Appendix B

Web Prolog predicate APIs

B.1 The actor API

Predicate: `self/1`

```
self(-Pid) is det.
```

Binds `Pid` to the process identifier of the calling process.

Predicate: `spawn/2-3`

```
spawn(+Goal, -Pid) is det.
spawn(+Goal, -Pid, +Options) is det.
```

Creates a new Web Prolog process running `Goal`. Valid options are:

- `node(+URI)`
  `URI` points to the Prolog Web node on which to create the process. Default is the current node `localnode`.
- `monitor(+Boolean)`
  Default is to not monitor.
- `link(+Boolean)`
  Default is to not link.
- `timeout(+IntegerOrFloat)`
  Terminates the spawned process (or the process of spawning a process) after `IntegerOrFloat` seconds.
- `src_list(+ListOfClauses)`
  Injects a list of Web Prolog clauses into the process.
- `src_text(+AtomOrString)`
  Injects the clauses specified by a source text into the process.

- `src_uri(+URI)`
  Injects the clauses specified in the source code located at `URI` into the process.
- `src_predicates(+List)`
  Injects the local predicates denoted by `List` into the process. `List` is a list of predicate indicators.
- `type(+Atom)`
  Indicates the type of the source to be injected into the process. Default is `web-prolog`. Note that some `src_*` options may not be compatible with other values of this option.

Predicate: `!/2`

```
+PidOrName ! +Message is det.
send(+PidOrName, +Message) is det.
```

Sends `Message` to the mailbox of the process identified as `PidOrName`. A message can be any Web Prolog term except a bare variable. The sending is asynchronous, i.e. `!/2` does not block waiting for a response but continues immediately. Also, `!/2` does not throw exceptions, so if a process named Pid does not exist, nothing happens.

Predicate: `raise/1`

```
raise(+Message) is det.
```

Sends `Message` to the mailbox of the current process. Defined as

```
raise(Message) :-
    self(Pid),
    Pid ! Message.
```

Predicate: `return/1`

```
return(+Message) is det.
```

Sends `Message` to the mailbox of the process that spawned the current process. Defined as

```
return(Message) :-
    '$parent'(Pid),
    Pid ! Message.
```

Predicate: `receive/1-2`

```
receive(+Clauses) is semidet.
receive(+Clauses, :Options) is semidet.
```

`Clauses` is a sequence of receive clauses delimited by a semicolon:

```
{  Pattern1 [when Guard1] ->
         Body1 ;
   ...
   PatternN [when GuardN] ->
         BodyN
}
```

Each pattern in turn is matched against the first message (the one that has been waiting longest) in the mailbox. If a pattern matches and the corresponding guard succeeds, the matching message is removed from the mailbox and the body of the receive clause is called. If the first message is not accepted, the second one will be tried, then the third, and so on. If none of the messages in the mailbox is accepted, the process will wait for new messages, checking them one at a time in the order they arrive. Messages in the mailbox that are not accepted are left in the mailbox without any change in their contents or order. Valid options:

- `timeout(+IntegerOrFloat)`
  If nothing appears in the current mailbox within `IntegerOrFloat` seconds, the predicate succeeds anyway. Default is no timeout.

- `on_timeout(+Goal)`
  If the timeout occurs, `Goal` is called.

Predicate: `exit/1-2`

```
exit(+Reason) is det.
exit(+PidOrName, +Reason) is det.
```

Executing `exit/1` terminates the current process. The predicate `exit/2` can be used to terminate any process with a known pid or registered name, but only by its owner.

B.2 The pengine API

Predicate: `pengine_spawn/1-2`

```
pengine_spawn(-Pid) is det
pengine_spawn(-Pid, +Options) is det
```

Spawns a pengine and binds `Pid` to its pid.

With just one exception, all options that can be passed to `pengine_spawn/2` are inherited from `spawn/3`. Thus, the options `monitor`, `link`, `src_list`, `src_text`, `src_uri` and `src_predicates` are valid here too.

The only new option that is added is:

- `exit(+Boolean)`
  Determines if the pengine session must exit after having run a goal to completion. Defaults to true.

Predicate: `pengine_ask/2-3`

```
pengine_ask(+Pid, +Goal) is det.
pengine_ask(+Pid, +Goal, +Options) is det
```

Calls pengine `Pid` with the goal `Goal`. Valid options are:

- `template(+Template)`
  `Template` is a variable (or a term containing variables) shared with the query. By default, the template is identical to the goal.

- `limit(+Integer)`
  Retrieve solutions in lists of length `Integer`. A value of 1 means a unary list (default). Other integers indicate the maximum number of solutions to retrieve in one batch.

`pengine_ask/2-3` is deterministic, even for queries that have more than one solution. Variables in `Goal` will not be bound. Instead, results and other kinds of output will be returned in the form of messages delivered to the mailbox of the process that called `pengine_spawn/2-3`.

- `success(Pid, Terms, More)`
  `Pid` refers to the pengine that succeeded in solving the query. `Terms` is a list holding instantiations of `Template`. `More` is either `true` or `false`, indicating whether or not we can expect the pengine to be able to return more solutions, would we call `pengine_next/1-2`.[1]

- `failure(Pid)`
  `Pid` is the pid of the pengine that failed for lack of (more) solutions.

- `error(Pid, Term)`
  `Pid` is the pid of the pengine throwing the exception. `Term` is the exception's error term.

- `output(Pid, Term)`
  `Pid` is the pid of a pengine running the goal that called `pengine_output/1`. Term is the term passed in the argument of `pengine_output/1` when it was called.

- `prompt(Pid, Term)`
  `Pid` is the pid of the pengine that called `pengine_input/2` and `Term` is the prompt.

- `down(Pid, Term)`
  `Pid` is the pid of the pengine that terminated and `Term` is the reason.

Predicate: `pengine_next/1-2`

```
pengine_next(+Pid) is det.
pengine_next(+Pid, +Options) is det
```

Asks pengine `Pid` for the next solution to `Goal`. The only valid option is:

- `limit(+Integer)`
  Retrieve solutions in lists of length `Integer`. A value of 1 means a unary list (default). Other integers indicate the maximum number of solutions to retrieve in one batch.

The messages delivered to the mailbox of the process that called `pengine_next/1-2` are the same as for `pengine_ask/2-3`.

[1] We are considering adding a fourth argument `Info`, where `Info` is a structure containing extra information such as timing information.

Predicate: `pengine_stop/1`

```
pengine_stop(+Pid) is det.
```

Asks pengine `Pid` to stop. If successful, delivers a message `stop(Pid)` to the mailbox of the process that called `pengine_spawn/2-3`.

Predicate: `pengine_abort/1`

```
pengine_abort(+Pid) is det.
```

Tells pengine `Pid` to abort any goal it currently runs. If successful, delivers a message `abort(Pid)` to the mailbox of the process that called `pengine_spawn/2-3`.

Predicate: `pengine_exit/1-2`

```
pengine_exit(+Reason) is det.
pengine_exit(+Pid, +Reason) is det.
```

Same as `exit/1` and `exit/2`.

Predicate: `pengine_output/1`

```
pengine_output(+Term) is det.
```

Sends a message `output(Pid,Term)` to the parent process. `Pid` is the pid of the current process. It is defined as follows:

```
pengine_output(Term) :-
    self(Pid),
    return(output(Pid, Term)).
```

Note that this is just a convenience predicate. A pengine, just like any other actor, may use `return/1` directly in order to send *any* term to its parent.

Predicate: `pengine_input/2`

```
pengine_input(+Prompt, -Term) is det.
```

Sends a message `prompt(Pid,Prompt)` to the parent process and waits for its input. `Prompt` may be any term (i.e. even a compound term). `Pid` is the pid of the current process. `Term` will be bound to the term that the parent process sends using `pengine_respond/2`.

Predicate: `pengine_respond/2`

```
pengine_respond(+Pid, +Input) is det.
```

Sends a response in the form of the term `Input` to a process that has prompted its parent process for input.

B.3 The RPC API

Predicate: `rpc/2-3`

```
rpc(+URI, +Query) is nondet.
rpc(+URI, +Query, +Options) is nondet.
```

Semantically equivalent to the sequence below, except that the query is executed in (and in the Prolog context of) the node referred to by `URI`, rather than locally.

```
copy_term(Query, Copy),
call(Copy),               % executed on node at URI
Query = Copy.
```

All options for `pengine_spawn/3` are valid for `rpc/3` as well, except for `node`, `exit` and `monitor`.

Predicate: `promise/3-4`

```
promise(+URI, +Query, -Reference) is det.
promise(+URI, +Query, -Reference, +Options) is det.
```

Makes an asynchronous RPC call to node `URI` with `Query`. This is a type of RPC which does not suspend the caller until the result is computed. Instead, a reference is returned, which can later be used by `yield/2-3` to collect the answer. The reference can be viewed as a promise to deliver the answer. Valid options are `template`, `offset`, `limit` and `timeout`.

Predicate: `yield/2-3`

```
yield(+Reference, ?Message) is det.
yield(+Reference, ?Message, +Options) is det.
```

Returns the promised answer from a previous call to `promise/3-4`. If the answer is available, it is returned immediately. Otherwise, the calling process is suspended until the answer arrives from the node that was called. The only valid option is `timeout`.

Note that this predicate must be called by the same process from which the previous call to `promise/3-4` was made, otherwise it will not return.