# Proof General Kit

**White Paper.**
**Version 1.6, 2003/07/09 22:34:36**

David Aspinall

February 2000 — July 2003

LFCS, University of Edinburgh, U.K.
Originated while visiting ETL, Osaka, Japan.
http://homepages.inf.ed.ac.uk/da

**Abstract**

This white paper describes proposals for the *Proof General Kit* , the evolution of the Proof General project. 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. The aim is to allow a flexible interoperability between various user interface elements (including Emacs), proof engines, and other proof tools. In the spirit of the present system, we want to achieve this using 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, by successive generalization.

At the moment this white paper is a working document; it contains some incomplete sections. It is intended to stimulate discussion and flesh out details. I welcome ideas or suggestions for improvements. Please send mail to David.Aspinall@ed.ac.uk.

## Contents

**1 Introduction** 2

**2 A New Architecture for Proof General** 2
2.1 Proof General Kit 3
2.2 Communication Protocols 4
2.3 Kit Components 4
2.4 Plan 4

**3 Protocols for interactive e-proof** 5
3.1 PGML, the Markup Language 5
3.1.1 Overview of PGML 5
3.1.2 PGML documents 5
3.1.3 State displays 6
3.1.4 Term display 6
3.1.5 Information and Errors 8
3.1.6 Terms 8
3.1.7 Atoms 8
3.1.8 Symbols 9
3.1.9 Formatting 9
3.1.10 Summary 9
3.2 PGIP, the Interface Protocol 10
3.2.1 Basic proof mechanism with PGIP 10
3.2.2 Message packet 11
3.2.3 Proof and control commands 11
3.2.4 Configuration messages 11
3.2.5 Status and error messages 12
3.2.6 Proof blocks 13
3.3 Interface configuration with PGIP 13
3.4 Script management with PGIP 14

**4 Components for Proof General Kit** 14
4.1 PGCOM, the Communication Layer 15
4.2 PGFILT, the Filter 15
4.3 PGIP for Emacs Proof General 15
4.4 PGMEX, the Multiplexer 15
4.5 PGDISP, the Displayer 16
4.6 New Features for Proof General Kit 16
4.6.1 Theory browser 16
4.6.2 Completion mechanism 16
4.6.3 Favourites and history 17
4.6.4 Configurable menus and toolbar 17

**5 Abstract syntax and pretty-printing** 17
5.1 PGTERM, the Term Representation 18
5.2 PGSYN, the Syntax Representation 18
5.3 PGPP, the Pretty Printer 18
5.4 PGDE, the Display Engine 18
5.5 Other term-based features 18

**6 Logic Genericity** 19
6.1 Logic General 19
6.2 Proof languages for Logic General 19

**7 Conclusions and related work** 19
7.1 Related Projects, past and present 19
7.2 Credits 21

**A Schemas for PGIP and PGML** 22
A.1 pgip.rnc 22
A.2 pgml.rnc 26

# 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 of proof assistants. Many proof assistants still have a primitive command line interface. Even when sophisticated GUI alternatives are available, we noticed that expert users often *prefer* the command line interface, and work more effectively with it. There are several possible reasons: because the GUIs are poorly engineered, 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 waste effort on learning an interface. Proof General targetted these expert users, by fitting closely with their existing models of interaction, but making those interactions more efficient and comfortable by adding short-cuts and centralising development around the target of the proof development, what Proof General terms the **proof script**. Later improvements such as buttons, menus, and symbol fonts 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*. The present system is aimed at users of systems based on type theory and logical frameworks, particularly LEGO, Coq, Isabelle, and HOL.[1]

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, 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 the local filesystem, so Proof General presents a similar interface to different proof assistants. This genericity is no empty claim or briefly tested design goal; Proof General is already in common use for the first three of the proof assistants mentioned, and support for others is on the way.

An important factor in the success of Proof General appears to be its development by *successive generalization* (in software engineering parlance, as a *product-line architecture*). Proof General began in 1997 as "LEGO mode", an interface for one system. Then support for Coq was added, generalizing the system to "Proof Mode". In 1999, support for Isabelle was added, generalizing further, and giving birth to "Proof General." Each stage of generalization involved some mix of modifying or 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, too, as features which are obvious or easy for the new system also get added back to previous systems, as innovations there; cross-fertilization in a novel and direct way.

I believe that this same engineering approach can be used for the future success of a new architecture for an improved Proof General.

# 2 A New Architecture for Proof General

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 provided, 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.

[1]Compared with some other proof systems, these systems have perhaps the strongest claim to fully formal proofs with careful correctness claims, yet they also have the weakest user interfaces. Systems developed in other communities have often had more elaborate UIs.

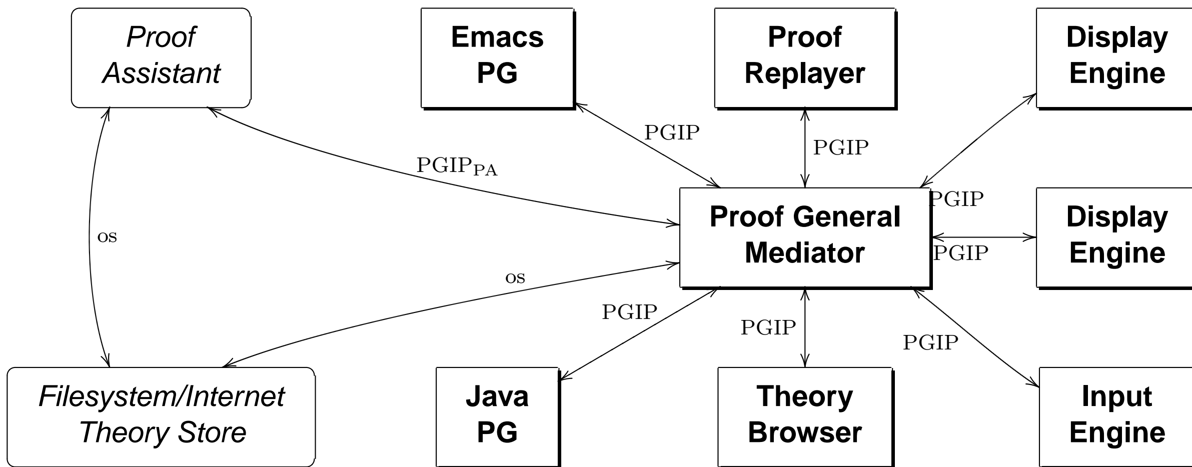


Figure 1: Architecture of Proof General Kit

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. (It has been drastically simplified with "easy-config" macros, but it still involves setting some tricky Emacs regular expressions and numerous flags and options, and understanding how those control Proof General's behaviour).

2. There is 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.

3. The implementor has too much choice, in a sense: each proof assistant is free to implement its own markup scheme. This makes it more complicated to configure Proof General and inhibits communication with other tools.

To address these drawbacks, we propose a new architecture for Proof General. First of all, 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 a yet more *general* system, we envisage a collection of alternative pluggable components which can serve as display or input engines of one sort or another. This architecture requires a common underlying interface language and interaction protocol.

Taking 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 some 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 really live up to its name: a generic system for doing proofs, with the underlying proof engine being almost arbitrary.

## 2.1 Proof General Kit

The *Proof General Kit* introduces a new architecture for Proof General. Instead of a monolithic implementation inside Emacs, Proof General will become a collection of communicating components.

An overview of the architecture of 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.

## 2.2 Communication Protocols

In the spirit of the present system, we want to use carefully designed lightweight protocols for Proof General Kit, which are easily supported by a range of present and future proof engines.

The overall protocol is called **PGIP** (standing for *Proof General Interactive Proof*). This is the protocol used to connect most of the components shown in Figure 1. PGIP is based on examining and clarifying the mechanisms currently implemented. It is introduced in Section 3.2 and developed through the rest of this white paper.

Part of PGIP protocol is concerned with transmitting documents representing information about the proof in progress, including terms and formulae displayed by the interface. These documents are annotated using **PGML** (standing for *Proof General Markup Language*). We give a description of PGML in Section 3.1.

## 2.3 Kit Components

In Figure 1, we can see that the *Mediator* has a central role: it interfaces the various other components to the proof assistant and filesystem. We expect that the other components have no access to the proof assistant other than via the mediator. The other components *may* have direct access to the filesystem (or internet) theory store, but this must be carefully sanctioned by the mediator. This separation allows the Mediator to organize the synchronization messages needed for script management, and maintain the set of locked files.

At the start of the project, the Mediator is even more important, because it provides extra interfacing mechanisms to connect to the proof assistant. 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 in Figure 1.

The remaining components are examples only. First, we have two "complete" front-end applications, *Emacs PG* and *Java PG*. These are intended to behave much as the present Emacs Proof General does, offering the user the possibility to edit proof scripts, process them with script management and display the results, amongst other facilities.

As well as these integrated applications, the new architecture allows the different functions to be split across different components. In a particular session we might connect several *Display Engines* (perhaps running remotely, or displaying different aspects). Although several may be connected, only one *Input Engine* is allowed to be *active for scripting* at a time, which means that it owns the current proof development.[2] The *Proof Replayer* is a restricted kind of input engine which only allows replay of previous proofs, perhaps over the internet so there is no need to install the proof assistant remotely. 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.

## 2.4 Plan

The overall plan to build Proof General Kit is broken into four phases, described in more detail in the following sections. The first two phases are covered in greatest detail; the second two phases require further research. Section 3, describes PGML and introduces PGIP, the common markup language and interaction protocol. Section 4 describes some of the basic components, and how they might be connected with PGIP. It also considers some extensions to PGIP which go beyond what is possible in the present Emacs Proof General. Section 5, marks the beginning of the "semantic aspect". At this point, we begin to consider features which require the interface to have more detailed semantic representations of the data structures used in theorem proving, and propose ways to transmit this information. Finally, Section 6 provides a sketch of the ideas for managing logics and syntax inside Proof General. This phase is dubbed *Logic General.* It will be developed more fully elsewhere.

[2] 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.

## 3 Protocols for interactive e-proof

The first phase for Proof General Kit is to develop a common output markup language and protocol for interactive electronic proof.

Previously, Proof General allowed each proof assistant to define its own markup and protocol elements. The idea was to allow configuration for proof assistants which cannot themselves be modified, perhaps because the source is not available, or simply because it would be an overly complicated undertaking. However, in practice, we often needed to add extra mark-up to the output of a proof assistant, anyway, to ensure robustness and implement extra features such as proof-by-pointing.

For Proof General Kit, we will start off by enforcing a common markup language and interface protocol. This is needed because we want to open up the architecture to allow interoperability between many components; a common interchange language is essential for sanity.

This phase involves some design (Sections 3.1–3.3) and some re-engineering (Sections 4.2–4.3). The design should be fairly straightforward because we will adapt the emergent design that evolved during the development of Proof General. The implementation effort is also relatively straightforward, because in this phase we stick with the Emacs front end as primary.

### 3.1 PGML, the Markup Language

The Proof General Markup Language is used for markup of output from the proof assistant which is to be displayed to the user. At this stage, we assume that the output is already in some concrete syntax and the reason to markup is to allow decoration of the syntax with special fonts, colours, etc, as well as to allow proof-by-pointing style interactions via subterm position annotations. We want to use a logical markup mechanism to make these things possible.

#### 3.1.1 Overview of PGML

Here is an overview of the markup scheme we propose. More precise details are given in the appendix, Section A.2, which gives a complete RELAX NG [4] schema specification for PGML.

To begin with, early versions of PGML will be custom XML-based document formats. Why do we need a new format? HTML is too low-level, of course; we want to reflect the logical aspects of proof displays. This will make it possible for different renderers to display proof assistant information in different ways, allowing variations in fonts, colours, special characters, hidden annotations, etc. On the other hand, there are emerging standards for XML-based document formats designed for displaying mathematics (MathML), and transferring mathematical content between applications (OpenMath). Later on, we hope to use each of these languages with PGML. At the moment, it is much easier to implement our own simple markup scheme since both MathML and OpenMath go further into the structure of terms than we need or can easily accommodate in a generic way.[3] So PGML begins as being a marked-up version of the *concrete* syntax output from proof assistants. Later on we consider richer forms of PGML when we consider the *abstract* syntax proposals in Section 5.1.

In summary, the first versions of our markup language, PGML 1.X, will be a decoration-oriented logical markup scheme for concrete syntax of proof displays, logical formulae, and terms. This is similar to the present ad hoc markup used by (some instances of) Proof General using non-ASCII characters.

#### 3.1.2 PGML documents

Markup in PGML uses the following elements and tags:

- `<pgml version="`*ver*`">`*text*`</pgml>`

[3] A proof assistant must be equipped with a way of marking-up its output syntax with annotations expressing the term structure; for some proof assistants this is highly non-trivial, and not supported at present. We believe that more proof assistants will provide structured output in the future.

- <statedisplay systemid="*id*" name="*nm*" kind="*kd*">*state text*</statedisplay>
- <termdisplay name="*nm*" kind="*kd*">*state text*</termdisplay>
- <information name="*nm*" kind="*kd*">*text*</information>
- <warning name="*nm*" kind="*kd*">*text*</warning>
- <error name="*nm*" kind="*kd*">*text*</error>
- <statepart name="*nm*" kind="*kd*">*part text*</statepart>
- <term pos="*p*" >*term text*</term>
- <type kind="*kd*">*type text*</type>
- <action kind="*kd*">*action text*</action>
- <atom kind="*kd*" fullname="*fn*">*name*</atom>
- <sym name="*nm*" alt="*SYM*">
- <br>

A PGML document is enclosed in a <pgml> element and may contain any number of <statedisplay>, <information>, <error>, and <warning> elements. The next few subsections describe these elements in more detail.

Notice that markup (and the induced decoration) applies uniformly to miscellaneous messages from the proof assistant, as well as to the logical information presented in state displays.

### 3.1.3 State displays

The main display item is a *statedisplay*, which is some text which gives some indication of the current status of a proof. We assume that a state display *refers to a particular internal state of the proof engine*. Proof-by-pointing (for example) only makes sense in some particular state. The display of a help message, or the a definition of an input term, might or might not.[4]

The state display may be freely split into a sequence of (possibly nested) <statepart> elements. Both <statedisplay> and <statepart> have optional tags giving a unique system-level identifier, a user-level name, and a classifying kind. The identifier will be important for ensuring synchronization: for example, it should reflect the internal position of the proof progress within the proof assistant, so that proof commands are anchored to a reference to the appropriate state of the proof assistant. For the top-level <statedisplay>, the user-level name is like the <title> tag in HTML; it might be used for a window title, or to navigate in a cache of previous states. For nested <statepart> elements, it serves similarly as a name which may help the display engine fold or unfold structure, for example.

Figure 2 shows an example of a state display for Isabelle with PGML markup.

### 3.1.4 Term display

A term display is like a state display except that it doesn't rely on an internal state of the proof system. (E.g. print output for a theorem, etc).

[4]It might not depend on the internal state in a way that matters for the interaction.

*Original display in shell window:*

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

*Skeleton of a marked-up version:*

```
<statedisplay name="Level 3"
      systemid="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>
```

*Isabelle has only one kind of proof state display, so there is no need for an outermost kind tag. On the other hand, it has a rich structure on the subgoal display, always including the overall goal and a list of local assumptions for each subgoal. The markup above makes this structure apparent within the concrete syntax. Notice that it is only a "skeleton" markup, because a full markup could also contain term-structure information, or variable kind annotations. But this is optional.*

Figure 2: Example of proof state markup for Isabelle

### 3.1.5 Information and Errors

Apart from state displays, the proof engine may output other subsidiary bits of information to the user, which are also marked in PGML. The distinction is the way they are displayed: a state display will be considered as a primary concern that the user is always interested in (it *persists*), while the subsidiary pieces of information are considered more transient and displayed in a less prominent window, a transient way, etc. This should be customizable to some extent.[5]

Subsidiary display items are triggered by `<information>` and `<error>` elements, which may also be tagged with user-level names and/or *kind* classifications. These serve to indicate what the information or error is. For example, information may be a system message, or the result of a user interaction of some kind. An error message may be a warning message or some fatal message. These auxiliary pieces of information could appear in a separate window on the screen, a popup window, or they could be shown on the same window as the proofstate display.

### 3.1.6 Terms

Terms can appear anywhere inside PGML text, marked with the `<term>` element and nested arbitrarily. To begin with, the `<term>` will be optional: its use is to add enough structure to annotate subterm position information, for mouse-sensitivity and proof-by-pointing actions. The optional *pos* annotation achieves this. It is not displayed or interpreted by the display engine, but should be recoverable by mouse pointing, so that it can be sent back to the proof engine. A character on the display has the *pos* annotation of the smallest enclosing `<term>` element, if there is one. As a configuration option, the display engine should be able to concatenate *pos* annotations on nested subterms when it indicates the position of the mouse, so that the proof assistant does not need to annotate with complete path information.

Some terms may be designated as being types with the `<type>` element; a print engine may allow for types to be coloured specially, optionally hidden, or displayed in balloon popups. A type is attached to a term by the convention of appearing immediately before a closing `</term>`. This is supposed to be compatible with the usual concrete syntax for types, as `a:T` or similar.

Terms may have *action* texts attatched to them. The convention is that one or more `<action>` texts appear immediately after an operning `<term>`. Different kinds of action tags are distinguished by their (compulsory) kind settings *kd*. A character has the actions of the smallest enclosing subterm, if it has any. The idea behind actions is that they allow additional hidden annotations which may be revealed to the user (e.g. proof hints), or used to generate commands to the proof assistant to allow finer-grained proof-by-pointing style interactions. At the moment, we leave unspecified the kinds of actions available.

### 3.1.7 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 particularly appropriate to use different kinds when the atom is named from a distinct (system) namespace (e.g. if we want to help the user distinguish between a free variable “x” and a bound variable “x”).

By convention, certain kinds are predefined. For variables, there are ordinary variables *var*, also metavariables *mvar*, free variables *fvar*, bound variables *bvar*, scheme variables *svar*, and parameters *pvar*. For constants, there are ordinary *const*, definition names *dconst*, and reserved keywords *rconst*.

One piece of extra information which can be attached to atoms is their *full names*. This may be useful when concrete syntax uses symbols, or hides path names to components in substructures, for example.

---

[5] In Proof General at present, one can choose between retaining a history of all information messages, or just keeping the latest one. One can also choose to hide the information messages as soon as possible, or keep them around on the screen.

Remember that this is a markup language only: the kinds of atoms are *hints* of logical distinctions without any precisely defined concrete semantic notion understood by Proof General. Different proof systems may use these tags in different ways according to their underlying logical framework.

### 3.1.8 Symbols

Proof General at present can use the *X Symbol* package 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 PGML 1.0, we will follow the X Symbol names for symbols (which are close to names used in LaTeX). For example,

```
<sym name="forall" alt="ALL">
<sym name="longrightarrow" alt="-->">
```

would display a $\forall$ symbol and a $\longrightarrow$ symbol if they were available, otherwise use the ASCII alternatives `ALL` and `->`. The `alt` attribute is optional; the other alternative is to display an Isabelle-style escape sequence based on the X-Symbol name, like `\<forall>` or `\<longrightarrow>`.

This mechanism is somewhat simplistic, of course; it assumes symbols always correspond to some ASCII equivalent. For later versions of PGML, we may use mechanisms from other markup languages (MathML) for mathematical symbols.

### 3.1.9 Formatting

We specify that PGML markup is not completely free-form, but that multiple spaces are obeyed in most circumstances. Specifically, spaces are *ignored* when:

1. At the start of a line;
2. At the end of a line;
3. Between successive opening tags;
4. Between successive closing tags.

This may need to be changed to fit with markup language standards for XML/SGML, in which case we will introduce an explicit tag to cause multiple horizontal spaces.[6]

Carriage returns are always ignored; the <br> tag causes an explicit carriage return and new line.

Moreover, we expect for PGML 1.0 that if all the tags are stripped, then the text is plain 7-bit ASCII.[7] In future version we may allow for other character sets or encodings (beyond the symbol scheme mentioned above).

We do *not* require that the stripped version of the text necessarily makes sense to display, although that would be a pleasant aspect. At the moment, only the <sym> tag would require special treatment.

### 3.1.10 Summary

This markup scheme legitimizes the antiquated way "special" 8-bit characters are used presently as markup in Emacs Proof General. Moreover, compared with that markup scheme, PGML is richer. The display of proof states can have more structure; <type> and <action> elements are new, and the classification of variables and constants is less ad hoc.

[6] Eventually, using a table formatted output in the prover would be a better solution.

[7] There should be no problem with 8-bit characters but to begin with we want to avoid confusion because of their historical use in Proof General.

## 3.2 PGIP, the Interface Protocol

PGIP is a protocol for conducting interactive proof. It 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 text at once. More details are given in Section 3.2.1 below.

PGIP is implemented via XML message passing. Although XML was original invented as a format for documents, the suggestion to use it for communication protocols was soon made. An example is the XML-RPC [6] remote procedure call protocol; the a prominent example, which has been taken forward into the defacto standard for web services, SOAP [5]. For PGIP it is ideal, since our messages are typically small, but they may contain embedded PGML documents.

In the future, it might be extended to a web service, for example, by wrapping the PGIP messages in SOAP envelopes. A RELAX NG schema for PGIP messages is given in the appendix, Section A.1.[8]

### 3.2.1 Basic proof mechanism with PGIP

PGIP makes an abstraction that the proof assistant is in one of several states.

*[Unfinished]* [diagram to be inserted]

The states of PGIP correspond to the progress of a proof, assuming that:

- A proof begins by issuing some target *goal*
- A proof proceeds by issuing successive *proof steps*
- To reverse the effect of a proof step, the proof engine has a command to *undo*.
- A proof is closed, abandoned and revoked, or left unfinished 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*.[9]

Proof scripts may have additional elements outside proofs, for example, to make definitions or assumptions. (If definitions and assumptions are allowed inside proofs we just 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*.[10]

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 Proof General.

PGIP is a protocol, but for the moment we will only deal with the messages on each side of the protocol and only hint at the rules of the interaction are. That remains for later elaboration.

Also for the moment, we show PGIP also in terms of an XML markup scheme, and consider it to be an extension of PGML, in the sense that PGML texts may appear as messages of the protocol. (However, this occurs in only one direction: we do not expect the proof assistant to receive PGML text.) Later on we *may* wish to implement a different format for encoding the protocol, particularly when we consider a communication medium for PGIP other than stream based i/o (see Section 4.1). However, since PGIP is not expected to be bandwidth intensive, and in any case, there exist methods for compression-on-thy-fly, passing XML messages will probably be fine.

[8] July 2003: experiments with PGIP are underway; the appendix and comments therein currently supersede the description below.

[9] Although we call the object of our developments proof scripts, they may well be *declarative* rather than *procedural* descriptions of proofs. It matters little to Proof General whether they are forwards or backwards proofs, or whether they use tactics or some more readable description of proof steps.

[10] At the moment, Proof General allows such additional structure but usually doesn't understand it, which can result in confusion with undo. There is an outstanding unimplemented mechanism proposed to solve this.

### 3.2.2 Message packet

A PGIP message is sent in a "packet", which is an XML document with root <pgip>.

- <pgip version="*ver*"class="*cls*"origin="*orig*"systemid="*id*">*text*</pgip>

The optional version attribute describes the version of PGIP being used. The required class describes the intended receiver(s) of the message. Standard classes are "pg", standing for Proof General (for messages sent from the proof assistant), and "pa", standing for the proof assistant. The optional origin attribute is a tag which may be added to identify the sender of the message. Finally, the message may be tagged with an optional id identity tag.

Several messages may be enclosed in the same packet.

We do not say anything here about how the communication stream is established between Proof General and the proof assistant, nor how sychronization is achieved or errors and re-transmission is dealt with. Instead we assume for the moment that these low-level details are dealt with properly, so that messages are always free of transmission errors, and arrive in the sequence in which they were sent. Managing these aspects this will be the concern of PGCOM, see Section 4.1.

### 3.2.3 Proof and control commands

*[Unfinished]* [details of responses from prover; types of control command]

Here are the PGIP messages which may be sent *to* the proof assitant to control proof:

- <proofcmd >*command text*</proofcmd>
- <controlcmd >*command text*</controlcmd>

It is the responsibility of the proof assistant to tell Proof General what kind of proof command has been issued. On the other hand, Proof General knows about several different kinds of control commands.

### 3.2.4 Configuration messages

Configuration messages allow Proof General to configure itself to the proof assisant to which it is connected. They also allow some amount of configuration in the opposite direction, so Proof General can adjust the setup of the proof assistant in certain ways.

When Proof General is first connected to a proof assistant, configuration messages will be sent before anything else.[11] Proof General drives this configuration phase by sending queries to the proof assistant, which responds by advertising the facilities it has available. The configuration will take a short time, before the prover and Proof General agree that they are both ready for work; how much configuration occurs depends on the sophistication of the proof assistant and its Proof General interface.

Here are the PGIP configuration messages which may be sent *from* the proof assistant:

- <usespgml version="*ver*"/>
  Advertises the version of pgml this proof assistant will use.
- <haspref type="*tp*"default="*def*"class="*cls*"descr="*dsc*"name="*nm*"/>
  Advertises that the proof assistant has a preference setting, called name, which should be unique.
- <prefval name="*nm*">*value*</prefval>
  Advertises the current value of the named preference setting.

[11] At the moment we do not preclude configuration messages being sent at other times, although it is likely that restrictions may be made on some configuration messages when the protocol is specified in detail.

- <idtable class="*cls*">*table*</idtable>
  Advertises an identifier class "cls" with a completion table consisting of space-separated identifiers.

- <addid class="*cls*">*ident*</addid>
  Advertises the addition of an identifier ident in the given class.

- <delid class="*cls*">*ident*</delid>
  Advertises the removal of an identifier ident in the given class.

- <menuadd path="*p*"name="*nm*">*text*</menuadd>

- <menudel path="*p*"name="*nm*">*text*</menudel>

Here are the PGIP configuration messages which may be sent *to* the proof assistant:

- <askpgml/>
  Ask the proof assistant which version of PGML it supports

- <askprefs class="*cls*"/>
  Ask the proof assistant for a list of preferences it supports

- <resetprefs class="*cls*"/>
  Reset preferences to their defaults.

- <setpref name="*nm*">*value*</setpref>
  Set a preference.

- <getpref name="*nm*"/>
  Ask for the value of a preference

**PGML configuration** Proof General will begin by asking the prover which version of PGML it supports, by sending the <askpgml> message. The prover will advertise using a <usespgml> message.

**Preferences** The proof assistant may advertise *preference settings* using the element <haspref>. This has a mandatory type attribute. Types understood are "boolean", "nat", "int", "string", and "choice($c_1, \ldots, c_n$)" where $c_1$ to $c_n$ are identifiers. More types may be added later. The <haspref> element has optional attributes default for specifying a default value for the preference, descr for giving a description for what the preference controls, and class, to categorize the preference. Unless a class is given, it is assumed that the kind is "user", indicating a user preference. Other kinds may be given to specify "internal" preferences are not adjustable by the user, or perhaps "expert" settings which are only revealed to "expert" users.

The current value for a particular preference may be advertised with a <prefval> message.

With the <askprefs> message, Proof General can ask for a complete list of the preference settings available, optionally restricted to those in a given class.

Preferences are set and retrieved from Proof General using the <setpref> and <getpref> messages.

### 3.2.5 Status and error messages

Status messages output from the proof assistant are simply <information> elements, which have particular kinds. Similarly, error messages are <error> elements of particular kinds.

- <information kind="*kd*" location="*loc*">*message*</information>

  kind="message" — general information message to user

  kind="urgentmessage" — important status message to user

kind="display" — text in PGML

- <error kind="*kd*"location="*loc*">*message*</error>

  kind="warning" — warning message, non-fatal

  kind="fatal" — fatal error message, e.g. failed command

  kind="interrupt" — signals an interrupt occurred

The optional location tag specifies a position in a file or some previous input, which indicates the position of the cause of the information, error or whatever. The *loc* should be a URL.[12]

### 3.2.6 Proof blocks

*[Unfinished]* Basics:

- Set-position messages, with names.
- Open-block and close-block messages, perhaps with names.
- Usually, the first open-block will be followed by a goal.

Motivations: we may have sections in proof script, as in Coq or LEGO. One can undo to start of a particular section. We may have block structure in proof, as in Isar. Each block represents a particular subtree of the proof. The degenerate case is the present Proof General, where we only consider a top-level block. For nested blocks, perhaps one can undo within a block, or block by block, and even switch context between blocks. This might break the current fixed linear constraint of Proof General a bit, and needs an enhancement of script management to consider several blue regions within a buffer, rather than just beginning up to finished point. But it seems manageable.

Suggestions?

## 3.3 Interface configuration with PGIP

We would like to allow *interface configuration* as part of PGIP. At present, (some aspect of) the syntax of the command language and proof script language of each proof assistant is hard-wired inside Emacs Proof General. Extra menu entries and special commands are also hard-wired. We want to lift this kind of configuration and add it to PGIP, so that the interface components can configure themselves.

Proof General now has about 80 configuration variables. These are currently set in the prover-specific Emacs lisp files. Some examples of the settings for prover syntax and interface configuration are shown in Table 1.

| | |
|---|---|
| proof-comment-start | String which starts a comment in the proof assistant command language. |
| proof-terminal-char | Character which terminates a command. Added to every command |
| proof-assistant-home-page | Web address for information on proof assistant. |
| proof-info-command | Command to ask for help or information in the proof assistant. |
| proof-showproof-command | Command to display proof state in proof assistant. |

Table 1: Interface configuration in PGIP (incomplete)

For the first version of the self configuration part of PGIP, we will simply use the present Emacs regular expression syntax and substitution mechanisms (for example, using a string with % s as a placeholder). Later on we could replace with standard regular expressions, since the semantics of Emacs regular expressions is a bit fuzzy at the edges. Some of the configuration variables are actually *function* variables, defining operation hooks, so we will need to invent new ways of configuring

[12] But we need special extensions to allow line numbers and character positions (no present standard for such a thing?), and to give a special name for unsaved files kept in an editor's memory (e.g. for Emacs, a machine name, process id and buffer name).

those. Examining the present use of function hooks, it looks likely that many functions can be replaced by a generalised schema which can be customized. Other configurations can be reduced to a choice of pre-defined functions.

At present, the configuration variables of Proof General are classified as follows:

1. User options, including faces (i.e. colours and fonts)
2. Menu entries and user-level commands
3. Script settings (controlling the syntax of proof script files)
4. Shell settings (controlling the interface protocol)
5. Goal display (the markup of term output)

There are subgroups according function and a couple of other miscellaneous groups.

For Proof General Kit, some of these settings become fixed. The goal display configuration will correspond to PGML, and most of the shell settings will correspond with the core of PGIP. The remainder of the settings are for the self-configuration extension of PGIP. We will design a similar partitioning of the settings to the one used above, but making some more careful separations, bearing in mind the future intention that different components may be built which will only understand a subset of the configuration.[13]

The interface configuration part of PGIP is solves one of the original drawbacks of Proof General: the duplication of information in the proof assistant and the interface.

### 3.4 Script management with PGIP

*[Unfinished]* Ideas:

- Add script management for files into the protocol. Just as we have proof blocks, we could have file blocks for opening and closing files.
- Unsaved development buffer can be a special file
- Add a specific feedback from the proof assistant to indicate a command is processed, rather than relying on output which is a "non-error".
- Something to interface to configuration management systems?

For the feedback messages, we could decorate each proof command with a line positions in a file, or in the case of the development buffer, a marker. Using a marker instead of a line position allows for the case of several locked regions in a buffer corresponding to subtrees of proofs, and then the possibility of editing a subproof which appears above the one currently being processed.

## 4 Components for Proof General Kit

The second phase for Proof General Kit is to build and connect together components which communicate using PGIP. This involves using some transport level for sending PGIP messages, as well as the infrastructure for connecting components together across this transport. A central component here is the mediator.

We also want to add more features to PGIP in this stage of the project, to implement some of the extensions that people have suggested or wanted for Proof General over the years.

[13]For example, Proof General presently overloads `proof-terminal-char` to mean both a character which terminates proof engine control commands, and a character which appears in the syntax of proof scripts to separate proof commands. Overloadings like this should be removed, to allow more freedom for proof script languages.

### 4.1 PGCOM, the Communication Layer

*[Unfinished]* Some ideas and notes:

- Want to be network transparent, should run across internet. In that case will need mechanisms for identification and authentication, but would hope to lift these from other tools (e.g. ssh?).
- Perhaps use sockets.
- Want Microsoft Windows compatibility.
- Component-based architecture.
- What are the possible frameworks to use?
  - CORBA — Markus suggests this is too heavyweight and complicated to expect people to understand. But maybe this objection goes away if we provide easy hooks into using it transparently, such as PGMEX.
  - COM or its successor.
  - Others?

Please make suggestions and comments here.

### 4.2 PGFILT, the Filter

The PGFILT component translates the interaction protocol used by a particular proof assistant into PGIP. It it used to connect systems which do not (yet) speak PGIP natively. Each such system needs a specific filter, which may be implemented in any suitable language.

For systems which already have support in Proof General, one possibility is to implement the filter inside Emacs using the settings which are already there. But this could lead to some confusion! A better idea is probably to translate the Emacs settings into Perl (or some other scripting language), and implement a configurable filter there.

### 4.3 PGIP for Emacs Proof General

As PGFILT is implemented and customized for one or more proof assistants, we can implement PGIP inside Emacs Proof General, as the first example interface component to use PGIP.

This implementation will consist largely of a new instantiation of Emacs Proof General, that is, some "prover specific" Emacs Lisp which in fact configures for the standard protocol PGIP. This prover-specific Elisp will suffice for the core of PGIP and the extension to script management. For the extension to self-configuration, a new class of messages must be added. This will also be a straightforward adaptation of the current implementation.

The other important aspect is interpreting PGML and displaying it inside Emacs. (It may be appropriate to use other tools to help with that).

### 4.4 PGMEX, the Multiplexer

The PGMEX component is intended to connect existing proof assistants into the new communication framework PGCOM, by translating between the message-passing communication mechanism and the stream-based command line interfaces.

New proof assistants may choose to implement PGCOM directly, others may move to it if it proves successful. But we expect to use PGMEX components in many cases.

We will need a PGMEX instance for each proof assistant which is connected, but since we have a standard language and protocol by now, the PGMEX component does not need to be configured specially itself. It may connect to a different PGFILT for each proof assistant.

### 4.5 PGDISP, the Displayer

PGDISP will be the first new application (or applet) which displays the status of the proof as it proceeds. Primarily, this involves rendering the PGML transmitted for each proof state display. The display application could be more flexible than just displaying, however. It ought to allow the copying action of cut-and-paste, proof-by-pointing actions, and perhaps extra features such as printing or saving the current display, and caching a history of displays.

One way to implement PGDISP would be inside a web browser, using an embedded scripting language with an interface to PGCOM (e.g. PHP), or perhaps a CGI script. In the latter case, some way to trigger display and refreshes would be needed. The job of rendering PGML in this case reduces to translation to HTML.

Another possibility would be to implement PGDISP directly as a Java application or KDE application; this is reasonably feasible as the markup language PGML is quite restricted. But it might not allow for future improvements so easily.[14]

Finally, although a brand new display application would be nice, we plan to maintain the Emacs based one, making it more modular now that we have introduced PGCOM.

Since the display is mostly passive, several displays could be connected at once (perhaps remotely).

### 4.6 New Features for Proof General Kit

There are several features not found in Proof General that we would like to integrate into Proof General Kit. Here is a brief outline of some of them.

#### 4.6.1 Theory browser

Proof General currently has limited mechanisms for helping the user find theorems and definitions during a proof. It has notion of displaying a “current context” for a proof, and configuration with a proof engine command for searching for theorems. It would be useful to extend these facilities with a *theory browser* for investigating the theories available, by querying a running proof assistant.

There is a conceptual distinction over whether the querying examines the filesystem to find theories, or is limited to the theories which have been loaded into the proof assistant’s memory in the present session. Proof General can safely ignore this distinction and leave the issue to the underlying proof assistant.

Work here involves designing a generic extension of PGIP for theory browsing, and perhaps extra markup components in PGML to display theories. Then the PGDISP component could be augmented or specialised to display theories.

Inside Emacs, a nice way to fit in theory browsing might be to use a `dired`-like buffer, by instantiating the virtual file system mechanism of `dired`. Note that theories usually have a dag-like structure rather than a tree-like structure. We could incorporate theories into a virtual file system viewer like `dired` by using a virtual sub-directory for each of the parent theories (in plase of `..`), and a special sub-directory `children` for the child theories of a theory (if the children are directly available in the system).

#### 4.6.2 Completion mechanism

Some proof assistants rely on the user remembering and typing long names for theorems, constants, or tactics. It would be useful to introduce a *completion table* mechanism to provide completion and selection menus for these names.

One way to do this would be to add transmission of completion tables to PGIP, treating it as an aspect of the interface configuration. But it is important that it should be dynamic, and that there is way of adding and removing names incrementally.

[14] For example, one can imagine adding support for the “flag” style of natural deduction proof output, which would need support for tables and hence more advanced layout engines.

Completion may also need to be context-sensitive, if the system uses several name spaces. In case parsing proof scripts is too difficult and not included in Proof General, an approximate way to deal with this would be to consider just the flattened name space for completion. There is no problem with selection from menus, the different namespaces could still be reflected in the interface.

#### 4.6.3 Favourites and history mechanisms

The user should be able to record his or her own commonly used proof commands, and select them from a menu. Additionally, a history mechanism for repeating an earlier proof step would be useful.[15]

The favourites and history mechanisms are purely interface issues.

#### 4.6.4 Configurable menus and toolbar

An important extension of the interface configuration in PGIP would be to allow configurable menus for the proof assistant.

At present, a default menu and toolbar is provided for Proof General, with little or nothing that is proof assistant specific. There is some strength in this poverty: it enforces the generic feel to the interface. But we recognize that in practice there are many useful commands and settings which are particular to the underlying engine, and that to be a good interface for the engine, Proof General should provide better access to them.[16]

One particularly easy (but low-level) way of adding menu configuration to PGIP would be to encode calls to the Emacs functions `add-menu` and `remove-menu`.

## 5 Abstract syntax and pretty-printing

The third phase of Proof General Kit will considerably widen the scope of Proof General, incorporating more features into the generic setting. Because we propose to add some features here which are implemented already in existing underlying proof engines, the components built in this phase may be more appropriate for new proof assistants in development, or ones which are weak in some area.

There is less re-engineering here and more fresh work than in the first two phases, since the programs needed perform functions that are not done at present in Proof General. But a starting point for implementing some of these would be to borrow good implementations from existing theorem proving systems.

By now, it is accepted that can be a good idea to separate a user interface for a proof assisstant (or other tool) from its underlying proof engine. This means a choice of interfaces can be provided, and it allows the implementor to focus on the logical and computational aspects of the proof engine using a language which is appropriate for that task, but maybe less appropriate for constructing a GUI. The question is: where should the division be? How much is a question of interface and how much should be deferred to the underlying proof engine?

We answer that question here by suggesting that more of the interface machinery should be pushed from the proof engine into the interface. But, because it is a design goal of Proof General to do as much as possible using *lightweight protocols*, we want to be careful before rushing in to do too much at this level, and avoid transmitting semantical information when it's not needed. This requires careful management of complexity and genericity, achieved using both design and experimentation.

[15] Proof General actually has some history mechanisms at the moment, but they ought to be more available and easier to use. One mechanism allows copying proof commands by a single click, higher up in a proof. Another mechanism allows repetition of previous non-proof commands, via the standard Emacs prompt history mechanism.

[16] One could always provide a way of switching off the engine specific menus, perhaps to provide a simpler interface for beginners.

### 5.1 PGTERM, the Term Representation

To begin with, we need to decide on a representation format for terms in the abstract syntax of a proof language.

Possible starting points:

- OpenMath, see `http://www.nag.co.uk/projects/openmath/omsoc/`
- MathML, see `http://www.w3.org/Math/`

Having decided the format for representing terms, we need to decide which terms are to be transmitted in this way. A starting point will be simply to take the part of PGML before which was annotated as concrete syntax for terms, with the <term> element. Note that in this case, the abstract syntax may already be enriched with annotations from the proof assistant providing hooks for proof-by-pointing, display of types, etc.

### 5.2 PGSYN, the Syntax Representation

To allow display of the terms, we must have some way of specifying the concrete syntax desired. The format for doing this is called PGSYN.

It isn't clear where we want to use PGSYN. But it seems likely that it must be integrated with the proof assistant since we may desire context-sensitive concrete syntax, when syntax is attached to particular theories (as in Isabelle).

### 5.3 PGPP, the Pretty Printer

PGPP takes syntax representations in PGSYN and terms in PGTERM and renders them into PGML.

PGPP will alleviate proof systems from implementing concrete syntax, and it will automatically implement term-structure markup used for proof-by-pointing and similar features.

### 5.4 PGDE, the Display Engine

We have the mechanisms for outputting abstract syntax (PGTERM), representation of concrete syntax (PGSYN), and printing from the two (PGPP). The last piece is to connect everything together, and for that we use the display engine, PGDE.

The job of the display engine is to sit following the PGIP communication (which must be extended to cope with the transmission of syntax specifications), and control the generation of PGML as necessary, because the display component(s) PGDISP only understand PGML.

One possibility is to use a display engine for each display. This makes sense because different displays may have different aspects. (We might even use voice rendering or natural language output as a "display"). The PGDISP component must become capable of informing others of its capabilities.

On the other hand, the job of pretty-printing may be fairly expensive, so another possibility is to retransmit the marked-up concrete syntax, so that several displays can be connected without needing to duplicate the pretty-printing effort. But then per-display customization of the pretty-printer is more difficult.

### 5.5 Other term-based features

Once Proof General has some access to the abstract structure of terms, it opens the way to many useful features, based on examining or transforming the trees.

However, to do anything really exciting, we must take another leap and attach some meaning to the terms. This happens in the final phase.

# 6 Logic Genericity

This phase of development approaches the pure idea behind Proof General: that one can have a generic front-end for conducting proof, connected to an *arbitrary* back-end proof assistant. The front-end hides the specifics of the proof assistant as far as possible, by using a *common logic* and a *common proof language*.

This phase is more tentative than the others and involves more research and much greater implementation efforts, really a project in itself. It may be possible or appropriate to link with one of the other projects in the theorem proving community (see Section 7.1).

## 6.1 Logic General

Logic General will be a family of "standard" logics (syntax and proof rules) which have adequate embeddings from the logics of each of the proof assistants we're interested in. For instance, HOL Logic General (HOL-LG for short) would be a version of higher-order logic with a particular syntax and set of proof rules. To use HOL-LG with Isabelle, we must provide an effective mapping into the syntax of Isabelle/HOL, and a careful argument that any proof in Isabelle/HOL could be mapped onto a proof in HOL-LG. Similarly for any other proof assistant we wish to use.

In the first instance we will be concerned with provability in a particular system, rather than proofs, since we consider the *proof text* constructed in Proof General to be the primary justification. Later on, we could consider an extension to construct *proof objects* which could be validated by a small and efficient independent proof checker.

## 6.2 Proof languages for Logic General

Logic General will be provided with one or more *proof languages* for conducting proofs. Just as a logic of Logic General must be connected to the underlying proof assistant logic, we must map the proof language onto the underlying proof commands of the proof assistant.

*[Unfinished]*

- Idea to have both "traditional" goal-tactic-qed scripts and "vernacular" scripts which look like read readable proofs.
- Wenzel's Isar may be a candidate.

# 7 Conclusions and related work

I have described a plan for Proof General Kit, a design for a open architecture of communicating *interactive* proof components, with an emphasis on constructing *proof scripts* as the target of proof developments.

So far, there is not yet a complete implementation of Proof General Kit. However, various experiments have been made, including the addition of PGML support to Isabelle, and the on-going development of an graphical interface based on ideas from the IsaWin interface [3].

Proof General Kit seeks to be an open, international collaborative effort. In the same way that Emacs Proof General was built by recruiting an experienced user for each of the supported systems, we hope to draw on the expertise of both implementors and users of each the currently supported systems. Additionally, we welcome new collaborators with or without system-specific affiliations.

## 7.1 Related Projects, past and present

There are several projects which are loosely related to Proof General Kit, especially in some of the later stages suggested above. These include:

**OMEGA** `http://www.ags.uni-sb.de/~omega/`

OMEGA connects together various theorem proving systems, using an extension of OpenMath called OMDoc, which adds additional semantical content to OpenMath's content dictionaries. The architecture is based on translations between native tool syntax and OMDoc documents, based on XML. Translations between the logics of different systems are supported by relativizations.

*(By contrast with OMEGA, Proof General Kit begins with less ambition, and aims to connect together systems from the bottom up. The later phases of Proof General Kit which are only touched upon here begin to approach ideas used in OMEGA, and indeed, it may be appropriate to use some of the OMEGA technology in achieving our aims there.)*

**Prosper** `http://www.dcs.gla.ac.uk/prosper/`

Prosper was an Esprit IV collaboration primarily between Glasgow, Cambridge, and Edinburgh. The resulting Prosper Toolkit allows various theorem proving tools (e.g. model checkers, BDD packages) to communicate data with the HOL theorem prover. Prosper is an enabling technology to allow engineers to build custom theorem proving systems by connecting different systems together.

*(Prosper and Proof General Kit address different domains; whereas Prosper helps with building custom theorem provers, Proof General Kit helps with interacting with theorem provers.)*

**OMRS** `http://www.mrg.dist.unige.it/omrs/index.html`

A project to support the construction of reasoning systems incrementally. The architecture of a system built this way is specified in terms of a logic part, a control part (describing inference strategies) and an interaction part (describing client-server interactions).

*(Proof General Kit is probably most closely related to the interaction part of an OMRS, which the OMRS project hasn't addressed much so far.)*

**HELM** `http://helm.cs.unibo.it/`

HELM aims to integrate tools for the automation of formal reasoning and the mechanization of mathematics (proof assistants and logical frameworks) with the most recent technologies for the development of web applications. Work so far has included the *Coq-on-line* effort of producing XML versions of Coq libraries which can be navigated with a web borwser.

Apart from projects specifically to do with theorem proving, there are some more general projects underway which have a broader scope, applying to many mathematics-related activities on computer:

**OpenMath** `http://www.nag.co.uk/projects/openmath/omsoc/`

OpenMath is a project instigated by the computer algebra community, as an attempt to define standards for representing mathematical objects on the web. Individual symbols (e.g. `RR`) are given by a name and a *content dictionary* from which they are taken (e.g. `Real`). The content dictionary can contain textual descriptions of the symbols meaning. There are some standard content dictionaries.

**MathWeb** `http://www.mathweb.org`

MathWeb follows the approach of OMEGA, using OMDoc and an XML-RPC protocol as enabling mechanisms to facilitate connections of mathematical tools to the web.

**MathML** `http://www.w3.org/Math/`

MathML is a markup language for mathematical text. It contains constructs to allow the description of display formulae and symbols. Plugins for browsers are beginning to appear.

There are a couple of aspects which set Proof General Kit apart from the projects already underway in the theorem proving community. First, we are aiming to do something slightly different. We take *proof scripts* and *interactive development* as our primitive concepts; then we build protocols and software to enable user-interaction with theorem proving systems. The second thing that sets Proof General Kit apart is that we (at least initially) address a different sub-community from some

of these projects. It is an unfortunate fact of life that the theorem proving community is somewhat fragmented at the moment. However, there are signs that this is changing and we hope that interoperability projects such as those above, and Proof General Kit, will bring various groups and tools closer together.

### 7.2 Credits

I'm grateful for feedback from past and present Proof General users and developers, some of whose ideas have found their way into this white paper. Particular mention is due to Paul Callaghan, Pierre Courtieu, Christoph Lüth, and Markus Wenzel. I wrote much of this paper in February 2000 while visiting ETL, Osaka, Japan, and I'm grateful to the people I spoke with when visiting Tokyo, Kobe and Kyoto and at the Arima Onsen Workshop, for stimulating discussions about Proof General.

I welcome ideas or suggestions for improvements. Send mail to David.Aspinall@ed.ac.uk.

## References

[1] D. Aspinall, H. Goguen, T. Kleymann, and D. Sequeira. Proof General, 2001. System documentation, see http://www.proofgeneral.org.

[2] David Aspinall. Proof General: A generic tool for proof development. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1785, pages 38–42, 2000.

[3] David Aspinall and Christoph Lüth. Proof General meets IsaWin. Informal proceedings of Workshop on User Interface for Theorem Provers, UITP'03, September 2003.

[4] OASIS RELAX NG technical committee. RELAX NG Specification, December 2001. http://relaxng.org/spec-20011203.html.

[5] Simple Object Access Protocol (SOAP) 1.1. W3C Note, May 2000. See http://www.w3.org/TR/SOAP/.

[6] Dave Winer. XML-RPC Specification. Published on the web site of UserLand Software, Inc., June 1999. See http://www.xmlrpc.com/spec.

# A Schemas for PGIP and PGML

## A.1 pgip.rnc

```
#
# RELAX NG Schema for PGIP, the Proof General Interface Protocol
#
# Authors:  David Aspinall, LFCS, University of Edinburgh
#           Christoph Lueth, University of Bremen
#
# Version: $Id: pgip.rnc,v 1.3 2003/07/09 17:42:33 da Exp $
#
# Status:  Experimental.
# For additional commentary, see the Proof General Kit white paper,
# available from http://www.proofgeneral.org/kit
#
# Advertised version: 1.0
#

# ========== PGIP MESSAGES ==========

pgip = element pgip {                           # A PGIP packet contains:
   pgip_attrs,                                  # attributes with header information;
   (provermsg                                   # either a message sent TO the prover,
    | kitmsg)}                                  # or an interface message

pgip_attrs =
  attribute origin { text }?,                   # name of sending PGIP component
  attribute id { text },                        # session identifier for component process
  attribute class { pgip_class },               # general categorization of message
  attribute refseq { xsd:positiveInteger }?,    # message sequence this message responds to
  attribute refid { text }?,                    # message id this message responds to
  attribute seq { xsd:positiveInteger }         # sequence number of this message


pgip_class = "pa"  # for a message sent TO the proof assistant
           | "pg"  # for a message sent TO proof general
           | string  # something else

provermsg =
    proverconfig    # query Prover configuration, triggering interface configuration
  | provercontrol   # control some aspect of Prover
  | proofcmd        # issue a proof command
  | filecmd         # issue a file command

kitmsg =
    kitconfig       # messages to configure the interface
  | proveroutput    # output messages from the prover, usually display in interface
  | fileinfomsg     # information messages concerning




#  ========== PROVER CONFIGURATION ==========

proverconfig =
    askpgip         #  what version of PGIP do you support?
  | askpgml         #  what version of PGML do you support?
  | askconfig       #  tell me about objects and operations
  | askprefs        #  what preference settings do you offer?
```

```
57    | setpref         #  please set this preference value
58    | getpref         #  please tell me this preference value
59
60  askpgip   = element askpgip   { empty }
61  askpgml   = element askpgml   { empty }
62  askconfig = element askconfig { empty }
63  askprefs  = element askprefs  { class_attr? }
64  setpref   = element setpref   { name_attr, class_attr?, text }
65  getpref   = element getpref   { name_attr? }
66
67
68  #  ========== INTERFACE CONFIGURATION ==========
69
70  kitconfig =
71       usespgip       # I support PGIP, version ..
72     | usespgml       # I support PGML, version ..
73     | haspref        # I have a preference setting ..
74     | prefval        # the current value of a preference is
75     | addids         # inform the interface about some known objects
76     | delids         # retract some known identifers
77     | menuadd        # add a menu entry
78     | menudel        # remove a menu entry
79     | guiconfig      # configure the following object types and operations
80
81  # version reporting
82  version_attr  = attribute version { text }
83  usespgml = element usespgml  { version_attr }
84  usespgip = element usespgip  { version_attr }
85
86  # Types for config settings
87  pgipbool   = element pgipbool { empty }
88  pgipint    = element pgipint  { empty }
89  pgipstring = element pgipstring { empty }
90  pgipchoice = element pgipchoice { pgipchoiceitem+ }
91  pgipchoiceitem = element pgipchoiceitem { tag_attr, text }
92  tag_attr = attribute tag { text }?
93
94  # preferences
95  default_attr = attribute default { text }
96  descr_attr   = attribute descr { text }
97  class_attr   = attribute class { text }
98  name_attr    = attribute name { text }
99
100 # FIXME: change haspref to empty
101 type_attr = attribute type { text }
102 haspref = element haspref  { type_attr?, default_attr?, descr_attr?, class_attr?, text }
103 prefval = element prefval  { name_attr, text }
104
105 # menu items (incomplete)
106 path_attr = attribute path { text }
107
108 menuadd = element menuadd  { path_attr?, name_attr?, text }
109 menudel = element menudel  { path_attr?, name_attr?, text }
110
111 # GUI configuration information: objects, types, and operations
112
113 # da: here we may want to require that certain standard operations
114 # are available: e.g. to construct a proof operation for opening a
115 # new theory.  We might even assume that the proof control and
116 # file commands described later are given using guiconfig.
117 # (perhaps guiconfig could be "uiconfig"?)
```

```
118  #
119  # da 1.12: I've changed objtype to include icon data optionally inside
120  # another element, as well as a descr element which could be used as
121  # a balloon popup hint for the object type.
122
123  icon  = element icon { text }   # FIXME: xsd type for icondata: ?
124  descr = element descr { text } # FIXME: could set max string length
125
126  objtype = element objtype { name_attr, icon?, descr? }
127  objtype_attr = attribute objtype { text }  # the name of an objtype
128
129  opsrc = element opsrc { text } # source types: a space separated list
130  optrg = element optrg { text } # the single target type
131  opcmd = element opcmd { text } # prover command, with printf-style "%1"-args
132
133  opn = element opn { name_attr, opsrc, optrg, opcmd }
134
135  # proof operations (no target sort: result is a proofcmd for script)
136  proofopn = element proofopn { name_attr, opsrc, opcmd }
137
138  # interactive operations - require some input
139  iopn = element iopn { name_attr, inputform, opsrc, optrg, opcmd }
140  inputform = element inputform { field+ }
141
142  # a field has a PGIP config type (int, string, bool, choice(c1...cn))
143  # and a name; under that name, it will be substituted into the command
144  # Ex. field name=number  opcmd="rtac %1 %number"
145
146  # the PCDATA is the prompt for the input field
147  field = element field { type_attr, name_attr, text }
148
149  guiconfig =
150    element guiconfig { objtype*, opn*, iopn*, proofopn* }
151
152
153  # identifier tables: these list known items of particular objtype.
154  # Might be used for completion or menu selection.
155  # May have a nested structure (but objtypes do not).
156
157  addids = element addids { objtype_attr, (identifier | idtable)* }
158  delids = element delids { objtype_attr, (identifier | idtable)* }
159
160  identifier = element identifier { text }
161
162
163
164
165  # ========== PROVER CONTROL ==========
166
167  provercontrol =
168      proverinit     # reset prover to its initial state
169    | proverexit     # exit prover
170    | startquiet     # stop prover sending proof state displays, non-urgent messages
171    | stopquiet      # turn on normal proof state & message displays
172
173  proverinit = element proverinit { empty }
174  proverexit = element proverexit { empty }  # exit prover
175  startquiet = element startquiet { empty }
176  stopquiet  = element stopquiet  { empty }
177
178
```

```
179 # ========== GENERAL PROVER OUTPUT ==========
180
181 proveroutput =
182     ready                   # prover is ready for input
183   | cleardisplay            # prover requests a display area to be cleared
184   | normalresponse          # prover outputs some display text
185   | errorresponse           # prover indicates an error condition, with error message
186   | metaresponse            # prover outputs some meta-information to interface
187
188 ready = element ready { empty }
189
190 displayarea = "message"                      # the message area (response buffer)
191             | "display"                      # the main display area (goals buffer)
192
193 cleardisplay =
194     element cleardisplay {
195        attribute area {
196           displayarea | "all" } }
197
198
199 include "pgml.rnc"                           # include PGML documents
200
201 displaytext = (text | pgml)*                 # grammar for displayed text
202
203 normalresponse =
204   element normalresponse {
205     attribute area { displayarea },
206     attribute category { text }?,            # optional extra category (e.g. tracing)
207     attribute urgent { "y" }?,               # whether to refresh display
208     displaytext
209 }
210
211 fatality = "nonfatal" | "fatal" | "panic"  # degree of errors
212
213 errorresponse =
214    element errorresponse {
215      attribute fatality { fatality },
216      attribute location { text }?,
217      attribute locationline { xsd:positiveInteger }?,
218      attribute locationcolumn { xsd:positiveInteger }?,
219      displaytext
220   }
221
222 metaresponse =
223    element metaresponse {
224       attribute infotype { text },       # categorization of data
225       text }
226
227
228 # ========== PROOF CONTROL COMMANDS ==========
229
230 proofcmd =
231      goal           # open a goal in ambient context
232    | proofstep      # a specific proof command (perhaps configured via opcmd)
233    | undostep       # undo the last proof step issued in currently open goal
234    | closegoal      # complete & close current open proof (succeeds iff goal proven)
235    | abortgoal      # give up on current open proof, close proof state, discard history
236    | giveupgoal     # close current open proof, record as proof obl'n  (sorry)
237    | postponegoal   # close current open proof, retaining attempt in script (oops)
238    | forget         # forget a theorem (or named target), outdating dependent theorems
239    | restoregoal    # re-open previously postponed proof, outdating dependent theorems
```

```
240
241 thmname_attr = attribute thmname { text }   # theorem names
242 aname_attr   = attribute aname { text }     # anchors in proof text
243
244 goal         = element goal { thmname_attr, text }       # text is theorem to be proved
245 proofstep    = element proofstep { aname_attr?, text }   # text is proof command
246 undostep     = element undostep { empty }
247
248 closegoal    = element closegoal { empty }
249 abortgoal    = element abortgoal { empty }
250 giveupgoal   = element giveupgoal { empty }
251 postponegoal = element postponegoal { empty }
252 forget       = element forget { thyname_attr?, aname_attr? }
253 restoregoal  = element restoregoal { thmname_attr }
254
255
256 # ========== THEORY/FILE HANDLING COMMANDS ==========
257
258 filecmd =
259     loadtheory      # load a file possibly containing a theory definition
260   | opentheory      # begin construction of a new theory.
261   | closetheory     # complete construction of the currently open theory
262   | retracttheory   # retract a theory.  Applicable to open & closed theories.
263   | openfile        # lock a file for constructing a proof text
264   | closefile       # unlock a file, suggesting it has been processed
265   | abortfile       # unlock a file, suggesting it hasn't been processed
266
267 fileinfomsg =
268    informfileloaded       # prover informs interface a particular file is loaded
269  | informfileretracted    # prover informs interface a particular file is outdated
270
271 url_attr     = attribute url     { text }       # typically: filename
272 thyname_attr = attribute thyname { text }       # a corresponding theory name
273
274 loadtheory    = element loadtheory    { url_attr?, thyname_attr? }
275 opentheory    = element opentheory    { thyname_attr, text }
276 closetheory   = element closetheory   { thyname_attr }
277 retracttheory = element retracttheory { thyname_attr }
278 openfile      = element openfile      { url_attr }
279 closefile     = element closefile     { url_attr }
280 abortfile     = element abortfile     { url_attr }
281
282 informfileloaded =
283    element informfileloaded    { thyname_attr, url_attr }
284 informfileretracted =
285    element informfileretracted { thyname_attr, url_attr }
```

### A.2 pgml.rnc

```
1  #
2  # RELAX NG Schema for PGML, the Proof General Markup Language
3  #
4  # Authors:  David Aspinall, LFCS, University of Edinburgh
5  #           Christoph Lueth, University of Bremen
6  # Version: $Id: pgml.rnc,v 1.2 2003/07/09 17:37:22 da Exp $
7  #
8  # Status:  Complete but experimental version.
9  #
10 # For additional commentary, see the Proof General Kit white paper,
11 # available from http://www.proofgeneral.org/kit
12 #
```

```
13 # Advertised version: 1.0
14 #
15
16 pgml_version_attr = attribute version { xsd:NMTOKEN }
17
18 pgml =
19   element pgml {
20     pgml_version_attr?,
21     (statedisplay | termdisplay | information | warning | error)*
22   }
23
24 termitem      = action | nonactionitem
25 nonactionitem = term | pgmltype | atom | sym
26
27 pgml_name_attr = attribute name { text }
28
29 kind_attr = attribute kind { text }
30 systemid_attr = attribute systemid { text }
31
32 statedisplay =
33   element statedisplay {
34     pgml_name_attr?, kind_attr?, systemid_attr?,
35      (text | termitem | statepart)*
36   }
37
38 br = element br { empty }
39 pgmltext = (text | termitem | br)*
40
41 information =
42   element information { pgml_name_attr?, kind_attr?, pgmltext }
43
44 warning     = element warning     { pgml_name_attr?, kind_attr?, pgmltext }
45 error       = element error       { pgml_name_attr?, kind_attr?, pgmltext }
46 statepart   = element statepart   { pgml_name_attr?, kind_attr?, pgmltext }
47 termdisplay = element termdisplay { pgml_name_attr?, kind_attr?, pgmltext }
48
49 pos_attr = attribute pos { text }
50
51 term = element term { pos_attr?, kind_attr?, pgmltext }
52
53 # maybe combine this with term and add extra attr to term?
54 pgmltype = element type { kind_attr?, pgmltext }
55
56 action = element action { kind_attr?, (text | nonactionitem)* }
57
58 fullname_attr = attribute fullname { text }
59 atom = element atom { kind_attr?, fullname_attr?, text }
60
61 alt_attr = attribute alt { text }
62 sym = element sym { pgml_name_attr?, alt_attr?, text }
```