Protocols for Interactive e-Proof*

David Aspinall

May 2000

LFCS, University of Edinburgh, U.K.
http://www.dcs.ed.ac.uk/home/da

Abstract

I motivate and describe work-in-progress on *Proof General Kit* , an evolution of the Proof General project. Proof General Kit introduces a new architecture for Proof General. Instead of the present monolithic implementation in Emacs Lisp, where each proof assistant is connected through a separate customization, Proof General will become a collection of pluggable communicating components using a standard protocol. The Kit is centred around this *protocol for interactive electronic proof*, dubbed PGIP. The design of PGIP has been influenced by the generic mechanisms that Proof General presently uses to communicate with various interactive proof assistants. PGIP is a rationalization of these mechanisms, and will be proposed as an open standard for future proof assistants to support. Messages in PGIP are small XML documents. PGIP comes with an associated markup language, PGML, for representing structural aspects of concrete syntax.

See the home page at http://www.lfcs.ac.uk/proofgen for more about Proof General.

1 Introduction

Proof General is a generic interface for interactive proof assistants, based on Emacs [1, 2].

Proof General was first built to address the needs of a particular class of users. Many proof assistants still have a primitive command line interface. Even when sophisticated GUI alternatives are available, it is observed that expert users often *prefer* the command line interface and work more effectively with it. This may be for several reasons: perhaps because the GUIs are poorly engineered, perhaps because they are overly restrictive or do not scale to large developments, or simply because current experts do not want to change their working practices and spend effort on learning an interface. (For more opinions, see [11]).

Proof General targetted expert users, by fitting quite closely with their existing models of interaction, but making those interactions more effective and convenient by adding short-cuts and centralising development around the end product — the *proof script*. Recent improvements such as buttons, menus, and symbol fonts have made Proof General more accessible for novice users too. It now provides a middle ground in interface technology, largely text-based rather than graphical, but with sophisticated features like *script management* and *proof by pointing* (both inspired by the CtCoq GUI [5]), implemented in a lightweight and efficient way. Proof General's implementation of script management is particularly impressive, allowing an integration with the file-handling system of the proof assistant.

The strategy of targetting experts as well as novices has been a success. Proof General is now widely used in teaching as well as research, and in industry as well as academia. Perhaps the greatest and most unique aspect of the success of Proof General is its *genericity*. It exploits the deep similarities between systems by hiding some of their superficial differences. Just as a web browser presents a similar interface to different protocols — http, ftp, or file, so Proof General presents a similar interface to different proof assistants. This genericity is no empty claim or briefly tested design goal; the system is already in common use for LEGO, Coq, and Isabelle. Support for HOL98 has recently been added.

2 A New Architecture

Although it has been a success, the present Proof General has some drawbacks and scope for improvement. For example, from the user's point of view:

1. The Emacs-centric nature puts off some people who don't use or don't like Emacs.
2. Not enough system-specific options are pro-

*Draft extended abstract, presented as a work-in-progress paper at TPHOLs 2000.

vided, for example, to invoke common tactics or commands in a system. The interface degenerates to offering a command-line prompt for these cases.

3. There is no allowance for interoperability with other tools, and only restricted facility for internet-based distributed development.

4. While Proof General presents a similar appearance for different proof assistants, ultimately the user must be familiar with most commands of the underlying proof assistant. Further abstraction should be possible.

From the implementors point of view, Proof General is not as easy to connect to proof assistants as we might like:

1. Customization and extension is exclusively in Emacs Lisp. (Recently it has been drastically simplified, but it still involves setting some tricky Emacs regular expressions).

2. The implementor has too much choice, in a sense: each proof assistant is free to use its own communication triggers and markup schemes.[1] This makes it complicated to configure Proof General and inhibits communication with other tools.

3. There is some duplication of information in the interface and the proof assistant, and some information is hard-coded into Emacs which would better belong in the proof assistant.

To address these drawbacks, we propose a new architecture for Proof General. First, we recognize that not everybody shares a love for Emacs, and that Emacs Lisp is not the best language for implementing some advanced aspects of the interface. To allow more generality, we envisage a collection of alternative pluggable components, implemented in various languages, which can serve as user interface elements, proof assistants, and eventually other proof tools. Second, to make this possible, we realize that the new architecture requires a common underlying interface language and interaction protocol.

To take things further, we want to deepen the abstraction that Proof General provides, so that the logic and proof language, as well as the interaction model, become generic. Ultimately the choice of logic and its syntax should fall under Proof General's organization, and we envisage implementing advanced aspects of logical frameworks: a logic morphism from the core logic of each supported proof assistant into Proof General's general form of the logic. Thus Proof General would truly live up to its name: a generic system for doing proofs, with the underlying proof engine being almost arbitrary.[2]

But that ultimate goal remains for the future; we prefer to build up towards it in stages. Right now, we are working on the first stage.

3 Proof General Kit

The Kit introduces a new architecture for Proof General. Instead of a monolithic implementation inside Emacs, Proof General will become a collection of communicating components. In the spirit of the present system, we want to use carefully designed lightweight protocols, which are easily supported by a range of present and future proof engines. The protocols and components will be developed in stages. The first stage is to develop the *PGIP protocol* and *PGML markup language*.

PGIP is the protocol that we propose for communicating the progress of an interactive proof; it is based on examining and clarifying the mechanisms currently implemented. We give a short description in Section 4 below.

PGML is the markup language used inside PGIP to annotate terms and formulae for display by the interface. We give a short description in Section 5 below.

System architecture An overview of the architecture of the Proof General Kit is shown in Figure 1. The proof assistant and filesystem are outwith the Kit; the other components are part of the system.

Notice the central role of the *Mediator* here: it interfaces the various other components to the proof assistant and filesystem, whereas the other components have no direct access to either. Only the mediator and the proof assistant use operating system calls to access the file system (or internet) theory store. This separation allows the Mediator to organize the synchronization messages needed for script management. At the start of the project, the Mediator is even more important, because it

[1] This is because we tried to make Proof General work with existing systems without requiring their modification. But this was only partially successful: for robustness and features like proof by pointing, it is necessary to modify the output routines of proof assistants anyway.

[2] This aspect is dubbed *Logic General*. Similar ideas appear elsewhere, e.g. in OMEGA [4].

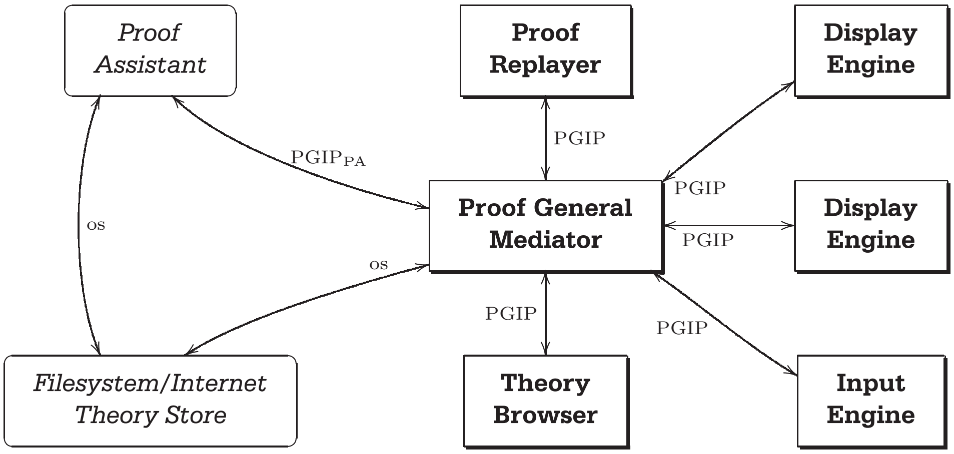

Figure 1: Simplified Architectecture of Proof General Kit

provides extra interfacing mechanisms to connect to the proof assistant.[3] This is because we begin from proof assistants which do not natively or fully handle PGIP, explaining the $\mathrm{PGIP}_{\mathrm{PA}}$ label on the arrow.

The remaining components are examples only. In a particular session we might connect several *Display Engines* (perhaps running remotely, or displaying different aspects). Only one *Input Engine* is allowed to be *active for scripting* at a time, which means that it owns the current proof development.[4] The *Proof Replayer* is a restricted kind of input engine which only allows replay of previous proofs, perhaps over the internet. The *Theory Browser* is a component which allows browsing of the theory store and/or the theories which are loaded into the running proof assistant.

4 PGIP

PGIP is a protocol for conducting interactive electronic proof. It has other features to enable the Kit architecture, for example, various querying operations on the filesystem or running proof assistant.

Ultimately we hope that PGIP will be adopted as a standard by proof assistant implementors;[5] the benefit to them will be that they can connect off-the-shelf interfaces, and with less effort than it takes to connect to Proof General currently.

Essentials PGIP makes certain assumptions about the way proof is conducted with a proof assistant. The essential aspect is that proof proceeds in steps, and that the proof assistant can be "fed" a step at a time, without needing to see the whole proof. So we assume that the proof assistant itself maintains some state representing the proof development. A proof using PGIP has these parts:

- Proof begins by issuing a target *goal*
- Proof proceeds by successive *proof steps*
- To reverse the effect of a proof step, the proof assistant has a command to *undo*.
- A proof is closed, aborted, or abandoned by issuing a *save-goal*, *quit-goal*, or *forget-goal*, respectively.

We consider the goal, save-goal, and forget-goal operations to be proof steps themselves, whereas undo and quit-goal are not. Together, the proof steps form a language for writing *proof scripts*. There is much freedom on exactly what a proof script is: for example, it may well be a *declarative* rather than a *procedural* description of a proof. It matters little to Proof General whether scripts are forwards or backwards proofs, or whether they use tactics or some more readable description of proof steps. The essential aspect is that they have a textual, linear representation.[6]

[3] In the Proof General Kit white paper [3], the Mediator is decomposed into several subcomponents.

[4] This is because we assume that the proof assistant is single threaded and maintains only a single proof state. Later on, we may treat proof assistants as a resource, and allow the Kit to connect to several at once; for the time being we will work within the current model of just a single proof assistant instance.

[5] This stands some chance, because it is being designed in conjunction with some of those very people!

[6] Many interfaces like to develop proofs graphically based on trees or graphs, since their authors argue that this is the true nature of a proof. Yet the abstract syntax of a program is also tree-based, and text-based linear description and development

Proof scripts may have additional elements outside proofs, for example, to make definitions or assumptions. (If definitions and assumptions are allowed inside proofs, we consider them as proof steps and they must be undoable). Moreover, proof scripts may have additional structure both inside and outside proofs, to allow *sections* or *block structure* which exposes some structure of the underlying proof tree and theory.

As well as controlling the progress of the interactive proof, PGIP is responsible for certain initialization and book-keeping tasks which require communication between the proof assistant and the rest of Proof General. It includes messages to configure (proof assistant specific) user-level menus and preferences; messages to display status or error dialogues, and messages to retrieve theory files.

For specifics of the PGIP format including an XML DTD, see [3].

Sending PGIP messages A PGIP message is sent in a "packet", which is an XML document with root <pgip>. We haven't yet decided on a particular transmission mechanism for packets. To begin with we are experimenting using Unix pipes and pseudo ttys; later we may use sockets and/or other mechanisms (probably http for internet connectivity; perhaps a simple CORBA interface for desktop connectivity). PGIP messages are themselves structured in XML, and may contain embedded PGML.

5 PGML

PGML is an XML-based document structure for transmitting marked-up *concrete* syntax output from proof assistants.

Most present proof assistants have their own favourite concrete syntax and syntax defining mechanisms, but little or no access to abstract syntax. So we assume that the output is already in concrete syntax; the reason to markup is to allow decoration of aspects of the syntax (e.g. special constants or variables), and also to allow proof-by-pointing style interactions via subterm position annotations. We want to use a logical markup mechanism to make these things possible, so that different renderers can display information in different ways, allowing variations in fonts, colours, special characters, hidden annotations, etc.

There are already emerging standards for XML-based document formats designed for displaying mathematics (MathML, [14]), and transferring mathematical content between applications (OpenMath, [12]). Later on, we hope to use these languages with PGML. At the moment, it is more feasible to implement our own simple markup scheme since both MathML and OpenMath go further into the *abstract* structure of terms than we need, or than we can easily accommodate in a generic way.

Term structure annotations are an effective halfway house, and can be included by those proof assistants capable of generating them. Already for systems like Isabelle it is a highly non-trivial task to relate the abstract syntax to the concrete, and generate annotations. But we believe that more proof assistants will provide structured output in the future.

PGML documents Here is a proof state display in Isabelle:

```
Level 3 (2 subgoals)
A & B --> B & A
 1. [| A; B |] ==> B
 2. [| A; B |] ==> A
```

In Figure 2, the skeleton of a PGML marked-up version of the same display is shown. Isabelle has a rich structure on the subgoal display, always including the overall goal and a list of local assumptions for each subgoal. The PGML markup makes this structure apparent within the concrete syntax. It is simple to implement this markup scheme in the output routines of Isabelle. And with the marked-up version of the subgoal display, it will become easy for display engines to hide or reveal parts of the display, quickly navigate through large displays, etc.

PGML can go a bit further into the structure of terms than shown in Figure 2, in particular using these elements:

- <term pos="*p*" >*term text*</term>
- <action kind="*kd*">*action text*</action>
- <atom kind="*kd*">*name*</atom>
- <sym name="*nm*" alt="*SYM*">

Terms The purpose of <term> is to add enough structure to annotate subterm position information, for mouse-sensitivity and proof-by-pointing actions. The *pos* annotation achieves this; typically it is a position indicator in the abstract syntax tree corresponding to the surounding term. The position is not displayed or interpreted by the display

of programs is hardly obsolete. Proof General takes the linear text to be primary, but may later offer adjuncts for browsing or developing proofs graphically, perhaps using *dependency analysis* [13] for exposing the structure of proof scripts.

```
<pgml>
<statedisplay name="Level 3" id="avocado.dcs.ed.ac.uk/951858790/12480/101">
    Level 3 (2 subgoals)
    <br/>
    <statepart kind="initialgoal">
        A & B --> B & A
    </statepart>
    <br/>
    <statepart kind="subgoal" name="1"> 1.
        <statepart kind="asms"> [| A; B |] </statepart>
        ==>
        <statepart kind="body"> A </statepart>
    </statepart>
    <br/>
    <statepart kind="subgoal" name="2"> 2.
        <statepart kind="asms"> [| A; B |] </statepart>
        ==>
        <statepart kind="body"> B </statepart>
    </statepart>
    <br/>
</statedisplay>
</pgml>
```

Figure 2: Example of outer structure PGML markup for Isabelle

engine, but should be recoverable by mouse pointing, so that it can be sent back to the proof engine.

Actions Terms may have *action* texts attached to them. Zero or more <action> texts appear immediately after an operning <term>. The idea behind actions is that they allow additional hidden annotations which may be revealed to the user (e.g. typing information, proof hints), or used to generate commands to the proof assistant to allow fine-grained proof-by-pointing style interactions.

Atoms Atoms in terms can be decorated by the display engine to indicate their logical status and to attach extra information. The logical status is suggested by the *kind* name, which may be used by the display engine to generate different colours or balloon popups. It is appropriate to use different kinds when the atom is named from a distinct namespace (e.g. if we want to help the user distinguish between a variable "x" and a constant "x").

Symbols Proof General at present can use the XEmacs package *X Symbol* [15] to display a variety of mathematical symbols, Greek letters, etc, which are not part of ASCII. PGML will allow these extra characters to appear via the <sym> tag.

For specifics of PGML and a DTD, see [3].

6 Design and Engineering

Careful design and good engineering are both important to achieve a usable system. And even when building research prototypes, we believe that software quality deserves serious attention.

One design question for a framework like PGIP is where to make the split between interface and proof engine. One might expect the part of the system which records the state of the proof development to be closely coupled to the interface, where it is always manipulated; perhaps with a functional API for constructing proofs in the engine. But experience with Proof General shows, perhaps surprisingly, that the stateful development mechanisms already implemented in diverse provers are compatible enough to have a generic interface at a higher level, sending proof commands. This is more efficient, since the data structures associated with proof development (e.g. list of current subgoals) are rather closely coupled with the proof engine. This explains the form of PGIP described in Section 4.

An engineering factor that seems important in the success of Proof General is the incremental way it has been constructed, by *successive generalization*.[7] It began as "LEGO mode", an interface for one system. Then support for Coq was added, generalizing to "Proof Mode". In the last cou-

[7]This is reminiscent of a *product line architecture* in software engineering.

ple of years, support for Isabelle was added, generalizing further, and giving birth to "Proof General." Each stage of generalization involved a mix of modifying and re-engineering the basic core of the system, and adding new features, all the while carefully maintaining support for previous proof assistants. This process is synergistic: supporting a new system typically *improves* support for the other systems. Features which are useful or easy to add for the new system get added back to previous systems, as innovations there; a direct and novel cross fertilization method.

We hope that this engineering approach can help the future success of Proof General Kit. That is why we start from a restricted and "bottom up" plan, rather than a "grand design" promising the world. It means we can refine and extend a series of useful working prototypes, rather than struggle for a long time on something unwieldy.

7 Conclusions

We have described and motivated ideas for the Proof General Kit. At the moment, we are at an early stage, conducting experiments and applying for funding. More details of what was described here appear in the white paper [3], which is being discussed on the Proof General developer's mailing list. Just now, we are considering the viability of PGIP for handling interaction methods other than text-based script processing; in particular, we hope to connect to a generic direct manipulation interface developed at the Universities of Bremen and Freiburg [10].

I welcome feedback and discussion on this paper and [3]. For information about the Proof General project or to try Emacs Proof General visit `http://www.lfcs.ac.uk/proofgen`.

Related work Interoperability is in the fashion, and several folk in the theorem proving community are taking it seriously. Other diverse projects include OMEGA [4], OMRS [8], Prosper [6] and HELM [7]. Compared to these, Proof General Kit has a rather different focus, concentrating on protocols for use during interactive proof, rather than communicating or storing semantic content. That said, we hope to consider semantic content as an extension to PGIP in the future, perhaps linking up to one of the other projects. Another difference of the projects mentioned is that most grew from other communities, which favour different proof assistants to the ones we are working with. But it seems that the interoperability drives are making links between the theorem proving cliques, at last, and we hope that Proof General Kit will add to this.

Acknowledgements I'm grateful to the Proof General developer team, and several others, who have expressed interest in the Kit project. In particular, for discussions on the content of [3] I'm grateful to Paul Callaghan, Pierre Courtieu, Christoph Lüth, and Markus Wenzel.

Proof General Credits Thomas Kleymann began LEGO mode, also worked on by Dilip Sequeira, and had the idea of making it generic. Healfdene Goguen added Coq support. Kleymann managed development until Aspinall took over; Aspinall added support for Isabelle and helped Proof General to its first widely distributed releases in late 1998. Since then Aspinall has developed Proof General together with helpers from the camps of the supported proof assistants (a deliberate policy), including M. Wenzel, D. von Oheimb, P. Callaghan, P. Loiseleur, and P. Courtieu. We are grateful to users and testers for their feedback and suggestions.

References

[1] D. Aspinall, H. Goguen, T. Kleymann, and D. Sequeira. Proof General. System documentation, see `http://www.dcs.ed.ac.uk/home/proofgen`, 2000.

[2] David Aspinall. Proof General: A generic tool for proof development. In Graf and Schwartzbach [9], pages 38–42.

[3] David Aspinall. Proof General Kit. White paper. See http://zermelo.dcs.ed.ac.uk/home/da/drafts/, 2000.

[4] C. Benzmüller et al. ΩMega: Towards a mathematical assistant. In William McCune, editor, *14th International Conference on Automated Deduction — CADE-14*, volume 1249 of *LNAI*. Springer, 1997.

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

[6] Louise A. Dennis et al. The PROSPER toolkit. In Graf and Schwartzbach [9].

[7] Andrea Asperti et al. Helm – an hypertextual electronic library of mathematics. Home page at `http://www.cs.unibo.it/~asperti/HELM/home.html`, 2000.

[8] Fausto Giunchiglia, Paolo Pecchiari, and Carolyn Talcott. Reasoning theories: Towards an architecture for open mechanized reasoning systems. In F. Baader and K.U. Schulz, editors, *"Frontiers of Combining Systems - First International Workshop" (FroCoS'96)*, Kluwer's Applied Logic Series (APLS), pages 157–174, 1996.

[9] Susanne Graf and Michael Schwartzbach, editors. *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1785. Springer-Verlag, 2000.

[10] C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2), 1999.

[11] N. Merriam and M. Harrison. What is wrong with GUIs for theorem provers? In Y. Bertot, editor, *User Interfaces for Theorem Provers UITP'97*. INRIA Sophia Antipolis. Electronic proceedings at `http://www.inria.fr/croap/events/uitp97-papers.html`, 1997.

[12] The OpenMath Society. `http://www.nag.co.uk/projects/openmath/omsoc/`.

[13] Olivier Pons, Yves Bertot, and Laurence Rideau. Notions of dependency in proof assistants. In *Proc. User Interfaces for Theorem Provers, UITP'98*, 1998.

[14] Mathematical markup language (MathML). W3C Recommendation, 1999. http://www.w3.org/TR/REC-MathML/.

[15] Christoph Wedler. Emacs package "X-Symbol". http://www.fmi.uni-passau.de/~wedler/x-symbol/.