Towards Merging *Plat*Ω and PGIP

David Aspinall

LFCS, School of Informatics, University of Edinburgh
Edinburgh, U.K. (homepages.inf.ed.ac.uk/da)

Serge Autexier

DFKI GmbH, 28359 Bremen, Germany (www.dfki.de/~serge)

Christoph Lüth

DFKI GmbH & FB Informatik, Universität Bremen
28359 Bremen, Germany (www.informatik.uni-bremen.de/~cxl)

Marc Wagner

DFKI GmbH, 28359 Bremen, Germany & FR Informatik, Universität des
Saarlandes, 66123 Saarbrücken, Germany (www.ags.uni-sb.de/~marc)

Abstract

The PGIP protocol is a standard, abstract interface protocol to connect theorem provers with user interfaces. Interaction in PGIP is based on ASCII-text input and a single focus point-of-control, which indicates a linear position in the input that has been checked thus far. This fits many interactive theorem provers whose interaction model stems from command-line interpreters. *Plat*Ω, on the other hand, is a system with a new protocol tailored to transparently integrate theorem provers into text editors like $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ that support semi-structured XML input files and multiple foci of attention. In this paper we extend the PGIP protocol and middleware broker to support the functionalities provided by *Plat*Ω and beyond. More specifically, we extend PGIP (i) to support multiple foci in provers; (ii) to display semi-structured documents; (iii) to combine prover updates with user edits; (iv) to support context-sensitive service menus, and (v) to allow multiple displays. As well as supporting $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$, the extended PGIP protocol in principle can support other editors such as OpenOffice, Word 2007 and graph viewers; we hope it will also provide guidance for extending provers to handle multiple foci.

Keywords: *Plat*Ω, Proof General, Mediator, Protocol, PGIP

1 Introduction

Proof General [2,3] is widely used by theorem proving experts for several interactive proof systems. In some cases, there is no alternative interface; in others, the alternatives are little different. Yet the limitations of Proof General are readily apparent and reveal its evolution from simple command line systems. For one thing, the input format is lines of ASCII-text, with the minor refinement of supporting Unicode or TeX-like markup. The presentation format during interaction is the same. For another thing, the proof-checking process has an overly simple linear progression with a single point-of-focus; this means that the user must explicitly undo and redo to manage changes in different positions in the document, which is quite tedious.

Meanwhile, theorem provers have increased in power, and the ability for workstations to handle multi-threaded applications with ease suggests that it is high time to liberate the single-threaded viewpoint of a user interface synchronised in lock-step to an underlying proof-checking process. Some provers now provide multiple foci of attention, or several prover instances might be run in concert. Text editors, too, have evolved beyond linear ASCII-based layout. The scientific WYSIWYG text editor T_EX_{MACS}, for example, allows editing TeX and LaTeX-based layout, linked to an underlying interactive mathematical system.

Significant experiments with theorem proving using richer interfaces such as T_EX_{MACS} have already been undertaken. In particular, the *Plat*Ω system [9,4] mediates between T_EX_{MACS} and the theorem prover ΩMEGA. While experiments with individual systems bring advances to those specific systems, we believe that many parts of the required technology are generic, and we can benefit from building standard protocols and tools to support provers and interfaces. The aim of this paper, then, is to integrate lessons learned from the *Plat*Ω system prototype with the mainstream tool Proof General and its underlying protocol PGIP, putting forward ideas for a new standard for theorem prover interfaces, dubbed here *PGIP 2*. Specifically, our contributions are to combine ideas of state-tracking from PGIP with semi-structured document models and menus as in *Plat*Ω, and to add support for possibly distributed multiple views.

1.1 PG Kit system architecture

The *Proof General Kit* (PG Kit) is a software framework for conducting interactive proof. The framework connects together different kinds of components, exchanging messages using a common protocol called *PGIP*. The main components are interactive provers, displays, and a broker middleware compo-

nent which manages proof-in-progress and mediates between the components. Fig. 1 shows the system architecture; for details of the framework, we refer to [3].

The PG Kit architecture makes some assumptions and design decisions about the components. Generalising from existing interactive provers (such as Isabelle, COQ, or *Lego*), it assumes that provers implement a single-threaded state machine model, with states *toplevel*, *file open*, *theory open* and *proof open*.

Fig. 1: PG Kit System Architecture

Displays, on the other hand, are assumed to be nearly stateless. Through the display, the user edits the proof text and triggers prover actions, e.g., by requesting that a part of the proof script is processed. Abstractly, the broker mediates between the nearly stateless display protocol PGIP_D, and the statefull prover protocol PGIP_P; it keeps track of the prover states, and translates display state change requests into sequences of concrete prover commands, which change the state of the prover as required.
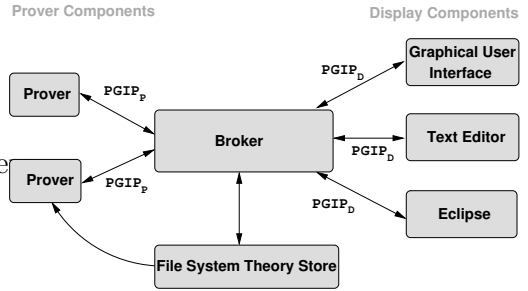
1.2 PlatΩ system architecture

The aim of the *Plat*Ω system is to support the transparent integration of theorem provers into standard scientific text editors. The intention is that the author can write and freely edit a document with high-quality typesetting without fully imposing a restricted, formal language; proof support is provided in the same environment and in the same format. The *Plat*Ω system is the middleware that mediates between the text editor and the prover and currently connects the text editor $\text{T}_{\!E}\!\text{X}_{\text{MACS}}$ and the theorem prover ΩMEGA. For the architecture of the system, see Fig. 2.
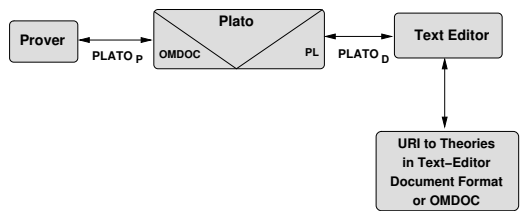

Fig. 2: *Plat*Ω System Architecture

1.3 Outline

The rest of the paper is structured as follows. In Section 2 we give a scenario for conducting a simple proof, and describe the interaction processes in *Plat*Ω and in Proof General. Section 3 begins discussion of our proposal to merge the two architectures, explaining how to extend PGIP to support documents
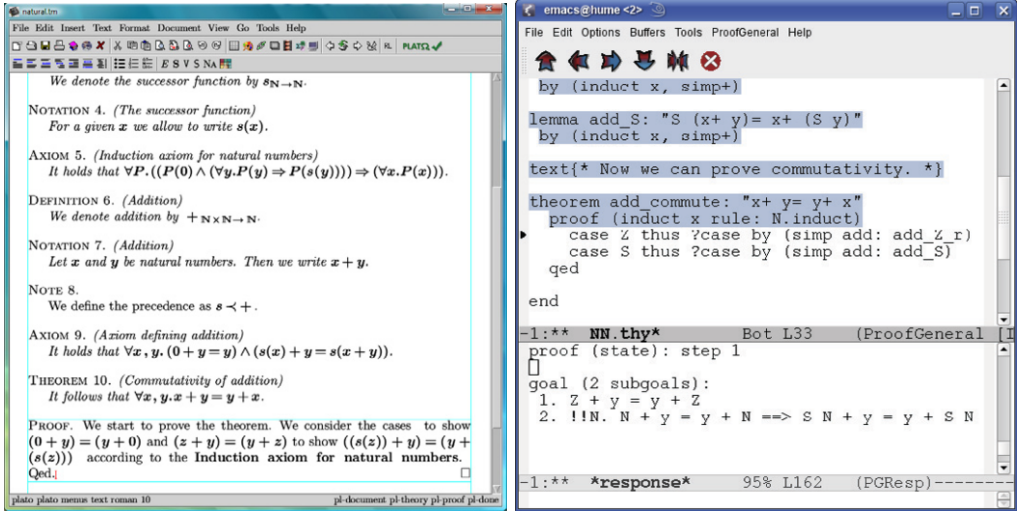
Fig. 3. Formalisation of the example scenario in $\TeX_{\text{MACS}}$ and PG Kit.

with more structure and multiple points of focus. Section 4 describes how to extend PGIP with a menu facility like that provided in *PlatΩ*, and Section 5 describes how to handle multiple displays, extending what is presently possible in *PlatΩ*. To complete our proposal, Section 6 explains how we can reconcile semi-structured documents with PGIP flat-structured documents, to connect theorem provers based on classical flat structured procedural proofs with our enhanced middleware for a richer document format. Section 7 discusses related work and future plans.

2   Interaction in *PlatΩ* and Proof General

We illustrate the overall functionality and workflow of *PlatΩ* and PG Kit with the following example, in which student Eva wants to prove the commutativity of addition in the standard Peano axiomatisation. Eva is typing this proof in a text editor, $\TeX_{\text{MACS}}$ or EMACS, and receives assistance from a *theorem prover*, ΩMEGA or Isabelle, for *PlatΩ* and PG Kit respectively (cf. Fig. 3). Eva's authoring process splits into the following five phases:

Phase 1. After having specified the theory and the conjecture

$$\forall x, y.x + y = y + x \tag{1}$$

in the text editor the document is passed to the theorem prover.

Phase 2. Eva begins to prove the conjecture. She does an induction on x and gets stuck with the subgoals: (1a) $0 + y = y + 0$ and (1b) $(z + y = y + z) \Rightarrow (s(z) + y = y + s(z))$.

Phase 3. She quickly realises that two lemmas are needed. Hence, she adds the following two lemmas somewhere in the document:

$$\forall x.0 + x = x + 0 \tag{2}$$

$$\forall x, y.(x + y = y + x) \Rightarrow (s(x) + y = y + s(x)) \tag{3}$$

Phase 4. Eva then tackles these lemmas one by one: for each, doing an induction on x and simplifying the cases proves the lemmas.

Phase 5. Eva then continues the proof of (1) by applying both lemmas to (1a) and (1b) respectively, which completes the proof.

2.1 PlatΩ

PlatΩ uses a custom XML document format called PL to connect to the text editor. The PL document contains markup for theories, theory items and linear, text-style proofs, and also notation definitions for defined concepts. Formulas in axioms, lemmas and proofs are in the standard, non-annotated LATEX-like syntax of T$_E$X$_{MACS}$. To connect to the theorem prover, *Plat*Ω uses OMDoc for the hierarchical, axiomatic theories and another custom XML format (TL) for the proofs.[1] *Plat*Ω holds the representations simultaneously, with a mapping that relates parts of the PL document to parts of the OMDoc(TL) document; a major task of the system is to propagate changes between the documents and maintain the mapping.

The text editor interface protocol (PLATO$_D$, see Fig.2) uses XML-RPC, with methods for complete document upload, service requests for specific parts of the PL document, and the execution of specific prover commands. On receiving a new document version, *Plat*Ω parses the live formulas using the document notations, producing *OpenMath* formulas. If a parse error occurs, an error description is returned to the editor. Otherwise *Plat*Ω performs an XML-based difference analysis [11] against the old PL document, resulting in a list of XUpdate modifications,[2] which are transformed into XUpdate modifications for the OMDoc(TL) document.

The interface to the theorem prover (PLATO$_P$) also uses XML-RPC, with methods for applying XUpdate modifications, service requests for parts of the OMDoc(TL) document, and executing specific prover commands. Applying an XUpdate modification may result in an error (e.g. a type error) or is simply acknowledged; either response is then relayed by *Plat*Ω to the display as an answer to the corresponding document upload method call. The result of a service request is a menu description in a custom XML format. That

---

[1] The next version of *Plat*Ω will use the OMDoc format for proofs, though still with ΩMEGA specific justifications for proof steps.

[2] see xmldb-org.sourceforge.net/xupdate/

menu is relayed to the display as a reply to the corresponding service request, rendering *OpenMath* formulas in the menu into T$_E$X$_{MACS}$ syntax using the notation information already used for parsing.

The result of executing a menu action is a list of XUpdates, which can either patch the menu (for lazy computation of sub-menus), or patch the document (for instance, inserting a subproof). *Plat*Ω transforms these OM-Doc(TL) patches into PL patches and renders occurring *OpenMath* formulas into T$_E$X$_{MACS}$ markup before sending the patch to the text editor.

Semantic Citation. A characteristic of *Plat*Ω is that everything that can be used comes from a document. Hence, there is a specific mechanism to "semantically" cite other T$_E$X$_{MACS}$ documents (see Fig. 2); these appear as normal citations in the editor but behind the scenes, are uploaded into *Plat*Ω, which then passes them to ΩMEGA. As a consequence, *Plat*Ω does not allow reuse of theories that are predefined in the theorem prover.

We now illustrate *Plat*Ω by describing the phases of the example scenario.

Phase 1. First, the whole document is passed from T$_E$X$_{MACS}$ to *Plat*Ω which extracts the formal content of the document including notational information to parse formulas. From the document, *Plat*Ω builds up the corresponding OMDoc theories and passes them as an XUpdate to ΩMEGA, which builds up the internal representation of the theory and initialises a proof for the open conjecture.

Phase 2. To start the proof of the theorem, Eva requests a menu from ΩMEGA, which returns a menu that lists the available strategies. Eva selects the strategy `InductThenSimplify`, which applies an induction on x to the open conjecture, simplifies the resulting subgoals terminates with the two open subgoals. This partial proof for Theorem (1) inside ΩMEGA is compiled into patch description and then passed to *Plat*Ω. *Plat*Ω transforms it into a patch for T$_E$X$_{MACS}$ by rearranging the obtained tree-like subproof representation into a linear, text-style proof representation using pseudo-natural language, and rendering the formulas using the memorised notational information.

Phase 3. After the two lemmas are written in the document, the whole document is uploaded and, after parsing, the difference analysis computes the patch to add the two lemmas. This is transformed into a patch description to add their formal counter-parts as open conjectures to the theory and sent to ΩMEGA. ΩMEGA, in turn, triggers the initialisation of two new active proofs.

Phase 4. Eva uses for both lemmas the strategy `InductThenSimplify` (again suggested by ΩMEGA in a menu) which succeeds in proving them. The resulting proof descriptions are again transformed by *Plat*Ω into proof patches

for the document and both lemmas are immediately available in the ongoing proof of Theorem (1).

Phase 5. ΩMEGA proposes in a menu to apply the lemma (2) to the subgoal (1a) and the lemma (3) to the subgoal (1b). Eva selects these suggestions one by one, which then completes the proof inside ΩMEGA. Subsequently, only the proof patch descriptions are transformed into patches for the T$_{E}$X$_{MACS}$ document as before.

2.2   Proof General

Unlike OMDoc, PGIP is not a proof format, nor does the PG Kit prescribe one. Instead, PGIP uses proofs written in the prover's native syntax, which are lightly marked up to exhibit existing implicit structure. The mark up divides the text into text spans, corresponding to prover commands which can be executed one-by-one in sequence. Different commands have different mark up, characterising e.g., start of a proof, a proof step, or (un)successful completion of a proof, as in:

  *<opengoal>***theorem** add_commute: "x+ y= y+ x"*</opengoal>*
   *<proofstep>***proof** (induct x rule: N.induct)*</proofstep>*

Elements like <opengoal> do not carry an inherent semantics (and they cannot be sent to the prover on their own), they merely make it clear that e.g. the command **theorem** add_commute: "..." starts the proof. Each of these text spans has a state; the main ones are parsed, processed and outdated. Proving a given theorem means to turn the whole proof into the processed state, meaning that the prover has successfully proved it. Returning to the scenario, we discuss the flow of events between the Emacs display, the PG Kit broker and the Isabelle prover.

Phase 1. Eva starts with an initial plain text Isabelle file, giving the definitions for the natural numbers, addition and the conjecture. She requests the file to be loaded, causing the broker to read it and send the contents to Isabelle for parsing. While this happens, the display shows the unparsed text to give immediate feedback. Isabelle returns the parsed file, which is then inserted into the Emacs buffer.

Phase 2. Eva now wants to prove the conjecture. She requests the conjecture to become processed so she can work on the proof (a command <setcmdstatus> is sent to the broker). This triggers sending a series of commands to Isabelle, ending with the conjecture statement. Isabelle answers with the open subgoal, which is then shown on the display.

    Eva attempts proof by induction. She writes the appropriate Isabelle com-

mands (proof (induct x rule: N.induct)). The new text is sent to the broker and then on to Isabelle for parsing. Once parsed the broker breaks the text into separately processable spans (here, only one), which is sent back to the display. Now Eva asks for the proof step to be processed, which sends the actual proof text to Isabelle, which answers with two open subgoals.

Phase 3. Realising she needs additional lemmas, and knowing Isabelle's linear visibility, Eva knows she has to insert two lemmas before the main theorem she is trying to prove. Since she cannot edit text which is in state *processed*, she first requests the text to change state to *outdated*. This causes a few undo messages to be sent to the prover to undo the last proof commands, resetting Isabelle's state back to where it has not processed the start of the main proof yet. Eva then inserts the needed lemmas in the document, and has them parsed as before.

Phase 4. Eva processes the lemma, and sees a message indicating that the proof worked. She finishes the other lemma similarly.

Phase 5. Eva returns to the main proof, editing the induction proof by inserting the induction base and induction step. Fig. 3 (right) shows the Emacs display at this point: the window is split in two parts, with the proof script in the upper part and the prover responses displayed below. The top portion of the proof script is blue, showing it has been processed, indicating the linear point of focus. After the induction step succeeds, Eva closes the proof with the command qed, which registers the theorem with the authorities. By turning the state of this closing command to *processed*, the proof is successfully finished.

3   Semi-Structured Documents

We have now seen how $Plat\Omega$ and the PG Kit handle documents. The architecture is similar: a central component handles the actual document, managing communication with the prover on one side and a user-interface component on the other side. The main differences are technical, summarised in the first two columns of Tables 1 and 2. Given the similarity, the question naturally arises: can we overcome these differences and provide a unified framework? This section will tentatively answer in the positive by extending PGIP on the prover side with the necessary new concepts (Section 3.1) and multiple foci (Section 3.2), and by using XUpdate pervasively on the display side (Section 3.3). The right-most columns of Tables 1 and 2 show the technical unification for the proposed *PGIP 2*.

	*Plat*Ω Display	PG Kit Display	*PGIP 2* Display
Document format	XML	Plain text	XML
Document syntax	T_EX_{MACS}	ASCII	Generic
Change protocol	XUpdate	PGIP$_D$	XUpdate
Change management	Dynamic Notation	Provided by prover	Provided by prover or display
Operations supported	Context-dependent menus	Global menus, typed operations	Context-dependent menus, typed operations

Table 1. Summary of differences between the Display Interfaces of *Plat*Ω and PG Kit

	*Plat*Ω Prover	PG Kit Prover	*PGIP 2* Prover
Document format	XML	Plain text	XML
Document syntax	OMDoc	Native prover syntax	Generic
Change protocol	XUpdate	PGIP$_P$	XUpdate/PGIP$_P$
Change management	Provided by Maya	Provided by Prover	Provided by Prover
Prover support	ΩMEGA	Generic (Coq, Isabelle, etc)	Generic (Coq, Isabelle, ΩMEGA, etc)
Operations supported	Context-dependent menus	Global menus, typed operations	Context-dependent menus, typed operations

Table 2. Summary of differences between the Prover Interfaces of *Plat*Ω and PG Kit

3.1 Document Formats

The two different *document formats* can both be treated as arbitrary XML, with the difference that for *Plat*Ω and OMDoc, there is deep structure inside the proof script (i.e., inside goals, proof steps etc) whereas in the case of PG Kit, there is only a shallow XML structure where the proof script is mainly plain text. To overcome this difference, we allow *PGIP 2* proof scripts to contain arbitrary marked-up XML instead of marked-up plain text, turning the document into a proper XML tree. Here is the present PGIP schema, excerpted and slightly simplified: [3]

```
opentheory  = element opentheory { thyname_attr, parentnames_attr?, plaintext }
closetheory = element closetheory { plaintext }
theoryitem  = element theoryitem { objtype_attr, plaintext }
openblock   = element openblock { objtype_attr, plaintext }
closeblock  = element closeblock { }
opengoal    = element opengoal  { thmname_attr?, plaintext }
proofstep   = element proofstep { plaintext }
closegoal   = element closegoal { plaintext }
```

The proposed *PGIP 2* amends this as follows, again excerpted:

```
theory          = element theory     { thyname_attr, parentnames_attr?, any }
theoryitem      = element theoryitem { objtype_attr, any }
block           = element block      { objtype_attr, xref_attr?, any }
```

---

[3] This XML schema is written in RELAX NG, which can be read much as a BNF grammar, with non-terminals named on the left and `element` and `attribute` introducing terminals; see http://relaxng.org/.

```
<assertion>theorem add_commute: &quot;x+ y= y+ (x::N)&quot;
<block objtype="proof body">
 <proofstep>proof (induct x rule: N.induct)</proofstep>
 <proofstep>case Z</proofstep><assertion>thus ?case
  <block objtype="proof body"><endproof status="proven">by (simp add: add_Z_r)</endproof>
  </block></assertion>
 <proofstep>case S</proofstep><assertion>thus ?case
  <block objtype="proof body"><endproof status="proven">by (simp add: add_S)</endproof>
  </block></assertion>
 <endproof status="proven">qed</endproof></block></assertion>
```

Fig. 4. Excerpt from the short example proof, marked up with *PGIP 2* (edited slightly for readability).

```
assertion       = element assertion  { thmname_attr?, id_attr?, any }
proofstep       = element proofstep  { xref_attr?, any }
endproof        = element endproof   { xref_attr?, proofstatus_attr?, any }

id_attr         = attribute xml:id
thmname_attr    = attribute thmname { xml:id }
thyname_attr    = attribute thyname { xml:id }
xref_attr       = attribute xref
proofstatus_attr = attribute ("proven"|"assert"|"unproven")

any             = ( text | anyElement ) *
anyElement      = element * { ( attribute * { text } | any ) * }
text            = element text { plaintext }
```

There are two major changes here: (i) arbitrary XML can occur where before only text was allowed; of course, the prover must understand whatever XML syntax is used here (e.g. ΩMEGA can understand OMDOC); (ii) instead of a flat list structure, we now use a proper tree; that is, a theory is not everything between an <opentheory> and <closetheory> element, but the contents of the <theory> element; and similarly, a proof is not everything between <opengoal> and <closegoal>, but the contents of the <block> element of type proofbody that belongs to an <assertion> element. The <endproof> element replaces <closegoal> and can be annotated with status information about the proof proven, assert, or unproven. Another extension is the corresponding attributes xml:id for the <assertion>, and xref for the <block> elements, which allow assertions to refer to proofs which are elsewhere in the document, and not directly following the assertion. These attributes are optional, and may only appear in the display protocol (i.e., between displays and the broker); we assume that provers always expect proof scripts to be in linear order, and it is the responsibility of the broker to rearrange them if necessary before sending them to be checked.

Furthermore, the broker must be able to divine the structure in an OM-DOC proof; e.g., the ΩMEGA prover or a component acting on its behalf must answer parse requests, and return XML documents using these elements. The revised version of our example proof with the *PGIP 2* markup is shown in Fig. 4.

3.2 Multiple Foci

The present PGIP prover protocol imposes an abstract state machine model which the prover is required to implement. ΩMEGA can be made to fit this model, but beyond that provides multiple foci. By this we mean that it can keep track of more than one active proof at a time and switch between them. Ignoring this would lose potential benefits (such as the ability to use a natively multi-threaded implementation of the prover) unnecessarily, and it is easy to accommodate into PGIP: we merely need to add an attribute to the prover commands to identify the focus. Some of these attributes already exist for the display protocol, where files are identified by a unique identifier (`srcid`). By adding unique identifiers also for theories and proofs, the prover can identify which ongoing proof a proof step belongs to, and use the appropriate thread to handle it. To allow fall-back to the simple case, we need a prover configuration setting to declare if multiple foci are available.

3.3 XUpdate

In the $PGIP_D$ protocol, changes in the document are communicated using specialised commands <`createcmd`> and <`editcmd`> from the display to the broker, and <`newcmd`>, <`delcmd`> and <`replacecmd`> from the broker to the display (so the protocol is asymmetric). We can rephrase this in terms of XUpdate; the unique identifier given by the broker to each command contained in the `cmdid` attribute allows to easily identify an object by the XPath expression `*[cmd=`c`]`. The key advantages of XUpdate are that it is standard, symmetric, and allows several changes to be bundled up in one <`xupdate:modifications`> packet that is processed atomically, adding a transaction capability to the display protocol.

Strict conformance to this protocol requires the displays to calculate or track differences, i.e., send only the smallest update. Not all displays (editors) are that sophisticated, and it is unrealistic to expect them to be; a basic design assumption of PG Kit is that the broker should contain the intelligence needed to handle proof documents, and displays should be easy to implement. Hence, displays can send back the whole document as changed, and expect the broker to figure out the actual differences (*whole-document* editing) using the XML difference mechanism from [11] that can take some semantics into account as already used by *Plat*Ω.

In the $PGIP_P$ protocol, changes in the document communicated via XUpdate must be mapped to changes in the prover state. In the previous version of PGIP, this was done by the broker, because the single-focus state model does not easily accommodate arbitrary changes to the document. However, the multiple-focus extensions as described in Sect. 3.2 amount to supporting

XUpdate on the prover side; if the prover offers this support, it should be exploited, otherwise we use PGIP$_P$.

3.4 Protocols

The underlying transport protocol of PGIP was custom designed, because communication with an interactive prover fits no simple standard single-request single-response protocol: the prover asynchronously sends information about proofs in progress, and we crucially need the ability to send out-of-band interrupts. However, on the display side these reasons do not apply; we might use XML-RPC or even plain HTTP in a REST architecture. REST (representational state transfer [6]) is an architecture style for distributed applications which, in a nutshell, is based on providing resources that are addressed using URIs and manipulated using four basic operations: creating, reading, updating and deleting ("CRUD"). The resources provided by the broker are as follows:

- The broker itself, with the list of all known provers, all loaded files, a global menu, and global configurations as attributes;
- each prover is a resource, with its status (not running or running, busy, ready, exited) as attributes, preferences for this prover, all identifiers for the prover, messages sent by the prover, its proof state, and prover-specific configurations such as types, icons, help documents, and a menu;
- and each file is a resource, containing the document as a structured text, and the status (saved or modified) as attributes.

Clients affect changes to the document by the XUpdate messages above, and trigger broker actions by changing the attributes. For example, to start a prover, the client will change the status of the prover resource from not running to running. Here, bundling up changes into one XUpdate modification becomes useful, as it allows displays to send several changes to the document resource in one transaction.

In the REST view, changes in the document produce a new version of the document; special links will always point to the latest version of the document, but may require the client to refresh them. This allows multiple displays; we will exploit this in Section 5. This REST-style interface is an *alternative* to the statefull protocol using PGIP or XML-RPC; in the long run, the broker will support both.

4 Service Menus

PGIP 2 supports context-sensitive service menus in the display for the interaction with the prover. The user can request a menu for any object in the

document; through the broker this triggers menu generation in the prover for the formal counterparts of the selected object. It remains to fix a format for menu descriptions.

Traditionally, menus are fully specified and include *all* submenus and the leafs are *all* actions with *all* possible actual arguments. Executing an action triggers modifications of the document and the menu is closed. For theorem provers, computing all submenus and action instances can be expensive and unduly delay the appearance of the menu. For example, a menu entry for applying a lemma would contain as a submenu all available lemmas, and for each lemma, all possibilities to apply it in the current proof situation. Once the user makes a choice, the other possibilities are discarded. So on-demand computation of submenus is desirable.

The *Plat*Ω system allows lazy menus, where actions executed in a menu can generate a submenu. The entire menu is modified by replacing the leaf action by the generated submenu. We adapt this model for *PGIP 2* also. However, not all displays are able to incorporate changes to live menus; therefore we do not impose the partial menu representation. Instead, the display specifies in the service request whether it will accept a lazy menu.

The description language for these menus is:

```
menu     = element menu { id,  name, for_attr, ((menu|action)+ | error) }
action   = element action { id, name, argument* }
argument = element argument { id, name, custom }
custom   = element custom { id, alt, any }
error    = element error { id, text }
```

(using the `any` element from above). A menu entry is rendered by its name and an action is rendered by its name and its arguments. Arguments are rendered with the given custom object, e.g., an *OpenMath* formula or some standard T$_E$X$_{MACS}$ markup. The `alt` attribute provides a fallback ASCII representation in case the custom object content cannot be displayed.

When the user chooses an action, it is executed on the specified arguments. The result of the action may be an XUpdate patch to the document. This is sent to the broker and then on to the display, which incorporates the patch and closes the menu. Alternatively it is a patch for the menu only: in this case the action is replaced in the menu by the new submenu. If a submenu is empty, i.e., there are no possibilities to refine the abstract action, then the submenu consists solely of an error that describes the cause, which should be displayed inside the menu.

Example 4.1 We illustrate the interactions when requesting a menu for a display that is able to deal with partial menus. In **Phase 5** of the scenario, Eva requests a menu for the subgoal (1a) $0 + y = y + 0$.

Menu Request: The menu is requested for a specific XPath of the document

and the broker maps it to a menu request to the prover for the corresponding formal object, that is, the open goal that corresponds to (1a) $0+y = y+0$. The prover generates a top-level menu with the actions "Apply Axiom or Lemma", "Apply Tactic" and returns that to the display via the broker.

Lazy Menu Deployment: Selecting "Apply Axiom or Lemma" triggers computing a submenu containing all available axioms and lemmas. That submenu is sent as an XUpdate patch to the display to replace the action "Apply Axiom or Lemma". Selecting Lemma (2) triggers the prover action that computes the possible ways to apply the lemma on the open goal. In this case the resulting submenu has a few entries for the cases where the lemma is applied from left to right and one case for the application of the lemma from right to left. The submenu is sent as an XUpdate patch to the display to replace the action "Apply Lemma (2)".

Menu Action Execution: The final top level action execution triggers applying the specific instance of the Lemma in the prover, modifying the formal proof. The modification is propagated via the broker to the display, either as an XUpdate patch for the document if the display is able to deal itself with these; otherwise the broker computes the new document version and forwards only the new document. Additionally, a patch description is sent for closing the menu.

5 Multiple Displays

The architecture of our new system inherits from the architecture of PG Kit (Fig. 1), which allows multiple displays to be connected to the broker. One use for this is to allow multiple views on a proof-in-progress, e.g., a display that shows a dependency graph, or a graphical interpretation of a proof (perhaps rendering geometric arguments diagrammatically), alongside the main proof editing display. These displays are prover-specific, but fit smoothly into the general architecture.

Another use for multiple displays is to support more than one display to change the document. For this we need a way to synchronise input from different displays. A way to do this is for the broker to act as a simple kind of source control repository, illustrated by example in Fig. 5. This works as follows:

- The broker maintains the latest revision (the head) of a document, and for each display, a copy of the latest revision acknowledged by that display. In Fig. 5, the head is Rev. 47.
- When Display 1 sends a change (Rev. 47'), the change is committed to the new head (Rev. 48), and the new revision broadcast to all displays.
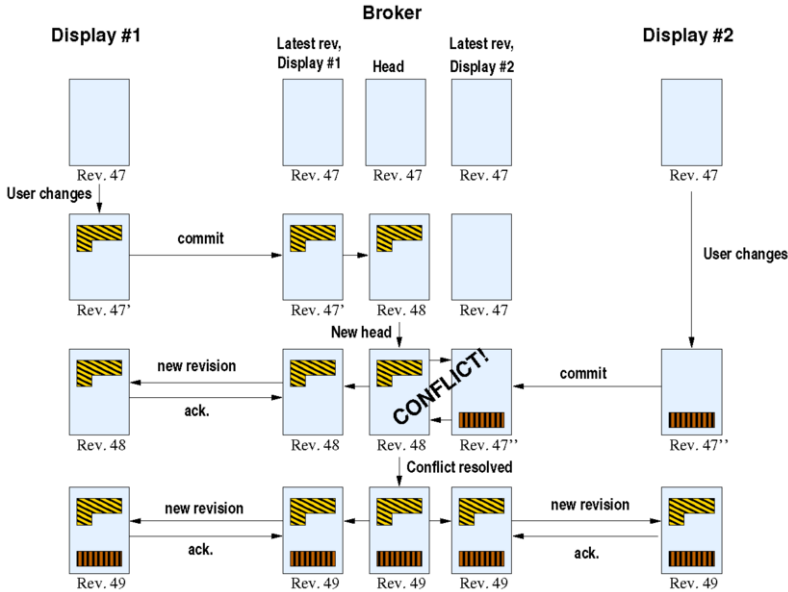
Fig. 5. Example for Editing via Multiple Displays

- Display 1 acknowledges the new revision. However, Display 2 has been changed meanwhile, so it does not acknowledge, instead attempting to commit its changes (Rev. 47"). The broker spots a potential conflict, and (in this case) merges the disjoint changes between 48 and 47" with respect to 47 into the current head revision without trouble. The merged document becomes the new head (Rev. 49), and is broadcast to all displays. Since no further changes have been made in the display, they both acknowledge.

If a conflict that cannot be merged occurs, the broker sends the merged document including conflict descriptions back to the display (using an extension to XUpdate to markup the conflicts, as in [11, Sect. 7.1.3]). The display (or the user) then needs to resolve the conflicts, and send in new changes.

This strategy is simple and flexible: displays could always send in changes to the whole document, and only acknowledge changes sent from the broker if the user has not edited the document at all. Alternatively, since this may create extensive conflicts without realising, displays might block between commit and acknowledge, or attempt to merge eagerly with new revisions sent by the broker.

6   Supporting Multiple Document Formats

So far the document format used with the display and the prover are essentially the same: for instance, in the classical PGIP with Isabelle, the document on

the display is an Isabelle input file with additional markup. With the extension for arbitrary XML document formats in Section 3, we could connect a display and prover that both use OMDoc. But we cannot yet connect two different formats, say, connecting the display based on a document format D, with a prover that works on a different format P. This is the final missing piece of the architecture for emulating $Plat\Omega$, which connects $D =$PL in the $\mathsf{PLATO_D}$ protocol to $\TeX_{\text{MACS}}$ through to $P =$OMDoc format as used in the $\mathsf{PLATO_P}$ protocol to $\Omega\textsc{mega}$.

To support multiple document formats at once, we propose to use a central structured document format B in the PG Kit broker that is annotated by PGIP markup. The broker does not need to know the semantics of the format B. Instead, dedicated translators are required for each target document format, translating $D \rightleftharpoons B$ and $B \rightleftharpoons P$. Each translator maintains a document representation mapping, and converts XUpdate-patches in either direction, much as the $Plat\Omega$ system does between the PL representation and the OMDoc representation as described in Section 2.1. The advantage of using the central format B is that provers do not need to be adapted to the document format of every display.

Experience with $Plat\Omega$ suggest the main difficulty lies in translating patch descriptions between the different document formats. Suppose we connect structured $\TeX_{\text{MACS}}$ documents with plain text Isabelle proof scripts, and choose OMDoc as the broker's central document format. On the display side we have a translator component that mediates between $\TeX_{\text{MACS}}$ documents and OMDoc. Prover side, a translator mediates between OMDoc and Isabelle ASCII text. We encode ASCII documents in XML as `<document><text>...</text>...<text>...</text></document>`, where text nodes are whitespace preserving.

Consider now the interactions when uploading and patching a document. Menu interactions are basically passed unchanged, but document patches must be translated. Since $Plat\Omega$ can already mediate between the $\TeX_{\text{MACS}}$ and OMDoc formats, we need only one new translator for OMDoc and Isabelle, implementing:

XUpdate flattening going from OMDoc to ASCII, the structured XML representation must be transformed into a linearised text representation. A mapping must be setup between XML ranges and text ranges, i.e., the start XPath maps to the start text position (relative to the last range) and the end XPath maps to the end text position (relative to the last range). Start and end XPaths have the same parent XPath by definition. To flatten patches, the affected XML ranges must be recomputed and the mapping adapted; additions in the patch are flattened similarly.

XUpdate lifting: going from ASCII to OMDOC, the text spans must be lifted to the XML representation. Generally, this is done by mapping text spans to the corresponding sequence of adjacent XML ranges. As an invariant it must be checked whether the resulting sequence can be expressed by start and end XPaths with the same parent XPath. Similar to flattening, the mapping has to be adapted between text ranges and XML ranges.

Of course, the devil lies in the detail: OMDOC allows some embedding of legacy formats, but to usefully translate to and from Isabelle, we must accurately interpret a subset of syntax that reflects theory structure, and have some confidence about the correctness of the interpretation.

On the other side, we can now provide translators for further displays with advanced layout possibilities, such as Word 2007. The translator component must abstract the display document format to simplify it for the broker: e.g. in Word 2007, the document body is extracted and information about fonts, colours and spacing is stripped. On the way back, annotations are extracted from the patches coming from the broker, which guide heuristics for layout of new or modified text.

7   Related Work, Conclusion and Next Steps

Many user interfaces to theorem provers are similar to the Proof General style of line-by-line and single focus interaction using ASCII input files in native theorem prover format. Often, a custom interaction protocol is used. The main novelties for *PGIP 2* proposed here are: (i) to handle semi-structured XML documents as input formats; (ii) to allow the user to work on different parts of a document in parallel by using multiple foci; (iii) to allow the theorem prover to change parts of the input document, possibly using menus, and (iv) to have multiple views and editing of the same document in different displays.

With respect to (i), the *MathsTiles* system [5] also allows to map semi-structured documents towards several special reasoning systems. However, the mapping is only unidirectional from the display to the reasoners and also does not support multiple displays and conjunctive editing. With respect to (ii), as far as we know, the ΩMEGA system is the only prover that currently supports semi-structured document input and multiple foci. State information describing which parts of the document have been checked by ΩMEGA is managed in an ad hoc style; making this explicit in the multi-threaded state machine model in *PGIP 2* markup improves this and suggests ways to migrate a single-threaded theorem prover to a multi-threaded mode.

The IAPP infrastructure introduced in [7] is a new architecture designed to

support asynchronous processing using a communication protocol that transfers the ownership of proof commands between the interface and the prover. Thus IAPP locks parts of the document to prevent conflicts, whereas we use a versioning-based approach where conflicts are resolved in the broker (using undo operations); the obvious advantage is that the interface does not have to wait for the prover to release parts of the document before editing. Additionally, IAPP tracks changes using a tight integration with the interface based on the assumption that the interface implements the OBSERVER pattern [8]. We do not impose such strong requirements on the interface because we use a difference analysis mechanism to compute the changes. This allows us to support multiple views equally on the interface side.

Multiple views have been used in various forms in different systems, but not in a clearly distributed way that also allows editing, as in *PGIP 2*. In LΩui [12] the display was split into a graph view on the proof and a display of the actual proof goals: those were based on pretty-printing and graph-visualisation tools built into the same display component. MATITA's user interface [1] has one proof script buffer and a display for the actual proof goal: the latter uses GtkMathview based on MathML representation of formulas that is generated from the internal representation of MATITA. GEOPROOF [10] allows one to generate COQ proofs from its internal, geometric representation which can be viewed in COQIDE [13]: this comes close to what we propose with multiple displays, except that currently there is no way back from COQ into GEOPROOF.[4] The infrastructure of *PGIP 2* and a (partial) mapping from COQ into GEOPROOF would allow for simultaneously working in GEOPROOF and COQIDE. Away from proof assistant systems, multiple views are familiar in IDEs for programming languages such as Eclipse and NetBeans: there the same file may be presented in different ways in different windows (e.g., code and model), and either updated dynamically in step, or at clearly defined points in the interaction (e.g., window activation).

The ability to extend the input document by incorporating information from the prover has also been supported in various ways before. An example besides the general change mechanism of *Plat*Ω/ΩMEGA is that of MATITA, which can generate a *tinycal* proof script from the GUI interactions on goals, and include it into the overall document. We hope that a generic infrastructure would allow functionality like this to be reused between systems. The facility to include information from the prover together with the multiple foci provide a good basis to use PG Kit for provers like Mizar, PVS and Agda that have different, non-linear interaction styles. The details of adapting to further

---

[4] This could, of course, only be a partial mapping since not all COQ-proofs are geometric proofs.

prover interaction styles is left to future work.

The main next step is to implement our planned *PGIP 2* and to rebuild *Plat*Ω's functionality on that basis. Future work will also be devoted to use Word 2007 and OpenOffice as displays and especially to build bi-directional transformers between prover-specific textual input files and corresponding OMDOC representations. We hope this will lead to a rich family of improved prover user interfaces.

References

[1] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007. Special Issue on User Interfaces in Theorem Proving.

[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. Springer, 2000.

[3] David Aspinall, Christoph Lüth, and Daniel Winterstein. A framework for interactive proof. In *Mathematical Knowledge Management MKM 2007*, LNAI 4573, pages 161– 175. Springer, 2007.

[4] S. Autexier, A. Fiedler, T. Neumann, and M. Wagner. Supporting user-defined notations when integrating scientific text-editors with proof assistance. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, LNAI. Springer, june 2007.

[5] W. Billingsley and P. Robinson. Student Proof Exercises using MathsTiles and Isabelle/HOL in an Intelligent Book. *Journal of Automated Reasoning*, 39(2):181–218, August 2007.

[6] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Archite ctures*. PhD thesis, University of California, Irvine, 2000.

[7] H. Gast. Managing proof documents for asynchronous processing. In S. Autexier and C. Benzmüller, editors, *8th Workshop on User Interfaces for Theorem Provers (UITP'08)*, August 2008.

[8] E. Grammar, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] C. Benzmüller M. Wagner, S. Autexier. Plato: A mediator between text-editors and proof assistance systems. In Christoph Benzmüller Serge Autexier, editor, *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*, volume 174(2) of *Electronic Notes on Theoretical Computer Science*, pages 87–107. Elsevier, april 2007.

[10] J. Narboux. A graphical user interface for formal proofs in geometry. *Journal of Automated Reasoning*, 39(2):161–180, 2007. Special Issue on User Interfaces in Theorem Proving.

[11] S. Radzevich. Semantic-based diff, patch and merge for XML documents. Master thesis, Saarland University, Saarbrücken, Germany, April 2006.

[12] J. Siekmann, S. Hess, C. Benzmüller, L. Cheikhrouhou, A. Fiedler, H. Horacek, M. Kohlhase, K. Konrad, A. Meier, E. Melis, M. Pollet, and V. Sorge. $\mathcal{LOUI}$: $\mathcal{L}$ovely Ω$\textsc{mega}$ $\mathcal{U}$ser $\mathcal{I}$nterface. *Formal Aspects of Computing*, 11:326–342, 1999.

[13] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.1*. INRIA, http://coq.inria.fr/doc-eng.html, 2008.