Proof General meets IsaWin

Combining Text-Based And Graphical User Interfaces

David Aspinall [1]

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

Christoph Lüth [2]

Department of Mathematics and Computer Science, Universität Bremen

Abstract

We describe the design and prototype implementation of a combination of theorem prover interface technologies. On one side, we take from Proof General the idea of a prover-independent interaction language and its proposed implementation within the PG Kit middleware architecture. On the other side, we take from IsaWin a sophisticated graphical metaphor using direct manipulation for developing proofs. We believe that the resulting system will provide a powerful, robust and generic environment for developing proofs within interactive proof assistants that also opens the way for studying and implementing new mechanisms for managing interactive proof development.

1 Introduction

Proof General is a generic interface for interactive proof assistants, built on the Emacs text editor [4,7]. It has proved rather successful in recent years, and is popular with users of several theorem proving systems. Its success is due to its genericity, allowing particularly easy adaption to a variety of provers (primarily, Isabelle, Coq, and LEGO), and its design strategy, which targets experts as well as novice users. Its central feature is an advanced version of script management, closely integrated with the file handling of the proof assistant. This provides a good work model for dealing with large-scale proof developments, by treating them similarly to large-scale programming

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

developments. Proof General also provides support for high-level operations such as proof-by-pointing, although these are less emphasised.

Proof General has some drawbacks, however. From the users' point of view, although the interface offers some high-level operations and tries to hide the shell-window process, interaction is still firmly based on editing text written in the proof assistant's (often cryptic) command language. While there are some menu entries and toolbar buttons to help initiate and complete a proof, there is little in the way of hints or templates to help in constructing the bulk of tactics and declarations used in writing proof scripts. From the developers' point of view, Proof General is rather too closely tied with the Emacs Lisp API which is somewhat unreliable, often changing, and exists in different versions across different flavours of Emacs.

IsaWin is the instantiation of a generic graphical user interface to Isabelle [24,23,25]. It aims at providing an abstract user interface based on a persistent visualisation metaphor and the concept of direct manipulation. Because tactic applications are generated automatically, users can be less concerned with the syntactic idiosyncrasies of the prover and can instead concentrate on the logical content of the proof. This abstract approach also allows high-level operations: for example, there is support for proof-by-pointing (folding or unfolding equations), term annotations (the system can display the type of sub-terms), or tactical programming (one can cut parts from the history of a proof to make it into an elementary tactic, which can be reapplied elsewhere).

While IsaWin is usually met with initial approval, in particular from theorem proving novices, it has some drawbacks. From the users' point of view, it has a rudimentary management of target proof scripts, and integrates foreign proof scripts only reluctantly. From the developers' point of view, customising or adapting it to other proof assistants requires Standard ML programming, and a good understanding of the data structures, so it is not possible to build a custom version gradually (unlike with Proof General), and it is hard to connect to provers not implemented in Standard ML.

Thus, Proof General and IsaWin can be considered as complimentary, with each offering advantages to compensate for the other's shortcomings. This paper describes an attempt to combine their advantages in one system based on the PG Kit infrastructure presented in [5,6]. The resulting system has an event-based architecture and focuses on managing proof scripts as the central underlying artefact. A generic interactive protocol language allows one to connect different user interfaces (called *displays*); thus, IsaWin becomes one particular user interface in this setting.

In the remainder of this paper, we describe the particular aspects of Proof General and IsaWin that we want to build on (Sections 2 and 3), followed by the design of the new system (Section 4) and a description of the prototype implementation (Sections 5 and 6). We close by briefly mentioning related work, and our vision for the future evolution of the project.

2 Proof General Basic Concepts

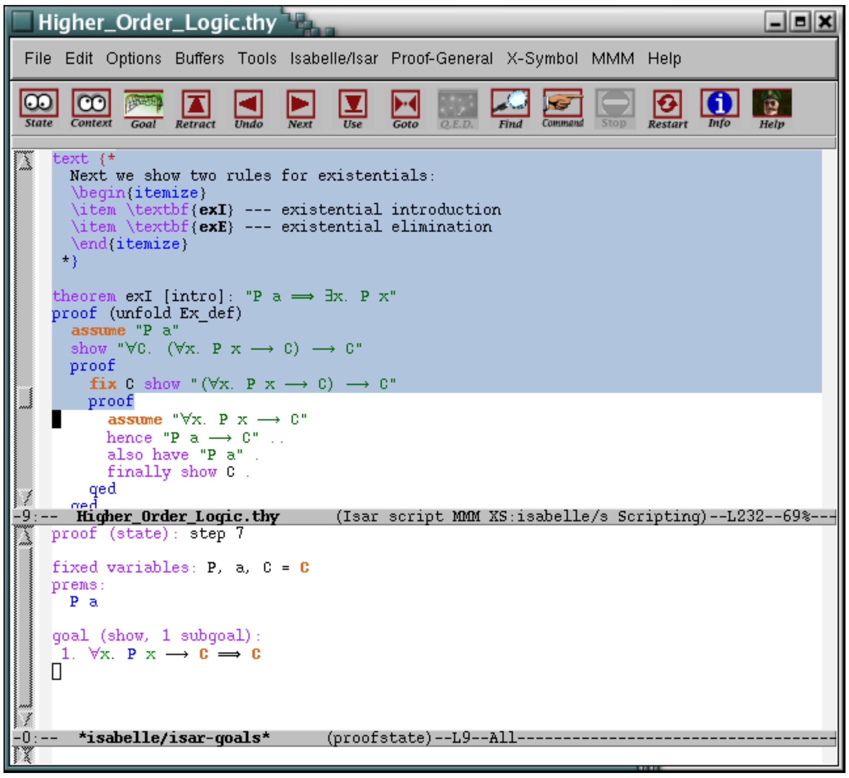
Emacs Proof General is the present incarnation of Proof General, based on the ubiquitous Emacs text editor.[3] For Proof General, a proof consists of the user-editable formal text (or *proof script*) which, when processed by the machine, either constructs a representation or checks for the existence of a proof in a formal system. The target proof script is the central focus of development.

It doesn't matter whether the proof script contains a tactic-style procedural proof, or a declarative proof description. The interface relies on the underlying proof assistant being able to process a proof script interactively and incrementally, in a textual dialogue: the user issues a proof command, and the system responds. A distinction is made between *proper* proof script commands, which belong in the text, and *improper* commands, which do not. Proper commands include statements of propositions to be proved, and the contents of their proofs. Improper commands include undo steps, commands for inspecting terms and for querying a database of available theorems. The proper commands are stored in the proof script file, and coloured according to progress in the proof: crucially, regions which have been processed by the proof assistant are coloured blue and should not be edited. This is the idea of *script management* as described in [12].

Proof General provides a simple browsing metaphor for replaying proofs, via a toolbar for navigating in a proof script, see Fig. 1 for a screenshot. Behind the scenes, this works by sending commands to issue proof steps and undo proof steps. The undo behaviour relies on the built-in history of the proof assistant, which typically forgets the history between steps of a proof once the proof has been completed, i.e. only the whole proof can be undone then, not single steps of it.[4] The undo behaviour splits the proof script into regions (or *spans*) of consecutive lines which are atomic for undo. Spans are treated linearly within the file, although they have a dependency structure which expresses dependencies within the proof development itself: it is possible to display this dependency within the interface, to highlight the auxiliary lemmas used in proving a theorem, for example.

[3] Several flavours of Emacs are supported, including XEmacs and GNU Emacs, running on numerous platforms.

[4] This is initially counter-intuitive to new users, although they soon get a feel for it. Proof General could work equally well with provers which keep more history. It would also be possible for Proof General to allow undo to arbitrary positions within proofs for provers which discard history, simply replaying forward until the target point is reached. The disadvantage with that design is that it could take any amount of time to perform the replay, contrary to users' expectations of "undo" operations being quick.


Higher_Order_Logic.thy
File Edit Options Buffers Tools Isabelle/Isar Proof-General X-Symbol MMM Help
State Context Goal Retract Undo Next Use Goto Q.E.D. Find Command Stop Restart Info Help

```
text {*
  Next we show two rules for existentials:
  \begin{itemize}
  \item \textbf{exI} --- existential introduction
  \item \textbf{exE} --- existential elimination
  \end{itemize}
 *}

theorem exI [intro]: "P a ⟹ ∃x. P x"
proof (unfold Ex_def)
  assume "P a"
  show "∀C. (∀x. P x ⟶ C) ⟶ C"
  proof
    fix C show "(∀x. P x ⟶ C) ⟶ C"
    proof
      assume "∀x. P x ⟶ C"
      hence "P a ⟶ C" ..
      also have "P a" .
      finally show C .
    qed
  qed
-9:--  Higher_Order_Logic.thy      (Isar script MMM XS:isabelle/s Scripting)--L232--69%---
```

```
proof (state): step 7

fixed variables: P, a, C = C
prems:
  P a

goal (show, 1 subgoal):
 1. ∀x. P x ⟶ C ⟹ C

-0:--  *isabelle/isar-goals*      (proofstate)--L9--All------------------------------------
```

Fig. 1. The Proof General interface, in a default configuration. The window is split into two panes, the bottom displaying the output from the proof assistant, the top displaying the proof script, with background coloured according to progress.

Proof General provides an advanced implementation of script management which synchronises file editing between the proof assistant and editor in a two-way communication. Files are used to store proof scripts, representing parts of developments. If the user completes processing a proof script file, Proof General informs the prover, if the prover reads another file during processing, it will inform Proof General, and similarly for undo operations at the file level.

Technically, Proof General is implemented mostly in Emacs Lisp, interfacing with other Emacs packages, notably including X-Symbol [37] for displaying mathematical symbols. A considerable effort has been made into making it easy to adapt Proof General to new proof assistants, and it is possible to configure by setting only a handful of variables, with little or no Emacs lisp programming. But the mechanism is ad hoc: we try to anticipate and cater for many different proof assistant behaviours within Proof General, supposing that the proof assistant itself will not be modified. This works only so far: for advanced features (such as proof-by-pointing [11] and proof-dependency visualisation [26,31]), dedicated support from the proof assistant is inevitably required.

To address the limits of the existing Proof General model, and particularly of the Emacs Lisp implementation, Proof General (PG) Kit has been designed [5,6]. The central idea is to use the experience of connecting to diverse provers to prescribe a uniform protocol for interaction. Instead of tediously adapting Proof General to each prover, Proof General calls the shots, by mandating a uniform *protocol for interactive proof*, dubbed **PGIP**, which each prover must support.

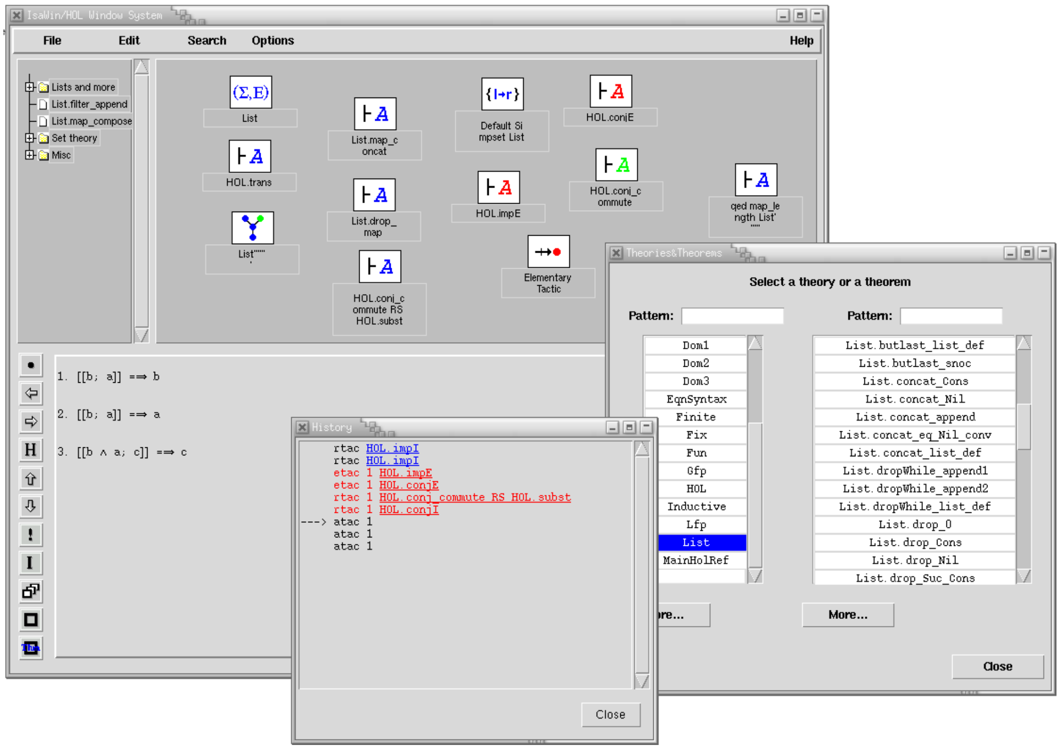
As part of the work described in this paper, we have implemented a prototype version of PG Kit. More details of the system architecture and PGIP protocol are described later, in Section 4.1.

3 IsaWin Basic Concepts

IsaWin is the instantiation of a generic graphical user interface called GenGUI for the Isabelle proof assistant. It provides a more abstract, less syntax-oriented interface to Isabelle (and related provers), based on the visual metaphor of a *notepad* [24]. All objects of interest, such as proofs, theorems, tactics, sets of rewriting rules etc. are visualised by icons on a notepad, and manipulated there using mouse gestures. More complex objects such as proofs can be manipulated by opening them in a separate sub-window. IsaWin offers self-contained history support, proof-by-pointing, dependency management and session management.

The interface is based on the concept of *direct manipulation*: a continuous representation of the objects and actions of interest with a meaningful visual metaphor and incremental, syntax-free operations, i.e., user gestures such as dropping an object onto another, pop-up menus or mouse clicks. Objects are visualised by icons on a notepad (see Fig. 2). The icon is given by the type of the object, which determines the available operations. Hence, when users see an object visualised by a theorem icon, they know that if they drop this object onto another theorem object, the theorem prover will attempt to combine them by forward resolution, whereas if the theorem is dropped onto on ongoing proof, a new proof step using backward resolution with this theorem will be attempted. The type of an object further determines the operations appearing in its pop-up menu and whether (and how) it can be opened into a separate sub-window. The interface also keeps track of dependencies, i.e. if one object is used as an argument to construct another object, and if objects are changed or outdated, all dependent objects are outdated as well.

Drag&drop is resolved by a table indexed with the types of the dropped and the receiving object respectively. The history is represented internally, as the sequence of operations used to construct an object. This has some advantages, as it allows an abstract manipulation of proofs; for example, we can cut out parts from proof scripts and make them into reusable tactics. But moreover it has severe disadvantages: as proof scripts are an external represen-



```
1.  [[b; a]] ⟹ b

2.  [[b; a]] ⟹ a

3.  [[b ∧ a; c]] ⟹ c
```

```
     rtac HOL.impI
     rtac HOL.impI
     etac 1 HOL.impE
     etac 1 HOL.conjE
     rtac 1 HOL.conj_commute RS HOL.subst
     rtac 1 HOL.conjI
---> atac 1
     atac 1
     atac 1
```

Select a theory or a theorem

Pattern:

Pattern:	Pattern:
Dom1	List.butlast_list_def
Dom2	List.butlast_snoc
Dom3	List.concat_Cons
EqnSyntax	List.concat_Nil
Finite	List.concat_append
Fix	List.concat_eq_Nil_conv
Fun	List.concat_list_def
Gfp	List.dropWhile_append1
HOL	List.dropWhile_append2
Inductive	List.dropWhile_list_def
Lfp	List.drop_0
List	List.drop_Cons
MainHolRef	List.drop_Nil
	List.drop_Suc_Cons

Fig. 2. The IsaWin interface. The main window shows the notepad and folder navigation bar on the top, and the ongoing proof below. Two auxiliary windows open here allow the search for theorems, and show the history of the proof.

tation of the history, a generic script management is not very straightforward to implement, and indeed IsaWin only offers limited script management, compared to Proof General. IsaWin has its own format to save and read proofs, and while it has some limited support to produce proof scripts in Isabelle's native format, it cannot parse them. This makes it hard to use IsaWin on proof scripts not developed using IsaWin, we can just load and replay them without manipulating the contained proofs.

Technically, IsaWin is the instantiation of the generic graphical user interface GenGUI with a particular Isabelle-specific module. GenGUI is implemented completely in Standard ML (SML); to customise it (adding or changing icons, shortcut buttons, or menu entries), one has to change the instantiating module. To adapt GenGUI to another proof assistant, one implements a different module with a given signature, containing the object types, operations and the drag&drop table. This requires a good understanding of the signature (and of SML), and also makes adaption to provers not implemented in SML cumbersome.

For us, the experience with GenGUI and IsaWin has shown that the implementation of an interface as an add-on to the proof assistant, even in the same programming language, can leads to a less modular architecture, which makes the interface difficult to reuse. It also makes the interface less robust: if the prover diverges or returns a run-time error, the interface diverges or stops as well. For these reasons, not many different adaptations of GenGUI exist, and in comparison with Proof General, GenGUI has not fully delivered on its promise of genericity.

A better architecture is to keep interface and proof assistant separate, and specify their interaction cleanly and in a language independent way — which is precisely what PGIP does. Proof scripts, understood broadly as the sequence of proof steps leading to a goal, are the main artefacts the user is constructing when working with a proof assistant, and as such should be represented and manipulated explicitly, rather than implicitly as in GenGUI.

4 The Proof General Kit Architecture

The novelty of the work described in this paper is twofold. We provide the fist implementation of a prover-independent protocol for interactive proof, and, on top of this, we introduce a new user interface which is a synthesis of two existing designs, combining text-based script processing with graphical direct manipulation.

This section describes the ideas behind the architecture; Sections 5 and 6 describe the implementation and user interface.

4.1 Communicating Interactive Proof Components

The spirit of Proof General Kit is to use lightweight protocols to connect together a range of components used for conducting interactive proof. Components are loosely coupled, and may be run on different machines. The specifics of the design are intended to work with existing theorem provers. Thus, where some of our design decisions appear fairly conservative, this is because we want to allow a gradual migration of existing theorem provers to our architecture while maintaining interoperability. In our experience, this is crucial for acceptance, both from users and theorem prover developers.

Fig. 3 illustrates the basic plan. The *mediator* has a central role: it interfaces the various other components to the proof assistant and filesystem[5]. Possible other components include a GUI for displaying and developing proof graphically, a text editor for editing a proof script, and a web browser for exploring the theory store or following a proof development remotely.

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 theory store, but (especially in case of write access) this must be carefully sanctioned by the mediator. This separation allows the mediator to organize

[5] or network theory store, if supported by the prover

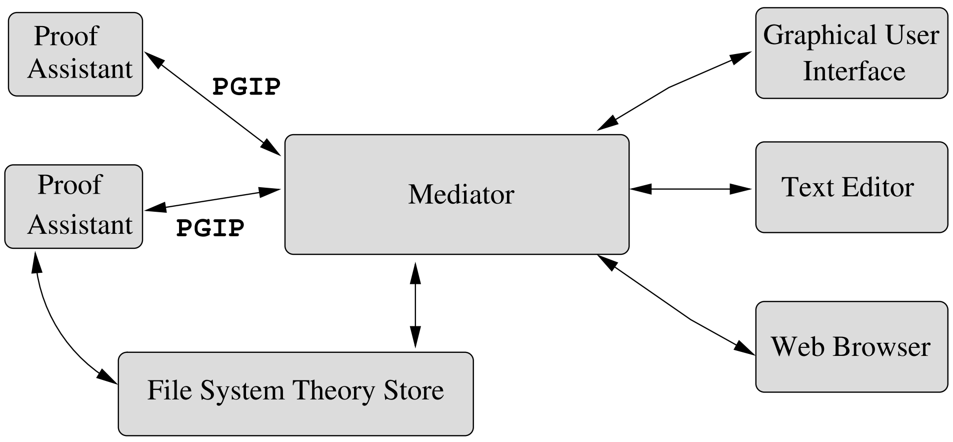


Fig. 3. Proof General Kit Architecture

the synchronization messages needed for interactive development, for example, maintaining a set of locked files.

4.2 PGIP: a Protocol for Interactive Proof

The main connecting protocol shown in Fig. 3 is called **PGIP** (standing for *Proof General Interactive Proof*), the syntax of which has been specified as an XML format. The design of PGIP began by isolating and clarifying the mechanisms currently implemented in Proof General.

PGIP messages can be divided into two classes: those sent to the prover and those originating from the prover. Messages sent to the prover consist of configuration commands, commands for inspecting various aspects of the proof assistant state, and actual proof commands. Messages sent from the prover include status display or error dialogues and messages to configure (proof assistant specific) user-level menus and preferences.

Fig. 4 shows an example PGIP interchange between a prover and the interface: the interface sends a proof command (`<proofstep>`), and the prover responds with a new state (`<proofstate>`). Command and response are wrapped up as `<pgip>` packets, containing meta-information such as a unique identity (`id`) and sequence number (`seq`), which allows responses to refer to earlier requests (with the `refseq` attribute). Notice that we generalise typical RPC schemes which are single-request single-response, by allowing possibly many responses to a request. The reason for this is so that the prover may emit information to the interface gradually, perhaps during the progress of a long proof. However, we do not allow the prover to initiate responses without a previous request. The second message from the prover, `<ready>`, indicates that it has completed processing a proof command. We expect that the main proof process is single threaded, updating a proof state which is the focus of interactive development. More generally, it is possible that secondary threads

```
<pgip origin="PG/Kit" id="sartre/cxl/13523" class="pa" seq="9">
   <proofstep>apply (rule allI)</proofstep></pgip>
```

```
<pgip origin="Isabelle/Isar" id="sartre/cxl/1067416425.138"
      class="pg" seq="21" refseq="9" refid="sartre/cxl/13523">
   <proofstate><pgml><statedisplay>proof (prove): step 1
goal (lemma (Foo), 1 subgoal):
 1. <sym name="And"/><atom kind="bound">x</atom>.
<atom kind="bound">x</atom> = <atom kind="bound">x</atom>
</statedisplay></pgml></proofstate></pgip>
<pgip origin="Isabelle/Isar" id="sartre/cxl/1067416425.138"
      class="pg" seq="22" refseq="9" refid="sartre/cxl/13523">
   <ready/></pgip>
```

Fig. 4. A real-life PGIP message exchange: above is a proof command sent to the prover, below is the prover's response.

would be available to inspect loaded theories, or that other processes would be used to off-load the processing of some PGIP commands.

One example of using an auxiliary process is to handle `<parsescript>` and `<unparsescript>` messages. We suppose that PGIP commands may be generated directly by the interface, or by the user entering text in the prover's native language, which the prover has to convert into PGIP format. Thus, there needs to be a way to parse proof script commands into PGIP commands and "unparse" them back. The parse and unparse commands might be handled by the prover itself, or instead with an additional component. For example, for simple enough proof script languages we could implement a generic parser based on regular expressions; this is similar to the way that the present Emacs Proof General works.

4.3 PGML: A Markup Language for Proof Display

Messages originating from the prover can contain a representation of the proof state, for example a list of subgoals to be solved to complete the proof. To communicate these, PG Kit includes another XML format, **PGML**, the Proof General Markup Language. The markup language is intended for displaying concrete syntax. It includes representations for mathematical symbols, along with the possibility of hidden annotations which express term structure. These annotations can be used to implement sub-term cut-and-paste, proof-by-pointing or similar features [11]. The prover response in Fig. 4 shows a PGML message embedded in the proof state display, with annotations for denoting the different kinds of variables.

There are already existing XML-based document formats designed for displaying mathematics (MathML, [35]), and transferring mathematical content between applications (OpenMath, [30]). Later on, we hope to use these languages with PGML. However, these formats require the abstract syntax of terms to be sent back and forth, with the markup into concrete syntax made in the mediator (or even the display). Although we believe that this is ultimately the right way, it cannot be easily accommodated by existing proof systems in a generic way, as these have often have their own (very elaborate) provisions for rendering concrete syntax. These mechanisms are supported immediately by PGML, while at the same time offering a migration route; this is one of the conservative design decisions mentioned above.

4.4 Incremental Proof Development

PGIP assumes an abstract model of incremental proof development, where we suppose there are four fundamental states occupied by the prover, with transitions between the states given by different kinds of proof commands.

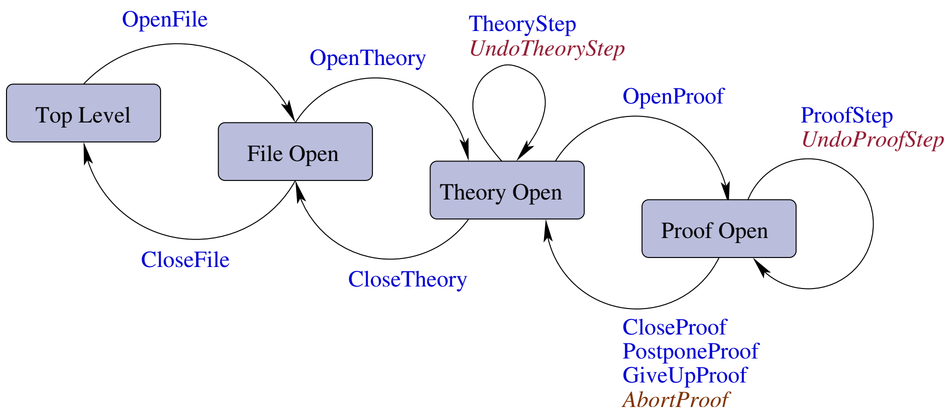


Fig. 5. Incremental proof development.

Fig. 5 shows the fundamental prover states, and the transitions between them. The four states illustrated are:

(i) the *top level* state where nothing is open yet;

(ii) the *file open* state where a file is currently being processed;

(iii) the *theory open* state where a theory is being built;

(iv) the *proof open* state where a proof is currently in progress.

The reason for distinguishing the states is that different commands are available in each state, and the prover's undo behaviour in each state can be different. In the theory state, for example, we may issue *theory steps* which add declarations or definitions to the theory, or we may undo the additions. In the proof state, we can issue *proof steps* and undo these steps, or complete the current proof attempt in a number of ways. These fundamental states also give rise to a hierarchy of objects: the top level may contain a number of files, files may contain theories, and theories may contain proofs.

Proof commands may take additional arguments. An `OpenTheory` command typically gets passed the name of the theory to built, and maybe parent theories (depending on the prover), whereas `OpenProof` will require a goal to be proven. Most proof commands correspond directly to textual elements which (usually) appear in the proof script language; these are the proper proof commands mentioned earlier. The improper commands are used for controlling the prover's state, and do not appear in the proof script being developed. These include the three italicized undo commands shown in Fig. 5.

Proof development proceeds by traversing the fundamental states, building up proof scripts along the way. This incremental development model is an abstraction of what occurs in a typical prover, based on the current model used in Proof General. Some provers may not implement all of this (for example, some provers do not know anything about files; others identify theories and files), and some provers may provide richer notions (for example, nested proofs or generic proofs) which are not captured in PGIP.

There is a difference here between provers whose proof scripts essentially record the individual interaction commands (for example, Isabelle and Coq), and those where final proof scripts may take a slightly different form. For example, in HOL scripts are encouraged to contain "batch" proofs which are executed more efficiently and directly outwith the interactive mode to reproof theorems in a single step. To handle this second possibility, we would employ automatic transformations between the two forms (although experienced HOL users might want to be able to hand-optimise their direct proofs).

Another (intentional) restriction in this first version of PGIP is to only allow sequential, single-threaded movement through the incremental proof development states shown in Fig. 5. The picture does not allow for the possibility of opening several files at once, or working on several open proofs within the same theory. This simplifies things for the protocol (and its implementation within the prover), but it does not preclude the mediator from implementing more flexible development mechanisms, hiding the underlying context switching from the user. Indeed, this is implemented in our prototype described in Section 5.

5 Implementing the PG Kit Architecture

The ideas behind the PG Kit system architecture have been outlined above. The architecture is described in more detail elsewhere [6,8], including XML schemas (written in RELAX NG [32]) for the PGIP and PGML languages. This section describes our prototype implementation of the design.

The *mediator* maintains the overall state, comprising the proof scripts under construction, their dependencies and an abstraction of the proof assistant's state. It sends commands to, and receives status messages from the proof assistant; at the same time, it receives user input from the display, and keeps the displays up-to-date. In principle, we can connect to more than one proof assistant at the same time, but presently we cannot interchange any data (such as proofs or theorems) between provers.

A *display engine* (or display for short) visualises the proof assistant's state, proofs, theorems, and so on. There can be more than one display connected to the main broker, and there are different kinds of displays. A simple *line-oriented display* displays lines of text and relays command lines typed by the user; this corresponds to Emacs Proof General. Slightly more sophisticated, a *graphical display* translates user gestures (drag&drop and so on) into textual prover commands; thus, IsaWin becomes another specific display module in this architecture. Other possible displays could include a web interface (either client-side by running an applet in a browser, or server-side by embedding the mediator into a web server via techniques such as servlets, ASP or CGI).

We further distinguish between *active* and *passive* displays. An active display allows the user to enter commands, whereas a passive display primarily visualises proofs, possibly using hyperlinked text.

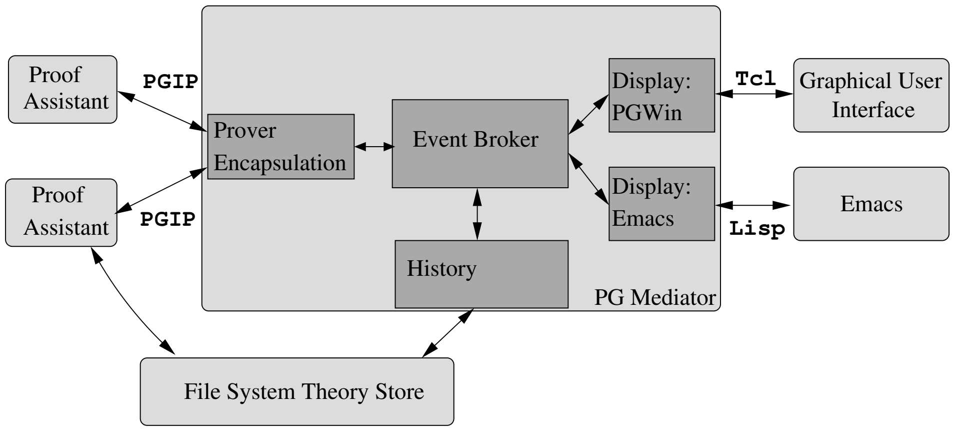


Fig. 6. System Architecture

Fig. 6 shows the architecture of the implemented system. Rounded boxes denote separate components, and square boxes denote the different modules of the mediator. The implementation is event based. Events are generated either from user input, or change of state in the prover. The mediator is the central event broker; it receives input events, updates its internal state, and sends on change events to other parts of the system as necessary. Events are structured: they contain PGIP messages.

We implement the mediator as a set of Haskell modules, loosely coupled by Unix pipes to the other components. The mediator contains one central event broker, and one event handling module for each module, such as displays and proof assistants (see Fig. 6). This architecture is flexible and portable (to support the Windows operating systems, one can either replace pipes by sockets, or use compatibility packages such as MinGW). The alternative would be a component framework, which would either restrict us in the operating system (e.g. DCOM or .NET for Windows only, DCOP is not for Windows) or programming language (e.g. JavaBeans) or is too heavy-weight (e.g. CORBA) for our purposes.

5.1 Events and Messages

The mediator is implemented in Concurrent Haskell, using the Glasgow Haskell Compiler, with events as first-class values in Haskell [33]. That is, we have a polymorphic datatype of events containing a value of any type, with operations to synchronise on an event, sequence an action with an event, combine two events via deterministic choice, and others:

```
Event a
 sync   :: Event a -> IO a
 (>>>=) :: Event a-> (a-> IO b)-> Event b
 (+>)   :: Event a-> Event a-> Event a
```

This allows us to write concurrent functional programs in a notation reminiscent of process algebras such as the π-calculus [28].

Events contain PGIP messages. To model PGIP faithfully in Haskell, we use HaXML [36]. From a given DTD, HaXML generates a series of Haskell datatypes, one for each element, along with functions to read and write XML. The advantage is that the type security given by the DTD extends into our program, making it nearly impossible to send messages containing invalid XML, and detecting the reception of invalid XML immediately. The broker does further validation on the messages, but it tries to be a robust as possible; e.g. if a response does not refer to a valid request, it is quietly discarded (and a log file entry is written).

There are different types of events, corresponding to different elements in the PGIP DTD, represented by different types in the mediator. These include:

- *Prover command events* (`ProverCmd`) are commands sent to the prover encapsulation, either generated from command events, or when replaying proof scripts.

- *Prover message events* (`ProverMsg`) are either messages from the proof assistant in response to proof commands, containing e.g. a new prover state or an error returned by the prover, or configuration messages from the proof assistant, adding or changing menus, shortcut buttons, icons, or the drag&drop table. They are interpreted by the event broker, which updates its internal states accordingly, and updates the connected displays.
- *Display command events* (type `DispCmd`) are commands input by the user. Commands are generated by active displays, either by interpreting gestures or by the user typing a command line (or mixtures of the two), and then handled by the event broker.
- *Display message events* (`DispMsg`) contain messages to be displayed, such as a new proof assistant state, error or warning messages. Display events are generated by the event broker from proof response events, or as an answer to user input (e.g. trying to browse beyond the end of the history).

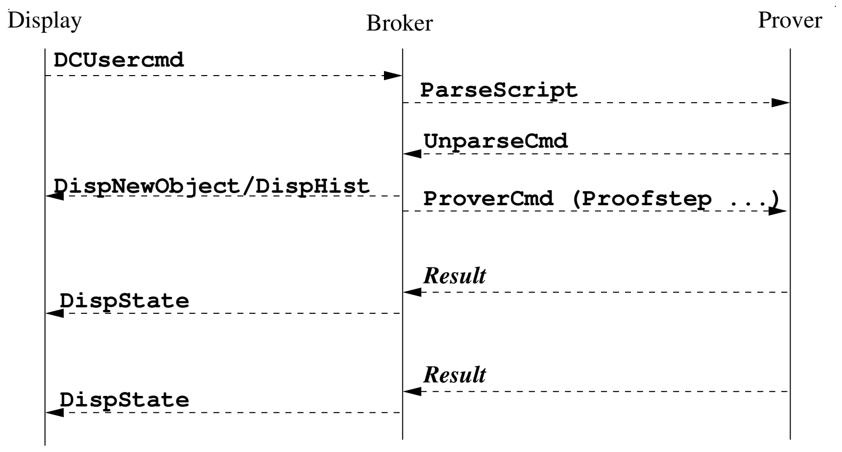


Fig. 7. A typical event sequence.

To illustrate the event concept, Fig. 7 shows a typical interaction sequence: the users enters a command, it gets parsed by the prover, a new object appears on the desktop, and as the prover computes the results of the command, the broker updates its internal state and the display.

As the interaction sequence shows, the prover may send its responses to one request in several PGIP messages, reporting on the progress of the proof. Interactive provers can take very long and sometimes even diverge; so it is important that these messages are displayed as soon as received, to keep the user informed.

To allow for the possibility of divergence, we must be able to interrupt the prover. Presently, the prover is run in a child process on the same machine, so we can use signals. For a distributed setup where the prover may run on another machine, we will need provision to transmit out-of-band interrupt signals, for example by running the prover as child process of the process listening on the actual socket.

5.2 Display Engines

All display engines, even such seemingly different ones like Emacs and PGWin, serve to visualise display messages originating mainly from the event broker. Crucially, display messages and user input may refer to earlier messages. For example, in later proofs users will usually want to use theorems they have proven earlier.

In order to subsume different display engines in a uniform framework, we use the notion of an *object* as introduced by the generic graphical user interface (see Sect. 3 above). An object is anything the system needs to keep track of: most prominently, theories, theorems and ongoing proofs, but also auxiliary objects such as tactics, rewrite rules, and so on. Objects are *typed*. Types comprise basic types (such as theories, theorems, and proofs), and prover-definable types (such as the auxiliary objects, which vary from prover to prover). Each display must visualise at least the basic types, but may not visualise all of the other types. Thus, while PGWin represents all objects by their icons (the icon forms part of the type definition), Emacs will only represent those types corresponding to a span, which is a particular line range in a particular text buffer.

All display engines have to implement certain operations, such as display warnings, errors and status messages; create a new object; update a previously displayed object; outdate an object; receive user input; etc. Technically, we define a type class `Display`, which defines the operations that a display has to implement, for example creating and updating objects, generating command events (`DispCmd`), and receiving display message events (`DispMsg`):

```
class Display d where
  newObject :: d-> ObjId -> ObjType -> DispAttr -> IO ()
  updObject :: d-> ObjId -> ObjText -> IO ()
  bind      :: d-> IO (Event DispCmd)
  send      :: d-> DispMsg-> IO ()
```

Every kind of display engine is modelled by a different type, but all are instances of the display class:

```
instance Display IsaWin where
  -- definitions of functions...
instance Display Emacs where
  -- definitions of functions...
```

All displays are kept in one heterogeneous list (using existential types [22]): display events are sent to all elements of this list, and the command events generated by all displays are the command events of each display combined with the obvious extension of the binary choice operator `(+>)` to lists of events.

Notice that implementing a notion of object within the interface allows us to keep context information with objects which specifies dependencies: effectively, a position within the linear PGIP protocol. This means that the user can switch between open proofs, for example, by selecting objects with the mouse. Behind the scenes the theory might be switched by aborting the currently open proof and closing its theory and file, before opening the file and theory of the newly selected proof. This is why displays must support outdating and updating operations on objects: if an earlier definition is undone, or an ancestor theory is retracted, all dependent objects will be outdated by the broker, and marked as temporarily unavailable in the interface.

At present, we have implemented one main display component, PGWin. It is described further in Section 6 below. There is also a *control display* which can launch other displays. Work is underway within the present Emacs version of Proof General to also support a PGIP and Lisp interface to Emacs as suggested in Fig. 6.

5.3 A PGIP-enabled version of Isabelle

The other piece needed for our prototype implementation is some proof assistant itself. By design, we want to interface with existing theorem provers with minimal effort, but recognising that they will need some customization.

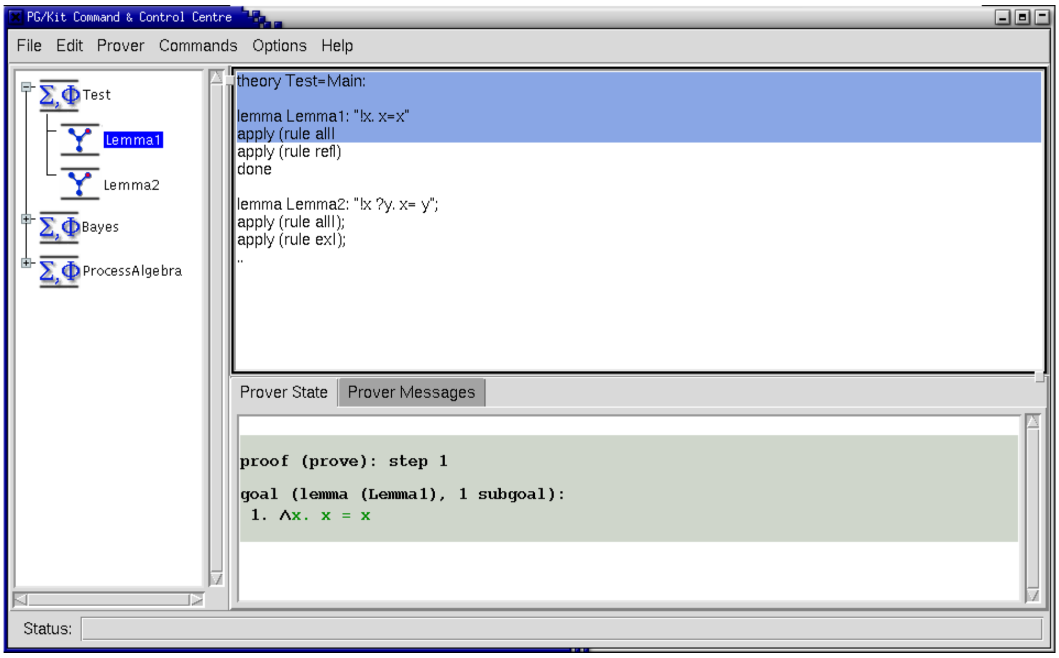
An experimental effort to PGIP-enable Isabelle/Isar [38] has been undertaken, with the help of the Isabelle development team. The implementation consists of a 500-line extension to the existing Standard ML module for interfacing with Proof General that is already supplied with Isabelle. There have also been some minor changes in other Isabelle modules.

Presently, we use Isabelle's (somewhat limited) built-in XML parser and XML output functions. Contrary to the strongly-typed approach used in the mediator, this implementation does not automatically guarantee to produce (nor parse) XML which conforms to the PGIP schema, since we have no appealing tool such as HaXML to automate the conversion to an SML datatype. Once the design of PGIP is settled, we will implement a suitable SML datatype manually.

The PGIP implementation inside Isabelle has been largely straightforward, although a few thorny issues have come to light. To produce correct XML, we have to be careful with escaping special characters; this requires checking all of the places where Isabelle outputs messages, in particular, to avoid the possibility of double-escaping because we already adjust the pretty-printer for Isabelle terms to produce PGML format. It also turned out to be surprisingly difficult to supply a parser for the Isar language, since the one inside Isabelle is constructed using parser combinators, which do not build a parse tree or record input position information. A final open issue is the question of equipping Isabelle terms with subterm structure information in the term printer. Although this had been implemented as part of the IsaWin effort [25], the code was unfortunately written for an older version of Isabelle. It is hoped that further work on the PGIP support in Isabelle will address these issues.

6 The PGWin Display Engine

In the PG Kit system architecture, PGWin is one particular display. Its design is a synthesis of Proof General and IsaWin, resulting in a novel interface which combines Proof General's text-based interaction with IsaWin's graphical user interface. The idea is to display both the icons and the text containing the textual representation of the objects. Thus, we show both an icon on the desktop for e.g. a proven theorem, and the text containing the proof. Since the text is inherently linearly ordered, the unordered display of icons on a notepad is replaced by a linearly ordered, hierarchical tree display.


PG/Kit Command & Control Centre
File Edit Prover Commands Options Help
Test
Lemma1
Lemma2
Bayes
ProcessAlgebra
Prover State
Prover Messages
Status:

```
theory Test=Main:

lemma Lemma1: "!x. x=x"
apply (rule allI
apply (rule refl)
done

lemma Lemma2: "!x ?y. x= y";
apply (rule allI);
apply (rule exI);
..
```

```
proof (prove): step 1

goal (lemma (Lemma1), 1 subgoal):
 1. ⋀x. x = x
```

Fig. 8. The interface of the PGWin display engine (prototype).

Fig. 8 shows a screenshot of the prototype. We can see the objects displayed by icons on the left, and the actual proof text in an editor on the right. Users can either edit the proof text directly in the editor (using the same script management techniques as Proof General), or manipulate the icons.

User gestures are translated into commands by a table indexed with the types of the objects. This table, along with the types of the proof assistant, the icons and many other details of the visual appearance are determined by display configuration messages, which form part of PGIP. Thus, when connecting a proof assistant to the mediator, the proof assistant sends configuration messages which tell the system the types of objects the proof knows about, and the commands triggered by drag&drop. There are further display configuration messages to add, change, or delete a custom menu's entries, configuration options, etc.

Fig. 9 shows an example of a PGIP configuration message sent by Isabelle/Isar (without the surrounding `<pgip>` packet). The first of these messages sets up the interface such that if a theorem is dropped onto another theorem, forward resolution is performed. To this end, the operation command (given in the `<opcmd>` element) is expanded such that the name of the first theorem (say `sym`) is inserted for the first argument (`%1`), and the name of the second theorem (say `refl`) is substituted for the second argument (`%2`). The string generated by this expansion (`sym THEN [refl]`) is then sent to the mediator, and from there to the proof assistant. The second of the configuration messages in Fig. 9 sets up backward resolution. The empty target type (`<optrg>`) means that it is a proof operation. When a theorem (say `allI`) is dropped onto the ongoing proof, the command `<proofstep>apply rule (allI)</proofstep` is generated (`<proofstep>` because this is a proof operation) and sent.

```
<guiconfig>
<opn name="forward resolution">
  <opsrc>theorem theorem</opsrc>
  <optrg>theorem<optrg>
  <opcmd>%1 [THEN %2]</opcmd>
</opn>
<opn name="apply rule">
  <opsrc>theorem</opsrc>
  <optrg></optrg>
  <opcmd>apply (rule %1)</opcmd></opcmd>
</opn>
</guiconfig>
```

Fig. 9. Configuring drag&drop: forward resolution and backward resolution

Users can freely mix graphical and textual interaction (for example, type one proofstep, and use drag&drop for the next one). This mixture has some distinct advantages: it accustoms users to the syntax of the command language, since they will see it appearing in the history, it gives full access to all of the prover's commands (in IsaWin, only those commands which had been configured could be used from the graphical interface), and it allows the interface to be used with any proof script.

The PGWin display engine is implemented in HTk, a functional encapsulation of the graphical user interface library and toolkit Tcl/Tk into Haskell [18]. Thus, the PGIP events are translated into Tcl code within Haskell, which is then sent on to the Tcl/Tk interpreter wish (see Fig. 6). A future, alternative graphical display engine might communicate externally in PGIP; but there is still a considerable amount of implementation work to organise the graphical interface which we believe is better done in Haskell than in an untyped scripting language like Tcl.

7 Conclusions and Outlook

This paper has described the concepts underlying the synthesis of the Proof General and IsaWin interfaces into one combined interface. The new interface has an event-based architecture based on the PG Kit, and consists of several components communicating in the PGIP proof protocol. The implementation of a first prototype of the PG Kit architecture has led to a many clarifications and extensions of the PGIP protocol; more are planned. The system comes with a new user interface, which allows a mixed graphical and textual interaction, combining the advantages of both Proof General and IsaWin. Moreover, the open architecture opens the way to more easily implementing new interaction mechanisms, and developing a truly generic high-level interface.

As it stands, the current implementation is a research prototype. We hope to release it to other researchers in the near future for further experimentation and feedback. Presently, the system supports Unix-like systems only, but a port to Windows is very possible; we have taken care in our architecture not to tie ourselves to one particular platform. An advantage of our prototype as it stands is the user does not need to install anything on his system apart from the compiled executable, the PGIP-enabled prover and Tcl/Tk (which is part of most modern Unix systems anyway).

In the longer run, we hope for further implementations of PG Kit. For example, work is beginning at the University of Edinburgh to integrate PG Kit into the Eclipse workbench [16], which is an extensible Integrated Development Environment designed to allow easy tool integration.

7.1 Related Work.

In recent years there have been a number of projects introduced with the aim of fostering interoperability between theorem proving-like tools, although not always considering interface aspects. We give a brief overview here, and then mention some particular interfaces.

HELM [3] and the more recent MoWGLI [29] are centered around semantically annotated mathematical hypertext documents (the "Semantic Math-Web" [2]), and tools supporting this; a user interface is part of this effort as well, but rather as a user front-end for the semantic math-web.

The OMEGA prover [9] started as a framework to integrate different automatic provers. It has developed into MathWeb, which uses the XML format OMDoc [20] as an exchange language, and the ActiveMath [27] project, which is a learning environment based on OMDoc. OMEGA has a user interface, LOUI, but it is not generic in our sense; MathWeb uses style sheets for visualisation. Nonetheless, the system architecture is not unlike ours: components loosely coupled over sockets using a customised middleware architecture (built from XML-RPC and KQML), a central broker component, and a store for theorems (called MBase [21]).

Logosphere [34] is a recently launched project aiming at connecting different provers, but with an emphasis on higher-order logic; the aim is to create a formally verified "Digital Library" of proof and theorems.

Prosper [15] uses another approach to interoperability, aiming at connecting different automated proof tools together: at the core of the system an LCF prover kernel is used to guarantee logical consistency. Tools are wrapped up as components, using the Prosper Integration Interface. Although this is a more logic-centred view, with an emphasis on the exchange of proofs and theorems, there are interesting similarities to our architecture: again, a light-weight customised middleware architecture implemented in a functional language and a central broker component with provers (and other tools) connected by loose coupling.

CtCoq [10], and its more recent descendent PCoq [1], are interfaces specifically for the Coq system, based around script management similar to that implemented in Proof General. In CtCoq and PCoq output is sent as abstract syntax to the interface, where pretty-printing and layout takes place; the transfer of abstract syntax also allows the implementation of features such as proof-by-pointing and rewrite by drag-and-drop.

We briefly mention a couple of other notable theorem prover interfaces. The Jape [13] system is a prover with a sophisticated graphical interface which can display sequent-style proofs as lists of inferences or using box-style. User interaction is made very straightforward by selecting premises or conclusions using mouse clicks, and choosing inference rules from menus. The PVS interface [19] is another Emacs-based interface written in Emacs lisp, notable because it provides ways of managing the proof obligations of PVS, so that theories may be constructed in a flexible order.

To sum up the review on related work: most interesting interfaces are geared towards one specific prover, even if potentially usable for more than one. So it is certainly desirable to look for an advanced interface which is truly generic and combines the best aspects of already existing interfaces; this is particularly pertinent when one bears in mind the lack of resource available to the community for work on interfaces. When combining different provers, the architecture of choice seems to be loose coupling via sockets or pipes, with a custom middleware often implemented in a functional or functional-logical language— quite similar to the architecture we have designed.

7.2 Outlook.

We envisage further extensions of PGIP in mainly three directions: covering the structure of theories as well as the structure of proofs; covering concrete syntax mechanisms for parsing and pretty-printing logical terms; and eventually, inventing a generic prover-independent scripting language.

Currently PGIP has a command to open a named theory, and a command to start proofs of named theorems in that theory. We would like to generalise this to consider naming other elements that appear in theories, such as declarations of types or constants within the logic — both of these are currently treated within PGIP as “theory steps” which are not further interpreted. In the first case, these additional named elements will allow more flexible processing of proofs scripts (for example, together with prover-supplied dependency information, refactoring operations to do renaming or move around definitions). In the second case, this will open the way to understanding abstract syntax trees for logical terms in the interface, and we can let the interface provide parsing and pretty-printing (rendering) facilities by adding concrete syntax directives to declarations. Presently, many provers offer these services as part of their syntax machinery, but it seems to us to be an extra-logical consideration which ought to be managed by the interface; so much for the better if there is a good generic implementation of such facilities.

One step further is the definition of a generic scripting language. Presently, proof scripts are still stored in the proof assistant’s native format; a generic scripting language would alleviate users from needing to learn different proof script languages when switching between proof assistants and their logics. It might ultimately even allow the exchange of *replayable* proofs, or parts of proofs, between different proof assistants. The definition of such a scripting language might be based on an already established formalism such as Wenzel’s Isar language [38], or might be designed afresh starting from a very simple language which offers a sequencing and a definitional operator, which describes the notion of *hiproofs* [14].

Acknowledgements. We’re grateful for the helpful suggestions for improvements to this paper provided by the referees, and for interesting discussions on the topic with Ewen Denney and various participants of UITP 2003. DA benefited from support provided by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing.

References

[1] Amerkad, A., Y. Bertot, L. Rideau and L. Pottier, *Mathematics and proof presentation in Pcoq*, in: *Proceedings of Proof Transformation and Presentation and Proof Complexities (PTP'01)*, 2001.

[2] Asperti, A., L. Padovani, C. S. Coen and I. Schena, *HELM and the semantic math-web*, in: R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics TPHOLs 2001*, Lecture Notes in Computer Science **2152** (2001), pp. 59–74.

[3] Asperti, A. et al., *HELM: Hypertextual electronic library of mathematics* (2002), University of Bologna, `http://helm.cs.unibo.it/`.

[4] Aspinall, D., *Proof General: A generic tool for proof development*, in: Graf and Schwartzbach [17], pp. 38–42, `http://homepages.inf.ed.ac.uk/da/papers/pgoutline/`

[5] Aspinall, D., *Protocols for interactive e-proof* (2000), `http://proofgeneral.inf.ed.ac.uk/kit/`

[6] Aspinall, D., *Proof General Kit* (2002), white paper, `http://proofgeneral.inf.ed.ac.uk/kit/`

[7] Aspinall, D., H. Goguen, T. Kleymann and D. Sequeira, *Proof General* (2003), system documentation, `http://proofgeneral.inf.ed.ac.uk/doc`

[8] Aspinall, D. and C. Lüth, *Commentary on PGIP* (2003), `http://proofgeneral.inf.ed.ac.uk/kit/`.

[9] Benzmüller, C. et al., *ΩMega: Towards a mathematical assistant*, in: W. McCune, editor, *14th International Conference on Automated Deduction — CADE-14*, LNAI **1249** (1997).

[10] Bertot, Y., *The CtCoq system: Design and architecture*, Formal Aspects of Computing **11** (1999), pp. 225– 243.

[11] Bertot, Y., T. Kleymann and D. Sequeira, *Implementing proof by pointing without a structure editor*, Technical Report ECS-LFCS-97-368, University of Edinburgh (1997), also published as Rapport de recherche de l'INRIA Sophia Antipolis RR-3286.

[12] Bertot, Y. and L. Théry, *A generic approach to building user interfaces for theorem provers*, Journal of Symbolic Computation **25** (1998), pp. 161–194.

[13] Bornat, R. and B. Sufrin, *A minimal graphical user interface for the Jape proof calculator*, Formal Aspects of Computing **11** (1999), pp. 244– 271.

[14] Denney, E., J. Power and K. Tourlas, *Hiproofs: A hierarchical notion of proof tree* (2004), draft.

[15] Dennis, L. A. et al., *The PROSPER toolkit*, in: Graf and Schwartzbach [17].

[16] The Eclipse Foundation, *Project web site*, `http://www.eclipse.org`.

[17] Graf, S. and M. Schwartzbach, editors, "Tools and Algorithms for the Construction and Analysis of Systems," Springer, 2000.

[18] *HTk — graphical user interfaces for Haskell programs*, `http://www.informatik.uni-bremen.de/htk`.

[19] Kiniry, J. and S. Owre, *Improving the PVS user interface*, in: *Proc. User Interfaces for Theorem Provers UITP 2003*, 2003, pp. 101– 122.

[20] Kohlhase, M., *OMdoc: Towards an OpenMath representation of mathematical documents*, `http://www.mathweb.org/omdoc/`.

[21] Kohlhase, M. and A. Franke, *MBase: representing knowledge and context for the integration of mathematical software systems*, Journal of Symbolic Computation **23** (2001), pp. 365– 402.

[22] Laufer, K., *Type classes with existential types*, Journal of Functional Programming **6** (1996), pp. 485 – 517.

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

[24] Lüth, C. and B. Wolff, *Functional design and implementation of graphical user interfaces for theorem provers*, Journal of Functional Programming **9** (1999), pp. 167– 189.

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

[26] McNeil, F., "On the Use of Dependency Tracking in Theorem Proving," Master's thesis, Division of Informatics, University of Edinburgh (2000).

[27] Melis, E., E. Andrès, J. Büderbender, A. Frischauf, G. Goguadze, P. Libbrecht, M. Pollet and C. Ullrich, *ActiveMath: a generic and adaptive web-based learning environment*, Artificial Intellligence in Education **12** (2001).

[28] Milner, R., "Communicating and Mobile Systems: the π-Calculus," Cambridge University Press, 1999.

[29] *MoWGLI. mathematics on the web: Get it right by logics and interfaces*, `http://www.mowgli.cs.unibo.it/`.

[30] *The OpenMath Society*, http://www.nag.co.uk/projects/openmath/omsoc/.

[31] Pons, O., Y. Bertot and L. Rideau, *Notions of dependency in proof assistants*, in: *Proc. User Interfaces for Theorem Provers, UITP'98*, 1998.

[32] *RELAX NG xml schema language* (2003), http://www.relaxng.org/.

[33] Russell, G., *Events in haskell and how to implement them*, in: *International Conference on Functional Programming ICFP'01* (2001).

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

[35] *Mathematical markup language (MathML)*, W3C Recommendation (1999), http://www.w3.org/Math/.

[36] Wallace, M. and C. Runciman, *Haskell and XML: Generic combinators or type-based translation?*, in: *International Conference on Functional Programming ICFP'99* (1999), pp. 148– 159.

[37] Wedler, C., *Emacs package X-Symbol*, http://x-symbol.sourceforge.net.

[38] Wenzel, M., "Isabelle/Isar — a versatile environment for human-readable formal proof documents," Ph.D. thesis, Technische Universität München (2001).