sent line-by-line to the prover with the prover responding at each step. It has been adapted to a variety of provers and is in common use for several, most notably Isabelle and Coq.

The central feature is an advanced version of *script management*. To interactively "run" a script like Fig. 1, 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 called the prover's *focus*. Script

[4] Very roughly: OMDoc is to PGIP as HTML is to HTTP.

```
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 short example proof script in Isabelle/Isar

management divides a proof script into three consecutive regions: a part which has been processed, a part which is currently being processed, and a part which has not yet been processed. Proof General displays this partitioning to the user by colouring processed text blue and busy (being-processed) text pink. Editing is prevented in the coloured region to ensure synchronisation with the prover. A toolbar provides buttons for navigating within the proof, moving the focus. The navigation buttons behave identically across numerous different systems, despite behind-the-scenes using rather different system-specific control commands.

Although successful, there are several drawbacks to the present Proof General. Users are required to learn Emacs and tolerate its idiosyncratic UI. Developers must contend with the Emacs Lisp API which is restrictive, often changing, and inconsistent across the many flavours of Emacs. For configuring provers, the instantiation mechanism has become fragile and too complex. This is because Proof General arose by successively generalising a common basis to a growing number of proof systems, with the design goal not to change the systems themselves. But this leaves the interface vulnerable to breakage by even small changes in the prover output format, and it does not itself offer a clear API, relying on regular expression matching of the prover output.

2.2 The framework architecture

Instead of trying to anticipate a range of slightly different behaviours, we propose a uniform protocol and model of proof development which captures behaviour reasonably common to all provers at an abstract level, and ask that each proof system implements that. We 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. We also want to allow connecting to more than one prover at once, to allow easy switching between different developments and systems. We even want to allow connecting other components that provide assistance during the proof process (e.g., for recommendation [13] or proof planning [8]). In the end, what we need is exactly a software framework: a way of connecting together interacting components customised to the domain.

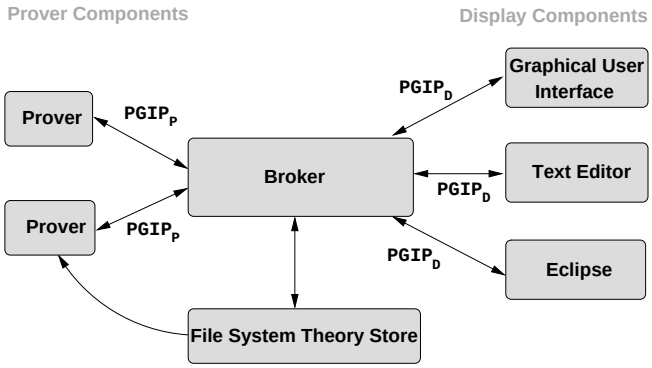

Fig. 2. PG Kit Framework architecture

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. 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 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 Interaction Protocol* [4]. It arose by examining and clarifying the communications used in the existing Proof General system. As we developed prototype systems following the ideas outlined above, the protocol has been revised and extended to encompass graphical front-ends, a document model markup for proof scripts, and authoring extensions [3–5]. PGIP is an abstraction of the communication between provers and interfaces. It allows for prover-specific behaviour and syntax (e.g. in the proof scripts), but specifies an abstract model of behaviour which all provers have to follow.

The syntax of PGIP messages is defined by an XML schema written in RELAX NG [17]. 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. PGIP comprises three sub-protocols, corresponding to the different types of components from Fig. 2:

– The *prover protocol* $\mathrm{PGIP_P}$ defines messages exchanged between provers and the broker. This includes: commands sent to the prover, which correspond

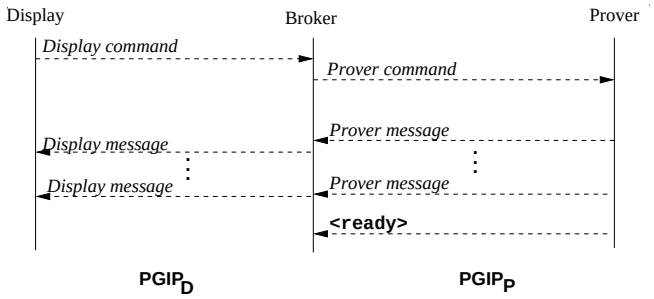

Fig. 3. Message exchange in the PGIP protocol.

to the commands in a conventional proof script and may affect the internal (proof-relevant) state of the prover; messages from the prover in reaction to these commands such as <`normalresponse`>, <`errorresponse`> or <`ready`>, which reflect the internal state; and configuration messages which describe some elements of its concrete syntax, preference settings available to the user, or which icons to use in a graphical interface.
- The *display protocol* $\mathrm{PGIP_D}$ defines messages exchanged between displays and the broker. This includes: display commands sent from the display to the broker, corresponding to user interaction, such as starting a prover, loading a file <`loadparsefile`>, or editing <`editcmd`>; and display messages, which contain output directed to the user, either relayed from the prover, or generated from the broker.
- The *inter-broker protocol* $\mathrm{PGIP_I}$ defines messages exchanged between different brokers, allowing running the prover on a remote machine (see Sec. 4).

The sub-protocols are not disjoint: some prover output (e.g., <`normalresponse`> or <`errorresponse`>) is relayed to the displays, so these messages are part of both $\mathrm{PGIP_D}$ and $\mathrm{PGIP_P}$. The broker analyses messages from the prover, and keeps an abstract view of the internal state of the prover which behaves according to a model described in Sect. 3.2. There is a secondary schema called *PGML*, for *Proof General Markup Language*, used to markup messages from the prover.[5]

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 proof may diverge (e.g. during proof search); it is essential that feedback can be displayed eagerly so the user can take action as soon as possible. The message exchange between the display and the broker is 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 sin-

[5] A standard markup language, e.g., MathML, could be used instead, but PGML is designed for easy support by existing systems by marking up concrete syntax.

```
<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.

gle 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 an edit-parse-prove cycle. The user enters a command via the display; it gets parsed and inserted into the proof script as parsed commands; and eventually it is evaluated, giving a new prover state. Repeating this builds up a sequence of prover commands inside the broker interactively, which form a proof script.

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 (mostly, humans with text editors). The basic principle for representing them in PGIP is to use the prover's native language and *mark up* the content with PGIP commands which explain the proof script structure. Fig. 4 shows the PGIP representation of the example proof script from Fig. 1 with the structural markup, including a PGML <sym> symbol element (we omit other PGML symbols and markup such as <whitespace> for white spaces for brevity). Notice the named and unnamed <opengoal> elements, and the indentation structure introduced by <openblock> and <closeblock>.

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 must 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. The broker does not know how to generate the prover-specific concrete syntax of proper commands; it is usually written directly by the user. However, the prover can offer a configuration of *prover types* and *prover operations* for building up commands which then enable interface features to help the

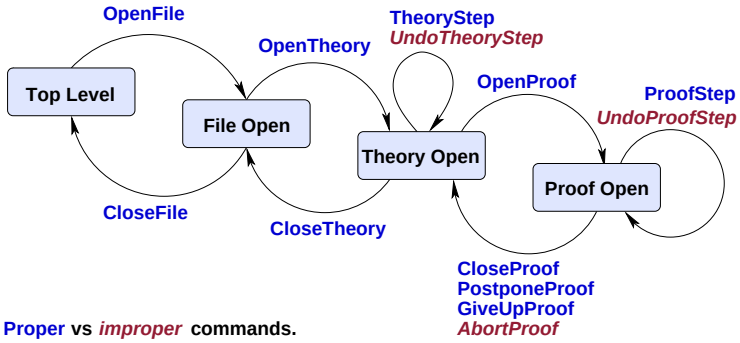

Fig. 5. Proof states during development.

user. The operations are defined by textual substitution. A trivial example for Isar is an operation taking an identifier *id* and a term string *tm*, and produces the command **lemma** *id* : "*tm*" which opens a goal. For textual interfaces, these operations allow a *template* mechanism; for graphical interfaces, they define operations which can be invoked when the user employs certain gestures.

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

3.2 The prover protocol: modelling the prover state

PG Kit has 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.

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. Theories in turn may contain theory items (declarations etc.) and proofs consisting of proofsteps. Within the fourth state, we allow arbitrary nesting (e.g., a proof that contains sub-lemmas).

The reason for distinguishing the states is that the undo behaviour is different in each state, and that different commands are available in each state. In the theory state, for example, we may issue *theory steps* which extend the theory,

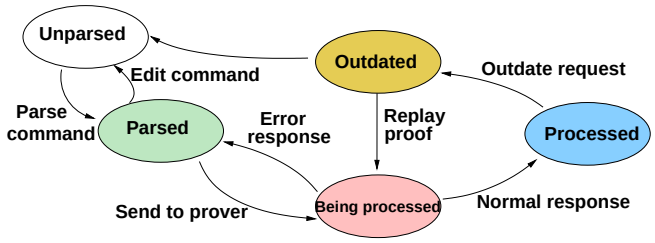

Fig. 6. Command state transitions.

or we may undo the additions. In the proof state, we can issue *proof steps* and undo these steps, or finish the current proof attempt in a number of ways. After finishing a proof, the history is forgotten, and we can only undo the whole proof.

This model is based on abstracting the common behaviour of many interactive proof systems, acting as a clearly specified virtual layer that must be emulated in each prover to cooperate properly with the broker.

3.3 The display protocol and 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 non-overlapping text spans each containing a prover command (see Fig. 4). Each text span has a status ranging over five main[6] possible values, shown in 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 *parsed*. To successfully process a command all commands it is depending on will have to been processed first. Similarly, when we *outdate* a command, all commands depending on it are outdated as well; the difference between outdated and parsed is that outdated regions have been successfully processed before. 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 tacitly.

The transitions between the commands refine the current script management in Proof General. By controlling the state of text spans independently, we can exploit a more fine-grained dependency analysis (if the prover reports the necessary dependency information): to process a command we only need to process those commands which are really needed. The broker handles all this dependency analysis behind the scenes. If the prover does not provide dependency information, the broker automatically assumes linear dependency, where every line potentially depends on all lines that come before.

[6] To be precise, there are other transient states besides *Being processed* but they are not distinguished to the user, so we omit them from Fig. 6.

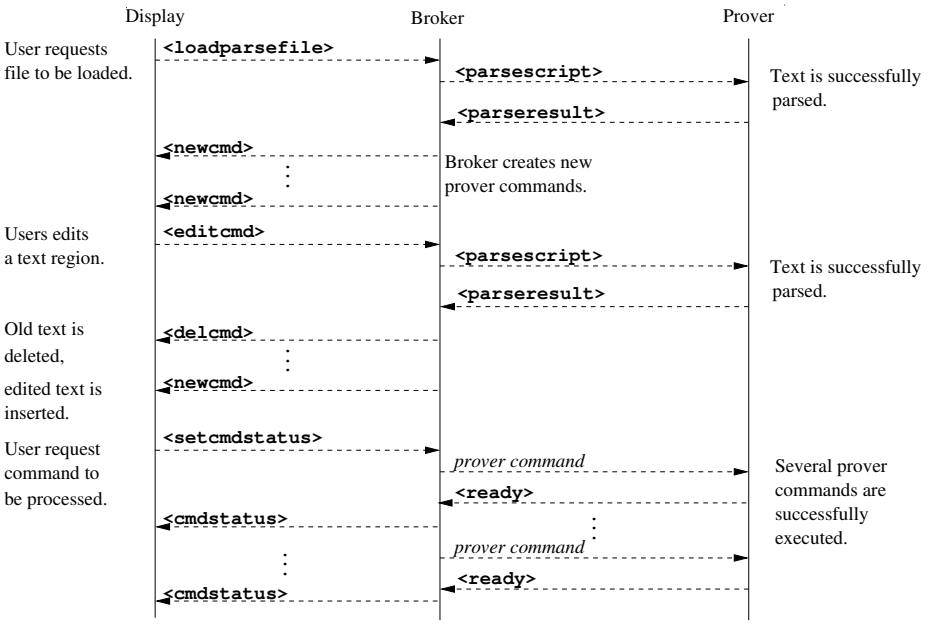

Fig. 7. The edit-parse-prove cycle in a typical situation.

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 requesting a command be processed (<`setcmdstatus`>), which causes 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 (and correct) it later.

4 Brokering electronic proof

The broker is the central middleware component of the PG Kit framework. It gathers input from the displays, sends prover commands to the provers, handles the responses and does the house-keeping, keeping 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 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. For each prover the broker models its state according to the abstract state model from Sect. 3.2. It 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 sends an <errorreponse>, the queue of pending messages is cleared, as it makes little sense to continue. On the other hand, displays have no internal notion of the prover state, but need to keep track of the displayed text and its state.

The broker sends parsing requests to the prover, and extracts the new commands from the answer, checking 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 the result fails this invariant, it inserts the dropped text. As long as only white spaces are dropped, this does not affect the proof.

Particular attention needs be paid to the ability to *interrupt* a running prover. When a prover diverges, it may not respond to messages anymore (including the PGIP <interruptprover> message), so when running a prover as subprocess, we send a Posix signal instead. This is not possible over a socket, so to run a prover remotely, the broker connects to another instance of itself on the remote machine called a *proxy*, using the PGIP_I inter-broker sub-protocol to communicate. This is also useful as broker and prover have to use the same filesystem.

The broker is implemented in Haskell (7k lines of code in 20 modules), using HaXml [25] for a well-typed embedding of the RELAX NG schema. This smoothly extends the schema typing into the Haskell implementation, making it impossible to send messages containing invalid XML.

5 Display components

The display components provide the front-ends with which the user interacts. Currently, an Emacs display and an Eclipse plugin are available.

5.1 Emacs Proof General revisited

The Emacs display for PG Kit will eventually replace the present Proof General. By moving complex functionality into the broker, the Elisp in Emacs can be greatly simplified. The Emacs display may be somewhat limited in facilities, but it has the advantage of portability, including functioning in a plain terminal.

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 re-parse 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 customisable protection against editing those which are in the "processed" state. Compared with the present Emacs interface, this now allows 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 available.

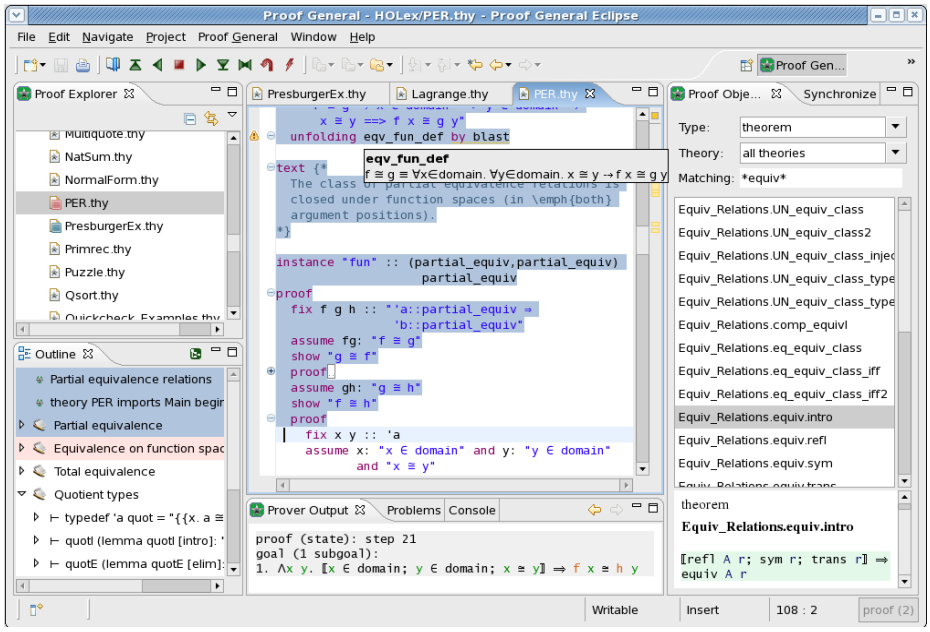

Fig. 8. Eclipse Proof General Display

5.2 Eclipse Proof General

Eclipse [20, 22] is an open-source IDE and tool integration platform written in Java. Most prominently it provides a powerful and attractive IDE for Java, but its plugins and extension points mechanism allows great adaptability. Many plugins are now available, supporting different programming languages, profiling and testing tools, graphical modelling, etc.

Eclipse Proof General is a truly powerful IDE for formal proof which, we hope, will enable a dramatic improvement in usability and productivity for proof development. Graphical views are possible [23], but the primary mode of working remains the editing and scripting management of proof script files.

A screenshot in Fig. 8 shows it in action. The main editor window displays the proof script `PER.thy`; a tool-tip hover shows a definition under the mouse. The Prover Output view below shows the latest subgoal message. The Problems view (obscured) in the tab behind lists outstanding problems, such as syntax errors or unfinished proofs. To the left of the editor window is an Outline View of the proof script showing its structure; above that, the Proof Explorer shows proof scripts in the present folder and indicates their status in the prover with coloured decorators. The colouring metaphor (blue means completed, pink means busy) is used in both of these views as well as the editor window. Above the editor, the toolbar buttons trigger proof or undo steps by sending appropriate PGIP instructions. On the right hand panel, the Proof Objects view allows

browsing the theories and theorems currently loaded in the running session. In the tab behind, the standard Synchronize view (obscured) allows synchronising the development with a version control system (e.g., CVS).

Further features include code folding to hide parts of the text (a sub-proof in the `PER.thy` file is folded in), integrated Javadoc-style help, and hyperlinked indexes for quick access to theorems, definitions and unfinished proofs. Completion is available for identifiers both found in proof script files and given in PGIP messages from the prover containing identifier tables. Completion also provides support for templates and mathematical symbols which are encoded by PGML symbols (or ASCII sequences). Two configurations are provided for symbols. One maps character sequences into Unicode sequences for display in the text editor (supporting provers whose syntax is restricted to poorer character sets). The other configuration is a stylesheet which maps PGML markup into HTML for fully-flexible output display used e.g. in the Prover Output view.

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 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. 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). In the second parsing phase, we call the broker with <`editcmd`> messages to obtain the PGIP mark-up structure. This can either happen in a low-priority background thread, or with specific user commands (such as evaluating a script).

The Eclipse PG plugin is implemented in Java (40k lines of code, 250 classes). Support for a new language in Eclipse is not as straightforward as one might hope, as much of the advanced functionality is still specific to Java. But, paralleling our own development of PGIP, the Eclipse platform is rapidly evolving to migrate Java functionality to platform-level generic mechanisms.

5.3 Other displays

A different kind of display is the lightweight "theorem proving desktop" providing a more abstract, less syntax-oriented interface based on direct manipulation and supported by the visual metaphor of a *notepad* [10]. 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. PGIP supports this style of GUI with the <`operationsconfig`> specification, which describes prover types and operations as mentioned in Sect. 3.1, and can also include icons and hints for selecting operations. We have implemented a prototypical graphical display called PGWin for an earlier version of PGIP [3], where display commands

and messages were not represented in XML. It is currently being adapted to the revised architecture, and made into a separate PGIP component.

Another display currently in development is a web-based display, which will allow users to connect to a running broker with a web-browser, and view the proof scripts as they are being developed. This is an example of a read-only display, which does not provide editing facilities.

6 Conclusions

The Proof General Kit is a framework for connecting interactive proof systems to interface tools and other components. This paper has provided an overview; elsewhere we provide full details including the RELAX NG schemas and protocol descriptions [4]. 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, the Emacs display and the Eclipse plugin are available as beta releases. These have been developed for the upcoming 2007 version of Isabelle, to which support for PGIP has been added by the first author. While straightforward in principle, supporting PGIP in Isabelle 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 are hard to unravel. We expect that this will usually be easier to do in other systems.

PG Kit is unique in proposing a generic framework customised for interactive proof, although there is related work in different settings. Efforts to publish formalised mathematical content on the web include HELM [1] and MoWGLI [15]. The MathWeb project [12] provides a standardised interface using OMDoc [9] as an exchange language. OMDoc elaborates the semantical content of documents, which goes beyond the scope of PG Kit. Other systems such as MONET [14], the MathBroker [18] and MathServe [27] have an architecture similar to ours, but integrate fully automated provers (Otter, Spass etc.) wrapped up as web services, with a broker orchestrating proofs between different provers with little user interaction during the actual proof. In contrast, PG Kit is geared towards connecting interactive theorem provers to user interfaces.

Other frameworks in theorem proving include Prosper [7], which connects several automatic provers to an LCF core to ensure logical consistency. The Prosper Integration Interface (PII) is similar to the low-level aspects of PG Kit, in particular in the way in which interrupts to running components are routed.

Other interfaces similar in spirit to ours include Alcor [6], which extends the Mizar system [24] with knowledge management services such as searching and authoring assistance, and Plato [11], which uses TeXmacs as authoring tool for the Omega system [21]. The architecture is somewhat similar to ours, with a middleware component mediating between prover and interface. Unlike PG Kit, both Alcor and Plato are geared to a specific prover.

There are many possible lines for future development. Foremost among them, we want to use the framework to investigate foundations for *Proof Engineering*, exploring an analogy with software engineering to study useful ways to support the construction, maintenance and understanding of large proof developments. Analogues of code browsing, refactoring, and model driven development would all be intriguing to investigate. Because proofs (in practice) are quite different beasts from programs, and their development is a rather different process, this is a significant research programme.

Another promising direction lies in pushing the generic aspects of the framework, by providing extra language layers or enhancements which work for different systems. For example, we have already designed a generate literate style markup or a document-driven development methodology [5]. We can also use the broker to control proof construction and search: PGIP contains almost enough functionality to support a tactic language at a generic level.

We welcome contact from researchers interested in working with us on future directions or in connecting their systems to PG Kit. Please contact either of the first two authors directly, or visit the Proof General web pages [16] for more information and software downloads.

Acknowledgments: We would like to acknowledge contributions over the years made to the Proof General project by its many users and past developers; not just bug reports, but useful suggestions for improvements, some of which have influenced work described here. Contributors to the Eclipse front-end have included Graham Dutton, Ahsan Fayyaz and Alex Heneveld. The Isabelle developers, particularly Makarius Wenzel, have provided essential help with supporting PGIP. DA benefited from support of the TYPES project (Types for Proofs and Programs, EU IST-2004-510996) and EPSRC platform grant GR/S01771 (The Integration and Interaction of Multiple Mathematical Reasoning Processes). DW was supported by a 2004 Eclipse Innovation Grant awarded by IBM to work on Eclipse Proof General.

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 2001 (TPHOLs '01)*, LNCS 2152, 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 and C. Lüth. Proof General meets IsaWin. In D. Aspinall and C. Lüth, editors, *User Interfaces for Theorem Provers UITP'03*, *Electronic Notes in Theoretical Computer Science* 103, 2003.
4. D. Aspinall and C. Lüth. Commentary on PGIP. Available from http://proofgeneral.inf.ed.ac.uk/kit/, 2003-7.
5. D. Aspinall, C. Lüth, and B. Wolff. Assisted proof document authoring. In M. Kohlhase, editor, *Mathematical Knowledge Management MKM 2005*, LNAI 3863, pages 65– 80. Springer, 2006.

6. P. Cairns and J. Gow. Integrating searching and authoring in Mizar. To appear in *Journal of Automated Reasoning*.
7. 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.
8. 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)*, LNCS 3223, pages 83–98. Springer, 2004.
9. M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. LNAI 4180. Springer, 2006.
10. C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2):167– 189, March 1999.
11. C. B. Marc Wagner, Serge Autexier. PLATO: A mediator between text-editors and proof assistance systems. In C. Benzmüller and S. Autexier, editors, *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*, Aug. 2006.
12. Mathweb homepage. http://www.mathweb.org/.
13. A. Mercer, A. Bundy, H. Duncan, and D. Aspinall. PG Tips, a recommender system for an interactive prover. Presented at MathUI workshop, 2006.
14. MONET — mathematics on the web. Home page at http://monet.nag.co.uk/.
15. MoWGLI. mathematics on the web: Get it right by logics and interfaces. http://www.mowgli.cs.unibo.it/.
16. Proof General Kit home page. http://proofgeneral.inf.ed.ac.uk/kit/.
17. RELAX NG XML schema language, 2003. Home page at http://www.relaxng.org/.
18. W. Schreiner, O. Caprotti, and R. Baraka. The MathBroker project, 2002. http://www.risc.uni-linz.ac.at/projects/basic/mathbroker/, Johannes-Kepler-Universität Linz.
19. C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381– 1405, Sept. 2005.
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. J. Siekmann, C. Benzmüller, and S. Autexier. Computer supported mathematics with OMEGA. *Journal of Applied Logic, special issue on Mathematics Assistance Systems*, 4(4), Dec. 2006.
22. The Eclipse Foundation. Project web site. http://www.eclipse.org.
23. E. Timiriassova. Tracking and visualizing dependency information within theories. Master's thesis, School of Informatics, University of Edinburgh, 2005.
24. A. Trybulec et al. The Mizar project, 1973. http://mizar.org, University of Bialystok, Poland.
25. 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.
26. F. Wiedijk. Digital math: systems implementing "mathematics in the computer". http://www.cs.ru.nl/~freek/digimath/.
27. J. Zimmer and S. Autexier. The MathServe system for semantic web reasoning services. In U. Furbach and N. Shankar, editors, *IJCAR 2006*, LNAI 4130, pages 140– 144. Springer, 2006.