Commentary on PGIP

[Version 1.4, 2003/09/25 07:11:25, LaTeX: September 25, 2003]

David Aspinall Christoph Lüth

September 25, 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.

`<proofstate>`

- The `proofstate` element reflects the current proof state. It should be displayed in the display area.
- The prover may send more than one `proofstate` element before sending a `ready` command; in this case, later elements supercede earlier ones. On the other hand, the prover is not required to send a `proofstate`.

`<normalresponse>`

- All other 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.
- `normalresponse` *vs* `proofstate`: the rule of thumb is that `normalresponse` output illustrates the progress of the proof, whereas `proofstate` displays the proof state after the current command has been processed.

`<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>`
	`<proofstate>`	`<normalresponse>`
	`<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 1.37 2003/09/25 09:11:49 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
#



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
 | proofctxt    # issue a context 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  name_attr = attribute name { token }            # identifiers must be XML tokens
70
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   { prefcat_attr? }
79  setpref    = element setpref    { name_attr, prefcat_attr?, text }
80  getpref    = element getpref    { name_attr, prefcat_attr? }
81
82
83
84  #  ========== INTERFACE CONFIGURATION ==========
85
86  kitconfig =
87      usespgip        # I support PGIP, version ..
88    | usespgml        # I support PGML, version ..
89    | pgmlconfig      # configure PGML symbols
90    | proverinfo     # Report assistant information
91    | hasprefs        # I have preference settings ...
92    | prefval         # the current value of a preference is
93    | guiconfig       # configure the following object types and operations
94    | setids     # inform the interface about some known objects
95    | addids          # add some known identifiers
96    | delids          # retract some known identifers
97    | idvalue     # display the value of some identifier
98    | menuadd         # add a menu entry
99    | menudel         # remove a menu entry
100
101 # version reporting
102 version_attr  = attribute version { text }
103 usespgml = element usespgml  { version_attr }
104 usespgip = element usespgip  { version_attr }
105
106 # PGML configuration
107 pgmlconfig = element pgmlconfig { pgmlconfigure+ }
108
109 # Types for config settings: corresponding data values should conform
110 # to representation for corresponding XML Schema 1.0 Datatypes.
111 # (This representation is verbose but helps for error checking later)
112
113 pgiptype    = pgipbool | pgipint | pgipstring | pgipchoice
114 pgipbool    = element pgipbool { empty }
115
116 pgipint     = element pgipint  { min_attr?, max_attr?, empty }
117 min_attr    = attribute min { xsd:integer }
```

```
118 max_attr    = attribute max { xsd:integer }
119 pgipstring = element pgipstring { empty }
120 pgipchoice = element pgipchoice { pgipchoiceitem+ }
121 pgipchoiceitem = element pgipchoiceitem { text }
122
123 icon   = element icon { xsd:base64Binary }  # image data for an icon
124
125 # preferences
126 default_attr = attribute default { text }
127 descr_attr    = attribute descr { text }
128
129 # icons for preference recommended size: 32x32
130 # top level preferences: may be used in dialog for preference setting
131 # object preferences: may be used as an "emblem" to decorate
132 # object icon (boolean preferences with default false, only)
133 haspref = element haspref  {
134    name_attr, descr_attr?,
135    default_attr?, icon?,
136    pgiptype
137 }
138
139 hasprefs = element hasprefs { prefcat_attr?, haspref* }
140
141 prefval = element prefval  { name_attr, text }
142
143 # menu items (incomplete)
144 path_attr = attribute path { text }
145
146 menuadd = element menuadd  { path_attr?, name_attr?, text }
147 menudel = element menudel  { path_attr?, name_attr?, text }
148
149
150 # GUI configuration information: objects, types, and operations
151 # NB: the following object types have a fixed interpretation
152 # in PGIP:  "comment", "thm", "theory", "file"
153
154 guiconfig =
155   element guiconfig { objtype*, opn* }
156
157 objtype = element objtype { name_attr, descr_attr?, icon?, hasprefs?, contains* }
158
159 objtype_attr = attribute objtype { token }          # the name of an objtype
160 contains = element contains { objtype_attr, empty }  #
161
162 opn    = element opn { name_attr, inputform?, opsrc, optrg, opcmd }
163
164 opsrc = element opsrc { list { token* } }  # source types: a space separated list
165 optrg = element optrg { token }?    # single target type, empty for proofstate
166 opcmd = element opcmd { text }      # prover command, with printf-style "%1"-args
167
168 # interactive operations - require some input
169 inputform = element inputform { field+ }
170
171 # a field has a PGIP config type (int, string, bool, choice(c1...cn))
172 # and a name; under that name, it will be substituted into the command
173 # Ex. field name=number  opcmd="rtac %1 %number"
174
175 field = element field {
176    name_attr, pgiptype,
177    element prompt { text }
178 }
```

```
179
180 # identifier tables: these list known items of particular objtype.
181 # Might be used for completion or menu selection, and inspection.
182 # May have a nested structure (but objtypes do not).
183
184 setids  = element setids { idtable } # (with an empty idtable, clear table)
185 addids  = element addids { idtable }
186 delids  = element delids { idtable }
187
188 # give value of some identifier (response to showid)
189 idvalue = element idvalue
190    { name_attr, objtype_attr, displaytext }
191
192 idtable    = element idtable { objtype_attr, (identifier | idtable)* }
193 identifier = element identifier { token }
194
195 # prover information:
196 # name, description, version, homepage,
197 # welcome message, docs available
198 proverinfo = element proverinfo
199    { name_attr, descr_attr?, version_attr?, url_attr?,
200      welcomemsg?, icon?, helpdoc* }
201 welcomemsg = element welcomemsg { text }
202 url_attr   = attribute url { text } # FIXME: schema for URL?
203
204 # helpdoc: advertise availability of some documentation, given a canonical
205 # name, textual description, and URL or viewdoc argument.
206 #
207 helpdoc    = element helpdoc { name_attr, descr_attr, url_attr?, text } # text is arg to "
208
209
210 # ========== PROVER CONTROL ==========
211
212 provercontrol =
213    proverinit      # reset prover to its initial state
214  | proverexit      # exit prover
215  | startquiet      # stop prover sending proof state displays, non-urgent messages
216  | stopquiet       # turn on normal proof state & message displays
217  | pgmlsymbolson   # activate use of symbols in PGML output (input always understood)
218  | pgmlsymbolsoff  # deactivate use of symbols in PGML output
219
220 proverinit     = element proverinit { empty }
221 proverexit     = element proverexit { empty }
222 startquiet     = element startquiet { empty }
223 stopquiet      = element stopquiet  { empty }
224 pgmlsymbolson  = element pgmlsymbolson { empty }
225 pgmlsymbolsoff = element pgmlsymbolsoff { empty }
226
227
228 # ========== GENERAL PROVER OUTPUT/RESPONSES ==========
229
230 proveroutput =
231    ready               # prover is ready for input
232  | cleardisplay        # prover requests a display area to be cleared
233  | proofstate         # prover outputs the proof state
234  | normalresponse      # prover outputs some display
235  | errorresponse       # prover indicates an error condition, with error message
236  | scriptinsert        # some text to insert literally into the proof script
237  | metainforesponse    # prover outputs some other meta-information to interface
238  | parseresult  # result of a <parsescript> request (see later)
239  | unparseresult # result of a <unparsecmds> request (see later)
```

```
240
241 ready = element ready { empty }
242
243 displayarea = "message"                    # the message area (response buffer)
244             | "display"                    # the main display area (goals buffer)
245
246 cleardisplay =
247    element cleardisplay {
248       attribute area {
249          displayarea | "all" } }
250
251
252 displaytext = (text | pgml)*               # grammar for displayed text
253
254 proofstate =
255   element proofstate { displaytext }
256
257 normalresponse =
258   element normalresponse {
259     attribute area { displayarea },
260     attribute category { text }?,          # optional extra category (e.g. tracing/debug)
261     attribute urgent { "y" }?,             # indication that message must be displayed
262     displaytext
263 }
264
265 fatality = "nonfatal" | "fatal" | "panic"  # degree of errors
266
267 errorresponse =
268    element errorresponse {
269      attribute fatality { fatality },
270      attribute location { text }?,
271      attribute locationline { xsd:positiveInteger }?,
272      attribute locationcolumn { xsd:positiveInteger }?,
273      displaytext
274   }
275
276 scriptinsert = element scriptinsert { text }
277
278
279 # metainformation is an extensible place to put system-specific information
280
281 value = element value { name_attr?, text }   # generic value holder
282
283 metainforesponse =
284    element metainforesponse {
285       attribute infotype { text },      # categorization of data
286       value* }                          # data values
287
288
289 # ========== PROOF CONTROL COMMANDS ==========
290
291 proofcmd =
292     properproofcmd | improperproofcmd
293
294 properproofcmd =
295     opengoal      # open a goal in ambient context
296   | proofstep     # a specific proof command (perhaps configured via opcmd)
297   | closegoal     # complete & close current open proof (succeeds iff goal proven)
298   | giveupgoal    # close current open proof, record as proof obl'n (sorry)
299   | postponegoal  # close current open proof, retaining attempt in script (oops)
300   | comment   # an ordinary comment: text ignored by prover
```

```
301   | litcomment    # a "literate" comment: text processed by prover, but no sidefx
302
303 improperproofcmd =
304     undostep      # undo the last proof step issued in currently open goal
305   | abortgoal     # give up on current open proof, close proof state, discard history
306   | forget        # forget a theorem (or named target), outdating dependent theorems
307   | restoregoal   # re-open previously postponed proof, outdating dependent theorems
308
309 thmname_attr = attribute thmname { text }   # theorem names
310 aname_attr   = attribute aname { text }     # anchors in proof text
311
312 opengoal     = element opengoal { thmname_attr, text }   # text is theorem to be proved
313 proofstep    = element proofstep { aname_attr?, text }   # text is proof command
314 undostep     = element undostep { empty }
315
316 closegoal    = element closegoal { empty }
317 abortgoal    = element abortgoal { empty }
318 giveupgoal   = element giveupgoal { empty }
319 postponegoal = element postponegoal { empty }
320 forget       = element forget { thyname_attr?, aname_attr? }
321 restoregoal  = element restoregoal { thmname_attr }
322 comment      = element comment { text }
323 litcomment   = element litcomment { text }
324
325
326 # ========= PROOF CONTEXT/ETC COMMANDS ===========
327
328 proofctxt =
329     askids    # please tell me about identifiers (given objtype in a theory)
330   | showid    # print value of an object
331   | bindid    # extend current context with identifer assignment
332   | parsescript    # parse a raw proof script into proofcmds
333   | unparsecmds    # unprase proofcmds into raw proof script
334   | showproofstate # (re)display proof state (empty if outside a proof)
335   | showctxt    # show proof context
336   | searchtheorems # search for theorems (prover-specific arguments)
337   | setlinewidth   # set line width for printing
338   | viewdoc    # request some on-line help (prover-specific arg)
339
340 thyname_attr = attribute thyname { text }       # theory name
341
342 askids = element askids  { thyname_attr?, objtype_attr }
343
344 showid = element showid  { thyname_attr?, objtype_attr, name_attr }
345 bindid = element setid   { name_attr, objtype_attr, setpref*, objval }
346 objval = element obvalue { text }   # text constructed with opcmds
347
348
349 # NB: parse/unparsing needs only be supported for "proper" proof commands,
350 # which may appear in proof texts.
351
352 properscriptcmd = properproofcmd | properfilecmd | bindid
353
354 parsescript = element parsescript { text }
355 parseresult = element parseresult { properscriptcmd* }
356
357 unparsecmds   = element unparsecmds { properscriptcmd* }
358 unparseresult = element unparseresult { text }
359
360 showproofstate = element showproofstate { empty }
361 showctxt       = element showctxt { empty }
```

```
362 searchtheorems = element searchtheorems { text }
363 setlinewidth    = element setlinewidth { xsd:positiveInteger }
364 viewdoc         = element viewdoc { text }
365
366
367 # ========== THEORY/FILE HANDLING COMMANDS ==========
368
369 filecmd =
370    properfilecmd | improperfilecmd
371
372 properfilecmd =
373     opentheory      # begin construction of a new theory.
374   | closetheory     # complete construction of the currently open theory
375
376 improperfilecmd =
377     aborttheory     # abort currently open theory
378   | retracttheory   # retract a named theory
379   | openfile        # lock a file for constructing a proof text
380   | closefile       # unlock a file, suggesting it has been processed
381   | abortfile       # unlock a file, suggesting it hasn't been processed
382   | loadfile        # load a file possibly containing theory definition(s)
383   | changecwd     # change prover's working directory (or load path) for files
384
385 fileinfomsg =
386    informfileloaded       # prover informs interface a particular file is loaded
387  | informfileretracted    # prover informs interface a particular file is outdated
388
389 # Below, url_attr will often be a file URL.
390 # We assume for now that the prover and interface are running on same
391 # filesystem
392
393 dir_attr      = attribute dir { text }   # Unix directory name
394
395 opentheory    = element opentheory    { thyname_attr, text }
396 closetheory   = element closetheory   { empty }
397 aborttheory   = element aborttheory   { empty }
398 retracttheory = element retracttheory { thyname_attr }
399
400 # FIXME: maybe add a command to ask prover for the file corresponding
401 # to some theory (prover searches it's search path / cwd).
402 loadfile      = element loadfile      { url_attr }
403 openfile      = element openfile      { url_attr }
404 closefile     = element closefile     { empty }
405 abortfile     = element abortfile     { empty }
406 changecwd     = element changecwd     { dir_attr }
407
408 informfileloaded =
409    element informfileloaded    { thyname_attr, url_attr }
410 informfileretracted =
411    element informfileretracted { thyname_attr, url_attr }
```

A.2 pgml.rnc

```
1 #
2 # RELAX NG Schema for PGML, the Proof General Markup Language
3 #
4 # Authors:  David Aspinall, LFCS, University of Edinburgh
5 #           Christoph Lueth, University of Bremen
6 # Version: $Id: pgml.rnc,v 1.5 2003/09/23 23:12:47 da Exp $
7 #
8 # 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)*
  }

pgmltext = (text | termitem)*

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 }

pos_attr = attribute pos { text }

term = element term { pos_attr?, kind_attr?, pgmltext }

# maybe combine this with term and add extra attr to term?
pgmltype = element type { kind_attr?, pgmltext }

action = element action { kind_attr?, (text | nonactionitem)* }

fullname_attr = attribute fullname { text }
atom = element atom { kind_attr?, fullname_attr?, text }


## Symbols

symname_attr = attribute name { text }
sym      = element sym { symname_attr }

# configuring PGML

pgmlconfigure = symconfig  # inform symbol support (I/O) for given sym
asciialt = attribute alt { text } # understanding of ASCII alt for given sym
```

```
70
71 symconfig = element symconfig { symname_attr, asciialt? }
```