Jekejeke Develop Interface

Version 1.3.4, January 16th, 2019

XLOG Technologies GmbH

Jekejeke Prolog

Development Environment 1.3.4

Programming Interface

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

Date: January 16th, 2019
Version: 0.7

Participants: None

Warranty & Liability

Rights & License

Trademarks

Table of Contents

Change History

Jan Burse, February 16th, 2012, 0.1:
- Initial Version.

Jan Burse, July 19th, 2012, 0.2:
- Environment toolkit moved to package platform.

Jan Burse, August 8th, 2012, 0.3:
- Some API fixes.

Jan Burse, March 4th, 2013, 0.4:
- Member index introduced.

Jan Burse, December 6th, 2013, 0.5:
- Some API fixes.

Jan Burse, January 30th, 2018, 0.6:
- New verbose command line option.

Jan Burse, January 16th, 2019, 0.7:
- New monitor command line option.

1 Introduction

This document gives a reference of the Jekejeke Prolog application programming interface as provided by the development environment.

- **Programming Examples:** We show some examples of the use of the Jekejeke Prolog application programming interface. We first show how to control the custom debugger from within a Java thread. We then show how to animate a Window from within a Java thread.

- **Integration Concepts:** t.b.d.

- **Headless API:** The development environment enhances the core set of predicates and executes code with instrumentation. It is represented by a singleton.

- **Appendix Example Listing:** The full source code of the Java classes and Prolog sources for the programming examples is given.

2 Programming Examples

We show some examples of the use of the Jekejeke Prolog application programming interface. We first show how to control the custom debugger from within a Java thread. We then show how to animate a Window from within a Java thread.

- **Port Sampling:** We first show how to control the custom debugger from within a Java thread. We will periodically poke an interpreter so that it does some port sampling via a custom debugger hook for us.

- **Memory Monitor:** We then show how to animate a Window from within a Java thread. The Window will show in real time a JFreeChart chart of the memory consumption of the development environment.

2.1 Port Sampling

We first show how to control the custom debugger from within a Java thread. We will periodically poke an interpreter so that it does some port sampling via a custom debugger hook for us.

- **Thread Interaction:** We give an overview of the components and the execution flow that we will use in the solution. There will be two active components.

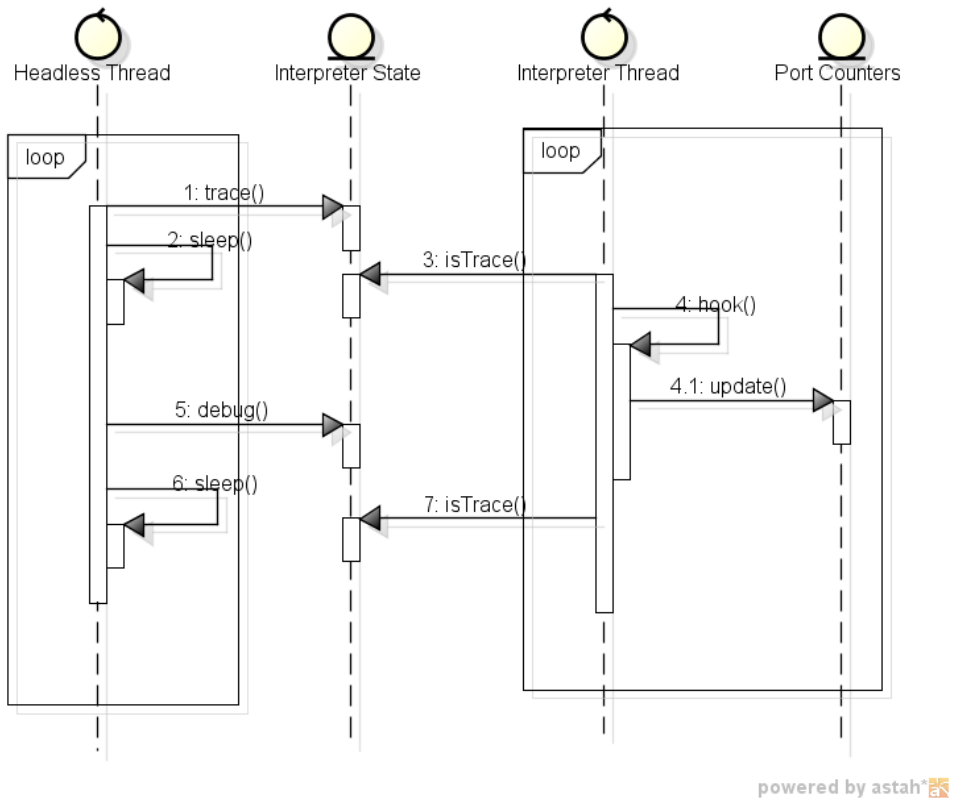
- **Relative Statistics:** We will use a customer debugger hook that will count the number of port events. There will be a simple statistic that will show the relative counts.

- **Headless Thread:** The component will periodically poke the interpreter. It will alternately let run the interpreter in debug mode and trace mode.

- **Example Uses:** The sampling thread can be invoked from within the development environment. Sampling gives approximate results in less time.

Thread Interaction

We give an overview of the components and the execution flow that we will use in the solution. There will be two active components. The first active component will be the sampling thread that will poke the interpreter. The second active component is the interpreter thread executing the Prolog text at hand.



Picture 1: Thread Interaction Port Sampling

Main Scenario Headless Thread:

1. The headless thread sets the interpreter state to trace.
2. The headless thread sleeps a larger amount of time.
3. The headless thread sets the interpreter state to debug.
4. The headless thread sleeps a smaller amount of time.
5. The flow continues with step 1.

Main Scenario Interpreter Thread:

1. The interpreter thread executes an instruction in trace mode.
2. The interpreter thread calls the debugger hook for a port.
3. The custom hook updates the port counters.
4. The flow continues with step 1.

Alternative Scenario Interpreter Thread

1a.1. The interpreter thread executes an instruction in debug mode.

1a.2. The flow continues with step 1.

For the interpreter thread there will be not much coding. The interpreter does already the polling of the trace flag and invokes the debugger hook for us. All we have to do is implement a custom hook that will update the port counters. We will draw upon the port statistics example already presented in the language reference manual of the development environment. We will present an enhanced version that will be able to show relative statistics.

Finally we will need to implement the main scenario of the frame thread. Similar to the interpreter thread this is basically a forever loop. We do not show here how the thread can be started and stopped from the command line of the development environment. The full source code can be found in the appendix. Certain aspects will be explained in the subsequent sections.

Relative Statistics

We will use a customer debugger hook that will count the number of port events. There will be a simple statistic that will show the relative counts. The Prolog text for the relative statistics is derived from the port statistics already found in the language reference document for the development environment. As a first step we have added a predicate time/1 to measure the used time. This predicate uses the new Java high resolution clock System.nanoTime(). We need this facility since our test program has execution time in the order of ~50ms.

As a test program we use the Einstein riddle. The Einstein riddle was first published under the heading "Who owns the Zebra?" in the Life International magazine on December 17, 1962 with solution given in the March 25, 1963 issue. The riddle given here is not a verbatim copy. We can use the time predicate to see how fast the riddle can be solved in Prolog. As a test machine we used a similar set-up as in the runtime library benchmark:

```
?- ['jekdev/study/ports/einstein.p'].
?- ['jekdev/study/ports/relative.p'].
?- time(test), time(test).
test  in 195.2 ms
test  in 47.7 ms
```

The difference between the first and second timing is probably due to the turbo boost feature of the test machine CPU. The CPU will automatically increase the clock rate on load and if only a single core is loaded. To perform the port sampling we will need to run the test program in debug mode. The test program runs a little slower in debug mode, since each instruction will be instrumented. But the overhead is not that great, since the Java Just-in-Time (JIT) compiler helps reducing the effort for the instrumentation code:

```
?- debug.
?- time(test), time(test).
test  in 121.3 ms
test  in 68.8 ms
```

As a next step we have changed the reporting predicate of the port statistics. The relative port statistics will use the exact same custom debugger hook as the absolute port statistics. But the reporting predicate will show relative port statistics. The shown relative number will express the ratio between the number of port events and the total number of port events for the same port type:

$$Intensity_{Indicator,Type} = \frac{Count_{Indicator,Type}}{\sum_{Indicator} Count_{Indicator,Type}}$$

We have realized the above formula in Prolog by first computing the sum as follows:

```
% sum_counts(+List,-CallExitRedoFail)
sum_counts([],0-0-0-0).
sum_counts([_-D|L],S) :-
    sum_counts(L,R),
    add_count(R,D,S).

% show
show :-
    findall(F/A-D,count(F,A,D),L),
    sum_counts(L,TR-TS-TT-TU),
```

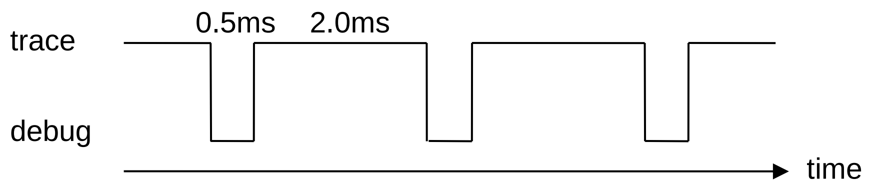
The sums TR, TS, TI and TU are then used to compute the relative port statistics as follows:

```
    write('Pred\tCall\tExit\tRedo\tFail'), nl,
    keysort(L,M),
    mem_counts(I-(R-S-T-U),M),
    RP is R*100 / TR,
    SP is S*100 / TS,
    TP is T*100 / TT,
    UP is U*100 / TU,
    ((RP >= 0.05; SP >=0.05; TP >=0.05; UP >=0.05) ->
```

The relative statistics in per cent are found in RP, SP, TP and UP. The keysort/2 predicate comes in handy to sort the results so that we will get a report that shows the predicate indicators in descending order. This makes it easier to compare different reports. The comparison with 0.05 is used to suppress result rows that would be shown as 0.0 0.0 0.0 0.0. This will assure that the report is not blown up by rarely called predicates.

Headless Thread

The component will periodically poke the interpreter. It will alternately let run the interpreter in debug mode and trace mode. The diagram below shows the realized timing. The debug mode will only last 0.5ms. During debug mode the measured program will nearly run at full speed. On the other hand the trace mode will last as long as 2.0ms. During trace mode the measured program will run much slower since the custom hook is called and the counts are accumulated. We therefore give more time to the trace mode than to the debug mode.



We will only poke one interpreter. A design that pokes multiple interpreters would also be possible. For the sake of simplicity we only show the simple solution. The interpreter under consideration will be stored in a field of the headless thread. The main loop of the headless thread will then use the setStatus() method to change the debugger mode:

```
                try {
                    inter.setStatus(
                       CapabilityDevelopment.STATUS_SYS_TRACE_MODE, on);
                } catch (InterpreterMessage x) {
                    throw new RuntimeException(x);
                }
```

The headless thread has then to sleep a given time. We opted for an implementation based on nano-seconds. Corresponding APIs are available since JDK 1.5 in connection with the new Java package java.util.concurrent. The thread will create a monitor via the class ReentrantLock and obtain a condition from it. The condition is then used to wait with a nano-second argument. Since a condition is allowed to yield for no reason we have implemented a loop that repeatedly waits:

```
                long sleep = SLEEP[on ? 1 : 0] -
                             (System.nanoTime() - lastTime);
                try {
                    while (sleep > 0)
                        sleep = cond.awaitNanos(sleep);
                } catch (InterruptedException x) {
                    return;
                }
```

The headless thread will not have any associated graphic element. We put it into the responsibility of the tester to manually start and stop the headless thread. To start the headless thread there will be a Java method that will return a handle to a newly created and started headless thread. We will store the handle in a thread local predicate. The corresponding Prolog code reads as follows:

```
:- foreign(start_sampler/1, 'jekdev.study.ports.ThreadAPI',
startSampler('Interpreter')).

:- dynamic handle/1.

start_sampler :-
  start_sampler(X),
  assertz(handle(X)).
```

The handle can later be retrieved to stop the thread again. There is a Java method for this purpose and corresponding Prolog code. More details can be found in the appendix where the full Java classes and Prolog text is listed.

Example Uses

The sampling thread can be invoked from within the development environment. Sampling gives approximate results in less time. Let’s verify this claim. So we will first run the collector without sampling. We will switch on the trace mode and run the tested program. As a result all port events will be recorded indiscriminately of the time frame. The following sequence runs the collector without sampling:

```
?- trace.
?- time(test), time(test).
test  in 1816.0 ms
test  in 1132.8 ms
?- nodebug.
```

We have run the test program twice to show the warm-up effect. The following sequence shows the relative port statistics. Since we have run the collector without sampling, the relative port statistics will be exact:

```
?- show.
```

```
Pred         Call         Exit         Redo         Fail
member / 2  26.3 %       25.9 %       25.8 %       26.2 %
nextTo / 3  5.7 %        21.6 %       21.6 %       5.7 %
rightTo / 3 68.1 %       52.5 %       52.6 %       68.1 %
```

Next we will run the collector with sampling. This time we will only switch on the debug mode. Before running the test program we will start the sampler. As a result port events will be only recorded during the trace intervals. The following sequence runs the collector with sampling:

```
?- reset.
?- ['jekdev/study/ports/thread.p'].
?- debug.
?- start_sampler.
?- time(test), time(test).
test  in 257.3 ms
test  in 207.0 ms
?- stop_sampler.
?- nodebug.
```

The collector with sampling does need less time than the collector without sampling. In the present case we see that only around a 1/4 of the time is needed. The following sequence shows the relative port statistics. Since we have run the collector with sampling, the relative port statistics will be approximations:

```
?- show.
Pred         Call         Exit         Redo         Fail
member / 2  26.2 %       24.3 %       24.0 %       25.8 %
nextTo / 3  5.6 %        21.9 %       22.1 %       5.7 %
rightTo / 3 68.2 %       53.8 %       54.0 %       68.5 %
```

Comparing the approximations with the exact values shows a difference in the range of 0.1% to 1.8%. By means of reporting the absolute values and with the help of a little math we could even extrapolate the exact values from the approximate values and a debug only run. But this would require more stable time measurements and thus possibly averaging the measurement over multiple runs. This would obviate our goal of a fast method.

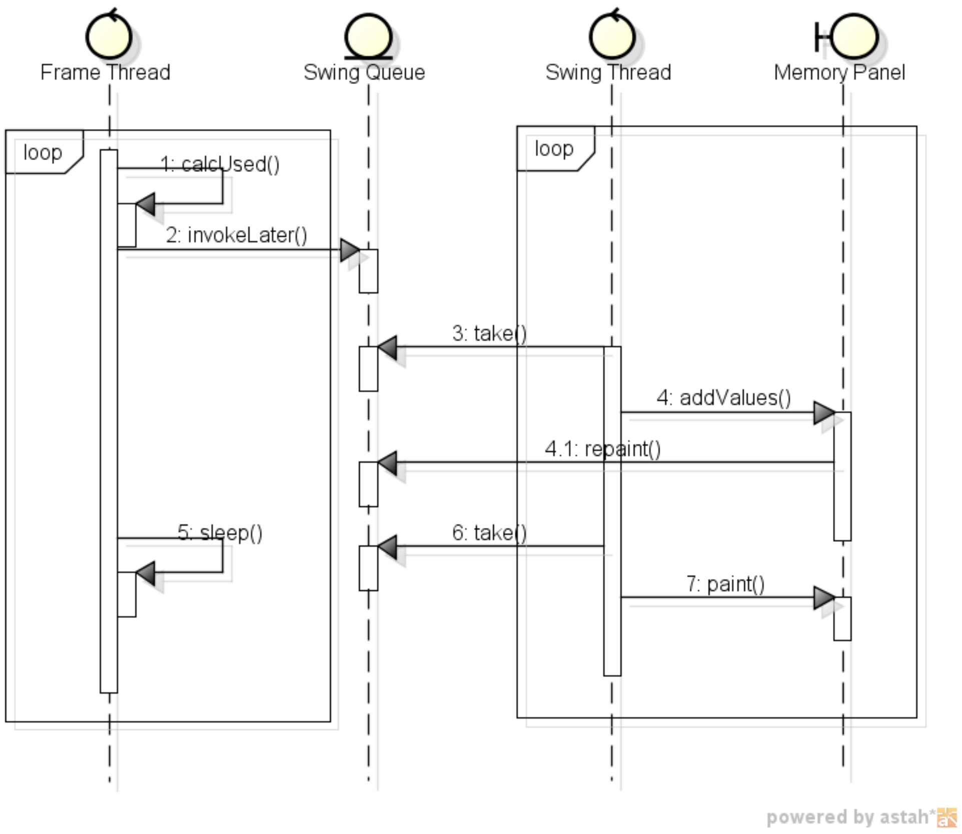
2.2 Memory Monitor

We then show how to animate a Window from within a Java thread. The Window will show in real time a JFreeChart chart of the memory consumption of the development environment.

- **Thread Interaction:** We give an overview of the components and the execution flow that we will use in the solution. There will be two active components.
- **Memory Panel:** This component will show the memory usage in a chart. The chart will roll from right to left and show a constant set of sample points.
- **Frame Thread:** This component will do the memory sampling and place a memory panel update on the event queue.
- **Memory Frame:** This component will show the memory usage in a window frame and provide a Java method for a foreign predicate to show the window frame.
- **Example Uses:** The memory monitor can be invoked from within the development environment. The memory usage pattern highly depends on the JVM.

Thread Interaction

We give an overview of the components and the execution flow that we will use in the solution. There will be two active components. The first active component will be the monitoring thread that will poll the memory usage. The second active component is the Swing event processing thread.



Picture 2: Thread Interaction Memory Monitor

The above diagram shows the parallel flow between the two active components. The two components communicate over the event queue. The frame thread is the event producer whereas the Swing thread is basically the event consumer. The Swing thread might also produce new events. In summary the parallel flow will work as follows:

Main Scenario Frame Thread:

1. The frame thread calculates the actual memory usage.
2. The frame thread posts an event with a new value for the memory panel.
3. The frame thread sleeps a particular amount of time.
4. The flow continues with step 1.

Main Scenario Swing Thread:

1. The swing thread receives a value event.
2. The swing thread requests the memory panel to add a new value.
3. The memory panel adds a new value.
4. The memory panel posts a repaint event.
5. The flow continues with step 1.

Alternative Scenario Swing Thread:

1a.3. The swing thread receives a repaint event.

1a.4. The swing thread requests the memory panel to paint itself.

1a.5. The memory panel draws itself.

1a.6. The flow continues with step 1.

For the alternative scenario of the Swing thread there will be not much coding on our side. We will use an out of the box component from JFreeChart. This component does all the drawing for us. The component does also necessary computations for us, such as determining min and max values and computing scale ticks.

The main scenario of the Swing thread does only require a little coding. The Swing queue and the Swing thread are already part of the Swing framework delivered with the Java runtime environment. But what will need to implement is a method to incorporate the data into the chart, since the chart does not know how we would like the data be placed.

Finally we will need to implement the main scenario of the frame thread. Similarly to the Swing thread it is basically a forever loop. We do not show here is how the thread is stopped by the memory frame. Also there is no depiction of the creation of the memory frame. The full source code can be found in the appendix. Certain aspects will be explained in the subsequent sections.

Memory Panel

This component will show the memory usage in a chart. The chart will roll from right to left and show a constant set of sample points. The dataset will be a set of XY-series, whereby we currently only need one XY-series. This dataset will be shown as a XY-area. Since the dataset will be sampled in seconds we will need a number X-axis. Similarly for the memory usage we will need a number Y-axis. The constructor of the MemoryPanel class will do the corresponding setup.

```
    /**
     * <p>Create a free chart panel.</p>
     * <p>The layout is basically:</p>
     * <pre>
     *     +--+------------------------------------+
     *     |N |                                    |
     *     |uA|                                    |
     *     |mx|                                    |
     *     |bi|             XYArea                 |
     *     |es|                                    |
     *     |r |                                    |
     *     +--+------------------------------------+
     *     |  |           NumberAxis               |
     *     +--+------------------------------------+
     * </pre>
     *
     * @param n The numnber of time series.
     */
    MemoryPanel(int n) {
```

More details on the internals of the constructor can be found in the appendix where a full listing of the Java code is given. We basically assemble a chart from the JFreeChart libraries. The chart will be stored in a field of the memory panel, so that it can be later accessed by the addValues() method. This method is responsible for filling the dataset. The method first appends new data to the right:

```
        for (int i = 0; i < dataset.getSeriesCount(); i++) {
            XYSeries series = dataset.getSeries(i);
            series.add(when, values[i]);
        }
```

Without further measures the dataset would accumulate more and more data. We limit the number of Y values by the constant VALUES_LIMIT. We then use a counter to check whether we have reached the limit. If the limit has reached we remove Y values and corresponding X values from the left. The VALUES_LIMIT is currently fixed to 200. Higher values mean more time spent by the chart adjusting and painting the chart. Which values guarantee a smooth scrolling of the chart depends on the machine performance and on the update frequency of the frame thread.

```
        count++;
        while (count >= VALUES_LIMIT) {
            for (int i = 0; i < dataset.getSeriesCount(); i++) {
                XYSeries series = dataset.getSeries(i);
                series.delete(0, 0);
            }
            count--;
        }
```

We had first experimented with the NetBeans/VisualVM chart. This chart was only able automatically increase the Y axis scale, but not to the decrease the Y axis scale. But this can happen since we use a sliding window. We therefore switched to JFreeChart.

Frame Thread

This component will do the memory sampling and place a memory panel update on the event queue. We have derived the frame thread from the Thread class. We also implemented the Runnable interface, so that we can simply start the frame thread by starting an instance. The frame thread will perform the loop that will sample the memory usage and create Swing events. The memory usage is computed from the total memory and the free memory in meg-abyte units:

```
            double[] values = new double[1];
            long total = Runtime.getRuntime().totalMemory();
            long free = Runtime.getRuntime().freeMemory();
            values[0] = (double) (total - free) / 1000000;
```

The little helper class MemoryJob then serves us in submitting an event to the Swing thread. We could also have used an anonymous inner class as a form of a closure. But this anony-mous inner class would have forced us to declare some local variables as final. In the end we would have needed a few more local variables. We think the MemoryJob helper class cleaner and more extensible in the future. More details on the helper class can be found in the appendix.

The frame thread has then to sleep a given time. We use the same nano-second approach as found in the headless thread of the previous port sampling example:

```
            long sleep = SLEEP - (System.nanoTime() - lastTime);
            try {
                while (sleep > 0)
                    sleep = cond.awaitNanos(sleep);
            } catch (InterruptedException x) {
                return;
            }
```

The frame thread can also be terminated. What is often seen in educational material is the use of a shared flag to control a primary thread by a secondary thread. What we instead ex-pect the secondary thread to do here is a thread.interrupt(). This will terminate the Thread.sleep() with an InterruptedException, eventually only on re-entering the call. We will catch the exception and terminate the loop. This design works as long as the loop does not perform long running non-interruptible calls, which is the case in the present.

Memory Frame

This component will show the memory usage in a window frame and provide a Java method for a foreign predicate to show the window frame. The memory frame has its own constructor which will place a memory panel inside the memory frame. The main method visible from Prolog is the startMonitor() method. This method will create a memory frame and a frame thread. It will then make the memory frame visible and start the frame thread:

```
    /**
     * <p>Java method called from Prolog.</p>
     */
    public static void startMonitor() {
        MemoryFrame frame = new MemoryFrame();
        frame.setSize(500, 300);
        frame.setVisible(true);
        frame.thread = new ThreadFrame(frame);
        frame.thread.start();
    }
```

We use a little Prolog text ‘memory.p’ to register the Java method as a foreign predicate. This Prolog text can be consulted from within the development environment. When doing so the JFreeChart libraries and the code for the class of this example have to be available. More details on what has to go into the class path can be found in the appendix. The main statement of the Prolog text reads as follows:

```
:- foreign(start_monitor/0, 'jekdev.study.memory.MemoryFrame',
startMonitor).
```

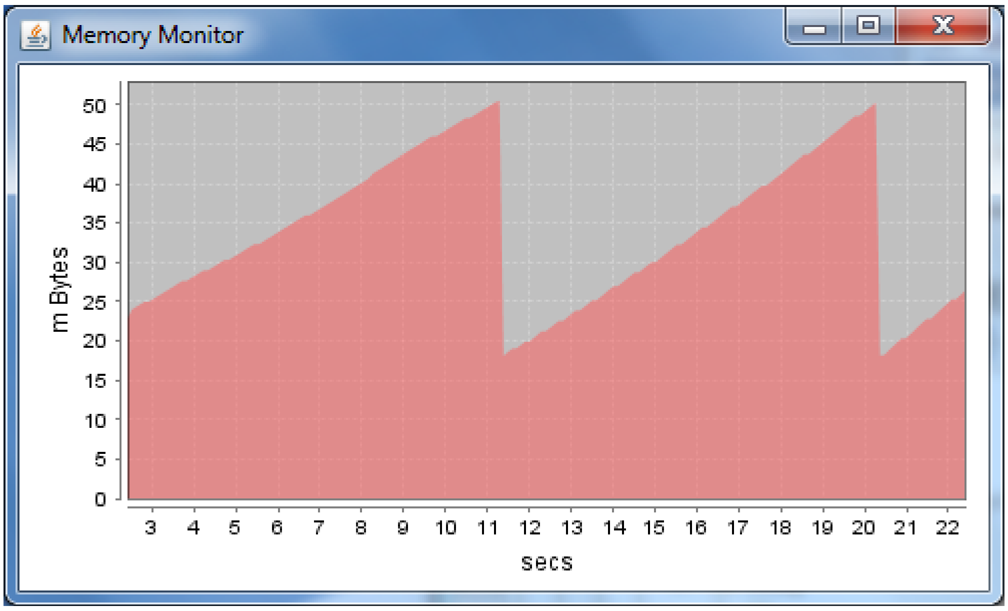
The Prolog text also registers a foreign predicate gc/0 that maps to the Java method System.gc(). We can experiment with calling explicitly the garbage collection. The Jekejeke Prolog system does in no place call the garbage collection, and it is not recommended to include explicit calls since the virtual machine might anyway ignore it.

Example Uses

The memory monitor can be invoked from within the development environment. The memory usage pattern highly depends on the JVM. Supposed the class path is correctly set, the easiest to start the memory monitor is to first consult the Prolog text ‘memory.p’. The memory monitor can then be invoked by the predicate start_monitor:

```
?- ['jekdev/study/memory/memory.p'].
Yes
?- start_monitor.
Yes
```

The typical memory usage pattern when the system is idle can be seen below. The system will consume memory objects by the frame thread itself. These memory objects will slowly eat up the memory until a larger garbage collection happens. The slope of the saw tooth depends on the effectiveness of the smaller garbage collections. The frequency of the saw tooth depends on the trigger condition for the larger garbage collection. Further 64-bit architectures might use more memory than 32-bit architectures.

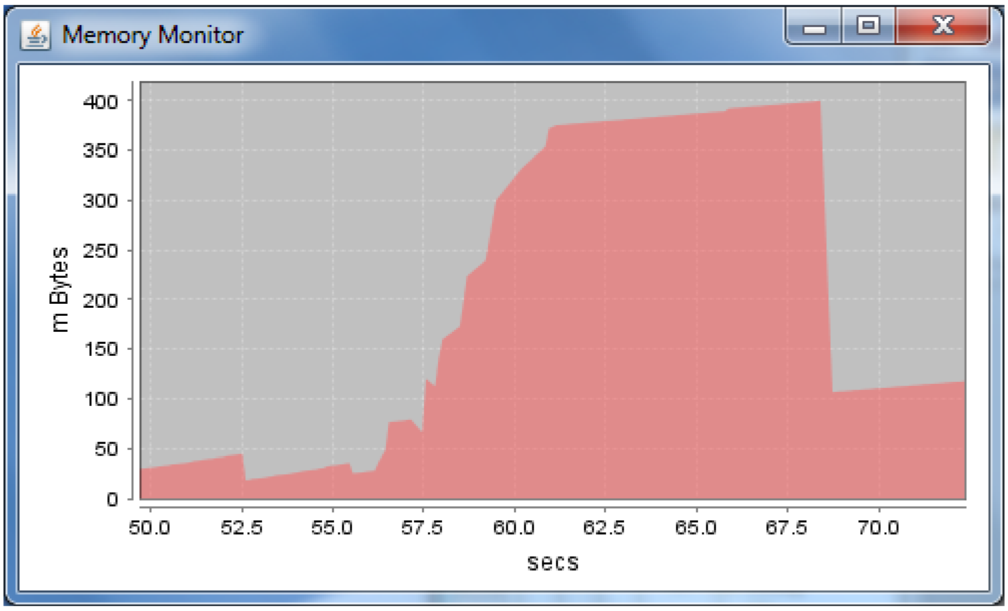


Picture 3: System Idle Memory Usage Pattern

We might now run a little test and verify the memory low alarm of Jekejeke Prolog. On both the Swing platform the runtime library and the development environment install a usage threshold. On the Android platform we have not yet figured out how to implement the threshold. We will run the factorial and look at the memory usage pattern. The threshold is 85% of the total memory. For a total memory of 512 m Bytes this amounts to 435 m Bytes.

We have run the following query:

```
?- fac(X,_).
X = n ;
X = s(n) ;
X = s(s(n)) ;
X = s(s(s(n))) ;
X = s(s(s(s(n)))) ;
X = s(s(s(s(s(n))))) ;
X = s(s(s(s(s(s(n)))))) ;
X = s(s(s(s(s(s(s(n))))))) ;
X = s(s(s(s(s(s(s(s(n)))))))) ;
X = s(s(s(s(s(s(s(s(s(n))))))))) ;
Error: Execution aborted since memory threshold exceeded.
      add / 3
      add / 3
      mul / 3
      mul / 3
      mul / 3
      mul / 3
      mul / 3
      mul / 3
      fac / 2
```



Picture 4: Memory Low Alarm Memory Usage Pattern

The above picture shows the memory usage pattern before and after the memory low alarm. All the queries for factorial do not use much more memory than 100 m Bytes up to the factorial of 8. The query for the factorial 9 then goes through the roof. Since the frame thread granularity is relatively low, we do not exactly see the peek that causes alarm. After the alarm the smaller garbage collections have already return some memory. But the system will remain with a high memory usage until a large garbage collection turns in.

In case there is no more use for the memory monitor the frame can simply be closed. It is also possible to first close the development environment. The memory monitor frame will then not automatically close, since we did not register with some event from the development environment. The JVM will only organically exit when all frames from the development environment have been closed and the memory monitor has been closed as well.

3 Integration Concepts

t,b.d.

- **Code Instrumentation:** t.b.d.

- **User Interfaces:** t.b.d.

3.1 Code Instrumentation

t.b.d.

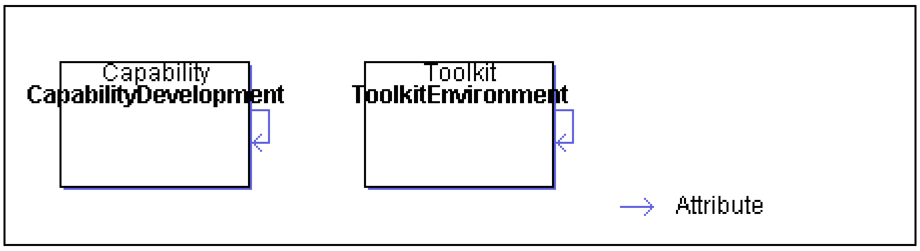
3.2 User Interfaces

t.b.d.

4 Headless API

The development environment enhances the core set of predicates and executes code with instrumentation. It is represented by a singleton.

This part of the API has the following class diagram:



Picture 5: Class Diagram Headless API

This part of the API consists of the following classes:

- Class ToolkitEnvironment
- Class CapabilityDevelopment

4.1 Class ToolkitEnvironment

The development environment toolkit provides a knowledge base and an interpreter that is instrumented for debugging. This class implements the singleton pattern. There is only one instance per Java class loader.

The class can be used to execute the development environment either with or without a graphical interface. For this purpose the main() method of the class can be statically called. The method will setup a knowledgebase and an interpreter, and then enter a query answer loop. When started without a graphical interface, the method will also install a ^C interrupt handler for the duration of the interactive session.

The Android version of the class ToolkitEnvironment does not provide a main() method and does not declare the constants for the command line options in its ToolkitLibrary class. Otherwise the class ToolkitEnvironment is identical to the Swing version.

Specific to the development environment is the command line option "-d". On the Swing platform, when a graphical user interface is used, this can be used to override the settings in the graphical user interface. The legal values for this flag are -1 for no thread monitor, 0 for a dynamic port of the thread monitor and a non-zero value for a static port.

The main() method for the Swing version recognizes the following options:

-h:
Don't create graphical user interface.
-v <level>:
Specify the console verbosity. The default value is “summary”.
-a <path>,...:
Add the paths <path>,....
-e <capability>,...:
Add the capabilities <capability>,....
-b <goal>:
Use <goal> as the console banner. The default value is “welcome”.
-t <goal>:
Use <goal> as the console top level. The default value is “prolog”.
-d <port>:
Use <port> as the server port for the thread monitor.

```
package jekdev.platform.headless;

import jekpro.tools.api.Toolkit;

public final class ToolkitEnvironment extends Toolkit {
    public static final ToolkitEnvironment DEFAULT;

    public final static String OPTION_DEBUG = "-d";
    public final static String PROP_SYS_MONITOR_CONFIG
                                = "sys_monitor_config";
    public final static String PROP_SYS_MONITOR_RUNNING
                                = "sys_monitor_running";

    public static void main(String[] args);
}
```

4.2 Class CapabilityDevelopment

The development environment capability provides the system predicates as defined in the language reference of the development environment. These system predicates enhanced the system predicates of the language reference of the runtime library. This class implements the singleton pattern. There is only one instance per Java class loader.

This capability is already predefined by a toolkit and it need not be added to the knowledge base after its initialization. This capability does need activation. It will not be useable without activation. The capability also declares some well-known interpreter properties.

```
package jekdev.platform.headless;

import jekpro.tools.api.Capability;

public final class CapabilityDevelopment extends Capability {
    public static final CapabilityDevelopment DEFAULT;

    public static final String PROP_DEBUG = "debug";
    public static final String PROP_SYS_TRACE_MODE = "sys_trace_mode";
    public static final String PROP_SYS_CLOAK = "sys_cloak";
}
```

5 Appendix Example Listings

The full source code of the Java classes and Prolog sources for the programming examples is given. We also give more details about compiling and executing the examples. The following source code has been included:

- Port Sampling
- Memory Monitor

5.1 Port Sampling

For the port sampling there are the following sources:

- **einstein.p:** The Prolog text that will be measured.
- **relative.p:** The Prolog text with the custom debugger hook.
- **RelativeAPI.java:** The Java methods for some statistics helpers.
- **thread.p:** The Prolog text with the thread foreign declarations.
- **ThreadAPI.java:** The Java methods for the thread objects.
- **ThreadHeadless.java:** The Java class for the sampling thread.

Prolog Text einstein

```
/**
 * Prolog code for the Einstein riddle.
 *
 * First published as "Who owns the Zebra?" in the Life International
 * magazine on December 17, 1962 with solution given in the March 25,
 * 1963 issue. The riddle given here is not a verbatim copy.
 *
 * Copyright 2012, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.3 (a fast and small prolog interpreter)
 */

member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

rightTo(L, R, [L,R | _]).
rightTo(L, R, [_ | Rest])
      :- rightTo(L, R, Rest).

nextTo(X, Y, List) :-
      rightTo(X, Y, List).
nextTo(X, Y, List) :-
      rightTo(Y, X, List).

einstein(Houses, FishOwner) :-
   Houses = [[house,norwegian,_,_,_,_],_,[house,_,_,_,milk,_],_,_],
   member([house,brit,_,_,_,red], Houses),
   member([house,swede,dog,_,_,_], Houses),
   member([house,dane,_,_,tea,_], Houses),
   rightTo([house,_,_,_,_,green], [house,_,_,_,_,white], Houses),
   member([house,_,_,_,coffee,green], Houses),
   member([house,_,bird,pallmall,_,_], Houses),
   member([house,_,_,dunhill,_,yellow], Houses),
   nextTo([house,_,_,dunhill,_,_], [house,_,horse,_,_,_], Houses),
   member([house,_,_,_,milk,_],Houses),
   nextTo([house,_,_,marlboro,_,_], [house,_,cat,_,_,_], Houses),
   nextTo([house,_,_,marlboro,_,_], [house,_,_,_,water,_], Houses),
   member([house,_,_,winfield,beer,_], Houses),
   member([house,german,_,rothmans,_,_], Houses),
   nextTo([house,norwegian,_,_,_,_], [house,_,_,_,_,blue], Houses),
   member([house,FishOwner,fish,_,_,_], Houses).

test :- einstein(_,_). * Prolog code for memory monitor.
```

Prolog Text relative

```
/**
 * Prolog code for the relative port statistics.
 *
 * The following data will be gathered:
 *    count(Fun, Arity, CallExitRedoFail).
 *
 * Copyright 2012, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.3 (a fast and small prolog interpreter)
 */

:- current_prolog_flag(source_file,X), set_source_property(X,sys_notrace).

:- foreign(format_float/3, 'jekdev.study.ports.RelativeAPI',
formatFloat('String','Double')).
:- foreign(nano_time/1, 'jekdev.study.ports.RelativeAPI', nanoTime).

% remove_count
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).
```

```
% sum_counts(+List,-CallExitRedoFail)
sum_counts([],0-0-0-0).
sum_counts([_-D|L],S) :-
    sum_counts(L,R),
    add_count(R,D,S).

% mem_counts(+Elem,+List)
mem_counts(X,[X|_]).
mem_counts(X,[_|Y]) :-
    mem_counts(X,Y).

% show
show :-
    findall(F/A-D,count(F,A,D),L),
    sum_counts(L,TR-TS-TT-TU),
    write('Pred\tCall\tExit\tRedo\tFail'), nl,
    keysort(L,M),
    mem_counts(I-(R-S-T-U),M),
    RP is R*100 / TR,
    SP is S*100 / TS,
    TP is T*100 / TT,
    UP is U*100 / TU,
    ((RP >= 0.05; SP >=0.05; TP >=0.05; UP >=0.05) ->
        write(I), write('\t'),
        format_float('0.0',RP,SR), write(SR), write(' %\t'),
        format_float('0.0',SP,SS), write(SS), write(' %\t'),
        format_float('0.0',TP,ST), write(ST), write(' %\t'),
        format_float('0.0',UP,SU), write(SU), write(' %'), nl; true),
    fail.
show.

% reset
reset :-
    remove_count.

% show(+Time)
show(T) :-
  format_float('0.0',T,U),
  write('\tin '),
  write(U),
  write(' ms'), nl.

% time(+Goal)
% Cannot be used to show time for redo/exit.
time(X) :-
  nano_time(T1),
  X,
  nano_time(T2),
  T is (T2-T1) / 1000000,
  write(X),
  show(T).
```

Java Class RelativeAPI

```
package jekdev.study.ports;

import java.math.BigInteger;
import java.text.DecimalFormat;

/**
```

```
 * <p>Java code for the API of the port sampler example.</p>
 * <p>This class provides an API to statistics helpers.</p>
 *
 * @author Copyright 2012, XLOG Technologies GmbH, Switzerland
 * @version Jekejeke Prolog 0.9.2 (a fast and small prolog interpreter)
 */
public final class RelativeAPI {

    /**
     * <p>Retrieve the nano time.</p>
     * <p>Java method called from Prolog.</p>
     *
     * @return The nano time.
     */
    public static BigInteger nanoTime() {
        return BigInteger.valueOf(System.nanoTime());
    }

    /**
     * <p>Format a double value.</p>
     * <p>Java method called from Prolog.</p>
     *
     * @param p The format specification.
     * @param d The double value.
     * @return The formatted value.
     */
    public static String formatFloat(String p, Double d) {
        DecimalFormat df = new DecimalFormat(p);
        return df.format(d);
    }

}
```

Prolog Text thread

```
/**
 * Prolog code for the sampling thread.
 *
 * Copyright 2012, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.3 (a fast and small prolog interpreter)
 */

:- current_prolog_flag(source_file,X), set_source_property(X,sys_notrace).

:- foreign(start_sampler/1, 'jekdev.study.ports.ThreadAPI',
startSampler('Interpreter')).
:- foreign(stop_sampler/1, 'jekdev.study.ports.ThreadAPI',
stopSampler('Term')).

:- thread_local handle/1.

start_sampler :-
  start_sampler(X),
  assertz(handle(X)).

stop_sampler :-
  retract(handle(X)),
  stop_sampler(X).
```

Java Class ThreadAPI

```
package jekdev.study.ports;

import jekpro.tools.api.*;

/**
 * <p>Java code for the API of the port sampler example.</p>
 * <p>This class provides an API to thread objects.</p>
 *
 * @author Copyright 2012, XLOG Technologies GmbH, Switzerland
 * @version Jekejeke Prolog 0.9.2 (a fast and small prolog interpreter)
 */
public final class ThreadAPI {
    private static final String OP_THREAD = "thread";
    private static final TermAtom ATOM_THREAD = new TermAtom(OP_THREAD);

    /**
     * <p>Start the sampler.</p>
     * <p>Java method called from Prolog.</p>
     *
     * @param inter The interpreter.
     * @return The output thread term.
     */
    public static Term startSampler(Interpreter inter) {
        ThreadHeadless thread = new ThreadHeadless(inter);
        thread.start();
        return wrapThread(inter, thread);
    }

    /**
     * <p>Stop the sampler.</p>
     * <p>Java method called from Prolog.</p>
     *
     * @param para The input thread term.
     * @throws InterpreterMessage Instantiation,
     *                            domain or permission error.
     */
    public static void stopSampler(Term para) throws InterpreterMessage {
        ThreadHeadless thread = (ThreadHeadless) unwrapThread(para);
        thread.interrupt();
    }

    /**
     * <p>Wrap a mutex object into the mutex term.</p>
     * <p>Will create a term of the following form:</p>
     * <pre>
     *     thread(ref)
     * </pre>
     *
     * @param inter The interpreter.
     * @param obj   The mutex object.
     * @return The mutex term.
     */
    private static TermCompound wrapThread(Interpreter inter, Thread obj) {
        return inter.createCompound(ATOM_THREAD,
                     new Term[]{new TermRef(obj)});
    }

    /**
     * <p>Unwrap a mutex object from the mutex term.</p>
     * <p>Will check whether the term has the following form:</p>
     * <pre>
     *     thread(ref)
     * </pre>
     *
     * @param para The stream term.
     * @return The stream object.
     * @throws jekpro.tools.api.InterpreterMessage
     *          Instantiation or domain error.
     */
    private static Thread unwrapThread(Term para)
                                throws InterpreterMessage {
        if (para instanceof TermVar)
            throw new InterpreterMessage(
                      InterpreterMessage.instantiationError());
        if (!(para instanceof TermCompound) ||
                ((TermCompound) para).getArity() != 1 ||
                !((TermCompound) para).getFunctor().equals(ATOM_THREAD))
            throw new InterpreterMessage(
                      InterpreterMessage.domainError(OP_THREAD, para));
        Term para2 = ((TermCompound) para).getArg(0);
        if (para2 instanceof TermVar)
            throw new InterpreterMessage(
                      InterpreterMessage.instantiationError());
        if (!(para2 instanceof TermRef) ||
                !(((TermRef) para2).getValue() instanceof Thread))
            throw new InterpreterMessage(
                      InterpreterMessage.domainError(OP_THREAD, para));
        return (Thread) ((TermRef) para2).getValue();
    }

}
```

Java Class ThreadHeadless

```
package jekdev.study.ports;

import jekdev.platform.headless.CapabilityDevelopment;
import jekpro.tools.api.Interpreter;
import jekpro.tools.api.InterpreterMessage;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * <p>The port sampler thread.</p>
 *
 * @author Copyright 2012, XLOG Technologies GmbH, Switzerland
 * @version Jekejeke Prolog 0.9.3 (a fast and small prolog interpreter)
 */
final class ThreadHeadless extends Thread {
    private static final long NANOS_PER_MILLI = 1000000;
    private static final long[] SLEEP = {NANOS_PER_MILLI / 2,
                                         2* NANOS_PER_MILLI};

    private Interpreter inter;
    private boolean on;
    private ReentrantLock lock = new ReentrantLock();
    private Condition cond = lock.newCondition();

    /**
```

```
     * <p>Create a port sampler thread.</p>
     *
     * @param i The interpreter.
     */
    ThreadHeadless(Interpreter i) {
        inter = i;
    }

    /**
     * <p>Constantly update the chart.</p>
     */
    public void run() {
        try {
            lock.lock();
            for (; ; ) {
                long lastTime = System.nanoTime();
                try {
                    inter.setStatus(
                      CapabilityDevelopment.STATUS_SYS_TRACE_MODE, on);
                } catch (InterpreterMessage x) {
                    throw new RuntimeException(x);
                }
                long sleep = SLEEP[on ? 1 : 0] -
                      (System.nanoTime() - lastTime);
                try {
                    while (sleep > 0)
                        sleep = cond.awaitNanos(sleep);
                } catch (InterruptedException x) {
                    return;
                }
                on = !on;
            }
        } finally {
            lock.unlock();
        }
    }

}
```

5.2 Memory Monitor

For the memory monitor there are the following sources:

- **memory.p:** The Prolog text with the foreign declaration.
- **MemoryFrame.java:** The Java class for the Swing frame.
- **MemoryJob.java:** The Java class for the event runnable.
- **MemoryPanel.java:** The Java class for the JFreeChart panel.
- **ThreadFrame.java:** The Java class for the monitor thread.

Prolog Text memory

```
/**
 * Prolog code for memory monitor.
 *
 * Copyright 2012, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.3 (a fast and small prolog interpreter)
 */

:- foreign(start_monitor/0, 'jekdev.study.memory.MemoryFrame',
startMonitor).
:- foreign(gc/0, 'jekdev.study.memory.MemoryFrame', gc).
```

Java Class MemoryFrame

```
package jekdev.study.memory;

import javax.swing.*;
import java.awt.event.WindowEvent;

/**
 * <p>The memory monitor frame.</p>
 *
 * @author Copyright 2012, XLOG Technologies GmbH, Switzerland
 * @version Jekejeke Prolog 0.9.3 (a fast and small prolog interpreter)
 */
public final class MemoryFrame extends JFrame {
    private ThreadFrame thread;
    private MemoryPanel panel;

    /**
     * <p>Create a memory monitor frame.</p>
     */
    private MemoryFrame() {
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                this_windowClosing(e);
            }
        });
        setTitle("Memory Monitor");
        panel = new MemoryPanel(1);
        getContentPane().add(panel);
    }

    /**
     * <p>Retrieve the memory panel.</p>
     *
     * @return The memory panel.
     */
    MemoryPanel getPanel() {
        return panel;
    }

    /**
     * <p>Handle window close event.</p>
     *
     * @param e The event.
     */
    private void this_windowClosing(WindowEvent e) {
        thread.interrupt();
        dispose();
    }

    /**
     * <p>Java method called from Prolog.</p>
     */
    public static void startMonitor() {
        MemoryFrame frame = new MemoryFrame();
        frame.setSize(500, 300);
        frame.setVisible(true);
        frame.thread = new ThreadFrame(frame);
        frame.thread.start();
    }

    /**
     * <p>Java method called from Prolog.</p>
     */
    public static void gc() {
        System.gc();
    }

}
```

Java Class MemoryJob

```
package jekdev.study.memory;

/**
 * <p>The memory monitor job.</p>
 *
 * @author Copyright 2012, XLOG Technologies GmbH, Switzerland
 * @version Jekejeke Prolog 0.9.3 (a fast and small prolog interpreter)
 */
final class MemoryJob implements Runnable {
    private double when;
    private double[] values;
    private MemoryFrame frame;

    /**
     * <p>Set the time.</p>
     *
     * @param n The time.
     */
    void setWhen(double n) {
        when = n;
    }
    /**
     * <p>Set the values.</p>
     *
     * @param v The values.
     */
    void setValues(double[] v) {
        values = v;
    }

    /**
     * <p>Set the memory frame.</p>
     *
     * @param f The memory frame.
     */
    void setFrame(MemoryFrame f) {
        frame = f;
    }

    /**
     * <p>Perform the chart job.</p>
     */
    public void run() {
        frame.getPanel().addValues(now, values);
    }

}
```

Java Class MemoryPanel

```
package jekdev.study.memory;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.DateAxis;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.data.time.FixedMillisecond;
import org.jfree.data.time.TimeSeries;
import org.jfree.data.time.TimeSeriesCollection;

import javax.swing.*;
import java.awt.*;

/**
 * <p>The memory monitor panel.</p>
 *
 * @author Copyright 2012, XLOG Technologies GmbH, Switzerland
 * @version Jekejeke Prolog 0.9.3 (a fast and small prolog interpreter)
 */
final class MemoryPanel extends JPanel {
    private static final int VALUES_LIMIT = 200;

    private final ChartPanel chartPanel;
    private int count;

    /**
     * <p>Add values to the chart.</p>
     *
     * @param when   The time point.
     * @param values The values.
     */
    void addValues(double when, double[] values) {
        JFreeChart chart = chartPanel.getChart();
        XYPlot plot = chart.getXYPlot();
        XYSeriesCollection dataset = (XYSeriesCollection)plot.getDataset();
        for (int i = 0; i < dataset.getSeriesCount(); i++) {
            XYSeries series = dataset.getSeries(i);
            series.add(when, values[i]);
        }
        count++;
        while (count >= VALUES_LIMIT) {
            for (int i = 0; i < dataset.getSeriesCount(); i++) {
                XYSeries series = dataset.getSeries(i);
                series.delete(0, 0);
            }
            count--;
        }
    }

    /**
     * <p>Create a free chart panel.</p>
     * <p>The layout is basically:</p>
     * <pre>
     *     +--+----------------------------------+
     *     |N |                                  |
     *     |uA|                                  |
     *     |mx|                                  |
     *     |bi|             XYArea               |
     *     |es|                                  |
     *     |r |                                  |
     *     +--+----------------------------------+
     *     |  |           NumberAxis             |
     *     +--+----------------------------------+
     * </pre>
     *
     * @param n The numnber of time series.
     */
    MemoryPanel(int n) {
        XYSeriesCollection dataset = new XYSeriesCollection();
        for (int i = 0; i < n; i++) {
            XYSeries series = new XYSeries(Integer.toString(i));
            dataset.addSeries(series);
        }

        JFreeChart chart = ChartFactory.createXYAreaChart(
                null,
                null,
                null,
                dataset,
                PlotOrientation.VERTICAL,
                false,
                false,
                false
        );

        XYPlot plot = chart.getXYPlot();
        NumberAxis domainAxis = new NumberAxis("secs");
        domainAxis.setAutoRangeIncludesZero(false);
        domainAxis.setLowerMargin(0);
        domainAxis.setUpperMargin(0);
        NumberAxis rangeAxis = new NumberAxis("m Bytes");
        plot.setDomainAxis(domainAxis);
        plot.setRangeAxis(rangeAxis);

        chartPanel = new ChartPanel(chart);
        setLayout(new BorderLayout());
        add(chartPanel, BorderLayout.CENTER);
    }

}
```

Java Class ThreadFrame

```
package jekdev.study.memory;

import javax.swing.*;

/**
 * <p>The memory monitor thread.</p>
 *
 * @author Copyright 2012, XLOG Technologies GmbH, Switzerland
 * @version Jekejeke Prolog 0.9.3 (a fast and small prolog interpreter)
 */
final class ThreadFrame extends Thread {
    private static final long NANOS_PER_MILLI = 1000000;
    private static final long SLEEP = 100 * NANOS_PER_MILLI;

    private final MemoryFrame frame;
    private ReentrantLock lock = new ReentrantLock();
    private Condition cond = lock.newCondition();

    /**
     * <p>Create a memory monitor thread.</p>
     *
     * @param f The memory frame.
     */
    ThreadFrame(MemoryFrame f) {
        frame = f;
    }

    /**
     * <p>Constantly update the chart.</p>
     */
    public void run() {
        long startTime = System.nanoTime();
        try {
            lock.lock();
            for (; ; ) {
                long lastTime = System.nanoTime();

                double[] values = new double[1];
                long total = Runtime.getRuntime().totalMemory();
                long free = Runtime.getRuntime().freeMemory();
                values[0] = (double) (total - free) / 1000000;
                int when = (int) ((lastTime - startTime +
                                NANOS_PER_MILLI / 2) / NANOS_PER_MILLI);

                MemoryJob job = new MemoryJob();
                job.setWhen((double) when / 1000);
                job.setValues(values);
                job.setFrame(frame);
                SwingUtilities.invokeLater(job);

                long sleep = SLEEP - (System.nanoTime() - lastTime);
                try {
                    while (sleep > 0)
                        sleep = cond.awaitNanos(sleep);
                } catch (InterruptedException x) {
                    return;
                }
            }
        } finally {
            lock.unlock();
        }
    }

}
```

Index

Pictures

Tables

References

[1]