Prolog/Java Bidirectional Interface

The CIAO System Documentation Series
Printed on: 21 April 2003
Version 1.9#78 (2003/4/21, 17:9:39 CEST)

Jesús Correas Fernández and The CLIP Group
clip@dia.fi.upm.es
http://www.clip.dia.fi.upm.es/
The CLIP Group
Facultad de Informática
Universidad Politécnica de Madrid

Table of Contents

Summary

This manual includes the complete reference to the low-level Prolog to Java interface. This interface allows a Prolog program to start a Java process and manipulate Java objects.

In this paper we present an interface between Ciao Prolog and Java to take advantage of the capabilities of the java programming language, avoiding problems related to compile-time linking, system dependencies, and static references. This interface fully achieves the objectives using a simple but powerful protocol between both languages. The communication is established by means of sockets, allowing the processes to be in separated machines, and thus providing a means for distributed processing.

This documentation corresponds to version 1.9#78 (2003/4/21, 17:9:39 CEST).

1 Introduction

The increasing diversity of platforms used today and the diffusion of Internet and the World Wide Web makes compatibility between platforms a key factor to run the software everywhere with no change. Java seems to achieve this goal, using a bytecode intermediate language and a large library of platform-dependent and independent classes which fully implements many. On the other hand, Prolog provides a powerful implementation of logic programming paradigm. This document includes the reference manual of the Prolog/Java bidirectional interface implemented in Ciao. In addition, it has been developed an application of this interface that makes use of an object oriented extension of Prolog to encapsulate the java classes, O'Ciao, both the ones defined in the JDK as well as new classes developed in Java. These classes can be used in the object oriented prolog extension of Ciao just like native O'Ciao classes.

The proposed interaction between both languages is realized as an interface between two processes, a Java process and a Prolog process, running separately. This approach allows the programmer to use of both Java and Prolog, without the compiler-dependent glue code used in other linkage-oriented approaches, and preserves the philosophy of Java as an independent language. The interface communication is based on a clean socket-based protocol, providing hardware and software independence. This allows also both processes to be run in different machines connected by a TCP/IP transport protocol, based on a client/server model that can evolve to a more cooperative model.

The present manual includes reference information about the Prolog side of the bidirectional Java/Prolog interface. The Java side of this interface is explained in the HTML pages generated by Javadoc.

1.1 Distributed Programming Model

The differences between Prolog and Java impose the division of the interface in two main parts: a prolog-to-java and a java-to-prolog interfaces. Most of the applications that will use this interface will consider that will be a "client' side that request actions and queries to a "server' side, which accomplish the actions and answer the queries. In a first approach, any of the both one-way interfaces implement a pure client/server model: the server waits for a query, performs the received query and sleeps until the next query comes; the client starts the server, carries out the initial part of the job initiating all the conversations with the server, and requests the server to do some things sometimes.

This model cannot handle correctly the tasks regarding an event oriented programming environment like java. A usual application of the prolog-to-java interface could be a graphical user interface server made in java, and a prolog client on the other side. A pure client/server model based on requests and results is not powerful enough to leave the prolog side managing all the application specific work of this example: some java specific stuff is needed to catch and manipulate properly the events thrown by the graphical user interface. This problem can be solved in a distributed context, on which both languages are clients and servers simultaneously, and can perform requests and do actions at a time. Using this model, the prolog side can add a prolog goal as listener of a specific event, and the java side launches that goal when the event raises.

In any case, the client/server approach simplifies the design of the interface, so both interfaces have been designed in such way, but keeping in mind that the goal is to reach a distributed environment, so each side do the things it is best designed for.

2 Prolog to Java interface

Author(s): Jesús Correas.

Version: 1.9#78 (2003/4/21, 17:9:39 CEST)

Version of last change: 1.9#67 (2003/3/14, 12:48:36 CET)

This module defines the Ciao Prolog to Java interface. This interface allows a Prolog program to start a Java process, create Java objects, invoke methods, set/get attributes (fields), and handle Java events.

This interface only works with JDK version 1.2 or higher.

Although the Java side interface is explained in Javadoc format (it is available at `library/javall/javadoc/` in your Ciao installation), the general interface structure is detailed here.

2.1 Prolog to Java Interface Structure

This interface is made up of two parts: a Prolog side and a Java side, running in separate processes. The Prolog side receives requests from a Prolog program and sends them to the Java side through a socket. The Java side receives requests from the socket and performs the actions included in the requests.

If an event is thrown in the Java side, an asynchronous message must be sent away to the Prolog side, in order to launch a Prolog goal to handle the event. This asynchronous communication is performed using a separate socket. The nature of this communication needs the use of threads both in Java and Prolog: to deal with the 'sequential program flow,' and other threads for event handling.

In both sides the threads are automatically created by the context of the objects we use. The user must be aware that different requests to the other side of the interface could run concurrently.

2.1.1 Prolog side of the Java interface

The Prolog side receives the actions to do in the Java side from the user program, and sends them to the Java process through the socket connection. When the action is done in the Java side, the result is returned to the user Prolog program, or the action fails if there is any problem in the Java side.

Prolog data representation of Java elements is very simple in this interface. Java primitive types such as integers and characters are translated into the Prolog corresponding terms, and even some Java objects are translated in the same way (e. g. Java strings). Java objects are represented in Prolog as compound terms with a reference id to identify the corresponding Java object. Data conversion is made automatically when the interface is used, so the Prolog user programs do not have to deal with the complexity of this tasks.

2.1.2 Java side

The Java side of this layer is more complex than the Prolog side. The tasks this part has to deal to are the following:

- Wait for requests from the Prolog side.
- Translate the Prolog terms received in the Prolog 'serialized' form to a more useful Java representation (see the Java interface documentation available at `library/javall/javadoc/` in your Ciao installation for details regarding Java representation of Prolog terms).
- Interpret the requests received from the Prolog side, and execute them.

- Handle the set of objects created by or derived from the requests received from de prolog side.
- Handle the events raised in the Java side, and launch the listeners added in the prolog side.
- Handle the exceptions raised in the Java side, and send them to the Prolog side.

In the implementation of the Java side, two items must be carefully designed: the handling of Java objects, and the representation of prolog data structures. The last item is specially important because all the interactions between Prolog and Java are made using Prolog structures, an easy way to standardize the different data management in both sides. Even the requests themselves are encapsulated using Prolog structures. The overload of this encapsulation is not significant in terms of socket traffic, due to the optimal implementation of the prolog serialized term.

The java side must handle the objects created from the Prolog side dinamically, and these objects must be accessed as fast as possible from the set of objects. The Java API provides a powerful implementation of Hash tables that achieves all the requirements of our implementation.

On the other hand, the java representation of prolog terms is made using the inheritance of java classes. In the java side exists a representation of a generic prolog term, implemented as an abstract class in java. Variables, atoms, compound terms, lists, and numeric terms are classes in the java side which inherit from the term class. Java objects can be seen also under the prolog representation as compound terms, where the single argument corresponds to the Hash key of the actual java object in the Hash table referred to before. This behaviour makes the handling of mixed java and prolog elements easy. Prolog goals are represented in the java side as objects which contain a prolog compound term with the term representing the goal. This case will be seen more in depth next, when the java to prolog is explained.

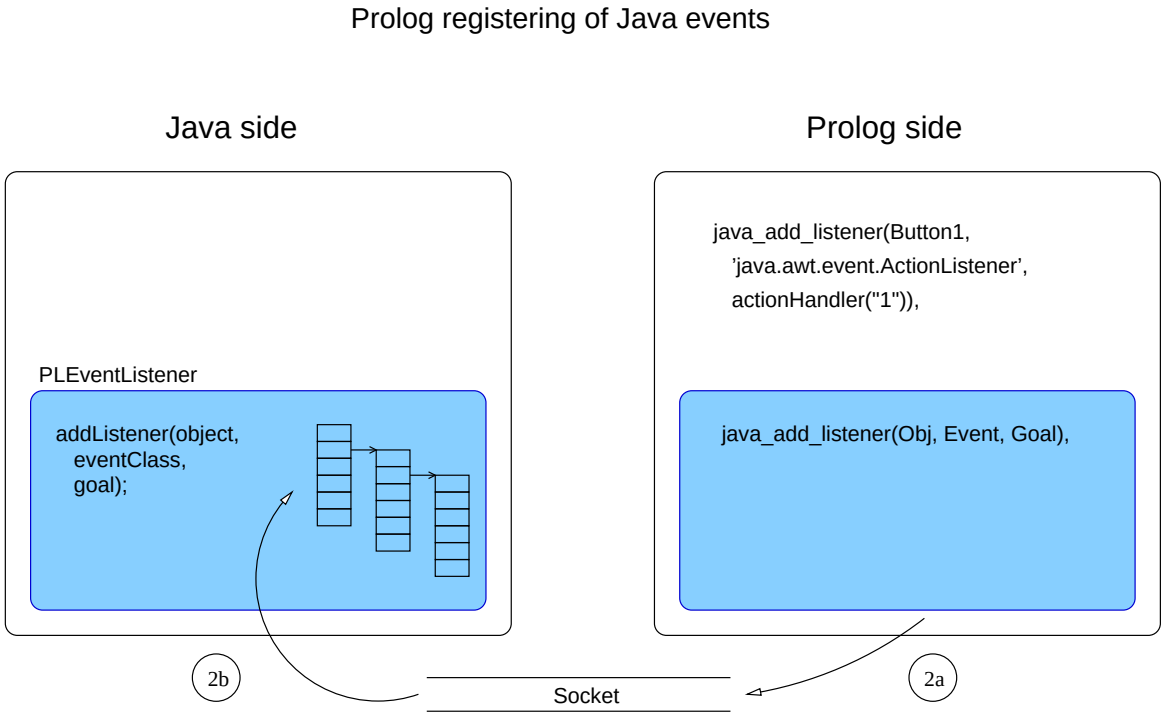
2.2 Java event handling from Prolog

Java event handling is based on a delegation model since version 1.1.x. This approach to event handling is very powerful and elegant, but a user program cannot handle all the events that can arise on a given object: for each kind of event, a listener must be implemented and added specifically. However, the Java 2 API includes a special listener (`AWTEventListener`) that can manage the internal java event queue.

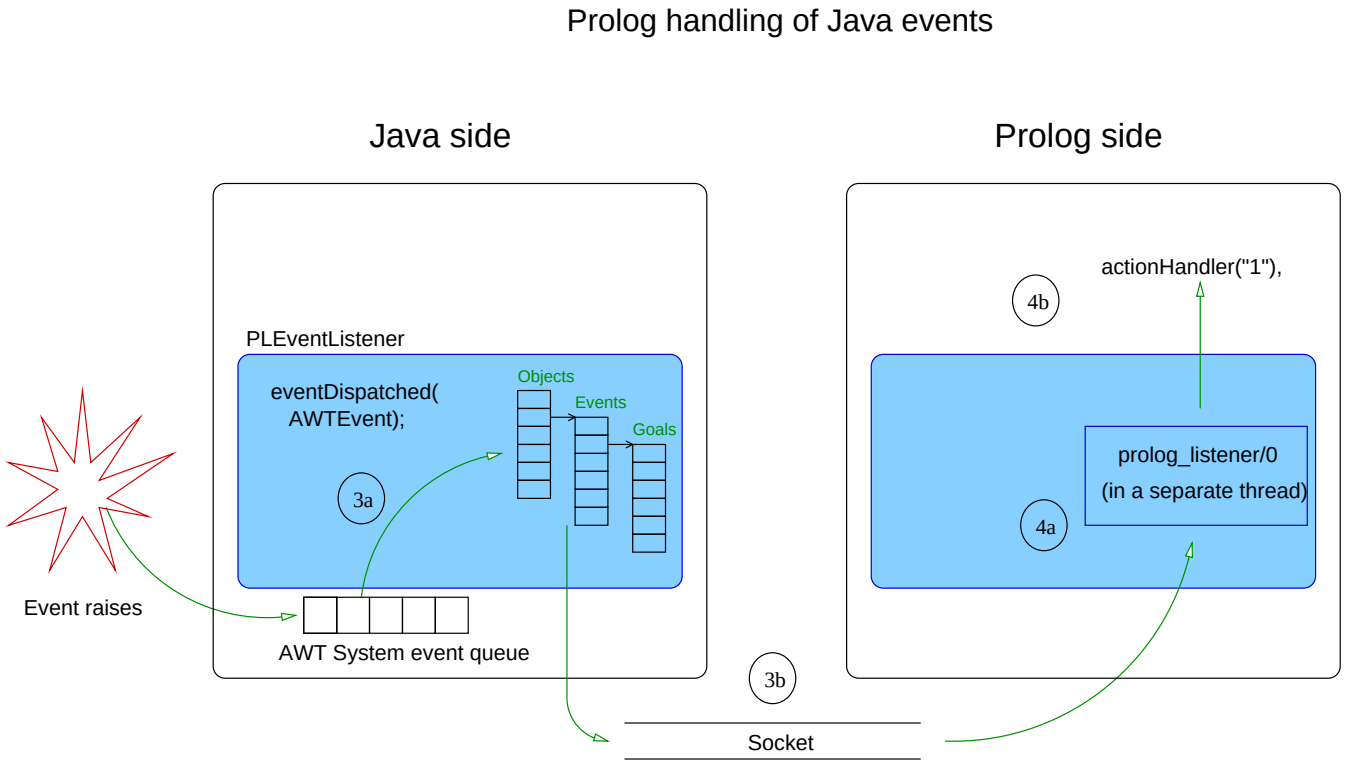
The prolog to java interface has been designed to emulate the java event handler, and is also based on event objects and listeners. The prolog to java interface implements its own event manager, to handle those events that have prolog listeners associated to the object that raises the event. From the prolog side can be added listeners to objects for specific events. The java side includes a list of goals to launch from the object and event type.

Due to the events nature, the event handler must work in a separate thread to manage the events asynchronously. The java side has its own mechanisms to work this way. The prolog side must be implemented specially for event handling using threads. The communication between java and prolog is also asynchronous, and an additional socket stream is used to avoid interferences with the main socket stream. The event stream will work in this implementation only in one way: from java to prolog. If an event handler needs to send back requests to java, it will use the main socket stream, just like the requests sent directly from a prolog program.

The internal process of register a Prolog event handler to a Java event is shown in the next figure:



When an event raises, the Prolog to Java interface has to send to the Prolog user program the goal to evaluate. Graphically, the complete process takes the tasks involved in the following figure:



2.3 Java exception handling from Prolog

Java exception handling is very similar to the peer prolog handling: it includes some specific statements to trap exceptions from user code. In the java side, the exceptions can be originated from an incorrect request, or can be originated in the code called from the request. Both exception types will be sent to prolog using the main socket stream, allowing the prolog program manage the exception. However, the first kind of exceptions are prefixed, so the user program can distinguish them from the second type of exceptions.

In order to handle exceptions properly using the prolog to java and java to prolog interfaces simultaneously, in both sides of the interface will be filtered those exceptions coming from their own side: this avoids an endless loop of exceptions bouncing from one side to another.

2.4 Usage and interface (javart)

- **Library usage:**
  `:- use_module(library(javart)).`
- **Exports:**
  - *Predicates:*
    `java_start/0`, `java_start/1`, `java_start/2`, `java_stop/0`, `java_connect/2`, `java_disconnect/0`, `java_use_module/1`, `java_create_object/2`, `java_delete_object/1`, `java_invoke_method/2`, `java_get_value/2`, `java_set_value/2`, `java_add_listener/3`, `java_remove_listener/3`.
  - *Regular Types:*
    `machine_name/1`, `java_constructor/1`, `java_object/1`, `java_event/1`, `prolog_goal/1`, `java_field/1`, `java_method/1`.
- **Other modules used:**
  - *System library modules:*
    `concurrency/concurrency`, `iso_byte_char`, `format`, `lists`, `read`, `write`, `javall/javasock`, `system`.
  - *Internal (engine) modules:*
    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

2.5 Documentation on exports (javart)

java_start/0: PREDICATE

Usage:

- *Description:* Starts the Java server on the local machine, connects to it, and starts the event handling thread.

java_start/1: PREDICATE

Usage: `java_start(+Classpath)`

– *Description:* Starts the Java server on the local machine, connects to it, and starts the event handling thread. The Java server is started using the classpath received as argument.
– *Call and exit should be compatible with:*
`+Classpath` is a string (a list of character codes). (`string/1`)

java_start/2: PREDICATE
Usage: `java_start(+machine_name,+classpath)`
– *Description:* Starts the Java server in machine_name (using rsh!), connects to it, and starts the event handling thread. The Java server is started using the classpath received as argument.
– *Call and exit should be compatible with:*
`+machine_name` is currently instantiated to an atom. (`atom/1`)
`+classpath` is a string (a list of character codes). (`string/1`)

java_stop/0: PREDICATE
Usage:
– *Description:* Stops the interface terminating the threads that handle the socket connection, and finishing the Java interface server if it was started using java_start/n.

java_connect/2: PREDICATE
Usage: `java_connect(+machine_name,+port_number)`
– *Description:* Connects to an existing Java interface server running in machine_name and listening at port port_number. To connect to a Java server located in the local machine, use 'localhost' as machine_name.
– *Call and exit should be compatible with:*
`+machine_name` is the network name of a machine. (`machine_name/1`)
`+port_number` is an integer. (`int/1`)

java_disconnect/0: PREDICATE
Usage:
– *Description:* Closes the connection with the java process, terminating the threads that handle the connection to Java. This predicate does not terminate the Java process (this is the disconnection procedure for Java servers not started from Prolog). This predicate should be used when the communication is established with java_connect/2.

machine_name/1: REGTYPE
Usage: `machine_name(X)`
– *Description:* `X` is the network name of a machine.

java_constructor/1: REGTYPE

Usage: `java_constructor(X)`

– *Description:* `X` is a java constructor (structure with functor as constructor full name, and arguments as constructor arguments).

java_object/1: REGTYPE

Usage: `java_object(X)`

– *Description:* `X` is a java object (a structure with functor '$java_object', and argument an integer given by the java side).

java_event/1: REGTYPE

Usage: `java_event(X)`

– *Description:* `X` is a java event represented as an atom with the full event constructor name (e.g., 'java.awt.event.ActionListener').

prolog_goal/1: REGTYPE

Usage: `prolog_goal(X)`

– *Description:* `X` is a prolog predicate. Prolog term that represents the goal that must be invoked when the event raises on the object. The predicate arguments can be java objects, or even the result of java methods. These java objects will be evaluated when the event raises (instead of when the listener is added). The arguments that represent java objects must be instantiated to already created objects. The variables will be kept uninstantiated when the event raises and the predicate is called.

java_field/1: REGTYPE

Usage: `java_field(X)`

– *Description:* `X` is a java field (structure on which the functor name is the field name, and the single argument is the field value).

java_use_module/1: PREDICATE

Usage: `java_use_module(+Module)`

– *Description:* Loads a module and makes it available from Java.

– *Call and exit should be compatible with:*

`+Module` is any term. (`term/1`)

java_create_object/2: PREDICATE

Usage: `java_create_object(+java_constructor,-java_object)`

– *Description:* New java object creation. The constructor must be a compound term as defined by its type, with the full class name as functor (e.g., 'java.lang.String'), and the parameters passed to the constructor as arguments of the structure.

– *Call and exit should be compatible with:*

`+java_constructor` is a java constructor (structure with functor as constructor full name, and arguments as constructor arguments). (`java_constructor/1`)

`-java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`java_object/1`)

java_delete_object/1: PREDICATE

Usage: `java_delete_object(+java_object)`

– *Description:* Java object deletion. It removes the object given as argument from the Java object table.

– *Call and exit should be compatible with:*

`+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`java_object/1`)

java_invoke_method/2: PREDICATE

Usage: `java_invoke_method(+java_object,+java_method)`

– *Description:* Invokes a java method on an object. Given a Java object reference, invokes the method represented with the second argument.

– *Call and exit should be compatible with:*

`+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`java_object/1`)

`+java_method` is a java method (structure with functor as method name, and arguments as method ones, plus a result argument. This result argument is unified with the atom 'Yes' if the java method returns void). (`java_method/1`)

java_method/1: REGTYPE

Usage: `java_method(X)`

– *Description:* `X` is a java method (structure with functor as method name, and arguments as method ones, plus a result argument. This result argument is unified with the atom 'Yes' if the java method returns void).

java_get_value/2: PREDICATE

Usage: `java_get_value(+java_object,+java_field)`

– *Description:* Gets the value of a field. Given a Java object as first argument, it instantiates the variable given as second argument. This field must be uninstantiated in the java_field functor, or this predicate will fail.

– *Call and exit should be compatible with:*

`+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`java_object/1`)

`+java_field` is a java field (structure on which the functor name is the field name, and the single argument is the field value). (`java_field/1`)

java_set_value/2: PREDICATE

Usage: `java_set_value(+java_object,+java_field)`

- *Description:* Sets the value of a Java object field. Given a Java object reference, it assigns the value included in the java_field compound term. The field value in the java_field structure must be instantiated.
- *Call and exit should be compatible with:*

  `+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`java_object/1`)

  `+java_field` is a java field (structure on which the functor name is the field name, and the single argument is the field value). (`java_field/1`)

java_add_listener/3: PREDICATE

Meta-predicate with arguments: `java_add_listener(?,?,goal)`.

Usage: `java_add_listener(+java_object,+java_event,+prolog_goal)`

- *Description:* Adds a listener to an event on an object. Given a Java object reference, it registers the goal received as third argument to be launched when the Java event raises.
- *Call and exit should be compatible with:*

  `+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`java_object/1`)

  `+java_event` is a java event represented as an atom with the full event constructor name (e.g., 'java.awt.event.ActionListener'). (`java_event/1`)

  `+prolog_goal` is a prolog predicate. Prolog term that represents the goal that must be invoked when the event raises on the object. The predicate arguments can be java objects, or even the result of java methods. These java objects will be evaluated when the event raises (instead of when the listener is added). The arguments that represent java objects must be instantiated to already created objects. The variables will be kept uninstantiated when the event raises and the predicate is called. (`prolog_goal/1`)

java_remove_listener/3: PREDICATE

Usage: `java_remove_listener(+java_object,+java_event,+prolog_goal)`

- *Description:* It removes a listener from an object event queue. Given a Java object reference, goal registered for the given event is removed.
- *Call and exit should be compatible with:*

  `+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`java_object/1`)

  `+java_event` is a java event represented as an atom with the full event constructor name (e.g., 'java.awt.event.ActionListener'). (`java_event/1`)

  `+prolog_goal` is a prolog predicate. Prolog term that represents the goal that must be invoked when the event raises on the object. The predicate arguments can be java objects, or even the result of java methods. These java objects will be evaluated when the event raises (instead of when the listener is added). The arguments that represent java objects must be instantiated to already created objects. The variables will be kept uninstantiated when the event raises and the predicate is called. (`prolog_goal/1`)

3 Java to Prolog interface

Author(s): Jesús Correas.

Version: 1.9#78 (2003/4/21, 17:9:39 CEST)

Version of last change: 1.9#65 (2003/3/14, 12:48:10 CET)

This module defines the Prolog side of the Java to Prolog interface. This side of the interface only has one public predicate: a server that listens at the socket connection with Java, and executes the commands received from the Java side.

In order to evaluate the goals received from the Java side, this module can work in two ways: executing them in the same engine, or starting a thread for each goal. The easiest way is to launch them in the same engine, but the goals must be evaluated sequentially: once a goal provides the first solution, all the subsequent goals must be finished before this goal can backtrack to provide another solution. The Prolog side of this interface works as a top-level, and the goals partially evaluated are not independent.

The solution of this goal dependence is to evaluate the goals in a different prolog engine. Although Ciao includes a mechanism to evaluate goals in different engines, the approach used in this interface is to launch each goal in a different thread.

The decision of what kind of goal evaluation is selected is done by the Java side. Each evaluation type has its own command terms, so the Java side can choose the type it needs.

A Prolog server starts by calling the `prolog_server/0` predicate, or by calling `prolog_server/1` predicate and providing the port number as argument. The user predicates and libraries to be called from Java must be included in the executable file, or be accesible using the built-in predicates dealing with code loading.

3.1 Usage and interface (`jtopl`)

- **Library usage:**
  `:- use_module(library(jtopl)).`
- **Exports:**
  - *Predicates:*
    `prolog_server/0`, `prolog_server/1`, `prolog_server/2`, `shell_s/0`, `query_solutions/2`, `query_requests/2`, `running_queries/2`.
- **Other modules used:**
  - *System library modules:*
    `concurrency/concurrency`, `system`, `read`, `write`, `dynamic`, `lists`, `format`, `compiler/compiler`, `atom2term`, `javall/javasock`, `prolog_sys`.
  - *Internal (engine) modules:*
    `internals`, `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

3.2 Documentation on exports (`jtopl`)

prolog_server/0: PREDICATE

Usage:

– *Description:* Prolog server entry point. Reads from the standard input the node name and port number where the java client resides, and starts the prolog server listening at the jp socket. This predicate acts as a server: it includes an endless read-process loop until the `prolog_halt` command is received.

However, from the low-level communication point of view, this Prolog server actually works as a client of the Java side. This means that Java side waits at the given port to a Prolog server trying to create a socket; Prolog side connects to that port, and then waits for Java requests (acting as a 'logical' server). To use this Prolog server as a real server waiting for connections at a given port, use `prolog_server/1`.

prolog_server/1: PREDICATE

Usage:

– *Description:* Waits for incoming Java connections to act as a Prolog goal server for Java requests.This is the only `prolog_server/*` predicate that works as a true server: given a port number, waits for a connection from Java and then serves Java requests. When a termination request is received, finishes the connection to Java and waits next Java connection request. This behaviour is different with respect to previous versions of this library. To work as before, use `prolog_server/2`.

Although it currently does not support simultaneous Java connections, some work is being done in that direction.

– *Call and exit should be compatible with:*

`Arg1` is an atom. (`atm/1`)

prolog_server/2: PREDICATE

Usage:

– *Description:* Prolog server entry point. Given a network `node` and a `port` number, starts the prolog server trying to connect to Java side at that `node:port` address, and then waits for Java requests. This predicate acts as a server: it includes an endless read-process loop until the `prolog_halt` command is received.

However, from the low-level communication point of view, this Prolog server actually works as a client of the Java side. This means that Java side waits at the given port to a Prolog server trying to create a socket; Prolog side connects to that port, and then waits for Java requests (acting as a 'logical' server). To use this Prolog server as a real server waiting for connections at a given port, use `prolog_server/1`.

– *Call and exit should be compatible with:*

`Arg1` is an atom. (`atm/1`)

`Arg2` is an atom. (`atm/1`)

shell_s/0: PREDICATE

Usage:

– *Description:* Command execution loop. This predicate is called when the connection to Java is established, and performs an endless loop processing the commands received. This predicate is only intended to be used by the Prolog to Java interface and it should not be used by a user program.

query_solutions/2: PREDICATE

No further documentation available for this predicate.

The predicate is of type *concurrent*.

query_requests/2: PREDICATE

No further documentation available for this predicate.

The predicate is of type *concurrent*.

running_queries/2: PREDICATE

No further documentation available for this predicate.

The predicate is of type *concurrent*.

4 Low-level Prolog to Java socket connection

Author(s): Jesús Correas.

Version: 1.9#78 (2003/4/21, 17:9:39 CEST)

Version of last change: 1.9#66 (2003/3/14, 12:48:24 CET)

This module defines a low-level socket interface, to be used by javart and jtopl. Includes all the code related directly to the handling of sockets. This library should not be used by any user program, because is a very low-level connection to Java. Use `javart` (Prolog to Java interface) or `jtopl` (Java to Prolog interface) libraries instead.

4.1 Usage and interface (`javasock`)

- **Library usage:**
  `:- use_module(library(javasock)).`
- **Exports:**
  - *Predicates:*
    `bind_socket_interface/1`, `start_socket_interface/2`, `stop_socket_interface/0`, `join_socket_interface/0`, `java_query/2`, `java_response/2`, `prolog_query/2`, `prolog_response/2`, `is_connected_to_java/0`, `java_debug/1`, `java_debug_redo/1`, `start_threads/0`.
- **Other modules used:**
  - *System library modules:*
    `fastrw`, `read`, `sockets/sockets`, `dynamic`, `format`, `concurrency/concurrency`, `javall/jtopl`, `sockets/sockets_io`.
  - *Internal (engine) modules:*
    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

4.2 Documentation on exports (`javasock`)

bind_socket_interface/1: PREDICATE

Usage: `bind_socket_interface(+Port)`

- *Description:* Given an port number, waits for a connection request from the Java side, creates the sockets to connect to the java process, and starts the threads needed to handle the connection.
- *Call and exit should be compatible with:*
  `+Port` is an integer. (`int/1`)

start_socket_interface/2: PREDICATE

Usage: `start_socket_interface(+Address,+Stream)`

- *Description:* Given an address in format 'node:port', creates the sockets to connect to the java process, and starts the threads needed to handle the connection.

– *Call and exit should be compatible with:*

`+Address` is any term. (`term/1`)

`+Stream` is an open stream. (`stream/1`)

stop_socket_interface/0: PREDICATE

Usage:

– *Description:* Closes the sockets to disconnect from the java process, and waits until the threads that handle the connection terminate.

join_socket_interface/0: PREDICATE

Usage:

– *Description:* Waits until the threads that handle the connection terminate.

java_query/2: PREDICATE

The predicate is of type *concurrent*.

Usage: `java_query(ThreadId,Query)`

– *Description:* Data predicate containing the queries to be sent to Java. First argument is the Prolog thread Id, and second argument is the query to send to Java.

– *Call and exit should be compatible with:*

`ThreadId` is an atom. (`atm/1`)

`Query` is any term. (`term/1`)

java_response/2: PREDICATE

The predicate is of type *concurrent*.

Usage: `java_response(Id,Response)`

– *Description:* Data predicate that stores the responses to requests received from Java. First argument corresponds to the Prolog thread Id; second argument corresponds to the response itself.

– *Call and exit should be compatible with:*

`Id` is an atom. (`atm/1`)

`Response` is any term. (`term/1`)

prolog_query/2: PREDICATE

The predicate is of type *concurrent*.

Usage: `prolog_query(Id,Query)`

– *Description:* Data predicate that keeps a queue of the queries requested to Prolog side from Java side.

– *Call and exit should be compatible with:*

`Id` is an integer. (`int/1`)

`Query` is any term. (`term/1`)

prolog_response/2: PREDICATE

The predicate is of type *concurrent*.

Usage: `prolog_response(Id,Response)`

- *Description:* Data predicate that keeps a queue of the responses to queries requested to Prolog side from Java side.
- *Call and exit should be compatible with:*

  `Id` is an integer. (`int/1`)

  `Response` is any term. (`term/1`)

is_connected_to_java/0: PREDICATE

Usage:

- *Description:* Checks if the connection to Java is established.

java_debug/1: PREDICATE

No further documentation available for this predicate.

java_debug_redo/1: PREDICATE

No further documentation available for this predicate.

start_threads/0: PREDICATE

Usage:

- *Description:* Starts the threads that will handle the connection to Java. This predicate is declared public for internal purposes, and it is not intended to be used by a user program.

References

Predicate/Method Definition Index

Property Definition Index

(Index is empty)

Regular Type Definition Index

J

M

P

Concept Definition Index