Commentary on PGIP

[Version 1.2, 2003/07/16 21:02:58, LaTeX: July 16, 2003]

David Aspinall Christoph Lüth

July 16, 2003

This document gives commentary on the definition of PGIP. The commentary is indended as a set of notes to help implementors of PGIP-enabled prover components; it does not (yet) form a complete description or motivation for the protocol. The RELAX-NG schemas for PGIP message and the PGML markup language are given in the appendix.

1 Basics

1. The PGIP protocol is intended as a mechanism for conducting interactive proof using PGIP-enabled software components. The aim of interaction is to produce one or more **proof scripts**.
2. A proof script has a textual representation as primary and resides in a file.

2 PGIP communication

1. A pair of components communicate by opening a channel (typically a Unix pipe or socket), where one end is designated the *proof assistant* (class `pa`; think server) and the other end is designated the *proof general interface* (class `pg`; think client).
2. PGIP communication proceeds by exchanging PGIP packets as XML documents belonging to the PGIP markup schema. A PGIP packet is contained by the `<pgip>` element.
3. The interface sends command requests to the prover, and processes responses which are returned. Unlike classical RPC conventions which are single-request single-response, a command request may cause several command responses, and it is occasionally possible that the prover generates "orphan" responses which do not correspond to any request from the interface.
4. Each PGIP packet contains a single PGIP message, along with identifying header information. The PGIP message may be a *command request* or a *command response*.
5. The interface should only attempt to send commands to the prover when it has received a `ready` message. On startup, the prover may issue some orphan responses, followed by a ready message.
6. Despite the classifications of `pa` and `pg`, the communicating components do not have to be exactly the prover and interface. It is also possible to have non-prover components which provide auxiliary services, and filtering components which process PGIP command streams.

3 PGIP and PGML markup

1. PGIP and PGML are separate document types:
   - PGML describes the markup for displayed text/graphics from the prover
   - PGIP describes the protocol for interacting with the prover
2. PGIP contains PGML in the same (default) namespace, so PGIP messages may contain PGML documents in certain places. PGML text is embedded with root `<pgml>`, which allows easy filtering by components concerned with display.

4 Prover to interface configuration

`<usespgip>`

- The prover reports which version of PGIP it supports.

`<usespgml>`

- The prover reports which version of PGML it supports.

`<pgmlconfig>`

- The prover reports its configuration for PGML.
- PGML can be configured for particular symbols. The prover reports the collection of symbols it will understand as input and omit as output, along with optional ASCII defaults. PGML symbol conventions define a large fixed set of named glyphs.

`<haspref>`

- The prover reports a user-level preference setting, along with a type and possible a default value.

`<prefval>`

- The prover reports a change in one of its preference settings, perhaps triggered by the interface.

`<guiconfig>`

- The prover specifies some basic object types it will let the interface manipulate (for example: theorem, theory, tactic, etc), together with the operations which are supported for those types.
- `opn` are commands which combine object values of the prover, in a functional manner. The `opcmd` should be some text fragment which produces the operation. The operations could be triggered in the interface by a drag-and-drop operation, or menu selection.
- `iopn` are operations which require some interactive input. They are configured
- `proofopn` are commands which produce text suitable for use as `<proofstep>`.
- As a general convention, if several operations are possible to produce a desired target object, then the prover will offer them in the choice that they were configured.

5 Prover control commands

`<proverinit>`

- Reset the prover to its initial state.

`<proverexit>`

- Exit the prover gracefully.

`<startquiet>`

- Ask the prover to turn off its output.

`<stopquiet>`

- Ask the prover to turn on its output again.

6 Prover output

`<ready>`

- The prover should issue a `ready/` message when it starts up, and each time it has completed processing a command from the interface.
- The interface should not send a command request until it has seen a `ready/` message. Input which is sent before then may cause buffer overflow, and more seriously, risks changing the prover state in an unpredictable way in case the previous command request fails.

`<displayarea>`

- PGIP assumes a display model which contains (at least) two display areas: the **message** area and the **displayarea**.
- Typically, both areas are shown in a single window. The display area is a possibly graphical area whereas the message area is a scrollable text widget that appears (for example) below the display area.
- The interface should maintain a display of all message area output that appears in response to a particular command. Between successive commands (i.e. on the first new message in response to the next command), the interface may (optionally) clear the message area.
- The interface should simply replace display area output whenever new display area output appears.
- Additional features may be desirable, such as allowing the user to keep a history of previous displays somehow (display pages by forwards/backwards keys; messages by text scrollbar).
- Occasionally, the prover may like to send hints that displays should be cleared, in `<cleardisplay>` commands. These should be obeyed.
- The interface is free to implement these displays in different ways, or even supress them entirely, insofar as that makes sense.

`<normalresponse>`

- All ordinary output from the prover appears under the `normalresponse` element. Typically the output will cause some effect on the interface display, although the interface may choose not to display some responses.
- A response which has attribute `urgent = "y"` should always be displayed to the user.
- A PGIP command may generate any number of normal responses, possibly over a long period of time, before the `ready` response is sent.

`<errorresponse>`

- The `errorresponse` element indicates an error condition has occurred.
- The `fatality` attribute of the error suggests what the interface should do:
  - a `nonfatal` error does not need any special action;
  - a `fatal` error implies that the last command issued from the interface has failed (a recoverable error condition);
  - a `panic` error implies an unrecoverable error condition: the connection between the components should be torn down.
- The location attibute allows for file/line-number locations to identify error positions, for example, for when a file is being read directly by the prover.
- A PGIP command may cause at most one error response to be generated. If an error response occurs, it must be the last response before a `ready` message.

`<scriptinsert>`

- This response contains some text which should be inserted literally into the proof script being constructed.
- The suggestion is that the interface immediately inserts this text, parses it, and sends it back to the proof assistant to conduct the next step in the proof. This protocol allows for "proof-by-pointing" or similar behaviour.

`<metainforesponse>`

- The `metainforesponse` element is used to categorize other kinds of prover-specific meta-information sent from the prover to the interface.
- At present, no generic meta-information is defined. Possible uses include output of dependency information, proof hints applicable for the current proof step, etc.
- Provers are free to implement their own meta-information responses which specific interfaces may interpret. This allows an method for extending the protocol incrementally in particular cases. Extensions which prove particularly useful may be incorporated into future versions.

Here are some example message patterns allowed by the PGIP message model:

provermsg	*provermsg*	*provermsg*
`<ready/>`	`<normalresponse>`	`<normalresponse>`
⋮	`<normalresponse>`	`<errorresponse>`
	`<ready/>`	`<ready/>`
	⋮	⋮

The *provermsg* is a message sent to the proof assistant and the responses are shown below. Responses all end in a ready message; the only possible exception is a panic error response, which indicates that the proof assistant has died (perhaps committed suicide) already.

7 Proof control commands

The PGIP proof model is to assume that the prover maintains a state which consits of a single possibly-open proof within a single possibly-open theory. [**FIXME**: explain further]

`<goal>`

- open a goal in ambient context

`<proofstep>`

- a specific proof command (perhaps configured via opcmd)

`<undostep>`

- undo the last proof step issued in currently open goal

`<closegoal>`

- complete & close current open proof (succeeds iff goal proven)

`<abortgoal>`

- give up on current open proof, close proof state, discard history

`<giveupgoal>`

- close current open proof, record as proof obl'n (sorry)

`<postponegoal>`

- close current open proof, retaining attempt in script (oops)

`<forget>`

- forget a theorem (or named target), outdating dependent theorems

`<restoregoal>`

- re-open previously postponed proof, outdating dependent theorems

Further notes:

1. Some of these operations have an effect on the proof script, namely: `<goal>` `<proofstep>` `<closegoal>` `<postponegoal>` `<giveupgoal>`. These operations will trigger a response which includes a `<scriptinsert>` message to insert the corresponding command into the proof script if it is successfully processed.
2. The other operations are meta-operations which correspond to script management behaviour: i.e., altering the interface's idea of "current position" in the incremental processing of a file.
3. As a later possibility, we may allow the prover provide a way to retain undo history across different proofs. For now we assume it does not, so we must replay a partial proof for a goal which is postponed.
4. We assume theorem names are unique amongst theorems and open/goals within the currently open theory. Individual proof steps may also have anchor names which can be passed to forget.
5. The interface manages outdating of the theorem dependencies within the open theory. By constrast, theory dependencies are managed by the prover and communicated to the interface.

8 Theory/file commands

PGIP assumes that the prover manages a notion of theory, and that there is a connection between theories and files. Specifically, a file may define some number of theories. The interface will use files to record the theories it constructs (but will only construct one theory per file).

PGIP assumes that the proof engine has three main states:

top level inspection/navigation of theories only

open theory may issue proof steps to construct objects, make defs, etc.

open theory & open proof may issue proof steps with aim of completing proof of some theorem. Prover records undo history for each step, but discards this history on proof completion.

This model only allows a single open theory. Nonetheless, it should be possible for the interface to provide extra structure and maintain an illusion of more than one open theory, without the prover needing to implement this directly. This can be done by judicious opening and closing of files, and automatic proof replay. Later on, we might extend PGIP to allow multiple open proofs to implemented within the prover to provide extra efficiency, to avoid too much proof replaying.

`<loadtheory>`

- load a file possibly containing a theory definition

`<opentheory>`

- begin construction of a new theory. The text allows some additional arguments to be given (e.g. ancestors)

`<closetheory>`

- complete construction of the currently open theory, saving it in the promised file.

`<retracttheory>`

- retract a theory. Applicable to open & closed theories.

`<openfile>`

- lock a file for constructing a proof text in the interface. The prover may check that the opened file does not already correspond to a processed theory.

`<closefile>`

- unlock a file, suggesting it has been processed completely (but incrementally via interface). A paranoid prover might want to check the file nonetheless.

`<abortfile>`

- unlock a file, suggesting it hasn't been processed

PGIP supposes that the interface has only partial knowledge about theories, and so the interface relies on the prover to send hints. Specifically, the next two messages may be sent *from* the prover. When the interface asks for a theory to be loaded, there may be a number of `<informtheoryloaded>` responses from the prover, and similarly for retraction.

`<informfileloaded>`

- prover informs interface a particular file is loaded

`<informfileretracted>`

- prover informs interface a particular file is outdated

A Schemas for PGIP and PGML

A.1 pgip.rnc

```
#
# RELAX NG Schema for PGIP, the Proof General Interface Protocol
#
# Authors:  David Aspinall, LFCS, University of Edinburgh
#           Christoph Lueth, University of Bremen
#
# Version: $Id: pgip.rnc,v 1.12 2003/07/16 20:57:00 da Exp $
#
# Status:  Experimental.
# For additional commentary, see the Proof General Kit white paper,
# available from http://www.proofgeneral.org/kit
#
# Advertised version: 1.0
#

## [ See rnc-temp-devel-notes.txt for possible changes to below - da ]


include "pgml.rnc"        # include PGML grammar

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

start = pgip | pgips        # pgips is the type of a log between
          # two components.

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

pgips = element pgips { pgip+ }

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


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

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

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


```

```
57
58
59  #  ========== PROVER CONFIGURATION ==========
60
61  proverconfig =
62     askpgip     #  what version of PGIP do you support?
63   | askpgml     #  what version of PGML do you support?
64   | askconfig     #  tell me about objects and operations
65   | askprefs     #  what preference settings do you offer?
66   | setpref     #  please set this preference value
67   | getpref     #  please tell me this preference value
68
69
70  name_attr = attribute name { token }  # identifiers must be XML tokens
71
72  prefcat_attr = attribute prefcategory { text}   # e.g. "expert", "internal", etc.
73        # could be used for tabs in dialog
74
75  askpgip    = element askpgip    { empty }
76  askpgml    = element askpgml    { empty }
77  askconfig = element askconfig { empty }
78  askprefs   = element askprefs   { empty }
79  setpref    = element setpref    { name_attr, prefcat_attr?, text }
80  getpref    = element getpref    { name_attr, prefcat_attr? }
81
82
83  #  ========== INTERFACE CONFIGURATION ==========
84
85  kitconfig =
86      usespgip     # I support PGIP, version ..
87    | usespgml     # I support PGML, version ..
88    | pgmlconfig      # configure PGML symbols
89    | haspref     # I have a preference setting ..
90    | prefval     # the current value of a preference is
91    | addids     # inform the interface about some known objects
92    | delids     # retract some known identifers
93    | menuadd     # add a menu entry
94    | menudel     # remove a menu entry
95    | guiconfig     # configure the following object types and operations
96
97  # version reporting
98  version_attr  = attribute version { text }
99  usespgml = element usespgml  { version_attr }
100 usespgip = element usespgip  { version_attr }
101
102 # PGML configuration
103 pgmlconfig = element pgmlconfig { pgmlconfigure+ }
104
105 # Types for config settings: corresponding data values should conform
106 # to representation for corresponding XML Schema 1.0 Datatypes.
107 # (This representation is verbose but helps for error checking later)
108
109 pgiptype    = pgipbool | pgipint | pgipstring | pgipchoice
110 pgipbool    = element pgipbool { empty }
111 pgipint     = element pgipint  { empty }
112 pgipstring = element pgipstring { empty }
113 pgipchoice = element pgipchoice { pgipchoiceitem+ }
114 pgipchoiceitem = element pgipchoiceitem { tag_attr?, pgiptype }
115 tag_attr = attribute tag { text }
116
117 # preferences
```

```
118 default_attr = attribute default { text }
119 descr_attr   = attribute descr { text }
120
121 haspref = element haspref  {
122    name_attr, descr_attr?, prefcat_attr?,
123    default_attr?, pgiptype
124 }
125
126 prefval = element prefval  { name_attr, text }
127
128 # menu items (incomplete)
129 path_attr = attribute path { text }
130
131 menuadd = element menuadd  { path_attr?, name_attr?, text }
132 menudel = element menudel  { path_attr?, name_attr?, text }
133
134 # GUI configuration information: objects, types, and operations
135
136 guiconfig =
137   element guiconfig { objtype*, opn*, iopn*, proofopn* }
138
139 objtype = element objtype { name_attr, descr_attr?, icon? }
140
141 icon  = element icon { xsd:base64Binary }  # image data for an icon
142
143 opsrc = element opsrc { list { token* } }  # source types: a space separated list
144 optrg = element optrg { token }     # the single target type
145 opcmd = element opcmd { text }              # prover command, with printf-style "%1"-args
146
147 opn  = element opn { name_attr, opsrc, optrg, opcmd }
148
149 # proof operations (no target sort: result is a proofcmd for script)
150 proofopn = element proofopn { name_attr, opsrc, opcmd }
151
152 # interactive operations - require some input
153 iopn = element iopn { name_attr, inputform, opsrc, optrg, opcmd }
154 inputform = element inputform { field+ }
155
156 # a field has a PGIP config type (int, string, bool, choice(c1...cn))
157 # and a name; under that name, it will be substituted into the command
158 # Ex. field name=number  opcmd="rtac %1 %number"
159
160 field = element field {
161    name_attr, pgiptype,
162    element prompt { text }
163 }
164
165
166
167 # identifier tables: these list known items of particular objtype.
168 # Might be used for completion or menu selection.
169 # May have a nested structure (but objtypes do not).
170
171 objtype_attr = attribute objtype { token }  # the name of an objtype
172
173 idtable = element idtable { objtype_attr, (identifier | idtable)* }
174
175 addids = element addids { idtable }
176 delids = element delids { idtable }
177
178 identifier = element identifier { token }
```

```
179
180
181
182
183 # ========== PROVER CONTROL ==========
184
185 provercontrol =
186    proverinit   # reset prover to its initial state
187  | proverexit   # exit prover
188  | startquiet     # stop prover sending proof state displays, non-urgent messages
189  | stopquiet      # turn on normal proof state & message displays
190
191 proverinit = element proverinit { empty }
192 proverexit = element proverexit { empty }  # exit prover
193 startquiet = element startquiet { empty }
194 stopquiet  = element stopquiet  { empty }
195
196
197 # ========== GENERAL PROVER OUTPUT/RESPONSES ==========
198
199 proveroutput =
200    ready  # prover is ready for input
201  | cleardisplay         # prover requests a display area to be cleared
202  | normalresponse       # prover outputs some display text
203  | errorresponse # prover indicates an error condition, with error message
204  | scriptinsert         # some text to insert literally into the proof script
205  | metainforesponse     # prover outputs some meta-information to interface
206
207 ready = element ready { empty }
208
209 displayarea = "message"      # the message area (response buffer)
210     | "display"      # the main display area (goals buffer)
211
212 cleardisplay =
213    element cleardisplay {
214       attribute area {
215          displayarea | "all" } }
216
217
218 displaytext = (text | pgml)*     # grammar for displayed text
219
220 normalresponse =
221   element normalresponse {
222     attribute area { displayarea },
223     attribute category { text }?,    # optional extra category (e.g. tracing/debug)
224     attribute urgent { "y" }?,     # indication that message must be displayed
225     displaytext
226 }
227
228 fatality = "nonfatal" | "fatal" | "panic"  # degree of errors
229
230 errorresponse =
231    element errorresponse {
232      attribute fatality { fatality },
233      attribute location { text }?,
234      attribute locationline { xsd:positiveInteger }?,
235      attribute locationcolumn { xsd:positiveInteger }?,
236      displaytext
237   }
238
239 scriptinsert =
```

```
240     element scriptinsert {
241        text
242     }
243
244  # metainformation is an extensible place to put system-specific information
245
246  value = element value { name_attr?, text }   # generic value holder
247
248  metainforesponse =
249     element metainforesponse {
250        attribute infotype { text }, # categorization of data
251        value* }    # data values
252
253
254  # ========== PROOF CONTROL COMMANDS ==========
255
256  proofcmd =
257      goal   # open a goal in ambient context
258    | proofstep   # a specific proof command (perhaps configured via opcmd)
259    | undostep   # undo the last proof step issued in currently open goal
260    | closegoal   # complete & close current open proof (succeeds iff goal proven)
261    | abortgoal   # give up on current open proof, close proof state, discard history
262    | giveupgoal    # close current open proof, record as proof obl'n  (sorry)
263    | postponegoal  # close current open proof, retaining attempt in script (oops)
264    | forget   # forget a theorem (or named target), outdating dependent theorems
265    | restoregoal   # re-open previously postponed proof, outdating dependent theorems
266
267  thmname_attr = attribute thmname { text }   # theorem names
268  aname_attr   = attribute aname { text }     # anchors in proof text
269
270  goal         = element goal { thmname_attr, text }       # text is theorem to be proved
271  proofstep    = element proofstep { aname_attr?, text }   # text is proof command
272  undostep     = element undostep { empty }
273
274  closegoal    = element closegoal { empty }
275  abortgoal    = element abortgoal { empty }
276  giveupgoal   = element giveupgoal { empty }
277  postponegoal = element postponegoal { empty }
278  forget       = element forget { thyname_attr?, aname_attr? }
279  restoregoal  = element restoregoal { thmname_attr }
280
281
282  # ========== THEORY/FILE HANDLING COMMANDS ==========
283
284  filecmd =
285      loadtheory     # load a file possibly containing a theory definition
286    | opentheory      # begin construction of a new theory.
287    | closetheory     # complete construction of the currently open theory
288    | retracttheory   # retract a theory.  Applicable to open & closed theories.
289    | openfile        # lock a file for constructing a proof text
290    | closefile       # unlock a file, suggesting it has been processed
291    | abortfile     # unlock a file, suggesting it hasn't been processed
292
293  fileinfomsg =
294     informfileloaded       # prover informs interface a particular file is loaded
295   | informfileretracted    # prover informs interface a particular file is outdated
296
297  url_attr     = attribute url     { text } # typically: filename
298  thyname_attr = attribute thyname { text } # a corresponding theory name
299
300  loadtheory    = element loadtheory    { url_attr?, thyname_attr? }
```

```
opentheory     = element opentheory     { thyname_attr, text }
closetheory    = element closetheory    { thyname_attr }
retracttheory = element retracttheory { thyname_attr }
openfile       = element openfile       { url_attr }
closefile      = element closefile      { url_attr }
abortfile      = element abortfile      { url_attr }

informfileloaded =
   element informfileloaded     { thyname_attr, url_attr }
informfileretracted =
   element informfileretracted { thyname_attr, url_attr }
```

A.2 pgml.rnc

```
#
# RELAX NG Schema for PGML, the Proof General Markup Language
#
# Authors:  David Aspinall, LFCS, University of Edinburgh
#           Christoph Lueth, University of Bremen
# Version: $Id: pgml.rnc,v 1.3 2003/07/16 14:56:35 da Exp $
#
# Status:  Complete but experimental version.
#
# For additional commentary, see the Proof General Kit white paper,
# available from http://www.proofgeneral.org/kit
#
# Advertised version: 1.0
#

pgml_version_attr = attribute version { xsd:NMTOKEN }

pgml =
  element pgml {
    pgml_version_attr?,
    (statedisplay | termdisplay | information | warning | error)*
  }

termitem       = action | nonactionitem
nonactionitem = term | pgmltype | atom | sym

pgml_name_attr = attribute name { text }

kind_attr = attribute kind { text }
systemid_attr = attribute systemid { text }

statedisplay =
  element statedisplay {
    pgml_name_attr?, kind_attr?, systemid_attr?,
     (text | termitem | statepart)*
  }

br = element br { empty }
pgmltext = (text | termitem | br)*

information =
  element information { pgml_name_attr?, kind_attr?, pgmltext }

warning      = element warning      { pgml_name_attr?, kind_attr?, pgmltext }
error        = element error        { pgml_name_attr?, kind_attr?, pgmltext }
statepart    = element statepart    { pgml_name_attr?, kind_attr?, pgmltext }
termdisplay = element termdisplay { pgml_name_attr?, kind_attr?, pgmltext }
```

```
48
49 pos_attr = attribute pos { text }
50
51 term = element term { pos_attr?, kind_attr?, pgmltext }
52
53 # maybe combine this with term and add extra attr to term?
54 pgmltype = element type { kind_attr?, pgmltext }
55
56 action = element action { kind_attr?, (text | nonactionitem)* }
57
58 fullname_attr = attribute fullname { text }
59 atom = element atom { kind_attr?, fullname_attr?, text }
60
61
62 ## Symbols
63
64 symname = attribute name { text }
65 sym     = element sym { symname }
66
67 # configuring PGML
68
69 pgmlconfigure = symconfig  # inform symbol support (I/O) for given sym
70 asciialt = attribute alt { text } # understanding of ASCII alt for given sym
71
72 symconfig = element symconfig { symname, asciialt? }
```