Jekejeke Develop Reference

Version 1.3.4, December 30th, 2018

XLOG Technologies GmbH

Jekejeke Prolog
Development Environment 1.3.4
Language Reference

Author: XLOG Technologies GmbH
Jan Burse
Freischützgasse 14
8004 Zürich
Switzerland

Date: December 30th, 2018
Version: 0.36

Participants: None

Warranty & Liability

Rights & License

Trademarks

Table of Contents

Change History

Jan Burse, April 25th, 2011, 0.1:
- Derived from the language reference and the console manual.

Jan Burse, April 30th, 2011, 0.2:
- Clause information and flags & properties section added.

Jan Burse, Mai 6th, 2011, 0.3:
- Interactions and Prolog flags enhanced.

Jan Burse, June 15th, 2011, 0.4:
- Input/output test predicates updated.

Jan Burse, August 19th, 2011, 0.5:
- Instruction table updated, flags updated and test predicates moved to runtime library.

Jan Burse, September 17th, 2011, 0.6:
- Internal database predicates moved to minimal logic and intermediate code changes.

Jan Burse, October 6th, 2011, 0.7:
- References, index dump and frame property section added.

Jan Burse, November 23th, 2011, 0.8:
- Metas & buddies and call back section added.

Jan Burse, December 2nd, 2011, 0.9:
- Conversation section removed, frame and clause property section introduced.

Jan Burse, December 27th, 2011, 0.10:
- Custom debugger, status handling and port statistics section introduced.

Jan Burse, March 12th, 2012, 0.11:
- Interrupt handling section introduced.

Jan Burse, June 4th, 2012, 0.12:
- Few enhancements.

Jan Burse, October 24th, 2012, 0.13:
- Few enhancements.

Jan Burse, November 19th, 2012, 0.14:
- Debug section before inspect section, tracing example and index section introduced.

Jan Burse, February 26th, 2013, 0.15:
- Stability analysis removed.

Jan Burse, May 7th, 2013, 0.16:
- New clause property sys_notrace.

Jan Burse, March 31st, 2014, 0.17:
- Module system introduced.

Jan Burse, August 7th, 2014, 0.18:
- More inspection predicates introduced.

Jan Burse, September 6th, 2014, 0.19:
- Inspection predicates improved.

Jan Burse, February 27th, 2015, 0.20:
- Frame properties improved and atom properties introduced.

Jan Burse, May 28th, 2015, 0.21:
- Title page introduced and error messages removed.

Jan Burse, July 3rd, 2015, 0.22:
- New debugger commands introduced.

Jan Burse, September 4th, 2015, 0.23:
- Improvements to the inspection.

Jan Burse, January 9th, 2016, 0.24:
- Predicates for evaluable functions removed and some additions to the inspection.

Jan Burse, March 6th, 2016, 0.25:
- Stack trace limit.

Jan Burse, April 13th, 2016, 0.26:
- New miscellaneous definitions section and new indexes section.

Jan Burse, May 21th, 2016, 0.27:
- New unify port and new modules tracker and cover introduced.

Jan Burse, August 28th, 2016, 0.28:
- Some improvements in instrumentation.

Jan Burse, November 29th, 2016, 0.29:
- Some changes because of late binding operator (::)/2 fixes.

Jan Burse, May 11th, 2017, 0.30:
- Clause reference access removed and moved to the Prolog runtime.

Jan Burse, November 02th, 2017, 0.31:
- Some improved clause index dump.

Jan Burse, May 10th, 2018, 0.32:
- New module notation and callable properties from runtime moved to here.

Jan Burse, August 16th, 2018, 0.33:
- Minor changes due to new reference counting.

Jan Burse, October 08th, 2018, 0.34:
- Deterministic debugger introduced.

Jan Burse, November 24th, 2018, 0.35:
- Package "notebook" to house DOM model modules from runtime.

Jan Burse, December 30th, 2018, 0.36:
- New package "wire".

1 Introduction

This document gives a reference to the programming language of the Jekejeke Prolog development environment. Here and there we will compare our definitions with the DEC-10 standard [1] and the ISO Prolog standard [2].

- **Environment Examples:** We show some examples of the use of the programming language in connection with the Jekejeke Prolog development environment.

- **Environment Conversations**: The Jekejeke Prolog development environment provides terminal based interactions. Among the interactions we find goal debugging.

- **Environment Syntax:** In this section we show what syntax the Jekejeke Prolog development environment accepts.

- **Development Packages:** The Jekejeke Prolog development environment provides a set of system predicates to control the debugger and the interpreter. These commands only assume a terminal.

- **Appendix Example Listing:** The full source code of Prolog examples is given.

2 Environment Examples

We show some examples of the use of the programming language in connection with the Jekejeke Prolog development environment. Readers who might be interested in getting a quick grip of the Jekejeke Prolog development environment and who have already a basic knowledge of Prolog might stick to this section only.

- **Deterministic Tracing:** We demonstrate deterministic tracing by the default debugger for a simple example.

- **Non-Deterministic Tracing:** We demonstrate non-deterministic tracing by the default debugger for a simple example.

- **Advanced Tracing:** We demonstrate some advanced tracing by the default debugger for a simple example.

- **Port Statistics:** We show how to replace the default debugger by a custom debugger that counts the goal ports.

2.1 Deterministic Tracing

We demonstrate deterministic tracing of the default debugger for a simple example. The execution of a predicate works in that it determines a substitution for the argument variables. This execution might or might not succeed and it then might or might not leave choice points. A deterministic execution is a successful execution without choice points.

Whether a predicate executes deterministically or not depends on pattern of arguments. Some pattern of arguments can execute deterministically other pattern of arguments might execute non-deterministically. We will use the following Prolog text, which defines a reverse predicate, as the working example to demonstrate deterministic tracing:

```
rev(X, Y) :- rev(X, [], Y).
rev([], X, X).
rev([X|Y], Z, T) :- rev(Y, [X|Z], T).
```

We can build trust that the predicate is correctly implemented by running a few test cases. This will be a black-box test, since we will only see the input argument and the output argument. The top-level will also immediately return to the prompt if an execution succeeded deterministically, so that we see that as well:

```
?- rev([1,2,3], X).
X = [3,2,1]
?-
```

The default debugger now allows performing a more white-box testing, in that it will halt on the call port and the exit port of each execution of a sub predicate. We could also archive this by placing write statements in the source code. The development environment does that instrumentation for us, it is a matter of the trace/0 debugger directive to start tracing:

```
?- trace.
Yes
?- rev([1,2,3], X).
    0 Call rev([1,2,3], X) ?
    1 Call rev([1,2,3], [], X) ?
    2 Call rev([2,3], [1], X) ?
    3 Call rev([3], [2,1], X) ?
    4 Call rev([], [3,2,1], X) ?
    4 Exit rev([], [3,2,1], [3,2,1]) ?
    3 Exit rev([3], [2,1], [3,2,1]) ?
    2 Exit rev([2,3], [1], [3,2,1]) ?
    1 Exit rev([1,2,3], [], [3,2,1]) ?
    0 Exit rev([1,2,3], [3,2,1]) ?
X = [3,2,1]
?-
```

In the above, we have just pressed the return button after each debugger prompt "?". Since the mode was trace mode, this is known as creeping among Prolog interpreters. When the prompt "?" appears the end-user can either use the mouse or the keyboard to issue debugger directives. For more information, see the next chapter.

2.2 Non-Deterministic Tracing

We demonstrate non-deterministic tracing of the default debugger for a simple example. Non-deterministic already happens if a predicate fails. The execution at one point then does not reach an exit port but instead a fail port. We can use the debugger directive nodebug/0 to switch off tracing. Without tracing, we only see a negative answer in the top-level:

```
?- nodebug.
Yes
?- rev([a|b], X).
No
```

An execution that fails or succeeds deterministically is called semi-deterministic. For semi-deterministic executions, the default debugger can deliver valuable information in that it can show the fail port where the predicate fails for the first time. We will see a sequence of call ports that are not anymore balanced by exit ports, but by fail ports instead:

```
?- trace.
Yes
?- rev([a|b], X).
    0 Call rev([a|b], X) ?
    1 Call rev([a|b], [], X) ?
    2 Call rev(b, [a], X) ?
    2 Fail rev(b, [a], X) ?
    1 Fail rev([a|b], [], X) ?
    0 Fail rev([a|b], X) ?
No
```

We will now use a different and simpler predicate to show non-deterministic execution. The predicate will only consist of a couple of facts. Facts are a simple way to represent lists of data and we will therefore name the corresponding predicate data/1. The only argument will consist of ground lists:

```
data([1,2]).
data([4,3]).
```

When using a variable call pattern the predicate data/1 will leave choice points. The top-level will then show the first answer substitution without a further prompt "?-". We need to enter the semicolon ";" to get further answer substitutions. The last result is deterministically returned, since the Prolog interpreter is clever enough to eliminate the choice point:

```
?- nodebug.
Yes
?- data(X).
X = [1,2] ;
X = [4,3]
?-
```

When an execution leaves some choice points, the default debugger will show redo ports. Not only will the top-level will show that a choice point was eliminated. The default debugger will also recognize the situation. As a result, the default debugger will only show a single redo port and not two redo ports for the example:

```
?- trace.
Yes
?- data(X).
    0 Call data(X) ?
    0 Exit data([1,2]) ?
X = [1,2] ;
    0 Redo data([1,2]) ?
    0 Exit data([4,3]) ?
X = [4,3]
?-
```

Instead of entering the semicolon ";" it is also possible to stop showing further answer substitutions while in debug mode. We use the recursive predicate from the previous section to generate a potential infinite stream of answer substitutions. In ordinary mode, the stream can be stopped by pressing return instead of entering the semicolon ";":

```
?- rev(X, Y).
X = [],
Y = [] ;
X = [_A],
Y = [_A] ;
X = [_A,_B],
Y = [_B,_A]
?-
```

This top-level functionality is also available while in debug mode. We run the same recursive predicate again, this time in debug mode. This will also demonstrate how additional variables can become visible. In the above example, a single new variable was named "_A". Because we look behind the scene, two new variables will be named "_A" and "_B":

```
?- trace.
Yes
?- rev(X, X).
    0 Call rev(X, X) ?
    1 Call rev(X, [], X) ?
    1 Exit rev([], [], []) ?
    0 Exit rev([], []) ?
X = [] ;
    0 Redo rev([], []) ?
    1 Redo rev([], [], []) ?
    2 Call rev(_A, [_B], [_B|_A]) ?
    2 Exit rev([], [_B], [_B]) ?
    1 Exit rev([_B], [], [_B]) ?
    0 Exit rev([_B], [_B]) ?
X = [_B]
?-
```

In the above, we have just entered a semicolon ";" or pressed return to control the stream of answer substitutions. When an answer substitution without a further prompt "?-" appears the end-user can either use the mouse or the keyboard to also issue debugger directives. For more information on debugger directives, see the next chapter.

2.3 Advanced Tracing

We demonstrate advanced tracing of the default debugger for a simple example. Determinism and non-determinism can be combined in ordinary execution and they are an important building block of Prolog search. We use the predicate from the previous sections and try a search problem. The first attempt goes wrong:

```
?- nodebug.
Yes
?- rev(X, [3,4]).
X = [4,3] ;
Error: Execution aborted since memory threshold exceeded.
          rev/3
          rev/2
```

By tracing the offending execution, we can get an idea why the search went astray. We need to use the debug/0 debugger directive to let the debugger accumulate tracing information. We can then switch to tracing after the first answer substitution and we will see all redo ports so far. We will also see how a call ports build-up:

```
?- debug.
Yes
?- rev(X, [3,4]).
X = [4,3] trace.
X = [4,3] ;
    0 Redo rev([4,3], [3,4]) ?
    1 Redo rev([4,3], [], [3,4]) ?
    2 Redo rev([3], [4], [3,4]) ?
    3 Redo rev([], [3,4], [3,4]) ?
    4 Call rev(_A, [_B,_C,_D], [3,4]) ?
    5 Call rev(_E, [_F,_B,_C,_D], [3,4]) ?
    6 Call rev(_G, [_H,_F,_B,_C,_D], [3,4]) ?
    7 Call rev(_I, [_J,_H,_F,_B,_C,_D], [3,4]) ?
    8 Call rev(_K, [_L,_J,_H,_F,_B,_C,_D], [3,4]) ? abort.
?-
```

We used the abort/0 debugger directive to abort the offending execution, so that we do not need to reach the memory threshold to return to the top-level. The lesson here is that simple predicates need not be bi-directional and always work for all call patterns. How to solve this problem depends on the actual requirements. Here is a simple solution with length/2:

```
?- nodebug.
Yes
?- use_module(library(basic/lists)).
% 1 consults and 0 unloads in 16 ms.
Yes
?- length(X, 2), rev(X, [3,4]).
X = [4,3]
```

The above example gave a scenario to use the debug/0 debugger directive. The above example also showed valuable use cases of debugger directives either after an answer substitution without a further prompt "?-" or after a debugger prompt "?". For more information on debugger directives, see the next chapter.

2.4 Port Statistics

In the following we will customize trace_goal/2. The idea is to provide a custom debugger call back that counts the invoked goal ports. We can do so by providing additional rules for the multi-file predicate goal_tracing/2. This predicate takes as an argument the port identifier and the goal frame. We would like to record the port invocation on a per goal predicate basis. We will use the following Prolog fact to record the invocation:

```
% count(Fun, Arity, CallExitRedoFail).
```

The third argument will be an aggregate of the number of invocations for the corresponding ports. We can retrieve the predicate indicator from a goal frame via the system predicate frame_property/2. The custom call-back then reads as follows:

```
% goal_tracing(+Port, +Frame)
goal_tracing(P, Q) :-
    frame_property(Q, sys_call_goal(G)),
    functor(G, F, A),
    get_delta(P, D),
    update_count(F, A, D).
```

The custom call back makes use of the predicates get_delta/2 and update_count/3. The predicate get_delta/2 will determine which part of the aggregate has to be incremented depending on the port. And the predicate update_count/3 will do the increment. Besides the custom call back we have also provided the commands show/0 and reset/0. More details on the implementation can be found in the appendix.

Let’s do some sample runs. We will look into the port statistics of the Peano factorial program. We first need to consult both the call back and the target program:

```
?- ['fac.p', 'count.p'].
```

We can now measure one particular predicate in that we set a spy point on it. This is done with the usual debugger command spy/2. We will measure the add/3 predicate:

```
?- spy(add/3).
```

All that remains to do is switching the debugger on and running our example program. To switch the debugger on we also use the usual debugger command debug/2. We will run the Peano factorial program for the Peano number 7:

```
?- debug.
?- fac(s(s(s(s(s(s(s(n))))))),_).
```

The program will behave nearly as executed without debugging. The only thing that can be seen is performance degradation and maybe a change in determinism. The performance degradation is the net effect of checking the debugger conditions and invoking the custom call back. The change in determinism is currently the effect of the execution event handlers which introduce additional choice points.

When the execution has finished we can proceed in looking at the results. We will first switch the debugger off so that our results are not spoiled by reporting function. We get the following results:

```
?- nodebug.
?- show.
Pred      Call      Exit      Redo      Fail
add / 3   5941      5941      0         0
```

Instead of setting a spy point we can also run the example program in trace mode. Then the call back will be invoked for every port of a traceable goal. But before we run our example program again we will reset the port statistics:

```
?- reset.
?- trace.
?- fac(s(s(s(s(s(s(s(n))))))),_).
```

The results will now not only include the add/3 predicate, but also the mul/3 and fac/2 predicate from the Peano factorial program. The result for the add/3 predicate should not differ from our previous result since ports are independently counted. We get the following results:

```
?- nodebug.
?- show.
Pred      Call      Exit      Redo      Fail
add / 3   5941      5941      0         0
mul / 3   35        35        0         0
fac / 2   8         8         0         0
```

The above port statistics for the Peano program does not show any redo or fail counts. The main reason is that we did not cause a redo of our main query for a factorial of the Peano 7. As a result after the success of the main query all the remaining choice points were removed. The process of removing choice points is not visible to the instrumentation.

We night ask whether a more refined port statistic is possible. Currently we support call site identification down to the line number of the active clause. It is planned to also allow the identification of the line number of the active goal, but this has not yet been implemented. The call site can be accessed via a further frame property. Each goal frame points to a text frame, which can in turn be used to access the associated clause.

We can retrieve the clause source file and line number from a clause reference via the system predicate clause_property/2. The modified call-back then reads as follows:

```
% goal_tracing(+Port, +Frame)
goal_tracing(P, Q) :-
      frame_property(Q, sys_call_goal(G)),
      functor(G, F, A),
      atom_property(F, source_file(O)),
      atom_property(F, line_no(L)), !,
      get_delta(P, D),
      update_count_predicate(F, A, D),
      update_count_source(F, A, O, L, D).
```

In the above code we update two counters, namely the predicate level and the call site level counters. The predicate level counter will thus show the sum of the call site level counters.

We could have opted for another design, where the sums are determined during the reporting. The call-back would thus have less work to do and therefore run faster. But for brevity we kept the solution simple. More details on the implementation can be found in the appendix.

If we run our Peano factorial program again we will get a more detailed port statistics. We get the following results:

```
?- show.
Pred      Source    Line      Call      Exit      Redo      Fail
add / 3                       5941      5941      0         0
          fac.p     9         5913      5913      0         0
          fac.p     12        28        28        0         0
mul / 3                       35        35        0         0
          fac.p     12        28        28        0         0
          fac.p     15        7         7         0         0
fac / 2                       8         8         0         0
          fac.p     15        7         7         0         0
                    3         1         1         0         0
```

The port statistics has multiple application areas. It can be used to empirically investigate into the complexity of a Prolog program. Or it can be used to determine the coverage of test cases. For the later purpose one simple runs the test cases with port statistics and can then check what predicates have been touched.

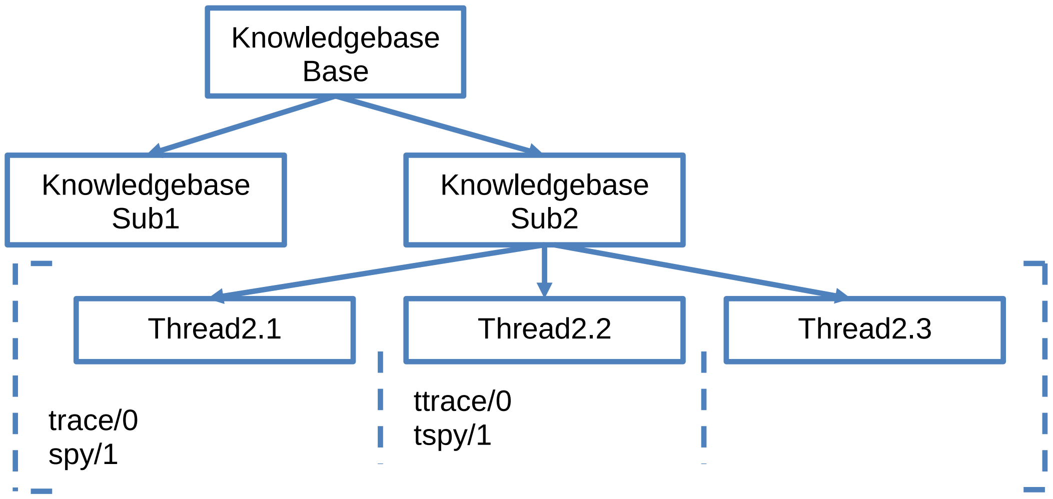
3 Environment Conversations

The Jekejeke Prolog development environment provides terminal based interactions. Among the interactions we find goal debugging.

- **Debugger Control:** General introduction to the debugger control.
- **Debugger Ports:** General introduction to the debugger ports.
- **Debugger Callback:** General introduction to the debugger callback.
- **Default Prompt:** The development environment supports execution debugging based on Byrd's box model.
- **Interrupt Handling:** The console allows manually interrupting the interpreter loop either during read or during execution.
- **Compatibility Matrix:** We compare our approach with the former DEC10 standard and the current ISO core standard.

3.1 Debugger Control

The debugger directives will modify or access the debugger control state. The debugger control state consists of flags, spy points and break points. Debugger control state exists on the level of knowledge bases and on the level of Prolog engines. The knowledge base control state applies to the thread group of the knowledge base:



Picture 1: Debugger Control Levels

The flags state affects the debug mode and the visible tracing. Together with the spy points and break points, the Prolog interpreter determines the reached ports. For this purpose, the Prolog interpreter uses instrumented clause forks of the predicates. For reached ports, the Prolog interpreter calls the current debugger callback.

Whether any port of a predicate is reachable further depends on its call-site:

- **sys_notrace:** Prolog text flag to disable port reaching.

The knowledge base level control state of the debugger can be modified and accessed by the debugger directives from the module "default". This module is preloaded and therefore available without any further import.

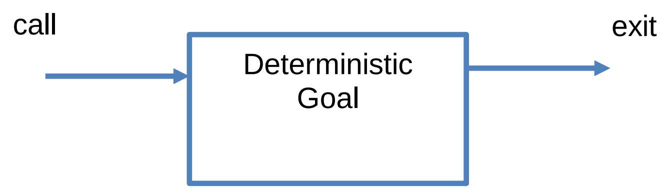
- **debug/0, trace/0, skip/0, out/0, nodebug/0:** Modify the debug mode.
- **visible/1:** Modify the visible tracing.
- **debugging/0:** Display the control state.
- **spy/1, nospy/1, spying/1:** Modify and access the spy points.
- **break/2, nobreak/2, breaking/2:** Modify and access the break points.

The Prolog engine level control state of the debugger can be modified and accessed by the debugger directives from the module "attach". This module is not preloaded and needs therefore an import before it can be used:

- **tdebug/0, ttrace/0, tskip/0, tout/0, tnodebug/0:** Modify the engine debug mode.
- **tvisible/1:** Modify the engine visible tracing.
- **tdebugging/0:** Display the engine control state.
- **tspy/1, tnospy/1, tspying/1:** Modify and access the engine spy points.
- **tbreak/2, tnobreak/2, tbreaking/2:** Modify and access the engine break points.

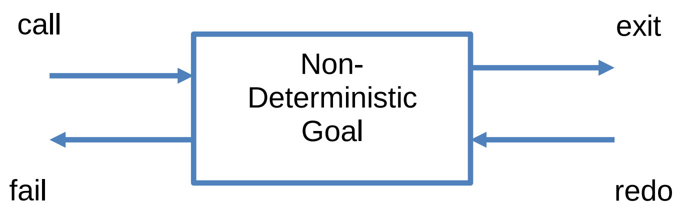
3.2 Debugger Ports

In the case of a deterministic goal, the debugger automatically reduces the number ports to "call" and "exit". This reduces the clutter during manual debugging and helps the Prolog to retain a deterministic execution of deterministic predicates. The "call" port happens when a goal is invoked for the first time, and the "exit" port happens when a goal succeeds:



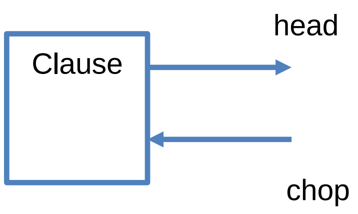
Picture 2: Deterministic Goal Ports

In case of a non-deterministic goal, the debugger automatically uses the full set of ports from the Byrd Box model. Non-deterministic goals are goals that succeed with leaving at least one choice point. In the case of a non-deterministic goal, the debugger will see to it that it will also provide the ports "redo" and "fail" to the port callback:



Picture 3: Non-Deterministic Goal Ports

It can happen that the choice points of a non-deterministic goal are later removed by a cut or exception in the continuation. The debugger will then abandon the "redo" port and "fail" port. Many Prolog systems provide further ports than only the ports from the Byrd Box model. We do as well and provide further ports to monitor clause unification:



Picture 4: Clause Ports Extension

For monitoring clause unification, we only use a half box so there is no monitoring when clause unification starts. The "head" port is called after a clause unification was successfully completed and before the attribute variable unify hooks are called. The "chop" port is called before the clause unification is undone again.

3.3 Debugger Callback

When the port callback is called a frame reference and a port name is handed over. The frame reference is an entry point to the call stack of the currently called goal. From there the currently called goal and further information can be accessed. By using a different port callback, it is possible to provide additional tooling.

The port callback is customizable via the multi-file predicate goal_tracing/2:

- **Default Callback:** The default callback leads to the default debugger prompt. This callback is defined in the module "default".
- **Profiling Callback:** The callback can be used to gather profiling information. See also our programming example for port statistics.
- **Coverage Callback:** The callback can be used to gather coverage information. See also our tooling from the module "tracker" and "cover".

Depending on the call-back further control state might determine the behaviour of the callback. This control state might be again scattered along the knowledge base and the Prolog engine and have different scopes. In case of the default debugger user interface the control state embraces the following settings.

Whether a predicate is visible to the default prompt depends on itself:

- **sys_notrace:** Predicate flag to disable the default prompt.

The following knowledge base level control state of the default prompt is available:

- **leash/1:** The leashing mode for the default prompt.

The following Prolog engine level control state of the default prompt is available:

- **tleash/1:** The engine leashing mode for the default prompt.

3.4 Default Prompt

Without debugging the interpreter will only interact with the end-user when input/output predicates for the console are encountered, or when a solution has been obtained. Otherwise during normal query execution the interpreter remains silent. The example predicate is taken from the naïve reverse Prolog text found in our benchmark suite:

```
?- concatenate(X, Y, [1,2,3]).
X = []
Y = [1, 2, 3] ;
X = [1]
Y = [2, 3]
```

Different debugging modes are available. Depending on the debugging mode the interpreter will run until a certain condition is met. When such a condition is met the debugger will show the met condition, the invocation depth, the current port and the current goal. Here is an example of trace mode with full leashing:

```
?- trace.
Yes
?- concatenate(X, Y, [1,2,3]).
   0 Call concatenate(X, Y, [1, 2, 3]) ?
   0 Exit concatenate([], [1, 2, 3], [1, 2, 3]) ?
X = []
Y = [1, 2, 3] ;
   0 Redo concatenate([], [1, 2, 3], [1, 2, 3]) ?
   1 Call concatenate(_G, Y, [2, 3]) ?
   1 Exit concatenate([], [2, 3], [2, 3]) ?
   0 Exit concatenate([1], [2, 3], [1, 2, 3]) ?
X = [1]
Y = [2, 3]
```

The available debugging and leashing modes are found in the documentation of the modules "default" and "attach". The predicates to add and remove spy points respectively break points are documented as well there. The end-user can interact with the debugger when the prompt (?) is shown. The end-user can then choose upon the following options.

```
EOF       = Exit the current session.
          = Continue running the program in the actual mode.
?         = Display this help text.\n\
<Goal>.   = Execute the <Goal> and prompt command again.
```

If the end-user chooses a goal, this goal will be executed and the end-user will be prompted again. The side effect of the goal can control the debugger state or the goal can be used to query the interpreter state. Besides that the end-user can also use session predicates such as abort/0, break/0, etc.. or even predicates he has defined on his own.

The debugger can be also controlled by the end-user through the menu, button and tool bar actions of the graphic environment. These actions will prepare a debugger directive and send it to the Prolog interpreter. The debugger directives will be visible in the console. The actions are documented in the Swing respectively Android documentation.

3.5 Interrupt Handling

The native console allows manually interrupting the interpreter loop either during read or during execution. The interrupt key (^C on Mac, Linux and Windows) will cause the invocation of an interrupt handler which will affect the interpreter.

For the development environment we decided to assign the pause function to the interrupt key. When the interrupt key is pressed the interrupt handler will change the debugging mode to step in and leash all ports. If the debugging mode was not off and if the program contains traceable predicates, this will cause a debugger prompt with user interaction.

```
>
Jekejeke Prolog, Development Environment 1.0.7
(c) 1985-2015, XLOG Technologies GmbH, Switzerland
?- debug.
Yes
?- repeat, fail.
^C
    0 Exit repeat ?
```

In the above example the interpreter hangs in an infinite computation from the repeat fail query. The interrupt key allows pausing this infinite computation.

3.6 Compatibility Matrix

We compare our approach with the former DEC10 standard and the current ISO core standard. The following compatibility issues persist for the interactions:

Table 1: Compatibility Matrix for Interactions

Nr	*Description*	*System*
1	Trace message has also invocation identifier.	DEC10
2	Trace message prompt dependent on leashing	DEC10
3	Trace message uses customizable print.	DEC10
4	Trace message indicates return from skip.	DEC10
5	Has retry, fail and redo debugger commands.	DEC10
6	Has quasi-skip which does not suspend spy points.	DEC10
7	Has back to choice point.	DEC10
8	Has print and display of the current goal.	DEC10
9	Has user goal execution at debugging prompt.	DEC10
10	Debugging is not part of the standard.	ISO

4 Environment Syntax

In this section we show what syntax the Jekejeke Prolog development environment accepts:

- **Miscellaneous Definitions:** The interpreter keeps track of flags and properties.

4.1 Miscellaneous Definitions

The interpreter also needs to keep track of flags and properties definitions. The following flags and properties are provided by the Jekejeke Prolog development environment:

- **Prolog Flags:** The predefined Prolog flags.
- **Thread Flags:** The predefined thread flags.
- **Predicate Properties:** The predefined predicate properties.
- **Source Properties:** The predefined source properties.
- **Callable Properties:** The predefined callable properties.
- **Frame Properties:** The predefined frame properties.

Prolog Flags

Prolog flags can be accessed via the system predicates current_prolog_flag/2 and set_prolog_flag/2. The following Prolog flags are supported by the Jekejeke Prolog development environment:

unknown:	See the default debugger section.
debug:	See the default debugger section.
sys_leash:	See the default debugger section.
sys_visible:	See the default debugger section.
sys_max_stack:	See the default debugger section.
sys_skip_frame:	See the default debugger section.
sys_query_frame:	See the stack properties section.
sys_cloak:	See the mode handling section.
sys_clause_instrument:	See the mode handling section.
sys_head_wakeup:	See the mode handling section.
sys_tleash:	See the module attach section.
sys_tvisible:	See the module attach section.
sys_monitor_config:	See the module monitor section.
sys_monitor_running:	See the module monitor section.
sys_monitor_logging:	See the module monitor section.

Thread Flags

Thread flags can be accessed via the system predicates current_thread_flag/2 and set_thread_flag/2. The following thread flags are supported by the Jekejeke Prolog development environment:

sys_tdebug:	See the module attach section.
sys_top_frame:	See the module attach section.
sys_thread_store:	See the module attach section.

Predicate Properties

Predicate properties can be accessed via the system predicates predicate_property/2, set_predicate_property/2 and reset_predicate_property/2. The following predicate properties are supported by the Jekejeke Prolog development environment:

sys_noinstrument:	See the mode handling section.
sys_nowakeup:	See the mode handling section.

Source Properties

Source properties can be accessed via the system predicates source_property/2, set_source_property/2 and reset_source_property/2. The following source properties are supported by the Jekejeke Prolog development environment:

sys_first_location:	See the module base section.
sys_location:	See the module base section.

Callable Properties

Callable properties can be accessed via the system predicates callable_property/2 and updated copies of callables can be obtained via the predicates set_callable_property/3 and reset_callable_property/3. The following callable properties are supported by the Jekejeke Prolog development environment:

source_file:	See the module provable section.
line_no:	See the module provable section
sys_context:	See the module provable section
sys_hint:	See the module provable section.
sys_fillers:	See the module provable section.
sys_raw_variables:	See the module stack section.
sys_variable_names:	See the module stack section.

Frame Properties

Frame properties can be accessed via the system predicates frame_property/2 and. The following frame properties are supported by the Jekejeke Prolog development environment:

sys_parent_frame:	See the module stack section.
sys_call_goal:	See the module stack section.

5 Development Packages

The Jekejeke Prolog development environment provides a set of system predicates to control the debugger and inspect the interpreter. The default debugger user interface only assumes a terminal:

- **Debug Package:** This package provides debugging of predicates.
- **Inspection Package:** This package provides inspection of the interpreter.
- **Testing Package:** This package provides unit testing of predicates.
- **System Package:** This theory is concerned with extending the Prolog system.
- **Notebook Package:** This theory is concerned with document stores.
- **Wire Package:** This theory is concerned with wiring a debugger.

5.1 Debug Package

This package provides debugging of predicates. It consists of the following modules:

- **Debug Mode:** Predicates can be executed in trace or debug mode.
- **Debug Plugins:** A callback can replace the default debugger prompt.
- **Module friendly:** Predicates to list the intermediate form.
- **Module dump:** Predicates to dump the clause index.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of the development theory.

Debug Mode

We can distinguish a couple of debugging modes. In any debugging mode the instrumented fork of a clause will be executed. Depending on the debugging mode the instrumentation will check a certain condition and when this condition is met the current execution is suspended and a callback is called. The debugging modes are:

Table 2: Debugging Modes

Mode	*Condition*
off	Run until paused or aborted
step in	Run until next port
step over	Run until predicate is executed
step out	Run until parent predicate is executed
on	Run until next spy or break point

When the current goal is shown additional information about the debugging mode, the invocation depth and the current port is shown as well. Last call optimization is currently defunct while debugging, the invocation depth is therefore larger than usual. The information is provided in the following format on the display console:

```
<mode><depth><port> <goal> ?

-: The debug mode is off
 : The debug mode is step in
=: The debug mode is step over
>: The debug mode is step out
*: The debug mode is on
```

Predicates that are invoked from within a source which has the sys_notrace source property set are as well ignored in any debugging mode. The system sources have the sys_notrace source property by default set.

The following debug mode predicates are provided:

debug:
The predicate switches to the on mode.

trace:
The predicate switches to the step in mode.

skip:
The predicate switches to the step over mode.

out:
The predicate switches to the step out mode.

nodebug:
The predicate switches to the off mode.

visible(L):
Show the ports that are listed in L, hide the ports that are not listed in L. In debug mode, hidden ports are not further debugged but simply continue. The following mnemonics work for the predicate:

off:	Never prompt.
loose:	Only prompt on call port.
half:	Only prompt on call and redo port.
tight:	Only prompt on call, redo and fail port.
full:	Prompt on call, exit, redo and fail port.
all:	Prompt on call, exit, redo, fail and head port.

debugging:
The predicate shows the debug mode, the spy points and the break points.

spy(P):
The predicate adds the predicate P to the spy points.

nospy(P):
The predicate removes the predicate P from the spy points.

spying(P):
The predicate succeeds in P for every spy point.

break(F, L):
The predicate adds the file F and the line number L to the break points.

nobreak(F, L):
The predicate removes the file F and the line number L from the break points.

breaking(F, L):
For every break point the predicate succeeds with the file F and the line number L.

The following debug mode Prolog flags are provided:

unknown: [ISO 7.11.2.4]
Legal values are error [ISO], fail [ISO] and warning [ISO]. The flag indicates how undefined predicates should be executed. Default value is error. Currently the value cannot be changed.

debug: [ISO 7.11.2.2]
The legal values are on [ISO], step_in, step_over, step_out and off [ISO]. The value indicates the debugging mode. Default value is off. The value can be changed.

sys_visible:
Legal values are lists of debugger ports. Legal debugger ports are call, exit, redo, fail, head and chop. The value indicates the tracing visibility. The default value is [call,exit,redo,fail]. The value can be changed.

sys_max_stack:
The property indicates the size of the stack trace that should be generated for an exception. Legal values are non-negative integers. The value can be changed.

sys_skip_frame:
The legal values are stack frames or the atom null. The value indicates the frame to skip to. Default value is null. The value can be changed.

Debug Plugins

A call back can replace the default debugger user interface. The instrumented code of a clause automatically checks for debugging conditions. The typical conditions are those from debug mode, spy points and break points described in the previous section. When the corresponding condition is met, the interpreter will call the predicate trace_goal/2.

The system predicate trace_goal/2 is customizable by the end-user via additional rules for the multi-file predicate goal_tracing/2. If the additional multi-file rules fail, the system predicate will invoke the default debugger user interface. The behaviour of the default debugger user interface can be further configured by the predicate leash/1.

Predicates that have the sys_notrace predicate property set can also meet some condition, but are not display by the default debugger user interface. A couple of predicates that resemble commands, e.g. listing/1, trace/1, etc., have the sys_notrace predicate property by set, so that they do not clutter interactive debugging.

The following debug plugins predicates are provided:

goal_tracing(P, F):
The predicate can be used to define a custom debugger call back for the port P and the frame F.

trace_goal(P, F):
The predicate invokes the current debugger call back for the port P and the frame F. If no call back is defined then the current goal is traced and optionally prompted.

store_changing(S):
The predicate can be used to define a custom debugger call back for the store S.

change_store(S):
The predicate invokes the current debugger call back for the store S. If no call back is defined then does nothing.

leash(L):
Leash the ports that are listed in L, unleash the ports that are not listed in L. When prompted unleashed ports do not await user interaction but simply continue. The predicate accepts the same mnemonics as the predicate visible/1.

The following debug plugins Prolog flags are provided:

sys_leash:
Legal values are lists of debugger ports. Legal debugger ports are call, exit, redo, fail, head and chop. The value indicates the prompt leashing. The default value is [call,exit,redo,fail]. The value can be changed.

Module friendly

Predicates are brought into intermediate form before execution. The intermediate form determines how the head of a clause is unified and how the goals of the body of a clause are invoked. A thread working on a clause will execute one block of code at a time. We identify these blocks as instructions and show them in linear form.

When the intermediate form of a clause is listed, first the Prolog text representation of the clause is given. Our instruction set then only demands two forms of operands, goal arguments and clause skeletons. The clause skeleton operands will be displayed by respecting the operator definitions. For readability, instructions are numbered:

```
instruction        --> integer name [ operand { "," operand } ]
operand            --> "_" integer
                       | term.
```

We do not find any branching instructions in our instruction set. Therefore, instructions need not be prefix by a label and labels cannot appear in the operands of an instruction. Labels are also not needed to invoke a goal. The called predicate is simply identified by the functor and arity of the goal. Further, we do not find any arithmetic or bit operation.

The arithmetic or bit operation are handled by invoking the corresponding built-ins. Most instructions are executed multiple times since choice points might succeed again and thus continuations might be re-executed. Some global or local code specialization [8] [9] would need to be done by some external pre-processing.

Here is a simple example of a clause and its intermediate form:

```
?- friendly(hello/1).
hello(X) :-
   write('Hello '),
   write(X), nl.
    0 call_goal write('Hello ')
    1 new_bind X
    2 unify_var _0, X
    3 last_goal write(X)
    4 dispose_bind X
    5 last_goal nl
    6 call_cont
```

Our instruction set is not derived from the WAM architecture [5] since terms are represented by a display and a skeleton. Therefore during unification in write mode we do not need to allocate compounds or lists. Instead our space effort is bound by the number of variable place holders that need to be created.

The optimization we implemented therefore tend to reduce the number of place holder allocations or to provide the Java virtual machine an opportunity to reuse place holders. The local optimizations are not based on n-grams [6]. Instead, we do a variable range analysis with far reaching code movements for some forms of argument unification.

The following instructions are part of the intermediate form:

new_bind V_1, .., V_n:
Create place holders for the variable skeletons V_1, .., V_n.
dispose_bind V_1, .., V_n:
If deterministic dereference the place holders for the variable skeletons V_1, .., V_n.
unify_term A, T:
Unify the argument A with the skeleton T.
unify_term A, B:
Unify the argument A with the argument B.
unify_var A, V:
Unify the argument A with the variable skeleton V.
call_goal T:
Invoke the skeleton T.
call_meta V:
Validate and invoke the variable skeleton V.
last_goal T:
Check for leapfrogging the parent frame. Invoke the skeleton T.
last_meta V:
Check for leapfrogging the parent frame. Validate and invoke the variable skeleton V.
call_cont:
Invoke the continuation.

The following intermediate form predicates are provided:

friendly:
The predicate lists the intermediate form of the clauses of the user predicates.
friendly(P):
The predicate lists the intermediate form of the clauses of the user predicate P.
instrumented:
Works like the predicate friendly/0 except that the debugger instrumented variant of the clause is shown.
instrumented(P):
Works like the predicate friendly/1 except that the debugger instrumented variant of the clause is shown.

Module dump

The shape of the clause index depends on the call pattern history of the predicate. We do not provide a programming interface to selectively inspect the clause index. Instead the end-user can dump the clause index for predicates in one go.

The detected call patterns can be read off from the detected argument positions. The clause index need not follow a simple collection of call patterns. Sub-indexes can have individual call patterns. Let's give a simple example:

```
?- [user].
p(7, a).
p(7, b).
p(9, c).
^D
?- p(7, a).
Yes
```

The query will deterministically succeed. This is an indicative that a clause index has been built that covers multiple arguments. Clause indexing based on first argument indexing only would not be able to detect this determinism. Although the clause index is multi argument, it does so only for the key "7":

```
?- dump(p/2).
-------- p/2 ---------
length=3
at=0
  key=7, length=2
    at=1
      key=a, length=1
      key=b, length=1
  key=9, length=1
```

Since release 1.2.5 of the Prolog runtime different data structures are used depending on a low and a high water mark. For small indexes a simple key-value pair list is used and no hash is computed. For large indexes a hash table is used.

The following index attributes are shown during a clause index dump:

length=<len>:	Gives the size of indexed clause set.
arg=<pos>:	Gives the argument position that is indexed.
map=<size>:	Gives the hash table size of the argument position.
<key>=:	Gives the key and corresponding sub index.
hash=<index>:	Gives the hash code module hash table size of the key.
nonguard:	Gives the non-guard hash table miss fallback index.
guard:	Gives the guard hash table miss fallback index.

The following clause index predicates are provided:

dump:
The predicate dumps the clause index of the clauses of the user predicates.

dump(P):
The predicate dumps the clause index of the clauses of the user predicate P.

Compatibility Matrix

The following compatibility issues persist for the development theory:

Table 3: Compatibility Matrix for the Development Theory

Nr	*Description*	*System*
1	Has leash predicate.	DEC10
2	Has operator definition for spy and nospy.	DEC10
3	Has maxdepth predicate.	DEC10
4	Has prompt/2 und version/1 predicate.	DEC10
5	Does not define a default debugger.	ISO
6	Does not define customizable debuggers.	ISO
7	Does not define an intermediate form.	ISO

5.2 Inspection Package

This package provides inspection of the interpreter. It consists of the following modules:

- **Module notation:** Predicates for Prolog module notations.
- **Module frame:** Predicates to access frame properties.
- **Module provable:** Predicates to access predicates and evaluable functions.
- **Module syntax:** Predicates to access syntax operators.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of the inspection theory.

Module notation

For debugging purpose it might be necessary to have direct access to our Prolog module notation. Our Prolog module notations are ordinary Prolog terms that are converted to interpreter objects. The slash notation combines a package name and a module name into a structured module name. The colon notation separates combines module name and a predicate name into a qualified predicate name.

Examples:

```
?- sys_atom_slash(X, foo/bar).
X = 'user$foo$bar'
?- sys_atom_slash(X, basic/lists).
X = 'jekpro.frequent.basic.lists'
```

The predicate sys_atom_slash/2 can be used to explicitly invoke the slash (/)/2 compound notation conversion, yielding a period (.) respectively dollar ($) characters in the resulting atom. The notation can be used to denote Prolog text modules and Java Classes. We additionally support for the {}/1 compound notation conversion as well, which can be used to denote Java Array Classes and yielding ([]) characters.

Examples:

```
?- sys_callable_colon(X, basic/lists:member(A,B)).
X = 'jekpro.frequent.basic.lists\bmember'(A,B)
?- sys_indicator_colon(X, basic/lists:member/2).
X = 'jekpro.frequent.basic.lists\bmember'/2
```

The predicates sys_callable_colon/2 can be used to explicitly invoke the colon (:)/2 and double colon (::)/2 notation conversion for a callable. The double colon notation combines the receiver module name by additionally prepending the receiver itself to the callable similar to the Python dynamic invocation. The predicate sys_indicator_colon/2 can be used to explicitly invoke to colon notation conversion for a predicate indicator.

The following base predicates are provided:

sys_atom_slash(S, T):
The predicate succeeds when S is a callable of the form ‘s_1. .. .s_n’ and T is a slash notation atom of the form s_1/../s_n, for 1 ≤ n. Besides the (/)/2 operator the {}/1 operator is supported as well.

sys_callable_colon(S, T):
The predicate succeeds when S is a callable of the form ‘p_{k-1}%p_k’(X_1, .., X_m) and T is a colon notation callable of the form p_1:..:p_k(X_1, .., X_m), for 1 ≤ k and 0 ≤ m.

sys_indicator_colon(S, T):
The predicate succeeds when S is an indicator of the form ‘p_{k-1}%p_k’/m and T is a colon notation indicator of the form p_1:..:p_k/m, for 1 ≤ k and 0 ≤ m.

Module frame

Since recently we have introduce hierarchical knowledge bases. They are already used in the Swing GUI, but not in the Android GUI. Every Swing console window runs in its own sub knowledge base which provides a separate class loader. The current knowledge base stack can be listed by the store/0 command:

Example, in Swing GUI:

```
?- stores.
Store-1
Store-0
```

Example, in Android GUI:

```
?- stores.
Store-0
```

Knowledge base properties can be query by the predicate store_property/2. The predicates set_store_property/2 and reset_store_property/2 serve updating knowledge base properties. This module also provides accessing thread stack frames via the predicate frame_property/2.

The following predicates for frame properties are provided:

rule_frame(H, B, F):
: The predicate succeeds with the user clauses that match H :- B. The predicate also unifies F with the new frame reference for the found clauses.

frame_property(F, P):
: The predicate succeeds for the properties P of the clause referenced by F.

store_property(F, P):
: The predicate succeeds for the properties P of the store F.

set_store_property(S, Q):
: The predicate assigns the property Q to the store S.

reset_store_property(S, Q):
: The predicate de-assigns the property Q from the store S.

stores:
: The predicate lists the store chain of the current thread.

The following frame properties for stack properties are provided:

sys_parent_frame(F):
: The property indicates that the stack frame has parent frame F. The property is single valued or can be missing. The property cannot be changed.

sys_call_goal(G):
: The property indicates that the stack frame has call goal G. The property is single valued or can be missing. The property cannot be changed.

The following callable properties for stack properties are provided:

sys_raw_variables(N):
The property indicates in N the raw variables of the clause associated with the callable. The property cannot be changed,

sys_variable_names(N):
The property indicates in N the variable names of the clause associated via the callable. The property cannot be changed.

Module provable

For debugging purposes it might be necessary to access predicates and evaluable functions that are not accessible from the top-level by the module system visibility rules. We provide predicates that allow direct access. The access is call-site independent, requires structured module names with package prefixes resolved and the module already loaded.

Examples:

```
?- current_predicate(basic/lists:member2/3).
No
?- current_provable(basic/lists:member2/3).
Yes
```

The directly accessible predicates can be tested and enumerated by the predicate current_provable/1. The predicates provable_property/2, set_provable_property/2 and reset_provable_property/2 are responsible for accessing and modifying properties of directly accessible predicates.

Example:

```
?- length(X, 0), callable_property(X, Y), writeq(Y), nl, fail; true.
sys_context('<path>/lists.px')
source_file('<path>/lists.px')
line_no(136)
Yes
```

Context and pretty printing information of an atom can be accessed and modified by the predicates callable_property/2, set_callable_property/3 and reset_callable_property/3. The predicates defined here generalize the predicates sys_get_context/2 and sys_replace_site/3 from the Jekejeke Prolog runtime.

The following provable access predicates are supported:

current_provable(P):
The predicate succeeds for the directly accessible predicates P.

provable_property(I, Q):
The predicate succeeds for the properties Q of the predicate I. The predicate will also try to access invisible predicates.

set_provable_property(I, Q):
The predicate assigns the property Q to the predicate I. The predicate will also try to access invisible predicates.

reset_provable_property(I, Q):
The predicate de-assigns the property Q from the predicate I. The predicate will also try to access invisible predicates.

callable_property(A, Q):
The predicate succeeds for the properties Q of the callable A.

set_callable_property(B, Q, A):
The predicate succeeds for a new callable B which is a clone of the callable A except for the property Q which is now set.

reset_callable_property(B, Q, A):
The predicate succeeds for a new atom B which is a clone of the callable A except for the property Q which is now reset.

The following callable properties are provided:

source_file(O):
The property indicates that the callable has the file origin O associated. The property is single valued and can be missing. The property cannot be changed.

line_no(L):
The property indicates that the callable has the line number L. The property is single valued and can be missing. The property cannot be changed.

sys_context(S):
The property indicates that this callable has the module source S associated. The property is single valued and might be missing.

sys_hint(H):
The property indicates that this callable has the hint H associated. The property is single valued and might be missing.

sys_fillers(F):
The property indicates that this callable has the fillers F associated. The property is multi valued and might be missing.

Module syntax

For debugging purposes it might be necessary to access syntax operators that are not accessible from the top-level by the module system visibility rules. We provide predicates that allow direct access.

The directly accessible syntax operators can be tested and enumerated by the predicate current_syntax/1. The predicates syntax_property/2, set_syntax_property/2 and reset_syntax_property/2 are responsible are responsible for accessing and modifying properties of directly accessible syntax operators.

The following syntax access predicates are supported:

current_syntax(P):
The predicate succeeds for each directly accessible syntax operator P.

syntax_property(O, Q):
The predicate succeeds for the properties Q of the syntax operator O. The predicate will also try to access invisible syntax operators.

set_syntax_property(O, Q):
The predicate assigns the property Q to the syntax operator O. The predicate will also try to access invisible syntax operators.

reset_syntax_property(O, Q):
The predicate de-assigns the property Q from the syntax operator O. The predicate will also try to access invisible syntax operators.

The following source properties for the module syntax are provided:

sys_first_location(I, O, L):
The property indicates that the source context has first predicates occurence for the indicator I, file origin O and line number L. The property is multi valued or can be missing. The property cannot directly be changed.

sys_location(I, O, L):
Same as sys_first_location/3 except that not only the first occurrence for each predicate is listed but that all occurences are listed.

Compatibility Matrix

The following compatibility issues persist for the inspection theory:

Table 4: Compatibility Matrix for the Inspection Theory

Nr	*Description*	*System*
1	Does not define assert options.	ISO
2	Does not define clause properties.	ISO
3	Does not define frame properties.	ISO
4	Does not define stack frame properties.	ISO
5	Does not define an index dump.	ISO

5.3 Testing Package

This package provides unit testing of predicate. It consists of the following modules:

- **Module summary:** This module allows the batch reporting of a summary.
- **Module runner:** This module allows executing test cases.
- **Module diagnose:** This module allows the online display of test results.
- **Module result:** This module allows the batch reporting of test results.
- **Module tracker:** This module allows tracking coverage analysis.
- **Module cover:** This module allows the batch reporting of coverage analysis.

Module summary

This module allows the batch reporting of a summary. The report shows the suite summary of the test results and the package summary of the coverage analysis. Beforehand the module runner needs to be used to produce the test results and the module tracker needs to be used to produce the coverage analysis.

The following summary predicates are provided:

summary_batch:
The predicate generates a file into the location pointed by the base_url Prolog flag.

Module runner

This module allows executing test cases. The test runner executes the test cases and summarizes the results. The test cases and the results are certain facts in the knowledge base. The test runner can be invoked via the predicate runner_batch/0.The coverage depends on how well the test cases are designed in respect of the probed execution paths.

The test cases are stored by as facts and rules in the following form:

```
:- multifile(test_ref/4).
:- discontiguous(test_ref/4).
% test_ref(Fun, Arity, Suite, Ref).

:- multifile(test_case/4).
:- discontiguous(test_case/4).
% test_case(Fun, Arity, Suite, Number) :- Body.
```

Note: Fun/Arity is used to denote predicates, and Fun/-Arity-1 is used to denote evaluable functions. Note: Fun/Arity is used to denote predicates, and Fun/-Arity-1 is used to denote evaluable functions. The test steps and the test validation points need to be implemented in the body of the predicate test_case/4. The body is assumed to terminate, the test runner doesn't impose some timeout currently. The body can be used to check a multitude of scenarios:

Examples:

```
% Check whether the goal succeeds:
    Goal
% Check whether the determinitic goal succeeds with result Value:
    Goal, Var==Value
% Check whether the non-determinitic goal first succeeds with result Value:
    Goal, !, Var==Value
% Check whether the goal fails:
    \+ Goal:
% Check whether the goal succeeds on redo:
    findall([],Goal,[_,_|_])
% Check whether the goal succeeds on redo with result Value:
    findall(Var,Goal,[_,Var|_]), Var==Value
% Check whether the goal fails on redo:
    findall([],Goal,[_])
% Check whether the goal throws the exception Value:
    catch(Goal,error(Var,_),true), Var==Value.
```

The testing is not limited to the above example scenarios. A particular application domain might need additional test helper predicates to express the desired test steps and test validation points. Examples are the CLP(FD) test cases which use the predicate call_residue/2. The test results are ok and not-ok counts, a non-conclusive count is currently not provided.

The results are stored by the following facts:

```
:- public result_summary/1.
:- dynamic result_summary/1.
% result_summary(OkNok).

:- public result_suite/2.
:- dynamic result_suite/2.
% result_suite(Suite, OkNok).

:- public result_predicate/4.
:- dynamic result_predicate/4.
% result_predicate(Fun, Arity, Suite, OkNok).

:- public result/5.
:- dynamic result/5.
% result(Fun, Arity, Suite, Number, OkNok).
```

The results are accessible by the diagnose module or the report module from the Jekejeke Prolog development environment. Or the tester might code its own analysis based on these facts. In the future we might as well provide additional tools, such as a coverage analysis tool or similar.

The following runner predicates are provided:

runner_batch:
The predicate executes the test cases, collects and summarizes the results.

Module diagnose

This module allows the online display of test results. Beforehand the module runner needs to be used to produce the test results. The predicate diagnose_online/0 will then first present a listing of the theories and their summarized test case success and failure count. The end-user can then choose a predicate and the summarized results will be showed there. Finally the end-user can inspect an individual test case.

The following diagnose predicates are provided:

diagnose_online:
The predicate starts an online drill down of the test results.

Module result

This module allows the batch reporting of test results. Beforehand the module runner needs to be used to produce the test results. The predicate result_batch/1 can then be used to generate a number of files that list and summarize the results in HTML format. The reporting tool makes an additional assumption about the suite names:

```
suite --> package "_" module.
```

The first level HTML page will thus present the results grouped by packages. The second level HTML page will thus present the results of a package grouped by modules. The current implementation shows success and failure counts not only as numbers but also as coloured bars. Furthermore links to the original test cases will be generated.

The following result predicates are provided:

result_batch(R):
The predicate generates a number of files into the location pointed by the base_url Prolog flag. Links to the test cases are generated relative to the argument R.

Module tracker

This module allows executing test cases and analysing the coverage of the tested code. The test cases are the same as for the module runner. But contrary to the module runner test results are not collected by this module. Instead this module installs a debugger hook and collects a coverage map.

The coverage map is stored by the following facts:

```
:- public cover_summary/1.
:- dynamic cover_summary/1.
% cover_summary(OkNok)

:- public cover_source/2.
:- dynamic cover_source/2.
% cover_source(Source, OkNok)

:- public cover_predicate/4.
:- dynamic cover_predicate/4.
% cover_predicate(Fun, Arity, Source, OkNok)

:- public cover/5.
:- dynamic cover/5.
% cover(Fun, Arity, Source, Line, OkNok)
```

The debugger hook slows down the execution of test cases by a factor of 3-4. The collection is done in two phases. First the predicate tracker_batch/0 has to be called. Then the predicate analyze_batch/0 has to be called. The later predicate needs text/1 facts that designate the sources that should appear in the coverage map.

The following tracker predicates are provided:

tracker_batch:
Run the test cases and collect the raw coverage map.

analyze_batch:
Relate the raw coverage map with the sources given as text/1 facts.

Module cover

This module allows the batch reporting of coverage analysis. Beforehand the module tracker needs to be used to produce the coverage analysis. The predicate cover_batch/1 can then be used to generate a number of files that list and summarize the results in HTML format. The reporting tool makes an additional assumption about the source names:

```
source --> package "/" module.
```

The first level HTML page will thus present the analysis grouped by packages. The second level HTML page will thus present the analysis of a package grouped by modules. The current implementation shows hit and miss counts not only as numbers but also as coloured bars. Furthermore links to the original source clauses will be generated.

The following cover predicates are provided:

cover_batch(R):
The predicate generates a number of files into the location pointed by the base_url Prolog flag. Links to the source code are generated relative to the argument R.

5.4 System Package

This theory is concerned with extending the Prolog system. The Prolog system can be extended by defining call backs.

- **Module protocol:** This module allows a transcript of a session.
- **Mode Handling:** A secondary thread can control a primary thread.
- **Module automatic:** The Java auto generated members can be listed.
- **Module memory:** This module provides non-random access memory streams.
- **Module charsio:** This module provides temporary input/output redirection.
- **Module attach:** This module provides engine debugger attachment.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of the system theory.

Module protocol

A transcript of the current session can be written to a file. The transcript captures what is read from the console and written to the console.

Example:

```
?- protocol('session.log').
Yes
... do something ...

?- noprotocol.
Yes
```

The transcript captures only the console input/output of the thread that is attached to the console where the command has been invoked. Other threads will not be visible as long as they don't input/output to this console as well.

The following protocol predicates are provided:

protocol(F):
Start transcript of the current session to the file F. If the file already exists the transcript is appended to the existing file. Otherwise a new file is created.

noprotocol:
Stop transcript of the current session. The current transcript file is closed.

Mode Handling

The Prolog interpreter activates the goals of the debug clause fork in case some debug mode is set. The Prolog flag sys_cloak allows temporarily disabling any debug mode for a Prolog interpreter. The predicate sys_ignore/1 will run a goal with this flag set.

The debug clause fork is pickled with the instrumentation built-ins sys_in/0, sys_out/0 and sys_at/0. The end-user is not supposed to use them. They need to have public access so that they can be called from any clause fork.

The following mode handling predicates are provided:

sys_ignore(A):
The predicate succeeds whenever A succeeds. The goal A is invoked with the mode cloak temporarily set to on.

sys_in:
This instrumentation hook should succeed. It is called before a goal is called every time a goal is called.

sys_out:
This instrumentation hook should succeed. It is called after a goal exits every time a goal exits.

sys_at:
This instrumentation hook should succeed. It is called after a head unification succeeds and before the attribute variable unify hooks are called.

The following Prolog flags for mode handling are provided:

sys_cloak:
Legal values are on and off. The flag indicates whether the interpreter currently should ignore the debug mode. Default value is off. The value can be changed.

sys_clause_instrument:
Legal values are on and off. The flag is per knowledge base and is inherited when predicates are created. Default value is on. The value can be changed.

sys_head_wakeup:
Legal values are on and off. The flag is per knowledge base and is inherited when predicates are created. Default value is on. The value can be changed.

The following mode handling predicate properties are provided:

sys_noinstrument:
The property indicates that the clauses of the predicate should not be instrumented for debugging. The value can be changed.

sys_nowakeup:
The property indicates that frozen goals should not be woken up after the head of the clause is unified. The value can be changed.

Module automatic

We provide some predicates that allow listing the Java auto generated members. The auto loader has two functions. Firstly it allows loading a Prolog text when a module or class has been specified in a (:)/2 respectively (::)/2 invocation. Secondly it allows the automatic generation of interface members for a Java class when class name has been specified in a (:)/2 respectively (::)/2 invocation.

The programming interface documentation of the Jekejeke runtime describes the generated interface members in more detail. Basically we generate a public foreign function for an evaluable function if all arguments are numbers otherwise we generate a public foreign function for a predicate. If the name is overloaded we generate branching Prolog code. Bridging and tunnelling is automatically provided by the Prolog interpreter.

Example:

```
?- X is 'Math':'PI'.
X = 3.141592653589793

?- system/automatic:generated(_:_/1).
% Math.class
:- package(foreign(java/lang)).
:- module('Math', []).
:- reexport(foreign('Object')).
:- sys_auto_load(foreign('Math')).

:- public foreign_const('E'/1,'Math','E').

:- public foreign_const('PI'/1,'Math','PI').

:- public foreign_fun(random/1,'Math',random).
```

The usual listing commands listing/0 and listing/1 suppress the automatically generated interface members. The commands generated/0 and generated /1 on the other hand allow listing the automatically generated interface members. Only predicates that match the given pattern are listed. The listing profits from our compact representation by stacked modifiers, shortened import specifiers and shortened class specifiers.

The following automatic predicates are provided:

generated:
The predicate lists the user clauses of automatic predicates.

generated(I):
The predicate lists the user clauses of automatic predicates that match the pattern I.

Module memory

This module provides non-random access memory streams. A new read memory stream can be created via the predicate memory_read/3. A new write memory stream can be created via the predicate memory_write/2. The content of write memory stream can be retrieved via the predicate memory_get/2.

Example:

```
?- memory_write([], S), write(S, foo), memory_get(S, atom(A)).
S = 0r1479d830,
A = foo

?- memory_read(atom('foo.\n'), [], S), read(S, T).
S = 0r28100c13,
T = foo
```

The current implementation is limited in that the streams cannot be created with the reposition property. It is therefore not possibly to use the methods set_stream_position/2 or set_stream_length/2. Otherwise all the byte, char, term and stream operations can be applied as if they were file or web streams.

The following memory predicates are provided:

memory_read(T, O, S):
The predicate succeeds with a new read memory stream S for the initial data T and the open options O. The predicate recognizes the following open options:

type(T): [ISO]	T is the type (text or binary), default value is text.

The predicate recognizes the following data formats:

atom(A):	The atom A is the text content of the stream.
bytes(L):	The byte list L is the binary content of the stream.

memory_write(O, S):
The predicate succeeds with a new write memory stream S for the open options O. The predicate recognizes the same open options as the predicate memory_read/3.

memory_get(S, T):
The predicate succeeds with the current data T of the write stream S. The predicate recognizes the same data formats as the predicate memory_read/3.

Module charsio

This module provides temporary input/output redirection. The predicate with_output_to/2 redirects the output for the given goal and retrieves the stream content for each success. The predicate with_input_from/2 redirects the input for the given goal.

Examples:

```
?- with_output_to(atom(A), (write(foo); write(bar))).
A = foo ;
A = bar ;
No

?- with_input_from(atom('foo.\n'), read(X)).
X = foo ;
No
```

The predicate with_output_to/2 is non-deterministic for non-deterministic goals and will return different data results for each success. The predicate with_output_to/2 and with_input_from/2 recognize the same data formats.

The following charsio predicates are provided:

with_output_to(C, G):
The predicate succeeds whenever the goal G succeeds and unifies C with each data result. The predicate recognizes the same data formats as the predicate memory_read/3.

with_input_from(C, G):
The predicate succeeds whenever the goal G succeeds where C defines the initial data. The predicate recognizes the same data formats as the predicate memory_read/3.

Module attach

This module provide debugger attachment for a Prolog engine. The ordinary spy points and break points from the default debugger are stored on the knowledge base level. The spy points and break points provided by this module are stored in the current engine.

The predicate tdebugging/0 allows listing all the Prolog engine locale spy points and break points. The predicates tspy/1 and tbreak/2 allow adding spy points respectively break points. The predicates tnospy/1 and tnobreak/2 allow removing them.

The following attach predicates are provided:

tclear:
The predicate switches the engine to the inherit mode.t

tdebug:
The predicate switches the engine to the on mode.

ttrace:
The predicate switches the engine to the step in mode.

tskip:
The predicate switches the engine to the step over mode.

tout:
The predicate switches the engine to the step out mode.

tnodebug:
The predicate switches the engine to the off mode.

tleash(L):
Leash the ports of the engine that are listed in L, unleash the of the engine ports that are not listed in L. When prompted, unleashed ports do not await user interaction but simply continue. The predicate accepts the same mnemonics as the predicate visible/1.

tvisible(L):
Show the ports of the engine that are listed in L, hide the ports of the engine that are not listed in L. In debug mode, hidden ports are not further debugged but simply continue. The predicate accepts the same mnemonics as the predicate visible/1.

tdebugging:
The predicate shows the engine spy points and the engine break points.

tspy(P):
The predicate adds the predicate P to the engine spy points.

tnospy(P):
The predicate removes the predicate P from the engine spy points.

tspying(P):
The predicate succeeds in P for every engine spy point.

tbreak(F, L):
The predicate adds the file F and the line number L to the engine break points.

tnobreak(F, L):
The predicate removes the file F and the line number L from the engine break points.

tbreaking(F, L):
For every engine break point the predicate succeeds with the file F and the line number L.

The following attach thread flags are provided:

sys_tdebug:
The legal values are inherit, on, step_in, step_over, step_out and off. The value indicates the debugging mode of the given thread. The default value is inherit. The value can be changed.

sys_top_frame:
The value is a stack frame or null. The value is the top frame of the given thread. The value cannot be changed.

sys_thead_store:
The value is a knowledge base. The value is the knowledge base of the given thread. The value cannot be changed.

The following attach Prolog flags are provided:

sys_tleash:
Legal values are lists of debugger ports. Legal debugger ports are call, exit, redo, fail, head and chop. The value indicates the engine prompt leashing. The default value is [call,exit,redo,fail]. The value can be changed.

sys_tvisible:
Legal values are lists of debugger ports. Legal debugger ports are call, exit, redo, fail, head and chop. The value indicates the engine tracing visibility. The default value is [call,exit,redo,fail]. The value can be changed.

Compatibility Matrix

The following compatibility issues persist for the system theory:

Table 5: Compatibility Matrix for the System Theory

Nr	*Description*	*System*

5.5 Notebook Package

This theory is concerned with document stores.

- **Module model:** The module provides access and modification of a DOM model.
- **Module serialize:** This module provides reading and writing of a DOM model.
- **Module transform:** Provides validation and transformation of DOM model.

Module model

The module provides access and modification of a DOM model. The DOM model is generalized so that it can be used to store both XML and JSON. This is done in that a text done can hold not only strings, but also other scalar data such as numbers. Further, the attribute axis of DOM element can also hold other DOM elements than only scalar data.

The following model predicates are provided:

node_is_elem(D):
The predicate succeeds if the DOM node D is a DOM element.

node_is_text(D):
The predicate succeeds if the DOM node D is a DOM text.

node_get_parent(D, C):
The predicate succeeds in C with the parent of the DOM node D.

node_get_key(D, K):
The predicate succeeds in K with the key name of the DOM node D.

node_set_key(D, K):
The predicate succeeds in setting the key name of the DOM node D to K.

node_copy(D, C):
The predicate succeeds in C with a copy of the DOM node D.

text_new(D):
The predicate succeeds in D with a new DOM text.

text_get_data(D, T):
The predicate succeeds in T with the data of the DOM text D.

text_set_data(D, T):
The predicate succeeds in setting the data of the DOM text D to T.

elem_new(D):
The predicate succeeds in D with a new DOM element.

elem_get_name(D, N):
The predicate succeeds in N with the name of the DOM element D.

elem_set_name(D, N):
The predicate succeeds in setting the name of the DOM element D to N.

elem_get_attr(D, A, V):
The predicate succeeds in V with the attribute A of the DOM element D.

elem_set_attr(D, A, V):
The predicate succeeds in setting the attribute A of the DOM element D to V.

elem_reset_attr(D, A):
The predicate succeeds in removing the attribute A from the DOM element D.

elem_attr(D, A):
The predicate succeeds in A for all the attribute names of the DOM element D.

elem_add_node(D, C):
The predicate succeeds in adding the DOM node C to the DOM element D.

elem_remove_node(D, C):
The predicate succeeds in removing the DOM node C from the DOM element D.

elem_node(D, C):
The predicate succeeds in C for all the DOM nodes of the DOM element D.

Module serialize

The rest of the predicates deal with reading/writing a DOM model. The predicate node_load/4 can be used to load a DOM model from a stream. The loading requires an already existing DOM node, which is then overwritten. The predicate node_store/4 can be used to store a DOM model to a stream.

The following serialize predicates are provided:

node_read(S, N):
node_read(S, N, O):

The predicate succeeds in loading the stream S into the DOM node D with the DOM options O. The following DOM options are recognized:

root(M):	M is the root mask.
type(E,T):	T is the complex type for the element E.
format(F):	F is the format.
comment(C):	C Is the writing comment.

The root mask can take the values "tree", "table", "document" and "fragment". The default value is "table". The complex type can take the values "none", "empty" and "any". The format can take the values "xml" and "json". The default value is "json".

node_write(S, N):
node_write(S, N, O):

The predicate succeeds in storing the DOM node D into the stream S with comment C and the DOM options O.

node_term(N, T):

If N is a variable then the predicate succeeds when N unifies with the DOM of T. Otherwise the predicate succeeds when T unifies with the term N.

Module transform

This module provides a couple of simple utilities to deal with the validation and application of XSL models. When a XSL model is validated, the referenced XML data is validated via its associated XSD schema. The predicates schema_new/1 and schema_digest/2 allow creating an XSD schema. An XML model is validated by the predicate data_check/3.

Example:

```
?- open('hello_buggy.xsl', read, S), elem_new(D),
   node_load(D, S, [root(text)]), close(S), assertz(my_data(D)).
?- my_data(D), sheet_check(D,[]).
Error: Undeclared XPath variable.
```

The XSL model loads referenced data or schema via reflection. The result should be an instance that implements the Java interface InterfacePath. We currently use the standard Java class loader to create the instance. Using the class loader from the Prolog knowledge base is planned for future releases of this module.

Example:

```
?- open('hello_english.xsl', read, S), elem_new(D),
   node_load(D, S, [root(text)]), close(S), assertz(my_data(D)).
?- my_data(D), current_output(S),
   sheet_transform(D, S, '', [variable(name,'John')]).
Welcome John!
```

XSL model select data fragments by XPath expressions. These expressions are parsed on the fly. During validation they advanced the current schema, whereas during application they advanced the current data. XSL models can be validated by the predicate sheet_check/2 and applied by the predicate sheet_transform/4.

The following xsl predicates are provided:

schema_new(S):
The predicate succeeds in S with a new XSD schema.

schema_digest(S, D):
The predicate succeeds in digesting the DOM element D into the XSD schema S.

data_check(D, S, O):
The predicate succeeds in checking the DOM node D versus the XSD schema S with the DOM options O.

sheet_transform(T, S, C, O):
The predicate succeeds in transforming the DOM node T into the stream S with comment C and the XSL options O. The following XSL options are recognized:

root(M):	M is the root mask.
type(E,T):	T is the complex type for the element E.
variable(N,V):	V is the value of the variable N.

The root mask can take the values list and text. The default value is text. The complex type can take the values empty and any. The variable values can be atoms and integers.

sheet_check(T, O):
The predicate succeeds in checking the DOM node T with the XSL options O.

5.6 Wire Package

This theory is concerned with wiring a debugger.

- **Module monitor:** The module provides a debugger monitor.

Module monitor

This module provides a HTTP object class to inspect the Prolog threads, Prolog stack frames and Prolog variables of a Prolog instance.

The HTTP object supports GET dispatch and GET upgrade to web sockets. The web sockets are used to notify the HTTP client of state changes.

The following monitor predicates are provided:

initialized(O, S):
The predicate is called when the server S is initialized for object O.
destroyed(O, S):
The predicate is called when the server S is destroyed for object O.
dispatch(O, P, R, S):
The predicate succeeds in dispatching the request for object O, with path P, with request R and the session S.
upgrade(O, P, R, S):
The predicate succeeds in upgrading the request for object O, with path P, with request R and the session S.

The following monitor Prolog flags are provided:

sys_monitor_config:
Legal values are -1 is no monitor, 0 is a dynamic monitor port and other values are static monitor ports. The default value is -1. The value can be changed.
sys_monitor_running:
Legal values are -1 is no monitor, 0 is a dynamic monitor port and other values are static monitor ports. The default value is -1. The value can be changed.
sys_monitor_logging:
Legal values are on and off. The default value is on. The value can be changed.

6 Appendix Example Listings

The full source code of the Java classes and Prolog texts for the examples is given. The following source code has been included:

- Tracing Example
- Port Statistics

6.1 Tracing Example

The following artefacts are listed:

- **trace.p:** The Prolog text for the tracing example.

Prolog Text trace

```
/**
 * Prolog code for the trace example.
 *
 * Copyright 2012, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.7 (a fast and small prolog interpreter)
 */

p.
r :- p.
q :- p.
s :- q, r.

% ?- s.
% Yes

% ?- trace.
% Yes

% ?- s.
%      0 Call s ?
%      1 Call q ?
```

6.2 Port Statistics

The following artefacts are listed:

- **count.p:** The Prolog text for the port statistics without call-site information.
- **count2.p:** The Prolog text for the port statistics with call-site information.

Prolog Text count

```
/**
 * Prolog code for the port statistics without call-site information.
 *
 * The following data will be gathered:
 *    count(Fun, Arity, CallExitRedoFail).
 *
 * Copyright 2011, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.2 (a fast and small prolog interpreter)
 */

remove_count :-
   retract(count(_,_,_)),
   fail.
remove_count.

% add_count(+CallExitRedoFail, +CallExitRedoFail, -CallExitRedoFail)
add_count(A-B-C-D, E-F-G-H, R-S-T-U) :-
    R is A+E,
    S is B+F,
    T is C+G,
    U is D+H.

% update_count(+Fun, +Arity, +CallExitRedoFail)
update_count(F, A, D) :-
    retract(count(F, A, R)), !,
    add_count(R, D, S),
    assertz(count(F, A, S)).
update_count(F, A, D) :-
    assertz(count(F, A, D)).

% get_delta(+Port, -CallExitRedoFail)
get_delta(call, 1-0-0-0).
get_delta(exit, 0-1-0-0).
get_delta(redo, 0-0-1-0).
get_delta(fail, 0-0-0-1).

% goal_tracing(+Port, +Frame)
goal_tracing(P, Q) :-
    frame_property(Q, sys_call_indicator(F,A)),
    get_delta(P, D),
    update_count(F, A, D).

% show
show :-
    write('Pred\tCall\tExit\tRedo\tFail'), nl,
    count(F, A, R-S-T-U),
    write(F/A), write('\t'),
    write(R), write('\t'),
    write(S), write('\t'),
    write(T), write('\t'),
    write(U), nl,
    fail.
show.

% reset
reset :-
    remove_count.
```

Prolog Text count2

```
/**
 * Prolog code for the port statistics with call-site information.
 *
 * The following data will be gathered:
 *    count_predicate(Fun, Arity, CallExitRedoFail).
 *    count(Fun, Arity, Origin, Line, CallExitRedoFail).
 *
 * Copyright 2011, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.2 (a fast and small prolog interpreter)
 */

remove_count_predicate :-
    retract(count_predicate(_,_,_)),
    fail.
remove_count_predicate.

remove_count :-
    retract(count(_,_,_,_,_)),
    fail.
remove_count.

% add_count(+CallExitRedoFail, +CallExitRedoFail, -CallExitRedoFail)
add_count(A-B-C-D, E-F-G-H, R-S-T-U) :-
    R is A+E,
    S is B+F,
    T is C+G,
    U is D+H.

% update_count_predicate(+Fun, +Arity, +CallExitRedoFail)
update_count_predicate(F, A, D) :-
    retract(count_predicate(F, A, R)), !,
    add_count(R, D, S),
    assertz(count_predicate(F, A, S)).
update_count_predicate(F, A, D) :-
    assertz(count_predicate(F, A, D)).

% update_count(+Fun, +Arity, +Origin, +Line, +CallExitRedoFail)
update_count(F, A, O, L, D) :-
    retract(count(F, A, O, L, R)), !,
    add_count(R, D, S),
    assertz(count(F, A, O, L, S)).
update_count(F, A, O, L, D) :-
    assertz(count(F, A, O, L, D)).

% get_delta(+Port, -CallExitRedoFail)
get_delta(call, 1-0-0-0).
get_delta(exit, 0-1-0-0).
get_delta(redo, 0-0-1-0).
get_delta(fail, 0-0-0-1).

% goal_tracing(+Port, +Frame)
```

```
goal_tracing(P, Q) :-
      frame_property(Q, sys_text_frame(T)),
      frame_property(T, sys_clause_ref(R)), !,
      clause_property(R, source_file(O)),
      clause_property(R, line_no(L)),
      frame_property(Q, sys_call_indicator(F,A)),
      get_delta(P, D),
      update_count_predicate(F, A, D),
      update_count(F, A, O, L, D).
goal_tracing(P,Q) :-
      frame_property(Q, sys_call_indicator(F,A)),
      get_delta(P, D),
      update_count_predicate(F, A, D),
      update_count(F, A, '', 0, D).

% show_shortname(+Path)
show_shortname(O) :-
    source_property(O, short_name(S)), !,
    write(S), write('\t').
show_shortname(_) :-
    write('\t').

% show_callsite(+Fun, +Arity)
show_callsite(F, A) :-
    count(F, A, O, L, R-S-T-U),
    write('\t'), show_shortname(O), write(L), write('\t'),
    write(R), write('\t'),
    write(S), write('\t'),
    write(T), write('\t'),
    write(U), nl,
    fail.
show_callsite(_,_).

% show
show :-
    write('Pred\tSource\tLine\tCall\tExit\tRedo\tFail'), nl,
    count_predicate(F, A, R-S-T-U),
    write(F/A), write('\t'), write('\t'), write('\t'),
    write(R), write('\t'),
    write(S), write('\t'),
    write(T), write('\t'),
    write(U), nl,
    show_callsite(F,A),
    fail.
show.

% reset
reset :-
    remove_count_predicate,
    remove_count.
```

Indexes

Public Predicates

Predicate	Module
asserta_opt/2	inspection/clause
assertz_opt/2	inspection/clause
clause_frame/3	inspection/frame
clause_property/2	inspection/clause
current_base/1	inspection/base
current_provable/1	inspection/provable
current_syntax/1	inspection/syntax
debug/0	debug/default
debugging/0	debug/default
dump/0	debug/dump
dump/1	debug/dump
expose_goal/3	debug/custom
frame_property/2	inspection/frame
friendly/0	debug/friendly
friendly/1	debug/friendly
generated/0	system/automatic
generated/1	system/automatic
goal_exposing/3	debug/custom
goal_tracing/2	debug/custom
leash/1	debug/default
memory_get/2	system/memory
memory_read/3	system/memory
memory_write/2	system/memory
nodebug/0	debug/default
(nopin)/1	debug/default
noprotocol/0	system/protocol
(nospy)/1	debug/default
out/0	debug/default
(pin)/1	debug/default
protocol/1	system/protocol
provable_property/2	inspection/provable
reset_clause_property/2	inspection/clause
reset_frame_property/2	inspection/frame
reset_provable_property/2	inspection/provable
reset_syntax_property/2	inspection/syntax
retract_frame/2	inspection/frame
set_clause_property/2	inspection/clause
set_frame_property/2	inspection/frame
set_provable_property/2	inspection/provable
set_syntax_property/2	inspection/syntax
skip/0	debug/default
(spy)/1	debug/default
syntax_property/2	inspection/syntax
sys_call_goal/1	inspection/stack
sys_callable_colon/2	inspection/provable
sys_callable_slash/2	inspection/base
sys_clause_term/1	inspection/frame
sys_ignore/1	system/mode
sys_in/0	system/mode
sys_indicator_colon/2	inspection/provable

sys_instance_clause/2	inspection/frame
sys_out/0	system/mode
sys_prepare_clause/2	inspection/stack
sys_repose_goal/3	debug/default
sys_trace/2	debug/default
sys_unfold_body/1	inspection/stack
trace/0	debug/default
trace_goal/2	debug/custom
with_input_from/2	testing/charsio
with_output_to/2	testing/charsio

Package Local Predicates

Predicate	Module

Non-Private Meta-Predicates

Predicate	Exp	Body	Rule	Module
asserta_opt(-1,?)	yes	no	no	inspection/clause
assertz_opt(-1,?)	yes	no	no	inspection/clause
clause_frame(-1,0,?)	no	no	no	inspection/frame
frame_property(?,0)	no	no	no	inspection/frame
retract_frame(-1,?)	no	no	no	inspection/frame
sys_call_goal(0)	yes	no	no	inspection/stack
sys_clause_term(-1)	yes	no	no	inspection/frame
sys_ignore(0)	yes	no	no	system/mode
with_input_from(?,0)	yes	no	no	testing/charsio
with_output_to(?,0)	yes	no	no	testing/charsio

Non-Private Closure-Predicates

Predicate	Module

Non-Private Syntax Operators

Level	Mode	Operator	Module
1150	fx	pin	debug/default
1150	fx	nospy	debug/default
1150	fx	nopin	debug/default
1150	fx	spy	debug/default

Pictures

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

Tables

References

[1] Bowen, D.L. et al. (1982): DECsystem-10 PROLOG USER'S MANUAL, Department of Artificial Intelligence University of Edinburgh, 10 November 1982
[2] ISO (1995): Prolog, Part 1: General Core, International Standard ISO/IEC 13211-1, First Edition, 1995-06-01
[3] Moura, P. et al. (2010): ISO/IEC DTR 13211 3:2006, Definite Clause Grammar Rules, Draft, April 1, 2010 http://www.sju.edu/~jhodgson/wg17/Drafts/DCGs/DCGs-DRAFT-2010-04-01.pdf
[4] Richard O'Keefe (2011): An Elementary Prolog Library, Draft 8, November 19, 2010 http://www.complang.tuwien.ac.at/ulrich/iso-prolog/pllib-2010-11-19.htm
[5] Warren, D.H.D. (1983): An Abstract Prolog Instruction Set, Technical Note 309, SRI International, 1983
[6] Nässén, H (2001): Optimizing the SICStus Prolog virtual machine instruction set, SICS Technical Report, March 2001
[7] Cousot, P. and Cousot, R. (1977): Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix-points. In Proc. Of SPPL’77, pages 238-252, Los Angeles, California, 1977
[8] Ferreira, M. and Damas, L. (1996): Unfolding WAM Code, Implementation Technology for Logic Programming Languages, Bonn, Germany, September 1996
[9] Ferriera, M. and Damas, L. (2003): WAM Local Analysis, International Symposium on Practical Aspects of Declarative Languages, 2003