Commentary on PGIP

[Version 1.30, 2007/07/11 10:28:22, LaTeX: July 11, 2007]

David Aspinall Christoph Lüth

July 11, 2007

PGIP is the *Proof General Interaction Protocol*, a message passing protocol for communicating proof components, primarily interactive theorem provers and their interfaces.

This document gives commentary on the definition of PGIP, version 2.0.X. The commentary is intended as a set of notes to help implementors of PGIP-enabled components; it does not form a complete description or motivation for the protocol. The RELAX-NG schemas for PGIP messages and the PGML markup language are given in the appendix.

> **Warning!** this is an evolving draft version, some details in the text may be out-of-step with schemas shown in the appendix and with the latest implementations. And you may not be looking at the latest version of this document! If in doubt, please contact the authors before relying on details here.

1 Basics

1.1 Overview

1. PGIP is an interaction protocol between different components of an architecture to support interactive proof development, and a markup language for proof scripts in such an environment.

   Fig. 1 shows the corresponding system architecture.

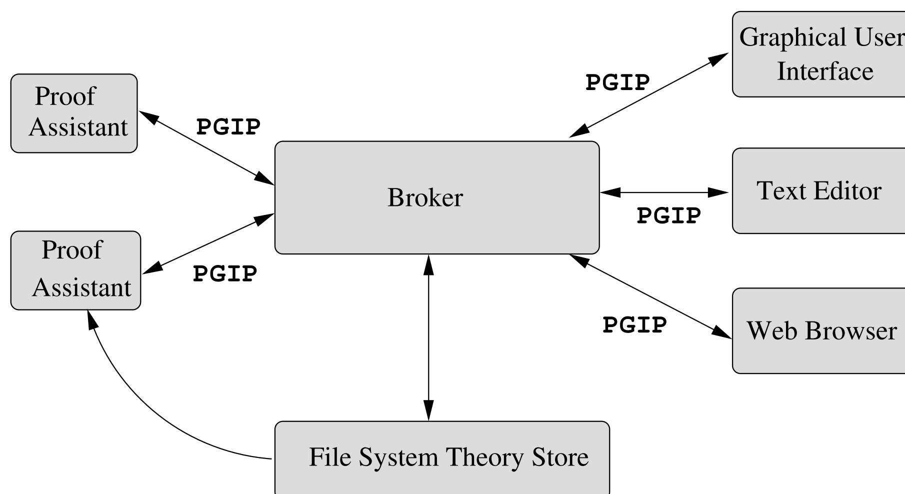

Figure 1: PG/Kit system architecture

2. 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**.

3. A proof script has a textual representation as primary and resides in a file. Proof scripts are written in the prover's native language.

4. Part of PGIP serves to make *explicit* the structure which is normally *implicit* in the proof script, by requiring the proof assistant to add certain XML mark up. The basic invariant is that by forgetting all the PGIP structure (applying the trivial forgetful stylesheet, if you will) we get back the original proof script.

1.2 PGIP communication

1. PGIP components connect to a central **Broker** process which helps manage interactive proof, and connection between front-ends and provers. The connection is a channel (typically a Unix pipe or socket).

2. PGIP components are designated as **prover**, **display**, or **auxiliary** (see Fig. 1).

3. Messages are classified: those which are sent *to* the PG Broker are given class `pg`. Messages which are sent to a proof assistant are given class `pa`, messages which are sent to a display are class `pd`.

4. PGIP communication proceeds by exchanging PGIP packets as XML documents belonging to the PGIP markup schema. A PGIP packet is contained in the `<pgip>` element.

5. 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.

6. 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*.

7. The broker will 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.

8. Each PGIP packet has the following attributes:

   `tag` A message tag, for example, the name of the origin component.

   `id` The session identifier of the originating component.

   `destid` The session identifier of the destination component (for a response packet).

   `class` The class of the message, as above.

   `seq` A unique (for this session of this component) sequence number. Sequence numbers never decrease over time.

   `refid` **and** `refseq` Both optional; if given, the session identifier, sequence number of a message this one refers to (see below).

   `origin` Only the broker will set this attribute, either to `broker` to indicate messages generated within the broker, or to the component identifier of the originating component if it is relaying message, e.g. from a prover to a display.

In the following section, we describe the single messages in more detail. We classify the messages in three categories: common messages (all classes), those exchanged between prover and broker (classes `pg` and `pa`), and those exchanged between prover and displays (classes `pg` and `pa`).

2 Common Interaction

This section describes messages common to all three main classes of components.

2.1 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.

2.2 Prover to interface configuration

`<usespgip>`

- The prover reports which version of PGIP it supports (only between prover and broker).

`<usespgml>`

- The prover reports which version of PGML it supports (only between prover and broker).

`<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 emit as output, along with optional ASCII defaults. PGML symbol conventions define a large fixed set of named glyphs.

`<hasprefs>`

- The prover reports some user-level preference settings, each one in an `<haspref>` element, giving its type and possible a default value. A collection of *global preferences* (dynamic switches inside the prover) are reported by the prover in response to an `<askprefs>` message, or are volunteered by the prover during configuration. Further *object-specific* preferences may be reported inside `<objtype>` messages which appear as part of the `<displayconfig>` configuration.

`<prefval>`

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

`<displayconfig>`

- The prover specifies a *display configuration* which contains items to configure the interface to the running prover, which are described by the remaining elements listed below. In particular, the `lexicalstructure` elements describe the lexical structure of proof scripts for the prover, while the `objtype` and `opn` operations describe the types of values that the prover wants the user to manipulate.

`<welcomemsg>`

- A simple textual message to inform the user about the underlying prover process.

<icon>

- an `icon` which is a Base64-encoded bitmap picture, representing the prover.

<helpdoc>

- An available item of documentation to make available to the user (e.g. in a menu). Includes either a URL or an argument for the `viewdoc` proof context command. There may be several or none.

<lexicalstructure>

- An element which configures (some part of) the proof script syntax. It may include any of the below elements:
  - `keyword`. Each keyword element specifies the word itself (`word` attribute), together with an optional category and long and short help texts.
    - The category is used, e.g., for subdividing menus or for configuring more than one colour for keyword syntax highlighting. It is not used for parsing. Recommended categories are:
      `major` (the default if none is mentioned) for main keywords;
      `minor` auxiliary keywords.
      with language-specific additions.
    - The `shorthelp` element gives a short tooltip-length help text for the keyword.
    - The `longhelp` gives a larger paragraph-length help for the keyword.
  - `stringdelimiter` elements

  Notice that the features of the lexical syntax which are supported by different displays may be affected by their individual capabilities (e.g., Emacs only allows one or two characters in comment-start and comment-end sequences).

<objtype>

- The prover specifies a basic object type it will let the interface manipulate.
- A type has a mandatory `name` attribute;
- it can have a `description` attribute, describing the type, and
- an `icon` which is a Base64-encoded bitmap picture (at the moment, the default and only format here are GIF bitmaps).
- optional preferences specified by a <hasprefs> element; these may indicate, for example, whether a theorem is flagged as being an introduction or elimination rule.
- Further, the types determine a hierarchical organisation of the prover objects; objects of each type may contain objects of other other types, as specified by the `contains` attribute.
- The following types are always predefined:
  - `"toplevel"` for toplevel objects;
  - `"theory"` for theory files;
  - `"theorem"` for theorems;
  - `"comment"` for comments.

  `theory` contains `theorem` and `theory`. Provers are free to redefine these types, e.g. to define new icons.

<opn>

- `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.
- As a general convention, if several operations are possible to produce a desired target object, then the interface should offer them in the choice that they were configured.

2.3 Identifiers

The PGIP model accommodates *identifiers*, i.e. the ability of the prover to name objects. The protocol makes the following assumptions about identifiers:

- Each identifier has a type (see above).
- Provers can arrange identifiers in a hierarchical fashion, so the name of one identifier can be valid in the *context* of another identifier, as specified by the `contains` attribute on the types.
- Identifiers can be used in a proof by referring to their name, and they can be displayed.
- The prover has a certain amount of pre-defined identifiers available upon starting, and can define more identifiers during run-time.

<askids>

- Ask prover to display identifiers, possibly in a given context and only of a given type; prover should respond with `setids` or `addids` message.

<showid>

- Ask prover to display value of an identifer. Prover should respond with `showid` message.

<idtable>

- A list of identifiers of a given type, possibly in a given context.

<setids>

- Argument is a list of `idtables`.
- For each `idtable`, deletes previously known identifiers of that type and context, and add given identifiers.

<addids>

- Argument is a list of `idtables`.
- For each `idtable`, add given identifiers.

<delids>

- Argument is a list of `idtables`.
- For each `idtable`, deletes previously known identifiers of that type and context.

<idvalue>

- Displays the value of a given identifier as a sequence of text of PGML packets.

3 Prover and Broker

This section describes the communication between broker and prover. The proof script commands are also handed on to the interface for display, but the interface is only supposed to use the markup to display structure.

3.1 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. This is intended to supress display of intermediate steps while processing a possibly large number of proof commands.

`<stopquiet>`

- Ask the prover to turn on its output again.

3.2 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.

`<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 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:

toprovermsg	*toprovermsg*	*toprovermsg*
<ready/>	<normalresponse>	<normalresponse>
⋮	<normalresponse>	<errorresponse>
	<normalresponse>	<normalresponse>
	<ready/>	<ready/>
	⋮	⋮

The *toprovermsg* 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.

3.3 Proof control commands

The PGIP proof model is to assume that the prover maintains a state which consists of a single possibly-open proof within a single possibly-open theory, see Section 3.4 for more explanation. We distinguish between **proper proof commands** which can appear in proof scripts, and **improper proof commands** which cannot.

In PGIP 1.0, proper commands were required to be interpreted by the theorom prover so they could be used to construct parts of proofs (e.g. the interface can provide an action for completing a proof). In PGIP 2.0 we rely on other mechanisms to construct proof script text, and proper commands are interpreted merely as markup which provides structure on a proof script document for the interface. The markup should be ignored by the prover when it processes a proper proof command. To simplify things for the interface, a proper proof commands may be wrapped in a <dostep> element which is ignored by the prover as well (and which is *not* allowed in markup for <parseresult>).

Proper proof commands.

<opengoal>

- open a goal in ambient context

<proofstep>

- a specific proof command (perhaps configured via <opcmd>)

`<closegoal>`

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

`<giveupgoal>`

- close current open proof, record as proof obligation (Isar's `sorry`).

`<postponegoal>`

- close current open proof, retaining attempt in script (Isar's `oops`)

`<comment>`

- a proof script comment; text is probably ignored by prover

`<whitespace>`

- a whitespace comment; text must be ignored by prover and may be discarded

`<litcomment>`

- a literate comment which is never passed to the prover

`<spuriouscmd>`

- a command which is ignored for undo purposes, and which could be pruned from a proof script (e.g. `print x`).

Improper proof commands.

`<dostep>`

- Used to issue a proper proof command to the prover, without passing in more specific markup. The prover should interpret `<dostep>` and any proper proof command simply by executing their contents.

`<undostep>`

- undo the last proof step issued in currently open goal.

`<redostep>`

- redo the last proof step issued in currently open goal. Optionally supported.

`<abortgoal>`

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

`<forget>`

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

  If theory name is omitted, we default to currently open theory. If ordinary name is omitted, default to whole theory. (Makes no sense for both names to be omitted). Optional type attribute is allowed for provers that have overlapping namespaces, that may allow forgetting different kinds of things within theories.

<restoregoal>

- re-open previously postponed proof, outdating dependent theorems. This is a context switch operation. Optionally supported.

Further notes:

1. The improper proof commands 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. The broker will enhance whatever native history management is supported by the prover, and try to exploit it.
2. In a later version, 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.
3. 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.
4. 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.

3.4 File-level and theory-level 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 may choose to only construct one theory per file).

PGIP assumes that the proof engine has four main states, which are nested:

top level inspection and navigation of theories only; no focus of active development.

open file a file is open for processing; we may declare theories now and other items which appear within files.

open theory a theory has been opened for construction; we may declare new definitions, types, and other commands which appear at the theory level. The prover may record an undo history of theory-level element declarations, but discards this history when the theory is complete.

open proof a proof has been opened for construction; we may issue proof steps which aim to complete the proof. The prover records an undo history for each step, but discards this history on proof completion.

This model only allows a single open item at each level (file, theory, or proof). Nonetheless, it should be possible for the interface to provide extra structure and maintain an illusion of more than one open item, 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 may extend PGIP to allow multiple open proofs to implemented within the prover to provide extra efficiency, to avoid too much proof replaying.

Similar to proof commands, theory and file commands divide into proper commands, which may appear in proof scripts, and improper commands, which may not. Proper file commands are treated as markup on parsed proof scripts, just as proper proof commands are.

Some improper commands are optional. Commands which are not supported in general should not be included in <acceptedpgipgelems> for the component. Where a command is not possible in a particular case, an error message should be given.

PGIP supposes that the interface has only partial knowledge about theory construction, and so the interface relies on the prover to send hints. Specifically, to help manage file-level control, there are are two information response messages which may be sent *from* the prover which help connect to the prover's internal file management system (if it has one).

Proper File Commands.

`<opentheory>`

- begin construction of a new theory.
  Attributes may provide the ancestor theory names.

`<closetheory>`

- complete construction of the currently open theory. When executed this discards any theory-level undo history.

`<theoryitem>`

- an item in a theory other than a binding or proof, for example type definition and data type declarations.

Improper File Commands.

`<doitem>`

- Process the given proper theory command (without passing in markup)

`<undoitem>`

- Undo the last step during theory construction (or the given named target, if possible).

`<redoitem>`

- Re-process the last undone theory-level item, if possible.

`<aborttheory>`

- Give up on the currently open theory (undo to before theoryopen, discarding history).

`<retracttheory>`

- Remove the named theory, and any dependent theories.

`<openfile>`

- Signal that the named file is being opened for constructing a proof text interactively.
  The prover may pay attention to this message if it tracks processed files: the file should not be one which is already processed.

`<closefile>`

- Signal that the currently open file has been completely processed.
  The prover may pay attention to this message if it tracks processed files. The file should now be considered fully processed, as if the prover had read it directly.

`<loadfile>`

- Read a file directly for processing in batch mode.
  The prover may load any required files automatically, but should indicate when this happens by sending appropriate `<informfileloaded>` messages to the broker.

`<retractfile>`

- Undo a processed file. Optionally supported.
  Only relevant if the prover tracks processed files. The prover may retract any files which are dependent on the given one, but if it does so, it should inform the broker of this by sending appropriate `<informfileretracted>` messages. Files which are not retracted automatically by the prover will be retracted by the broker.

`<changecwd>`

- change prover's current working directory.
  In case the prover may read subsidiary files, the interface can set the directory that the proof script under construction is destined for. Normally this would be the path of the URI given in an `openfile` command which might precede `openfile`.

`<systemcmd>`

- An escape which allows to send an arbitrary command to the prover. A certain security hole.

File loader information responses.

`<informfileloaded>`

- prover informs interface a particular file is loaded.
  When the interface asks for a file to be loaded, or when some proper proof script command triggers file loading, a number of `<informtheoryloaded>` responses can be sent from the prover. These indicate dependency of the loaded file on whatever caused the loading, and also suggest that the loaded files should be considered processed by the prover.

`<informfileretracted>`

- prover informs interface a particular file is outdated.
  Conversely, when the interface asks for a file to be retracted, the prover may automatically retract dependent files. If it does so, it should inform the broker with `<informfileretracted>` messages. The broker will retract remaining dependencies.

3.5 Parsing

Proof scripts are written in the prover's native language. By *parsing*, we mean enriching them with PGIP markup to make their structure explicit. The component which does the parsing need not always be the prover, it could be an auxiliary component which filters out the `<parsescript>` requests from the input stream and answers them.

`<parsescript>`

- Requests the prover to parse one or more command lines. (This can be a whole proof script.) Optional attributes are the starting position; these are for error reporting, such that `<errorresponse>` answer can carry the correct location wrt to the original source.

`<parseresult>`

- Returns the result of a parsing request.

`<singleparseresult>`

- A single parse result is an `<unparseable>` element, or an `<errorresponse>` containing error messages, or a *proper script command*, which is either a proper proof command, a proper file command, or a delimiter (`<openblock>`/`<closeblock>` below).

  One `<parsescript>` must result in exactly one `<parseresult>`. There is an invariant that by discarding all markup on the result of the parse, we get back the original proof script. The prover may leave out text when returning the parsed elements (these are then considered unparseable), but the *order* must be maintained.

`<unparseable>`

- Returns text the parser can not make head or tails of.

`<openblock>`

- Opens a layout block. The proof script can have layout structure annoted with basic `<openblock>` and `<closeblock>` elements. A basic `<openblock>` element has no attributes except optionally a name. This block structure suggests intendation or a tree-like structure for layout purposes. A basic `<openblock>` is taken to be associated with the immediately preceding document element; block delimiters themselves are empty.

  Blocks can alternatively be treated special positions in the document which may be replaced by text in document-based development. These blocks are indicated by `<openblock>` with a non-empty `metavarid` or `objtype` element. The meta-variables is used to communicate a document position back to the prover; an object type allows the interface to replace the contents of the document element. These kind of blocks are *not* used for layout.

`<closeblock>`

- Closes the matching block. Once parsing has been completed, there will be a matching `<closeblock>` for each `<openblock>`, but in a single `<parseresult>` this may not be the case for layout blocks, as we may only parse part of a document. For document element blocks

4 Broker and Display

This section describes the communication between broker and display (classes `pg` and `pd`). In the following, **display messages** go from the broker to the display, and **display commands** go from the display to the broker.

4.1 The PGIP edit-prove cycle

The PGIP system architecture supports an edit-prove cycle in which the user can interactively edit and construct proof scripts, residing in files.

Internally, the broker component is responsible for the actual handling of the source documents. The user editing commands are translated into PGIP display commands (class `pd`), which are sent to the broker for the actual editing.

For the display, the source is broken up into *objects* [**FIXME**: bad name, heavily overloaded. What about items, units or atoms?] which contain a PGIP element each. Each object has an identifier, which the display has to keep track of, as subsequent edit and change commands pertain to the object identifier.

Each object has a status, which ranges over five possible values (see Fig. 2). The different transitions between the objects are a refinement of the script management as implemented by Proof General. The object status can change by the results of the interaction with the system. After editing, an object becomes either *parsed* or *unparseable*; after being processed by the prover it becomes *processed* and can become *outdated* later on. Note that while being processed the object cannot be edited; if it has been processed it needs to be outdated first. The `unparseable` status is for objects which cannot be parsed (because they are syntactically malformed) but which we still need to be part of the proof script.

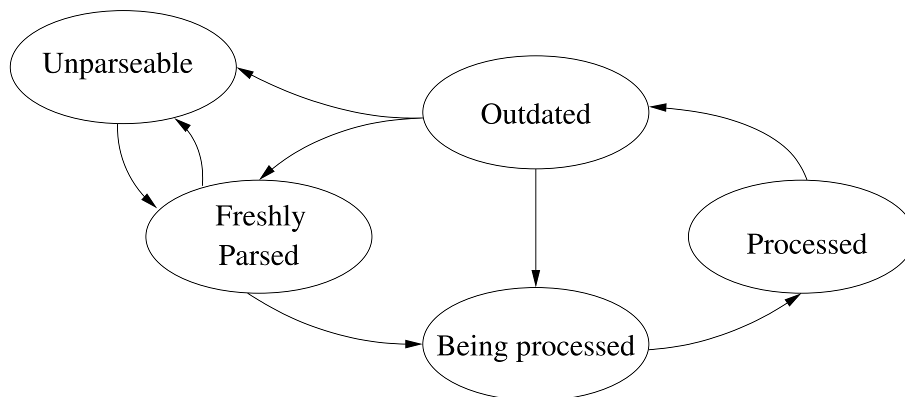

Figure 2: Object Status Transitions

Here is a typical edit-prove cycle:

1. Loading a file.
   (a) The user wants to edit an old file and selects the corresponding menu button or cryptical key sequence. This results in the display sending a `<loadfile>` message to the broker.
   (b) The broker actually reads the file, locks it while it is being edited, and sends it to the prover to be parsed (with a `<parsescript>` request).
   (c) The prover responds with one or more `<parseresult>` packets, containing the marked-up proof script. The broker actually checks that the marked-up proof script is the same as the original, to make sure no source code gets lost.
   (d) The broker now sends a series of `<newobj>` messages to the display. This will actually display the source file.
   (e) To save a file, the display sends a `<savefile>` message, which causes the broker to assemble the source file and write it to a file.
2. Editing.

(a) The user can edit an object. In a graphical display, the user may actually see the object structure as it is graphically represented, but in a pure textual interface (such as the revered Emacs editor) only text can be edited. In any case, the display must keep track of the positions of the object in the display, such that any edit operation on part of the user can be translated into a `<editobj>` request being sent back to the broker. `<editobj>` contains a range of object identifiers, and the new text to be inserted for these identifiers.

(b) The broker now sends `<delobj>` messages for all objects in the span being edited, signalling to the display that they are not needed any more.

(c) The broker decides wether this edit is *allowed* or not: it is always allowed for comments, outdated and unparsed objects, never allowed for objects being processed (and if possible, the display should not allow it to start with), and for processed objects it corresponds to a request to outdate the object, followed by the actual edit. If editing is not allowed, the broker sends an error message to the display, and `<newobj>` messages which reinstate the original content. [**FIXME**: this is debatable, as the user loses his changes? Should not have made them in the first place?]

(d) If editing is allowed, the text is sent to the prover to parse. We make the assumption here that the prover does not need a context to parse single commands like this, as this would considerably complicate the protocol; this seems a reasonable assumption as no typecheck etc. needs to be performed. The prover either returns a `<parseresult>` element if the parse is successful, or an error message otherwise which is sent on to the display.

(e) If the broker received an error message, it sends a `<newobj>` message inserting an `unparseable` object. The user can then try again.

(f) If the parsing suceeds with a series of `<parseresult>`, the broker sends on corresponding `<newobj>` message (one for each object) to the display to insert the new objects into the buffer.

3. Proving: processing a file.

(a) Processing a file means that the user requests the status of a particular object to be changed (e.g. to `processed` to make it up-to-date, or to `outdated` if it needs to be edited). The display sends a `<setobjstatus>` request to the broker.

(b) The broker checks the status of the object, and translates the request into a number of PGIP commands sent to the prover. (To bring a single object up-to-date, an arbitrary number PGIP commands may be needed, e.g. a long proof script be worked through).

(c) The resulting object changes (not only for this object as other objects may change status as well, e.g. objects this depends on being brought up-to-date) are sent back to the display as `<objstatus>` messages. The last one of these message will change the status of the required object, unless an error has occured.

4.2 Control Messages

Attribute: `prvid`

- A unique *prover identifier*. Once a prover has started, it is referred to by a such an identifier. The Broker can handle more than one prover at a time, but each source file can only contain theories and hence objects from a single prover.

`<componentspec>`

- specifies a PGIP component. It gives a unique identifier (e.g. `"isa2004-HOL"`, a textual name under which is visible and meaningful to the user (e.g. `"Isabelle 2004/HOL"`), the type (allowed are `provertype`, `displaytype`, and `auxiliary`) and specifies how to connect to or start it.

- Only the broker knows the specifiation of the components. The present assumption is that the broker reads a configuration file containing a list of `<componentspec>` on startup; this should later be extended to allow dynamic configuration, e.g. by allowing for new components to announced to the broker.

- On starting a display, the broker will send it a <brokerstatus> message (containg in particular a list of known provers) as part of the initial configuration sequence.

Display Commands:

<launchprover>

- request a new prover to be launched; attribute componentid is the component identifier for a known prover.

<exitprover>

- requests prover to be stopped; attribute is the prvid.

<restartprover>

- requests prover to be restarted; attribute is the prvid.

<shutdownbroker>

- requests broker to perform shutdown; optional attribute is force. The broker will not shut down if there are still unsaved changes in source files, unless 'force' attribute is given (with value true).

<brokerstatusquery>

- request broker status information.

Display Messages:

<brokerstatus>

- status information from broker, containing known provers, running provers, and some textual information;

<proveravailable>

- a new prover is available under name and componentid given as attributes (must be launched before use.)

<proverstarted>

- a new prover has just started under proverid given as attribute.

  The proverid is the same as the componentid used to start this prover. Thus, we cannot run multiple copies of the same prover concurrently.

<proverstatus>

- prover status has changed. Displays are urged to display ready/busy information, as provers may diverge and users will want to notice.
- Attributes:

  prvid unique prover identifier

  proverstatus can be busy, ready or exitus

4.3 File Handling

Attribute: `srcid`

- A unique *source file identifier*. The broker can handle more than one source file, and so must the display. The source file identifier is assigned by the broker once the file has been loaded or created, and referred to when saving the file, or inserting new objects. This corresponds, very roughly, to an Emacs buffer.

Display Commands:

`<loadparsefile/>`

- user requests an existing source file to be loaded into a specific prover.
- Attributes:

  `url` URL of file to be loaded. (The Broker will not understand anything else than the `file` scheme for a while.)

  `prvid` identity of prover which should read the file.

`<dispopenfile/>`

- user requests a new source file to be created.
- Attributes:

  `url` URL of file to be created.

  `overwrite` overwrite if file exists (otherwise gives an error).

  `prvid` identity of prover which the source file is to be created for.

`<newfilewith/>`

- user requests a new source file to be created, with given text content. In contrast to `dispopenfile`, the file may be written out later, and the user may not commit to any file name yet. In that case, a subsequent `savefile` must give a URL.
- Attributes:

  `url` URL of file to be created.

  `prvid` identity of prover which the source file is to be created for.

`<savefile/>`

- user requests changes to be commited
- Attributes:

  `srcid` unique source file identifier

  `url` (optional) URL of where to save; default is to save where it has been created or loaded it from.

`<discardfile/>`

- User wants to discard changes to this file.
- Attributes:

  `srcid` unique source file identifier

Display Messages:

`<newfile/>`

- The broker has created a new file or loaded an old file, and announces the new source file identifier.
- Attributes:

  `srcid` unique source file identifier

`<filestatus/>`

- Informs the display that the status of a file has changed. The interface can use this to display 'needs-save' information.
- Attributes:

  `srcid` unique source file identier

  `filestatus` new status: `saved`, `changed`, or `discarded`.

  `datetime` Time and date in XML Schema format (briefly, `CCYY-MM-DDThh:mm:ss` followed by `Z` for Zulu time or a timezone offset like `-01:00`). It indicates the time the status change took effect (in particular, when a file was written).

- A `discarded` message is the last in which this `srcid` will be used.

4.4 Object Management

Attribute: `objstatus`

- Attribute which can have five possible values:
  - `unparseable`
  - `parsed`
  - `being_processed`
  - `processed`
  - `outdated`
- The interface should if possible stop the user from editing objects which are `being_processed`, as the broker will reject these edits.

Attribute: `objid`

- A *unique object identifier*, which is unique for this broker during this session, across all objects in all source files and for all provers.

Display Messges:

`<newobj>`

- The broker has created a new object and requests it being displayed. The contents of this element is a proper script command, and the attributes are as follows:

  `srcid` identifies the source file in which this object is being inserted.

  `objid` unique object identifier, to be used in subsequent editing operations;

  `objposition` optional `objid` of object before which this object is inserted. If empty, insert at the end of the current text.

  `objstatus` object status (see above; freshly created objects are either of type `parsed` or `unparseable`);

  `objparent` optional `objid` of a parent object, in case the display wants to display objects hierarchically. These are generated from the block structure given by the parser, and are only for layout; they carry no semantic meaning.

`<delobj/>`

- Signals that the broker has deleted all references to this object identifier, and requests display to delete it if it has not already done so.
- Attributes:

  `srcid` unique source file identifier

  `objid` unique object identifier

`<objectstatus/>`

- State of this object has changed in the broker
- Attributes:

  `srcid` unique source file identifier

  `objid` unique object identifier

  `objstatus` new object status.

Display commands:

`<editobj>`

- The user has edited one or more objects. The content of this message contains the edited text.
- Attributes:

  `editfrom` (optional) unique object identifier of object where the editing started; if not given edit starts from first object.

  `editto` (optional) unique object identifier of object where the editing ended; if not given edit ends with last object.

  `srcid` (optional) identifies the source file in which the objects are edited; can be omitted if one of `editfrom` and `editto` is given.

`<createobj>`

- The user has entered text which is not an edit of existing objects; in other words, the user requests new objects to be created. The content of the message contains the entered text.
- Attributes:

  `srcid` (optional) identifies the source file in which this object is created.

  `objposition` (optional) unique object identifier before which the new objects are to be inserted. If empty we add resulting objects to the end of the buffer.

  At least one of `srcid` or `objposition` has to be given. Since the object determines the source file, if `objposition` is given, it takes precedence over `srcid`.

`<setobjstatus/>`

- The user has requested the status of this object to be changed.
- Attributes:

  `objid` unique object identifier

  `objstatus` new object status. Only `processed` and `outdated` status can be requested.

5 PGML markup language (v 2.0+)

PGML is a simple markup language for annotating prover input and output text. Annotations can be used to provide pretty printing hints (blocks and line breaks), or additional structure on already pretty-printed text. The markup is intended to be further transformed (e.g. by system-specific stylesheets) to produce HTML or other display forms.

PGML is not intended to be a complete language for fully flexible page layout. Instead, if needed, more complex layout instructions can be embedded in another XML language using the `<embed>` element: for example, MathML for math fragments or HTML for ready-rendered web pages.

Text is split into *terms* which can have a nested subterm structure. A `<subterm>` is a position for attaching annotations of various kinds. Annotations can include:

Attribute: `subterm`

name a user-level global identifier

objtype (obj)type information

place placement directives (`sub`, `super`, `above`, `below`)

decoration decoration hints for fonts (`bold`, `italic`) or colouring (e.g. standard *error*, *warning* and *information* attributes for markup with red/yellow/blue squiggles)

action actions for activation (e.g. triggering a context `menu`, `button` to show a button, or `toggle` for folding).

pos a position pointer to the some content form (e.g. true abstract syntax before rendering) which is used to communicate back to the prover (also used for menu request or button pushing).[1]

xref a cross reference to a defining declaration (typically a file URL with a numerical dotted fragment for the line and column number, e.g. `file:///home/da/Test.thy#15.55`.

Leafs of terms can be strings, `<atom>` elements or (atoms containing) `<sym>` elements. The `<sym>` element is a named reference to a symbol (or composite symbol) which has been configured in a symbol configuration specific to the prover/display.

Pretty printing `<box>` annotations are included, modelled after those of BoxML in GtkMathView (see `http://helm.cs.unibo.it/mml-widget/qna.html`).

See comments in the schema for more information.

`<pgml>`

- This element encloses a PGML document which is displayed in a window.
  Attributes:

  `area` for output markup, used to indicate a window where the document should appear.

  `version` version of PGML this document uses.

  `systemid` System identifier token. This can be used by the display to control the interpretation of prover-specific style elements.

  The `version` and `systemid` attributes will not normally be used by the prover, since the surrounding PGIP packet should determine these. The broker may add them for the purposes of sending on to other components.

[1] TODO: not yet harmonised with PGIP messages

Attribute: `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).
- The interface is free to implement these displays in different ways, or even supress them entirely, insofar as that makes sense.

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 L th , University of Bremen
#
# Advertised version: 2.1 (pre-1)
#
# CVS Version: $Id: pgip.rnc,v 3.45 2007/07/11 10:28:12 da Exp $
#
# Status:   Prototype.
#
# For additional commentary, see accompanying commentary document available at
# http://proofgeneral.inf.ed.ac.uk/Kit/docs/commentary.pdf
#
#
# Contents
# ========
#
#  0. Prelude
#  1. Top-level
#  2. Component Control messages
#  3. Display Commands
#  4. Prover Configuration
#  5. Interface Configuration
#  6. Prover Control
#  7. Proof script markup and proof control
#
#
# ==============================================================================
#
# Note on datatypes.  (see e.g. http://books.xmlschemata.org/relaxng):
#
#  token  : any string possibly with spaces, but spaces are normalised/collapsed
#           (i.e. tokenised).  Same as XML Schema xs:token
#  string : any string, whitespaces preserved.  Same as XML Schema xs:string
#           (NB: attributes are normalised by XML 1.0 parsers so
#            spaces/newlines must be quoted)
#  text   : text nodes/mixed content (whitespace may be lost in mixed content)
#
# So: attributes should usually be tokens or more restrictive; (sometimes: strings for printing)
#     element contents may be string (preserving whitespace), token (tokenising),
#       or text (which may contain further nodes).

# ==============================================================================
# 0. Prelude
# ==============================================================================

# Namespace: this breaks DtdToHaskell, currently removed
#default namespace pgip = "http://proofgeneral.inf.ed.ac.uk/pgip"

include "pgml.rnc"                          # include PGML grammar

name_attr = attribute name { token }         # names are user-level textual identifiers (space-collapse
thyname_attr = attribute thyname { token }   # names for theories (special case of name)
thmname_attr = attribute thmname { token }   # names for theorems (special case of name)

datetime_attr =
   attribute datetime { xsd:dateTime }       # CCYY-MM-DDHH:MM:SS plus timezone info
url_attr  = attribute url { xsd:anyURI }     # URLs  (often as "file:///localfilename.extn")
dir_attr  = attribute dir { string }         # Unix-style directory name (no final slash)

systemdata_attr  =
            attribute systemdata { token }?   # system-specific data (useful for "stateless" RPC)
```

```
65
66 objname = token      # an identifier name (convention: any characters except semi-colon)
67 objnames = token     # sequence of names in an attribute: semi-colon separated
68
69 #objnames = string                          # A sequence of objnames
70 #termobjname  = string { pattern = "[^;]+;" } # unfortunately these declarations don't
71 #objnames = objname | (termobjname, objname)  #  work with the RNC->DTD tool trang
72 #objnames = objname+
73
74
75 # ==============================================================================
76 # 1. Top-level Messages/documents
77 # ==============================================================================
78
79 start = pgip                               # Single message
80       | pgips                              # A log of messages between components
81       | displayconfig                      # displayconfig as a standalone element
82       | pgipconfig                         # pgipconfig as a standalone element
83       | pgipdoc                            # A proof script document
84
85 pgip = element pgip {                      #  A PGIP packet contains:
86    pgip_attrs,                             #   - attributes with header information;
87    (toprovermsg | todisplaymsg |           #   - a message with one of four channel types
88     fromprovermsg | fromdisplaymsg
89    | internalmsg )
90   }
91
92 pgips = element pgips { pgip+ }
93
94 pgip_attrs =
95  attribute tag { token }?,                 # message tag, e.g. name of origin component (diagnostic)
96  attribute id { token },                   # (unique) session id of this component
97  attribute destid { token }?,              # session id of destination component
98  attribute class { pgip_class },           # general categorization of message
99  attribute refid { token }?,               # component id this message responds to (usually destid)
100 attribute refseq { xsd:positiveInteger }?, # message sequence this message responds to
101 attribute seq { xsd:positiveInteger }      # sequence number of this message
102
103 pgip_class = "pg"      # message sent TO proof general broker (e.g. FROM proof assistant).
104            | "pa"      # message sent TO the proof assistant/other component
105            | "pd"      # message sent TO display/front-end components
106
107 toprovermsg =          # Messages sent to the prover (class "pa"):
108    proverconfig        #   query Prover configuration, triggering interface configuration
109  | provercontrol       #   control some aspect of Prover
110  | improperproofcmd    #   issue a proof command
111  | improperfilecmd     #   issue a file command
112  | properproofcmd      #   [ NB: not strictly needed: input PGIP processing not expected ]
113  | properfilecmd       #   [ NB: not strictly needed: input PGIP processing not expected ]
114  | proofctxt           #   issue a context command
115
116 fromprovermsg =        # Messages from the prover to PG (class "pg"):
117    kitconfig           #   messages to configure the interface
118  | proveroutput        #   output messages from the prover, usually display in interface
119  | fileinfomsg         #   information messages concerning file-system access / prover state
120
121 todisplaymsg =         # Messages sent to display components (class "pd"):
122    brokermsg           #   status reports from broker
123  | dispmsg             #   display commands
124                        #   - Further, all fromprovermsg can be relayed to display
125
126 fromdisplaymsg =       # Messages sent from display components (class "pg"):
127    dispcmd             #   display messages
128  | brokercontrol       #   messages controlling broker & prover processes
129                        #   - Further, all toprovermsg to be relayed to prover
130
131 # ==============================================================================
132 # 2. Component Control
133 # ==============================================================================
```

```
134
135 #
136 # Idea:  - broker knows how to manage some components (inc provers) as child processes,
137 #          communicate via pipes.  Configured by a fixed PGIP config file read on startup.
138 #        - other components may connect to running broker
139 #
140 # TODO: - describe startup protocol for component connecting to to running broker dynamically.
141
142 # This is the element contained in the configuration file read by the
143 # broker on startup.
144 pgipconfig = element pgipconfig { componentspec* }
145
146 componentspec =
147    element componentspec {
148       componentid_attr,          # Unique identifier for component class
149       componentname_attr,        # Textual name of component class
150       componenttype,             # Type of component: prover, display, auxiliary
151       startupattrs,              # Describing startup behaviour
152       systemattrs,               # System attributes for component
153       componentconnect           # How to connect to component
154    }
155
156 componentid_attr   = attribute componentid   { token }
157 componentname_attr = attribute componentname { token }
158
159 componenttype = element componenttype {
160      provercomponent
161    | displaycomponent
162  # | filehandlercomponent
163    | parsercomponent
164    | othercomponent }
165
166 provercomponent  = element provercomponent { empty }
167 displaycomponent = element displaycomponent { attribute active { xsd:boolean}? }
168 parsercomponent  = element parsercomponent  { componentid_attr }
169 othercomponent   = element othercomponent { empty }
170
171 componentconnect =
172    componentsubprocess | componentsocket | connectbyproxy
173
174 componentsubprocess =
175    element syscommand { string }
176 componentsocket =
177    (element host { token }, element port { xsd:positiveInteger })
178 connectbyproxy =
179    (element proxy { attribute host { token }    # Host to connect to
180                   , attribute connect {
181                          "rsh" | "ssh" # Launch proxy via RSH or SSH, needs
182                                        # authentication
183                        | "server"  # connect to running proxy on given port
184                        }?
185                   , attribute user { token } ? # user to connect as with RSH/SSH
186                   , attribute path { token } ? # path of pgipkit on remote
187                   , attribute port { xsd:positiveInteger } ? # port to connect to running proxy
188                   , componentconnect
189                   })
190
191 # Attributes describing when to start the component.
192 startupattrs =
193   attribute startup {          # what to do on broker startup:
194             "boot"   |         # always start this component (default with displays)
195             "manual" |         # start manually (default with provers)
196             "ignore"           # never start this component
197             }?
198
199 # System attributes describing behaviour of component.
200 systemattrs = (
201     attribute timeout { xsd:integer }?  # timeout for communications
202   , attribute sync { xsd:boolean }?     # whether to wait for ready
```

```
203     , attribute nestedgoals { xsd:boolean}? # Does prover allow nested goals?
204     )
205
206 # Control commands from display to broker
207 brokercontrol =
208       launchprover                  # Launch a new prover
209     | exitprover                    # Request to terminate a running prover
210     | restartprover                 # Request to restart/reset a running prover
211     | proversquery                  # Query about known provers, running provers
212     | shutdownbroker                # Ask broker to exit (should be wary of this!)
213     | brokerstatusquery             # Ask broker for status report
214     | pgipconfig                    # Send config to broker
215
216 provername_attr    = attribute provername { provername }
217 provername         = token
218
219 proverid_attr      = attribute proverid  { proverid }
220 proverid           = token
221
222 launchprover       = element launchprover { componentid_attr }
223 exitprover         = element exitprover { proverid_attr }
224 restartprover      = element restartprover { proverid_attr }
225 proversquery       = element proversavailable { empty }
226 brokerstatusquery = element brokerstatusquery { empty }
227 shutdownbroker     = element shutdownbroker { attribute force { xsd:boolean }? }
228
229 # Control messages from broker to interface
230 brokermsg  =
231       brokerstatus                  # response to brokerstatusquery:
232     | proveravailmsg                # announce new prover is available
233     | newprovermsg                  # new prover has started
234     | proverstatemsg                # prover state has changed (busy/ready/exit)
235
236 brokerstatus  = element brokerstatus
237                       { knownprovers, runningprovers, brokerinformation }
238
239 knownprover   = element knownprover   { componentid_attr, provername }
240 runningprover = element runningprover { componentid_attr, proverid_attr, provername }
241
242 knownprovers   = element knownprovers { knownprover* }
243 runningprovers = element runningprovers { runningprover* }
244 brokerinformation = element brokerinformation { string }
245
246 proveravailmsg  = element proveravailable { provername_attr,
247                                             componentid_attr }
248 newprovermsg    = element proverstarted { proverid_attr
249                                         , componentid_attr
250                                         , provername_attr
251                                         }
252 proverstatemsg = element proverstate {
253                       proverid_attr, provername_attr,
254                       attribute proverstate {proverstate} }
255
256 proverstate    = ("ready" | "busy" | "exitus")
257
258 # FIXME: This only allows provers to be available which are configured.
259 #        In the long run, we want to change configurations while running.
260
261
262 # ==============================================================
263 # 3. Display Commands
264 # ==============================================================
265
266 # Messages exchanged between broker and display
267
268
269 dispcmd = dispfilecmd | dispobjcmd  # display commands go from display to broker
270 dispmsg = dispfilemsg | dispobjmsg  # display messages go from broker to display
271
```

```
272
273 dispfilecmd =
274     loadparsefile            # parse and load file
275   | newfilewith              # create new source file with given text
276   | dispopenfile             # open (or create) new file
277   | savefile                 # save opened file
278   | discardfile              # discard changes to opened file
279
280 dispfilemsg =
281     newfile                  # announce creation of new file (in response to load/open)
282   | filestatus               # announce change in status of file in broker
283
284 # unique identifier of loaded files
285 srcid_attr = attribute srcid { token }
286
287 loadparsefile = element loadparsefile { url_attr , proverid_attr }
288 newfilewith   = element newfilewith   { url_attr , proverid_attr , string }
289 dispopenfile  = element dispopenfile { url_attr ,
290                                        proverid_attr ,
291                                        attribute overwrite { xsd:boolean }?}
292 savefile      = element savefile { srcid_attr ,
293                                    url_attr? }
294 discardfile   = element discardfile { srcid_attr }
295
296 newfile       = element newfile  { proverid_attr , srcid_attr , url_attr }
297 filestatus    = element filestatus  { proverid_attr , srcid_attr , newstatus_attr , url_attr ?,
298                                       datetime_attr}
299
300 newstatus_attr = attribute newstatus { "saved" | "changed" | "discarded" }
301
302 dispobjcmd =
303     setobjstate              # request of change of state
304   | editobj                  # request edit operation of objects
305   | createobj                # request creation of new objects
306 # Suggested May 06: probably add re-load flags instead
307 #  | reloadobjs               # request relisting of all objects
308   | inputcmd                 # process the command (generated by an input event)
309   | interruptprover          # send interrupt or kill signal to prover
310
311 dispobjmsg = element dispobjmsg {
312       newobj+                # new objects have been created
313     | delobj+                # objects have been deleted
314     | replaceobjs            # objects are being replaced
315     | objectstate+           # objects have changed state
316     }
317
318 newobj = element newobj {
319             proverid_attr ,
320             srcid_attr ,
321             objid_attr ,
322             attribute objposition { objid } ?,
323             objtype_attr ?,
324             attribute objparent { objid }?,
325             attribute objstate { objstate },
326             # FIXME: would like to include metainfo here
327             # as (properscriptcmd , metainfo*) | unparseable
328             (properscriptcmd | unparseable) }
329
330 replaceobjs = element replaceobjs {
331                         srcid_attr ,
332                         attribute replacedfrom { objid }? ,
333                         attribute replacedto { objid }?,
334                         delobj*,    # actually , either of delobj or
335                         newobj* }   # newobj can be empty but not both.
336
337 delobj = element delobj   { proverid_attr , srcid_attr , objid_attr }
338
339 ## update to display
340 objectstate = element objstate
```

```
341                         { proverid_attr , srcid_attr , objid_attr ,
342                           attribute newstate {objstate} }
343
344 ## update from display
345 setobjstate = element setobjstate
346                   {  objid_attr , attribute newstate {objstate} }
347
348 editobj = element editobj { srcid_attr ?,
349                             attribute editfrom { objid }?,
350                             attribute editto   { objid }?,
351                             string }
352 createobj = element createobj { srcid_attr ?,
353                                 attribute objposition { objid }?,
354                                 string }
355
356 # Suggested May 06: probably add re-load flags instead
357 # reloadobjs = element reloadobjs { srcid_attr }
358
359 inputcmd       = element inputcmd { improper_attr , string }
360 improper_attr  = attribute improper { xsd:boolean }
361
362 interruptprover = element interruptprover
363                           { interruptlevel_attr , proverid_attr }
364
365 interruptlevel_attr  = attribute interruptlevel { "interrupt" | "stop" | "kill" }
366
367 objid_attr = attribute objid { objid }
368 objid      = token
369
370 objstate =
371   ( "unparseable" | "parsed" | "being_processed" | "processed" | "outdated" )
372
373
374 # ==============================================================================
375 #  4. Prover Configuration
376 # ==============================================================================
377
378 proverconfig =
379    askpgip                      # what version of PGIP do you support?
380  | askpgml                      # what version of PGML do you support?
381  | askconfig                    # tell me about objects and operations
382  | askprefs                     # what preference settings do you offer?
383  | setpref                      # please set this preference value
384  | getpref                      # please tell me this preference value
385
386
387
388 prefcat_attr = attribute prefcategory { token }   # e.g. "expert", "internal", etc.
389                                                   # could be used for tabs in dialog
390
391 askpgip   = element askpgip   { empty }
392 askpgml   = element askpgml   { empty }
393 askconfig = element askconfig { empty }
394 askprefs  = element askprefs  { prefcat_attr? }
395 setpref   = element setpref   { name_attr , prefcat_attr?, pgipvalue }
396 getpref   = element getpref   { name_attr , prefcat_attr? }
397
398
399
400 # ==============================================================================
401 #  5. Interface Configuration
402 # ==============================================================================
403
404 kitconfig =
405     usespgip                    # I support PGIP, version ..
406   | usespgml                    # I support PGML, version ..
407   | pgmlconfig                  # configure PGML symbols
408   | proverinfo                  # Report assistant information
409   | hasprefs                    # I have preference settings ...
```

```
410     | prefval                   # the current value of a preference is
411     | displayconfig             # configure the following object types and operations
412     | setids                    # inform the interface about some known objects
413     | addids                    # add some known identifiers
414     | delids                    # retract some known identifers
415     | idvalue                   # display the value of some identifier
416     | menuadd                   # add a menu entry
417     | menudel                   # remove a menu entry
418
419  # version reporting
420  version_attr  = attribute version { token }
421  usespgml = element usespgml  { version_attr }
422  usespgip = element usespgip  { version_attr
423                               , activecompspec
424                               }
425
426  # These data from the component spec which an active component can override, or which
427  # components initiating contact with the broker (e.g. incoming socket connections).
428  # There are some restrictions: if we start a tool, the componentid and the type must be the
429  # same as initially specified.
430  activecompspec =  ( componentid_attr?   # unique identifier of component class
431                    , componentname_attr? # Textual name of this component (overrides initial spec)
432                    , componenttype?      # Type of component
433                    , systemattrs         # system attributes
434                    , acceptedpgipelems?  # list of accepted elements
435                    )
436
437
438  acceptedpgipelems = element acceptedpgipelems { singlepgipelem* }
439
440  singlepgipelem    = element pgipelem {
441     attribute async { xsd:boolean }?,  # true if this command supported asynchronously (deflt false)
442                                        # (otherwise part of usual ready/sync stream)
443     attribute attributes { text }?,    # comma-separated list of supported optional attribute names
444                                        # useful for: times attribute
445     text }                             # the unadorned name of the PGIP element (*not* an element)
446
447  # PGML configuration
448  pgmlconfig = element pgmlconfig { pgmlconfiguration }
449
450  # Types for config settings: corresponding data values should conform to canonical
451  # representation for corresponding XML Schema 1.0 Datatypes.
452  #
453  # In detail:
454  #  pgipnull   = empty
455  #  pgipbool   = xsd:boolean = true | false
456  #  pgipint    = xsd:integer = (-)?(0-9)+     (canonical: no leading zeroes)
457  #  pgipstring = string  =  <any non-empty character sequence>
458  #  pgipchoice = cf xs:choice  = type1 | type2 | ... | typen
459
460  pgiptype   = pgipnull | pgipbool | pgipint | pgipstring | pgipchoice | pgipconst
461
462  pgipnull   = element pgipnull  { descr_attr?, empty }
463  pgipbool   = element pgipbool  { descr_attr?, empty }
464  pgipint    = element pgipint   { min_attr?, max_attr?, descr_attr?, empty }
465  min_attr   = attribute min { xsd:integer }
466  max_attr   = attribute max { xsd:integer }
467  pgipstring = element pgipstring { descr_attr?, empty }
468  pgipconst  = element pgipconst { name_attr, descr_attr?, empty }
469  pgipchoice = element pgipchoice { pgiptype+ }
470
471  # Notes on pgipchoice:
472  # 1. Choices must not be nested (i.e. must not contain other choices)
473  # 2. The description attributes for pgipbool, pgipint, pgipstring and pgipconst
474  #    are for use with pgipchoice: they can be used as a user-visible label
475  #    when representing the choice to the user (e.g. in a pull-down menu).
476  # 3. A pgipchoice should have an unambiguous representation as a string. That means
477  #    all constants in the choice must have different names, and a choice must not
478  #    contain more than one each of pgipint, pgipstring and pgipbool.
```

```
479
480 pgipvalue  = string
481
482 icon  = element icon { xsd:base64Binary }  # image data for an icon
483
484 # The default value of a preference as a string (using the unambiguous
485 # conversion to string mentioned above). A string value will always be quoted
486 # to distinguish it from constants or integers.
487 default_attr = attribute default { token }
488
489 # Description of a choice.  If multi-line, first line is short tip.
490 descr_attr   = attribute descr { string }
491
492 # icons for preference recommended size: 32x32
493 # top level preferences: may be used in dialog for preference setting
494 # object preferences: may be used as an "emblem" to decorate
495 # object icon (boolean preferences with default false, only)
496 haspref = element haspref  {
497    name_attr, descr_attr?,
498    default_attr?, icon?,
499    pgiptype
500 }
501
502
503
504 hasprefs = element hasprefs { prefcat_attr?, haspref* }
505
506 prefval = element prefval { name_attr, pgipvalue }
507
508 # menu items (incomplete, FIXME)
509 path_attr = attribute path { token }
510
511 menuadd  = element menuadd  { path_attr?, name_attr?, opn_attr? }
512 menudel  = element menudel  { path_attr?, name_attr? }
513 opn_attr = attribute operation { token }
514
515
516 # Display configuration information:
517 # basic prover information, lexical structure of files,
518 # an icon for the prover, help documents, and the
519 # objects, types, and operations for building proof commands.
520
521 # NB: the following object types have a fixed interpretation
522 # in PGIP:
523 #       "identifier":   an identifier in the identifier syntax
524 #          "comment":   an arbitrary sequence of characters
525 #          "theorem":   a theorem name or text
526 #          "theory" :   a theory name or text
527 #          "file" :     a file name
528
529 displayconfig =
530   element displayconfig {
531      welcomemsg?, icon?, helpdoc*, lexicalstructure*,
532      objtype*, opn* }
533
534 objtype = element objtype { name_attr, descr_attr?, icon?, contains*, hasprefs?  }
535
536 objtype_attr = attribute objtype { token }           # the name of an objtype
537 contains = element contains { objtype_attr, empty }  #  a container for other objtypes
538
539 opn = element opn {
540    name_attr,
541    descr_attr?,
542    inputform?,    # FIXME: can maybe remove this?
543    opsrc*,        # FIXME: incompat change wanted: have list of source elts, not spaces
544    optrg,
545    opcmd,
546    improper_attr? }
547
```

```
548 opsrc =
549    element opsrc {
550       name_attr?,          # %name as an alternative to %number
551       selnumber_attr?,     # explicit number for %number, the nth item selected
552       prompt_attr?,        # prompt in form or tooltip in template
553       listwithsep_attr?,   # list of args of this type with given separator
554       list { token* } }    # source types: a space separated list
555                            # FIXME incompat change wanted: just have one source here
556                            # FIXME: need to add optional pgiptype
557
558 listwithsep_attr =  attribute listwithsep { token }
559 selnumber_attr =    attribute selnumber { xsd:positiveInteger }
560 prompt_attr =       attribute prompt { string }
561
562 optrg =
563    element optrg { token }?           # single target type, empty for proof command
564 opcmd =
565    element opcmd { string }           # prover command, with printf-style "%1"-args
566                                            #  (whitespace preserved here: literal text)
567
568 # interactive operations - require some input
569 inputform = element inputform { field+ }
570
571 # a field has a PGIP config type (int, string, bool, choice(c1...cn)) and a name; under that
572 # name, it will be substituted into the command Ex. field name=number opcmd="rtac %1 %number"
573
574 field = element field {
575    name_attr, pgiptype,
576    element prompt { string }
577 }
578
579 # identifier tables: these list known items of particular objtype.
580 # Might be used for completion or menu selection, and inspection.
581 # May have a nested structure (but objtypes do not).
582
583 setids  = element setids { idtable* }   # (with an empty idtable, clear table)
584 addids  = element addids { idtable* }
585 delids  = element delids { idtable* }
586
587 # give value of some identifier (response to showid; same values returned)
588 idvalue = element idvalue
589    {  thyname_attr?, name_attr, objtype_attr, pgml }
590
591 idtable    = element idtable { context_attr?, objtype_attr, identifier* }
592 identifier = element identifier { token }
593
594 context_attr = attribute context { token } # parent identifier (context)
595
596 # prover information:
597 # name, instance (e.g. in case of major parameter of invocation);
598 # description, version, homepage,  welcome message, docs available
599 proverinfo = element proverinfo
600    { name_attr, version_attr?, instance_attr?,
601      descr_attr?, url_attr?, filenameextns_attr?,
602 ## TEMP: these elements are duplicated in displayconfig, as they're
603 ## moving there.
604      welcomemsg?, icon?, helpdoc*, lexicalstructure* }
605
606 instance_attr = attribute instance { token }
607
608 welcomemsg = element welcomemsg { string }
609
610 # helpdoc: advertise availability of some documentation, given a canonical
611 # name, textual description, and URL or viewdoc argument.
612 #
613 helpdoc = element helpdoc { name_attr, descr_attr, url_attr?, token } # token string is arg to "viewdoc
614
615 filenameextns_attr = attribute filenameextns { xsd:NMTOKENS } # space-separated extensions sans "."
616
```

```
617 # lexical structure of proof texts
618 lexicalstructure =
619    element lexicalstructure {
620       keyword*,
621       stringdelimiter?,
622       escapecharacter?,
623       commentdelimiter*,
624       identifiersyntax?
625    }
626
627 keyword = element keyword {
628    attribute word { token },
629    shorthelp?,
630    longhelp? }
631
632 shorthelp = element shorthelp { string }   # one-line (tooltip style) help
633 longhelp  = element longhelp { string }    # multi-line help
634
635 stringdelimiter = element stringdelimiter { token }  # should be a single char
636
637 # The escape character is used to escape strings and other special characters - in most languages it is
638 escapecharacter = element escapecharacter { token }  # should be a single char
639
640 commentdelimiter = element commentdelimiter {
641    attribute start { token },
642    attribute end { token }?,
643    empty
644   }
645
646
647 # syntax for ids: id = initial allowed*  or id = allowed+ if initial empty
648 identifiersyntax = element identifiersyntax {
649    attribute initialchars { token }?,
650    attribute allowedchars { token }
651 }
652
653 # ==============================================================================
654 # 6. Prover Control
655 # ==============================================================================
656
657 provercontrol =
658    proverinit                   # reset prover to its initial state
659  | proverexit                   # exit prover
660  | setproverflag                # set/clear a standard control flag (supersedes above)
661
662 proverinit     = element proverinit { empty }
663 proverexit     = element proverexit { empty }
664
665 setproverflag  = element setproverflag { flagname_attr,
666                                          attribute value { xsd:boolean } }
667 flagname_attr  =
668    attribute flagname { "quiet"
669                       | "pgmlsymbols"
670                       | "metainfo:thmdeps"
671                       }
672
673 # General prover output/responses
674
675 # Prover output has an otional proverid_attribute. This is set by the broker when relaying
676 # prover output to displays. When producing output, provers can and should not set this
677 # attribute.
678
679 proveroutput =
680    ready                        # prover is ready for input
681  | normalresponse               # prover outputs, no error condition
682  | errorresponse                # prover indicates an error/warning/debug condition, with message
683  | scriptinsert                 # some text to insert literally into the proof script
684  | metainforesponse             # prover outputs some other meta-information to interface
685  | parseresult                  # results of a <parsescript> request (see later)
```

```
686
687
688 ready = element ready { empty }
689
690 normalresponse =
691   element normalresponse {
692     proverid_attr?,                                  # if no proverid, assume message is from broker
693     pgml
694 }
695
696
697
698 ## Error messages: these are different from ordinary messages in that
699 ## they indicate an error condition in the prover, with a notion
700 ## of fatality and (optionally) a location. The interface may collect these
701 ## messages in a log, display in a modal dialog, or in the specified
702 ## display area if one is given
703 ##
704 ## Error responses are also taken to indicate failure of a command to be processed, but only in
705 ## the special case of a response with fatality "fatal". If any errorresponse with
706 ## fatality=fatal is sent before <ready/>, the PGIP command which triggered the message is
707 ## considered to have failed. If the command is a scripting command, it will not be added to
708 ## the successfully processed part of the document. A "nonfatal" error also indicates some
709 ## serious problem with the sent command, but it is not considered to have failed. This is the
710 ## ordinary response for
711
712 errorresponse =
713    element errorresponse {
714      proverid_attr?,                                # ... as above ...
715      attribute fatality { fatality },
716      location_attrs,
717      pgml
718    }
719
720
721 fatality =          # degree of error conditions:
722     "info"          # - info message
723   | "warning"       # - warning message
724   | "nonfatal"      # - error message, recovered and state updated
725   | "fatal"         # - error message, command has failed
726   | "panic"         # - shutdown condition, component exits (interface may show message)
727   | "log"           # - system-level log message (interface does not show message; written to log file)
728   | "debug"         # - system-level debug message (interface may show message; written to log file)
729
730 # attributes describing a file location (for error messages, etc)
731 location_attrs =
732      attribute location_descr    { string }?,
733      attribute location_url      { xsd:anyURI }?,
734      attribute locationline      { xsd:positiveInteger }?,
735      attribute locationcolumn    { xsd:positiveInteger }?,
736      attribute locationcharacter { xsd:positiveInteger }?,
737      attribute locationlength    { xsd:positiveInteger }?
738
739 # instruction to insert some literal text into the document
740 scriptinsert = element scriptinsert { proverid_attr?, metavarid_attr?, string }
741
742
743 # metainformation is an extensible place to put system-specific information
744
745 value = element value { name_attr?, text }        # generic value holder [ deprecated: use metainfo ]
746 metainfo = element metainfo { name_attr?, text }  # generic info holder
747
748 metainforesponse =
749    element metainforesponse {
750       proverid_attr?,
751       attribute infotype { token },       # categorization of data
752       (value | metainfo)* }               # data values/text
753
754
```

```
755  # ==========================================================================
756  # 7. Proof script markup and proof control
757  # ==========================================================================
758
759  # properproofcmds are purely markup on native proof script (plain) text
760  properproofcmd =
761      opengoal          # open a goal in ambient context
762    | proofstep         # a specific proof command (perhaps configured via opcmd)
763    | closegoal         # complete & close current open proof (succeeds iff proven, may close nested pf)
764    | giveupgoal        # close current open proof, retaining attempt in script (Isar oops)
765    | postponegoal      # close current open proof, record as proof obl'n  (Isar sorry)
766    | comment           # a proof script comment; text probably ignored by prover
767    | doccomment        # a proof script document comment; text maybe processed by prover
768    | whitespace        # a whitespace comment; text ignored by prover
769    | spuriouscmd       # command ignored for undo, e.g. "print x", could be pruned from script
770    | badcmd            # a command which should not be stored in the script (e.g. an improperproofcmd)
771    | litcomment        # a PGIP literate comment (never passed to prover)
772    | pragma            # a document generating instruction (never passed to prover)
773
774  # improperproofcmds are commands which are never stored in the script
775  improperproofcmd =
776      dostep          # issue a properproofcmd (without passing in markup)
777    | undostep        # undo the last proof step issued in currently open goal
778    | redostep        # redo the last undone step issued in currently open goal (optionally supported)
779    | abortgoal       # give up on current open proof, close proof state, discard history
780    | forget          # forget a theorem (or named target), outdating dependent theorems
781    | restoregoal     # re-open previously postponed proof, outdating dependent theorems
782
783
784  scriptmarkup = pgml | text
785
786  opengoal     = element opengoal     { thmname_attr?, scriptmarkup } # TODO: add objprefval
787  proofstep    = element proofstep    { name_attr?, objtype_attr?, scriptmarkup }
788  closegoal    = element closegoal    { scriptmarkup }
789  giveupgoal   = element giveupgoal   { scriptmarkup }
790  postponegoal = element postponegoal { scriptmarkup }
791  comment      = element comment      { scriptmarkup }
792  doccomment   = element doccomment   { scriptmarkup }
793  whitespace   = element whitespace   { scriptmarkup }
794
795  spuriouscmd  = element spuriouscmd  { scriptmarkup }  # no semantic effect (e.g. print)
796  badcmd       = element badcmd       { scriptmarkup }  # illegal in script (e.g. undo)
797  nonscripting = element nonscripting { scriptmarkup }  # non-scripting text (different doc type)
798
799  litcomment  = element litcomment { format_attr?, (text | directive)* }
800  directive   = element directive { (proofctxt,pgml) }
801  format_attr = attribute format { token }
802
803  pragma       = showhidecode | setformat
804  showhidecode = element showcode { attribute show { xsd:boolean } }
805  setformat    = element setformat { format_attr }
806
807  dostep       = element dostep { string }
808  undostep     = element undostep { times_attr? }
809  redostep     = element redostep { times_attr? }
810  abortgoal    = element abortgoal { empty }
811  forget       = element forget { thyname_attr?, name_attr?, objtype_attr? }
812  restoregoal  = element restoregoal { thmname_attr }
813
814  times_attr   = attribute times { xsd:positiveInteger }
815
816  # empty objprefval element is used for object prefs in script markup
817  objprefval = element objprefval { name_attr, val_attr, empty }
818  val_attr   = attribute value { token }
819
820
821
822
823  # ====================================================
```

```
824 # Inspecting the proof context, etc.
825
826 # NB: ids/refs/parent: work in progress, liable to change.
827
828 proofctxt =
829     askids          # tell me about identifiers (given objtype in a theory)
830   | askrefs         # tell me about dependencies (references) of an identifier
831 #  | askparent      # tell me the container for some identifier
832   | showid          # print the value of some object
833   | askguise        # tell me about the current state of the prover
834   | parsescript     # parse a raw proof script into proofcmds
835   | showproofstate # (re)display proof state (empty if outside a proof)
836   | showctxt        # show proof context
837   | searchtheorems # search for theorems (prover-specific arguments)
838   | setlinewidth    # set line width for printing
839   | viewdoc         # request some on-line help (prover-specific arg)
840
841 askids = element askids  { thyname_attr?, objtype_attr? }
842         # Note that thyname_attr is *required* for certain objtypes (e.g. theorem).
843         # This is because otherwise the list is enormous.
844         # Perhaps we should make thyname_attr obligatory?
845         # With a blank entry (i.e. thyname="") allowed for listing theories, or for when
846         # you really do want to see everything (could be a shell-style glob)
847
848
849 # askids:    container -> identifiers contained within
850 # askparent: identifier + type + context -> container
851 # askrers:   identifier + type + context -> identifiers which are referenced
852 #
853 askrefs = element askrefs { url_attr?, thyname_attr?, objtype_attr?, name_attr? }
854 # TODO: maybe include guises here as indication of reference point.
855 # setrefs in reply to askrefs only really needs identifiers, but it's nice to
856 # support voluntary information too.
857 setrefs = element setrefs { url_attr?, thyname_attr?, objtype_attr?, name_attr?, idtable*, fileurl* }
858 fileurl = element fileurl { url_attr }
859 # telldeps = element telldeps { thyname_attr?, objtype_attr, name_attr?, identifier* }
860 # Idea: for a theory dependency we return a single file (URL), the containing file.
861 #       for a file dependency we return urls of parent files,
862 #       for theorem dependency we return theory
863 #       for term dependency we return definition (point in file)
864
865
866 showid = element showid  { thyname_attr?, objtype_attr, name_attr }
867
868 askguise = element askguise { empty }
869
870 showproofstate = element showproofstate { empty }
871 showctxt       = element showctxt { empty }
872 searchtheorems = element searchtheorems { string }
873 setlinewidth   = element setlinewidth { xsd:positiveInteger }
874 viewdoc        = element viewdoc { token }
875
876
877 # =====================================================
878 # Proof script documents and parsing proof scripts
879
880 # A PGIP document is a sequence of script commands, each of which
881 # may have meta information attached.
882 properscriptcmdmetainfo = properscriptcmd, metainfo*
883 pgipdoc = element pgipdoc { properscriptcmdmetainfo* }
884
885
886 # NB: parsing needs only be supported for "proper" proof commands,
887 # which may appear in proof texts. The groupdelimiters are purely
888 # markup hints to the interface for display structure on concrete syntax.
889 # The location attributes can be used by the prover in <parsescript> to
890 # generate error messages for particular locations; they can be used
891 # in <parseresult> to pass position information back to the display,
892 # particularly in the case of (re-)parsing only part of a file.
```

```
893 # The parsing component MUST return the same location attributes
894 # (and system data attribute) that was passed in.
895
896 parsescript = element parsescript
897                  { location_attrs, systemdata_attr, string }
898
899 parseresult = element parseresult { location_attrs, systemdata_attr,
900                                     singleparseresult* }
901
902 # Really we'd like parsing to return properscriptcmdmetainfo as a single
903 # result (and similarly for newobj).
904 # Unfortunately, although this is an XML-transparent extension, it
905 # messes up the Haskell schema-fixed code rather extensively, so for
906 # now we just treat metainfo at the same level as the other results,
907 # although it should only appear following a properscriptcmd.
908
909 singleparseresult  = properscriptcmd | metainfo | unparseable | errorresponse
910
911 unparseable = element unparseable { scriptmarkup }
912 properscriptcmd = properproofcmd | properfilecmd | groupdelimiter
913
914
915
916
917 groupdelimiter = openblock | closeblock
918 openblock  = element openblock {
919    name_attr?, objtype_attr?,
920    metavarid_attr?, positionid_attr?,
921    fold_attr?, indent_attr?,
922    display_attr? }
923 closeblock = element closeblock { empty }
924
925 metavarid_attr  = attribute metavarid { token }         #
926 positionid_attr = attribute positionid { token }        #
927 fold_attr       = attribute fold { xsd:boolean }        # whether to enable folding for this block
928 indent_attr     = attribute indent { xsd:integer }      # whether to indent this block [attrib in
929 display_attr    = attribute nodisplay { xsd:boolean }  # whether to display in documentation
930
931
932 # =====================================================
933 # Theory/file handling
934
935 properfilecmd =     # (NB: properfilecmds are purely markup on proof script text)
936     opentheory      # begin construction of a new theory.
937   | theoryitem      # a step in a theory (e.g. declaration/definition of type/constant).
938   | closetheory     # complete construction of the currently open theory
939
940 improperfilecmd =
941     doitem          # issue a proper file command (without passing in markup)
942   | undoitem        # undo last step (or named item) in theory construction
943   | redoitem        # redo last step (or named item) in theory construction (optionally supported)
944   | aborttheory     # abort currently open theory
945   | retracttheory   # retract a named theory
946   | openfile        # signal a file is being opened for constructing a proof text interactively
947   | closefile       # close the currently open file, suggesting it has been processed
948   | abortfile       # unlock a file, suggesting it hasn't been processed
949   | loadfile        # load (i.e. process directly) a file possibly containing theory definition(s)
950   | retractfile     # retract a given file (including all contained theories)
951   | changecwd       # change prover's working directory (or load path) for files
952   | systemcmd       # system (other) command, parsed internally
953
954 fileinfomsg =
955    informfileloaded        # prover informs interface a particular file is loaded
956  | informfileretracted     # prover informs interface a particular file is outdated
957  | informguise             # prover informs interface about current state
958
959 opentheory    = element opentheory    { thyname_attr, parentnames_attr?, scriptmarkup }
960 closetheory   = element closetheory   { scriptmarkup }
961 theoryitem    = element theoryitem    { name_attr?, objtype_attr?, scriptmarkup } # FIXME: add objpref
```

```
962
963 doitem         = element doitem        { string }
964 undoitem       = element undoitem      { name_attr?, objtype_attr?, times_attr? }
965 redoitem       = element redoitem      { name_attr?, objtype_attr?, times_attr? }
966 aborttheory    = element aborttheory   { empty }
967 retracttheory  = element retracttheory { thyname_attr }
968
969 parentnames_attr = attribute parentnames { objnames }
970
971
972 # Below, url_attr will often be a file URL. We assume for now that
973 # the prover and interface are running on same filesystem.
974 #
975
976 openfile       = element openfile   { url_attr }     # notify begin reading from given file
977 closefile      = element closefile  { empty }        # notify currently open file is complete
978 abortfile      = element abortfile  { empty }        # notify currently open file is discarded
979 loadfile       = element loadfile   { url_attr }     # ask prover to read file directly
980 retractfile    = element retractfile { url_attr }    # ask prover to retract file
981 changecwd      = element changecwd  { url_attr }     # ask prover to change current working dir
982
983 # this one not yet implemented, but would be handy. Perhaps could be
984 # locatethy/locatefile instead.
985 #locateobj     = element locateobj { name_attr, objtype_attr } # ask prover for file defining obj
986
987 informfileloaded    = element informfileloaded    { completed_attr?,
988                                                     url_attr } # prover indicates a processed file
989 informfileretracted = element informfileretracted { completed_attr?,
990                                                     url_attr } # prover indicates an undone file
991 informfileoutdated  = element informfileoutdated { completed_attr?,
992                                                     url_attr } # prover indicates an outdated file
993
994 informfilelocation  = element informfilelocation  { url_attr } # response to locateobj
995
996 completed_attr = attribute completed { xsd:boolean } # false if not completed (absent=>true)
997                                                      # (the prover is requesting a lock)
998
999
1000
1001 informguise =
1002   element informguise {
1003     element guisefile { url_attr }?,
1004     element guisetheory { thyname_attr }?,
1005     element guiseproof { thmname_attr?, proofpos_attr? }?
1006   }
1007
1008 proofpos_attr = attribute proofpos { xsd:nonNegativeInteger }
1009
1010 systemcmd      = element systemcmd     { string }        # "shell escape", arbitrary prover command!
1011
1012 # ======================================================================
1013 # 8. Internal messages: only used between communicating brokers.
1014 # ======================================================================
1015 internalmsg  = launchcomp | stopcomp | registercomp | compstatus
1016
1017 launchcomp   = element launchcomponent { componentspec }
1018                      # request to start an instance of this component remotely
1019 stopcomp     = element stopcomponent { attribute sessionid { token } }
1020                      # request to stop component with this session id remotely
1021
1022 registercomp = element registercomponent { activecompspec }
1023                      # component has been started successfully
1024 compstatus   = element componentstatus { componentstatus_attr    # status
1025                                        , componentid_attr?       # component id (for failure)
1026                                        , element text { string }? # user-visible error message
1027                                        , element info { string }? # Additional info for log files.
1028                                        }
1029                      # component status: failed to start, or exited
1030
```

```
1031 componentstatus_attr = attribute status { ("fail"   # component failed to start
1032                                           |"exit"  # component exited
1033                                           )}
1034
1035 # Local variables:
1036 # fill-column: 95
1037 # End:
1038 # ============================================================================
1039 # end of `pgip.rnc'.
```

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 L th, University of Bremen
6  #
7  # Version: $Id: pgml.rnc,v 3.12 2007/07/11 10:21:12 da Exp $
8  #
9  # Status:  Complete, prototype.
10 #
11 # This is a simple markup language for annotating prover input and output
12 # text.  Annotations can be used to provide pretty printing or additional
13 # structure on already pretty-printed text.  The markup is intended to be
14 # further transformed (e.g. by system-specific stylesheets) to produce
15 # HTML or other display forms.
16 #
17 # For additional commentary, see accompanying commentary document
18 # (available at http://proofgeneral.inf.ed.ac.uk/kit)
19 #
20 # With thanks to:
21 #   Cezary Kaliszyk, Makarius Wenzel
22 #
23 # Advertised version: 2.0
24 #
25
26 # Namespace: this breaks DtdToHaskell, currently removed
27 #default namespace pgip = "http://proofgeneral.inf.ed.ac.uk/pgip"
28
29 ##
30 ## PGML terms with pretty printing, actions, references and embedded objects.
31 ##
32 term            = sym | atom | box | break | subterm | alt | embed | text
33
34 ## A named symbol may have a previously configured alternative
35 ## rendering (see below) or be rendered as the given text
36 sym             = element sym { symname_attr, text }
37 symname_attr    = attribute name { xsd:NMTOKEN }        # names must be [a-zA-Z][a-zA-Z0-9]+
38
39 ## An atom is some text (maybe including symbols), with an optional
40 ## kind which may control its printing by a prover-specific style
41 ## sheet.
42 atom            = element atom { kind_attr?, (text | sym)* }
43 kind_attr       = attribute kind { token }
44
45 ## A box whose children may be displayed with different layout strategies, using
46 ## the directives of BoxML (see http://helm.cs.unibo.it/mml-widget/qna.html).
47 ## The children of the box may have: fixed horizontal or vertical layout,
48 ## horizontal or vertical with inconsistent breaking ("hov", the default),
49 ## or horizontal or vertical with consistent breaking ("hv").
50 ## Inside a box, the given indentation space is applied to children
51 ## if a line break is added.  Note and spaces/newlines in the markup are
52 ## NOT obeyed (explicit spaces must be given with atoms).
53 box             = element box { orient_attr?, indent_att?, term* }
54 orient_attr     = attribute orient { "hov" | "h" | "v" | "hv" }
55 indent_att      = attribute indent { xsd:integer }
56
57 ## A line/column breaking hint.  The indentation is added if the line
58 ## is not broken.  Consistent breaking will take all breaks at the same level.
59 break           = element break { mandatory_attr?, indent_att? }
60 mandatory_attr = attribute mandatory { xsd:boolean }
61
62 ## An annotated subterm.  This may be used to:
63 ##
64 ## - subdivide large displays (e.g. goal state by subgoals, assumptions)
65 ##     using the kind and parameter tokens (system specific interpretation);
66 ## - mark an identifer or term with an objtype (and perhaps a global name);
```

```
67  ## - decorate a region with a bold, italic, or other highlight (e.g.,
68  ##     to markup error/warning/info areas with squiggles red/yellow/blue).
69  ## - reference to a content model with position annotations
70  ## - indicate an action such as a system-specific menu, folding ("toggle")
71  ##     or render a button
72  ## - give a cross reference to the defining point of some text
73  ##    (e.g. file URL with line.column fragment)
74  ##
75  subterm        = element subterm { skind_attr?, sparam_attr?,
76                                 splace_attr?, sdec_attr?, sobjtype_attr?, sname_attr?,
77                                 saction_attr?, spos_attr?, sref_attr?, term }
78  skind_attr    = attribute kind { token }
79  sparam_attr   = attribute param { token }
80  splace_attr   = attribute place { "sub" | "super" | "above" | "below" }
81  sobjtype_attr = attribute objtype { token }
82  sname_attr    = attribute name { token }
83  sdec_attr     = attribute decoration { "bold" | "italic" | "error" | "warning" | "information" | token
84  saction_attr  = attribute action { "toggle" | "button" | "menu" }
85  spos_attr     = attribute pos { token }
86  sref_attr     = attribute xref { xsd:anyURI }
87
88
89  ## Alternative subtrees. This may be used to select between subtrees based on
90  ## the kind, for example, alternate renderings according available space, or
91  ## alternate embedded objects according to target language.
92  alt            = element alt { altkind_attr, term* }
93  altkind_attr   = attribute kind { token }
94
95  ## An embedded object.  The object embedded would typically be in a
96  ## different markup language (e.g. MathML).
97  embed          = element embed { text }
98
99
100
101 ##
102 ## PGML documents
103 ##
104
105 pgml = element pgml { pgml_version_attr?, systemid_attr?,
106                       area_attr?, term* }
107
108 ## Version of PGML.  Useful for standalone documents.  Broker may add.
109 pgml_version_attr = attribute version { xsd:NMTOKEN }
110
111 ## System that produced PGML.  Useful for standalone documents.  Broker may add.
112 systemid_attr     = attribute systemid { token }
113
114 ## Display area (e.g., window) for showing message.  Only relevant for output;
115 ## default area is message in this case.
116 area_attr =
117    attribute area {
118                  "status"      # a status line
119                | "message"     # the message area
120                | "display"     # the proof state display area
121                | "tracing"     # tracing output
122                | "internal"    # debug/log message, not displayed
123                | token         # prover-specified window name
124                           }
125
126 ## This schema is embedded into the PGIP schema.  If we would
127 ## keep them separate, we would add the start production here:
128 ##
129 # start = pgml
130
131
132
133 ##
134 ## Symbol configuration
135 ##
```

```
136  ## The sym element can be rendered in three different ways,
137  ## in descending preference order:
138  ##
139  ## 1) one of the previously configured alternatives for the symbol,
140  ##    available in a prover-specific configuration, see below.
141  ## 2) the PGML symbol given by the name attribute, if one of
142  ##    standard (TeX-ish) names given elsewhere (deprecated).
143  ## 3) the text content of the SYM element, if non-empty
144  ##
145  ## The symbol configuration allows flexible choice between
146  ## different renderings.
147  ##
148
149  ## A configuration of PGML is a sequence of symbol configurations
150  pgmlconfiguration = symconfig*
151
152  ##
153  ## A symbol configuration provides:
154  ##
155  ##  a (unique) symbol name, used in the <sym> element
156  ##  a family name (user documentation only)
157  ##  a unicode character sequence
158  ##  an HTML alternative using HTML entities (in plain ASCII)
159  ##  a plain text alternative for impoverished ASCII displays
160  ##  a token text which can be used in the document in place of the unicode sequence
161  ##    to store files in a poorer ASCII/latin/other encoding
162  ##  a typing shortcut that may be supported by editors in displays
163  ##  an xml rendering (in some other XML schema)
164  ##
165  ## e.g.
166  ##
167  ## <symconfig name="Forall" family="Symbols" unicode="   " shortcut="all "
168  ##             html="&forall;" token="\<forall>" alt="ALL" />
169  ##
170  ## Symbol configurations are prover specific.  They may be shared
171  ## between displays, although different displays may use different
172  ## parts of configurations.
173
174  symconfig =
175     element symconfig { symname_attr, family_attr?,
176        unicode_attr?, html_attr?, textalt_attr?,
177        token_attr?, shortcut_attr?, xmlrender? }
178
179  family_attr   = attribute family { string }
180  unicode_attr  = attribute unicode { string }
181  html_attr     = attribute html { string }
182  textalt_attr  = attribute alt { string }
183  token_attr    = attribute token { string }
184  shortcut_attr = attribute shortcut { string }
185  xmlrender     = element xmlrender { text }
```