Survey on Different Prolog Implementations Libraries and tuProlog Socket Communication

Mirco Bordoni

Abstract

The aim of this paper is to explain every step I have gone through to complete the practical project of Computational Models and Languages LM.
I will start from a brief overview of the main libraries found in different Prolog implementations. In this way it is possible to have an idea of which are the functionalities not supported by tuProlog and that can be useful to implement. We will then focus on Socket Library, giving a deeper description of its structure in other Prologs and an analysis of how it can be implemented in *tuProlog*.
Then I will describe how I have implemented the socket support for *tuProlog* and the problems found.

1 Overview of Different Prolog Implementations Libraries

In the following I will list the libraries provided by four different Prolog implementations: GNU Prolog, SWI-Prolog, SICStus Prolog and Ciao Prolog. I am not going to show all of them, but only the noteworthy ones not implemented in tuProlog or that implement much more functionalities than the respective implemented in tuProlog. To have a complete list just have a look in the bibliography.

A brief discussion on these libraries and on the possible features for AI will follow these lists.

1.1 GNU Prolog

GNU Prolog is a free Prolog compiler including different libraries with great support for Finite Domain Constraint Solving.

Follows a list of the main libraries not implemented in tuProlog:

- *Finite Domain Constraint Solver*: used to solve Constraint Based Problems where a set of constraints and an objective function are given. Variables included in the problem have to be in a finite domain.
  Many Java libraries exist to solve this kind of problem, but none included in the official Oracle JDK.

- *Global Variables*: a global variable is a variable associated with each atom. It can be a copy of a term, a link to a term or an array of objects.

- *Operative System Interface*: interaction with File System, access to command line arguments, creation of processes and management of the interaction and communication via pipe etc.

- *Sockets I/O*: provides networking functionalities.

- *Interfacing Prolog to C*

1.2 SWI-Prolog

This implementation of Prolog is licensed under the Lesser GNU Public License. It comes with a really wide variety of libraries. As you will notice some of them does not carry a short description as often the name gives an idea of the functionalities.

- *Global and Attributed Variables*: as for GNUProlog a global variable in an association between a name and a term. Attributed variables are especially used in Constraint Logic Programming e.g. to specify domains

- *Constraint Handling Rules*: used to write constraint solvers

- *Multi-Threading*: this library is based on POSIX Thread Standard and let the user define multi-threaded application. Each thread has its own stack and only share the Prolog heap: predicates, records, flags etc.

- *Broadcast and Receive Events Notifications*: each Prolog agent can listen or generates events. This library was first introduced to realise GUI applications that use Prolog as a database.
- *Constraint Logic Programming over Finite Domains, Rationals and Reals*
- *Advanced Lists Manipulation*: this library provides many functions to access and manage lists
- *Non-Backtrackable Set*: defines Non-Backtrackable sets implemented as binary trees.
- *Command Line Parsing*: implements a way of building interactive applications through a command line parser.
- *Ordered Set Manipulation*: manipulation of ordered lists with unique elements
- *Operation on Key-Value lists*
- *Cross-Reference Data Collection*: used to track dependencies between files to reveal the structure of unknown programs or detect missing components at compile time.
- *Solve Linear Programming Problems (PLI)*
- *Resource Bounded Thread Management*: creation of pools with a limited number of threads to realise multi-threaded applications (e.g. web servers) to cope with limited resources infrastructures
- *Unweighted Graphs Modelling*
- *Analysing and Constructing URLs*

Additional packages are also available:

- *SWI-Prolog C Library*: this library provides a way to use C language from Prolog.
  - Unix Process Manipulation Library: implemented to manage processes

  - Process Library: provides processes creation and I/O redirection
  - Extended Operation on Files
  - User and Group management on Unix systems
  - Socket library: TCP and UDP inet-domain socket management to provide client and server-side communication
  - URI library: to process URIs and to provide high-performance C-based primitives for manipulating URIs.
  - CGI support library
  - MIME library: Parse MIME documents
  - Password Encryption Library
  - SHA1 and SHA2 Secure Hash Algorithms Library
  - Memory Files library: memory files are an alternative to Temporary Files and they are usually faster. They also do not suffer from security riscs and name conflicts
  - Time and Alarm library
  - Limiting Process resources library: provides an interface to the POSIX getrlimit()/setrlimit() API that control the maximum resource-usage of a process or group of processes.

- *HTTP Support*
  - HTTP Client Library: support to client-side http protocol
  - HTTP Server Library: library to support server-side http protocol. It is made of three parts, two of them obligatory and the last one optional. The first part deals with connection management, the second implements a generic wrapper for deconding the http request, calling user code and encode the answer. The optional part can be used to assign http locations to predicates
  - Transfer encoding: message encoding support
  - JSON support

- *Natural Language Processing Primitives*: this library provides some well known basic routines for natural language processing and information retrieval.

- *ODBC Interface*: provides an interface to the Microsoft DataBase Connectivity. It is made of two layers: the first encapsulates ODBC core functions (connection, queries...) and the second one exploits relations between Prolog predicates and database tables to provide a natural Prolog view of the data. The last part has not been implemented yet

- *C++ Interface to SWI-Prolog*: it provides an interface between SWI-Prolog and C++ even if not all functionalities of C interface have been implemented

- *SWI-Prolog Source Documentation*: implements a source code documentation infrastructure called PlDoc, loosely based on JavaDoc. It can create HTML+CSS and LaTeX documentation files

- *Prolog Unit Tests*: this library provides a support to automatic testing of sotware written in Prolog

- *Google's Protocol Buffers Library*: Protocol Buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data

- *RDF Parser*: provides a support to convert RDF-XML documents to triples and viceversa.

- *SWI-Prolog Semantic Web Library*: a library for dealing with standards from the W3C for the Semantic Web (RDF, OWL, XML etc.)

- *SWI-Prolog SGML/XML Parser*

- *SWI-Prolog Spatial Indexing*: SWI-Prolog interface to Spatial Index and GEOS libraries, providing spatial indexing of URI's. Supports import and export to GML, KML, and RDF with GeoRSS Simple, GeoRSS GML, and W3C WGS84 vocabulary properties

- *SWI-Prolog SSL Interface*: provides secure socket communication to Prolog applications. It can also be used to provide authentication and secure data exchange between Prolog processes over the network

1.3 SICStus Prolog

SICStus Prolog is a proprietary software and comes with a big quantity of libraries. Some of them are similar to the libraries I have listed above, so I will only mention them without explanation.

- *Debugging Features*: many debugging features are supported, like visualization of control flow, trace the program with spypoints, setting advice-points to carry out some actions at certain point of execution indipendently of the tracing activity, other control and information options
- *C/C++ Bidirectinal Support*: with this library Prolog can use C or C++ and viceversa
- *Java and .NET Interfacing library*
- *Aggregation Operator for Data-Base-Style Queries*: provides aggregation operator like in SQL queries, but for Prolog database.
- *Association Lists*: creates non balanced binary trees of association lists, that is trees of key-value pairs
- *Attribute Variables*: provides a mean of associating arbitrary attributes to variables
- *AVL Trees*: it is used to create "association lists" balanced binary trees
- *Bags or Multisets*: this library is used to create sets of elements
- *External Storage of Terms (Berkeley DB)*: this library module handles storage and retrieval of terms on files. By using indexing, the store/retrieve operations are efficient also for large data sets. The package is an interface to the Berkeley DB toolset
- *Accessing Files and Directories*: advanced operations on files and directory
- *List Operations*
- *Array Operations*: this libary module provides extendible arrays with logarithmic access time

- *The Object Package*: this library enables programmers to write object-oriented programs in SICStus Prolog. A class is a collection of encapsulated data and methods, accessible via message passing.
- *ODBC Interface Library*
- *Ordered and Unordered Set Operations*: operations on sets represented as ordered/unordered lists. Ordered lists have no duplicates.
- *Process Management*
- *Queue Operations*: this module provides an implementation of queues and many operations on them
- *Generic Sorting*: sorting of lists
- *Socket I/O*
- *Updatable Binary Tree*: management and usage of binary trees with logarithmic access time
- *Unweighted/Weighted Graph Operations*: operations on directed unweighted/weighted graphs
- *Parsing and Generating XML*
- *Process Communication*: Linda-like communication, based on server and client library implementation
- *Constraint Handling Rules*
- *Constraint Logic Programming over Finite Domain, Booleans, Rationals and Reals*
- *Jasper Interface*: bidirectional interface between SICStus Prolog and Java
- *Plunit Interface*: Prolog unit-test framework

1.4 CIAO Prolog

CiaoProlog has a GPL license and has several basic libraries:

- *Assertions*: this is a noteworthy functionality. It gives to the user the chance of writing assertions on Prolog code to describe predicates, properties, modules and applications. This assertions may be used in Unit Test declarations defining preconditions, postconditions and global properties. Furthermore, assertions are used to automatically generate software documentation.
- *Parse and Return Command Line Options*: this library provides support to the parsing of command line specified arguments while the program is started
- *Socket Interface*
- *Operating System Interface*

Many Extension Libraries are available:

- *Extended Multi-Threading Support*
- *Delaying Predicates*: executing predicate when given conditions hold
- *Active Module library*: provides an high-level model of inter-process communication and distributed execution. An active module is essentially a daemon: it is started at the operating system level. Communication with active modules is implemented using sockets
- *Basic Support to Agents Definition*: based on Active Modules
- *Breadth-First Execution*
- *Depth First Iterative-Deepening Execution*
- *Constraint Programming over Rationals and Reals*
- *Support to Fuzzy Logic*
- *Object-Oriented Programming*

Many Other Libraries contributed by users of Ciao system are available:

- *Interfaces with other Languages and Systems* like C, Sockets, Tcl/tk, Web Languages (http, html, xml, etc.), SQL/ODBC systems, Java, emacs
- *Constraint Programming over Finite Domains*
- *Automatic Tester*
- *XDR Handle library*
- *XML Query library*

1.5 AI Useful Libraries

As shown by previous paragraphs, many libraries are available in the Prolog implementations explored. Artificial Intelligence is a multidisciplinary research area and it can take advantage of a broad range of functionalities.

Some frameworks widely used in AI already exist, e.g. ECLiPSe. It implements a powerful Constraint Solver but, as we saw in the previous lists, almost every implementation of Prolog provides a library for Constraint Programming over Finite Domains. This is a feature that could be useful to add in tuProlog. I have been looking for an official Java library implementing Constraint Solver, but it seems there is not. Some libraries are provided only by users, like JACOP, JACK etc.

Of course Prolog mechanisms, like backtracking, can be really useful for AI. For example to implement meta-heuristics algorithms, expert systems, planning, informed and uninformed search or, more in general, tree search algorithms. Using tree structures in Java is quite easy and some basic implementations already exist.

In this survey we are interested in libraries, so we should focus on what kind of useful functions it is possible to implement in Java+Prolog. For AI some features that can be implemented are:

- Environments to provide collaboration between Prolog programs, e.g. Centralised or Distributed Tuple Centers, Networking etc.
- Custom Data Structures and their usage
- Introduction of special operators to model graphs or relations between states

Many other features of course could be realised.

2 Socket Communication in Prolog

All the implementations of Prolog previously analysed provide a support for communication via Socket, obviously with different characteristics.

GNU Prolog and SWI-Prolog implement a similar socket communication library. The idea is to map 1:1 each predicate with the respective C function. Both use streams to execute inputs and outputs on the sockets.

2.1 GNU Prolog Socket I/O

Follows a list of the predicates implemented in GNU Prolog and a brief description:

- *socket(+Domain, -SocketId)*: Socket creation
- *socket_close(+SocketId)*: an active socket should be eventually closed with this predicate
- *socket_bind(+SocketId, +Socket_Address)*: binds the socket identified by SocketId to a specific address
- *socket_connect(+SocketId, +Socket_Address, -StreamIn, -StreamOut)*: connect a socket to a specified address, obtaining two streams, one to read data from the socket and the other to send data through it
- *socket_listen(+SocketId, +Length)*: this predicate is used to set the number of maximum pending connections (Length) for a specific socket
- *socket_accept(+SocketId, -Client, -StreamIn, -StreamOut)*: accept a connection to socket SocketId obtaining the address of the connecting client, a stream for input and another for output. This is a blocking operation

As shown from this list, networking with GNU Prolog is quite simple and uses really a few predicates.

Streams are used to exchange data and they are the same used for files, pipes, devices etc.

To cope with blocking predicates there is only one way: using different processes. GNU Prolog does not support multi-threading, but it implements a library to create new processes in a C-like fashion.

Our purpose is to implement networking in tuProlog with Java. For this reason we do not have to worry about different processes, because multi-threading is fully supported.

2.2 SWI-Prolog Socket I/O

SWI-Prolog provides a more advanced socket library, focusing more on client and server sides, implementing a basic support for UDP messages too.

An important feature provided by SWI-Prolog is the chance of creating multiple threads executing with their own control flow. Each thread has its own stack and only shares the Prolog heap: predicates, records, flags and other global data.

The main predicates defined for networking are:

- *tcp_socket(-SocketId)*: create a new socket identified by SocketId
- *tcp_close_socket(+SocketId)*: close an opened socket identified by SocketId
- *tcp_open_socket(+SocketId, -InStream, -OutStream)*: given a socket identifier, this predicates provides both Input and Output Streams to communicate through the socket
- *tcp_bind(+Socket, ?Port)*: bind the socket to a specific port on the current machine. If port is unbound the system binds the socket to a free port
- *tcp_listen(+Socket, +Backlog)*: set the number of maximum allowed pending connections for the specified socket
- *tcp_accept(+Socket, -Slave, -Peer)*: this predicate waits for a client connection request to the socket. On success, it creates a new socket for the client and binds the identifier to Slave. Peer is bound to the IP-address of the client.
- *tcp_connect(+Socket, +Host:+Port, -Read, -Write)*: Client-interface to connect a socket to a given Port on a given Host. Port is either an integer or the name of a service
- *tcp_connect(+Socket, +Host:+Port)*: same as the previous one, but to use the socket we have to call tcp_open_socket/3

Server Example

```
create_server(Port) :-
        tcp_socket(Socket),                 %Opening Socket
        tcp_bind(Socket, Port),             %Bind it to a Port
        tcp_listen(Socket, 5),              %Maximum pending connections
                                            %set to 5
        tcp_open_socket(Socket, AcceptFd, _), %Opening an Input
                                                %stream on the socket
        <dispatch>                          %user-defined behaviour
```

The predicate *tcp_open_socket/3* opens an Input Stream on the Socket.

The wait for an incoming connection is implemented with a blocking predicate, *tcp_accept/3*. To cope with this waiting the suggested ways are: starting a new thread or a new process (only in UNIX Systems).

```
dispatch(AcceptFd) :-
        tcp_accept(AcceptFd, Socket, _Peer), %wait for an incoming connection
        thread_create(process_client(Socket, Peer), _,  %Create Thread
                      [ detached(true)
                      ]),
        dispatch(AcceptFd).   %back to waiting for a connection

process_client(Socket, Peer) :-     % Thread Behaviour
        setup_call_cleanup(tcp_open_socket(Socket, In, Out),
                           handle_service(In, Out),
                           close_connection(In, Out)).

close_connection(In, Out) :-        % Close streams
        close(In, [force(true)]),
        close(Out, [force(true)]).

handle_service(In, Out) :-
        ...
```

In the example above the predicate *dispatch/1* iteratively waits for a new incoming connection. Once received it creates a new thread with goal *process_client/2* and goes back to a new connection waiting.
Each thread executes its behaviour defined through the meta-call predicate

setup_call_cleanup(Setup,Goal,Cleanup) that calls (once(Setup), Goal). If Setup succeeds, Cleanup will be called exactly once after Goal is finished. So, first of all, Socket's Input and Output Streams are retrieved and then some kind of service is executed. When finished, the connection is closed.

Client Example

```
create_client(Host, Port) :-
        setup_call_catcher_cleanup(tcp_socket(Socket),
                                   tcp_connect(Socket, Host:Port),
                                   exception(_),
                                   tcp_close_socket(Socket)),
        setup_call_cleanup(tcp_open_socket(Socket, In, Out),
                           chat_to_server(In, Out),
                           close_connection(In, Out)).

close_connection(In, Out) :-
        close(In, [force(true)]),
        close(Out, [force(true)]).

chat_to_server(In, Out) :-
        ...
```

In this example a Client connects to a Server and starts chatting with it. To setup the connection a meta-call predicate is used: *setup_call_catcher_cleanup(Setup, Goal,Catcher,Cleanup)*. It is quite similar to the *setup_call_cleanup/3* we saw before, but Cleanup is called only if Catcher unifies with the termination code. So in our example a new socket is created and then connected to the server. *tcp_close_socket/1* is called only if an exception is raised.

If the connection succeded, a *setup_call_cleanup/3* is called to retrieve the Streams, chat with the server and then close the connection.

2.3 SICStus Prolog

In SICStus Prolog there is not a direct mapping between predicate and socket functions. Each predicate encapsulates more than one function. The main predicates provided are:

- *socket_client_open(+Addr, -Stream, +Options)*: open a new client socket and try to connect to a specified address. The result is a Stream that can be used to communicate over the socket.
  For example to create a binary stream to some web server *www.sics.se*:

  ```
  ?- socket_client_open('www.sics.se':80, Stream, [type(binary)]).
  ```

- *socket_server_open(?Addr, -ServerSocket, +Options)*: create a Server Socket and bind it to the specific address where it will be listening.

- *socket_server_accept(+ServerSocket, -Client, -Stream, +StreamOptions)*: it is a blocking predicate and waits for a connection to ServerSocket. When a client connects to this socket, its address will be bound to Client and a Stream is created. By default the stream is binary

- *socket_server_close(+ServerSocket)*: close the server socket ServerSocket and stop listening on its port

- *socket_select(+ServerSockets,-SReady, +ReadStreams,-RReady, +WriteStreams,-WReady, +Timeout)*: check for server sockets with incoming connections, streams on ReadStreams ready for input, and streams on WriteStreams ready for output. The streams can be any kind of streams, they need not to be socket streams. The ready server sockets are returned (in the same order) in SReady, the ready input streams in RReady, and the ready output streams in WReady. An input (output) stream is ready for input (output) when an item can be read (written) without blocking

In SICStus Prolog there is not a clear support for multi-threading like we saw in SWI-Prolog. For this reason documentation does not provide suggestions on the best practices for socket management.

2.4 CIAO Prolog

CIAO Prolog implements a Socket I/O library really similar to a mix of the previous we explored. It only provides two more predicates to write to and read from the stream associated to the socket.

3 Socket I/O in tuProlog analysis

We have now explored several Socket libraries and the time to think about the same library for tuProlog has come.

What I suggest in this paper is to take the best from the previous libraries and try to bring it in Java, which of course is a really powerful environment to use sockets and multi-threading.

So my idea is to create a library implementing a TCP Client-Server model like in SICStus Prolog, adding the chance to create threads to avoid unnecessary idles.

A possible interface to implement could be:

```
public interface ISocketLib {
    public boolean tcp_socket_client_open_2(Struct Address, Term
    Socket);

    public boolean tcp_socket_server_open_3(Struct Address, Term
    Socket, Struct Options);

    public boolean tcp_socket_server_accept_3(Term ServerSock,
    Term Client_Addr, Term Client_Slave_Socket);

    public boolean tcp_socket_server_close_1(Term serverSocket);

    public boolean read_from_socket_3(Term Socket, Term Msg,
    Struct Options);

    public boolean write_to_socket_2(Term Socket, Term Msg);

    public boolean aread_from_socket_2(Term Socket, Struct Options);
}
```

The main operations provide a way to:

- create a Client socket and connect to the Server socket bound to an address passed in the form *Address:Port*
- create a Server socket bound to address expressed by *Address:Port*

- accept a connection to a Server socket, resulting in a client address and a slave socket associated to the client. This should be a blocking predicate
- close sockets
- read from and write to sockets. In tuProlog the concept of stream is already implemented, but it seems it does not provide blocking predicates, or at least it fails if nothing is read. What I need is a blocking predicate that waits until something is received from the stream associated to a socket.
- asynchronous read from a socket if the user wants to avoid blocking predicates

To work with sockets it might be useful to implement new typed Terms. An UML model of new types is in Figure 1. AbstractSocket is the abstract

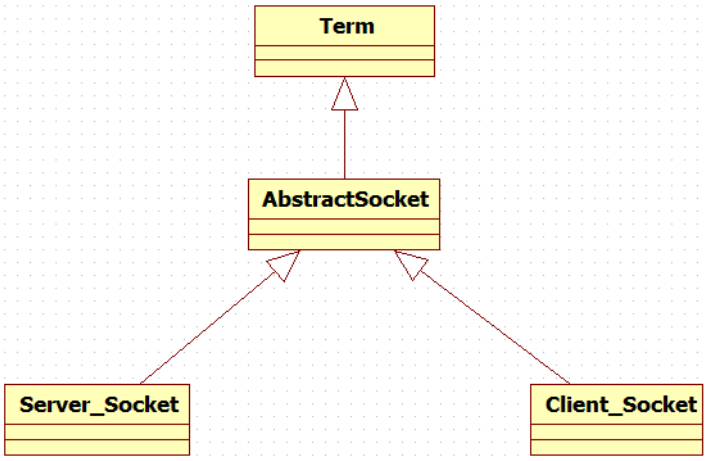

Figure 1: New Typed Terms

class defining a socket. In Java there are two kinds of socket: ServerSocket and Socket. As the name suggests, the first is used to define a Server socket, while the second defines a Client socket. For this reason AbstractSocket has to be specialized by Server_Socket and Client_Socket.

A multi-threading support has to be introduced if we want to avoid useless waitings. A good approach to multi-threading might be the one introduced

in SWI-Prolog, where a new thread is started with a given goal to execute. *tuProlog* engine is thread safe and many methods are executed as atomic, so it becomes quite difficult to run different control flows sharing the same engine, theory and flags. In fact if we try to solve two goals using the same engine we observe that the second goal is not solved until the first is completely solved.

An immediate solution could be running a new engine with the same theory of the caller giving a goal to solve. In this way a new thread behaves as a new process, given that it does not share any kind of information with its creator.

To implement a really multi-threading support, Prolog engine has to be changed and an analysis of its behaviour is needed. What is especially needed is to find where mutual exclusion is really required.

I tried to explore Prolog engine and I found that what is really important for multiple flows is to share the same theory. In fact, each of them has to carry its own binding map and its own execution context because they are solving different goals. For this reason I believe that to enhance a safe multi-threading support we may consider to change the way each flow access the theory, making those parts thread safe.

I will not explore this issue because it may take longer than the estimated time for this project.

4 tuProlog Socket Library Implementation

The implementation of a library for *tuProlog* consists of a class providing the desired predicates implemented as methods, with a well defined signature.

I have indeed implemented the class *SocketLib* that extends the basic class *Library* and implements the interface *ISocketLib* seen before.

The library has been implemented with the client-server model in mind and it focuses on connection oriented communication (using TCP as transport protocol).

4.1 New Types

New typed Terms can be really useful to manage sockets, so I decided to implement the same types shown in Figure 1.

This means I have created three new classes: *AbstractSocket*, *Server_Socket* and *Client_Socket*.

AbstractSocket extends *Term* and overrides its main methods. It then declare four new abstract methods:

```
- public abstract boolean isClientSocket();

- public abstract boolean isServerSocket();

- public abstract Object getSocket();

- public abstract InetAddress getAddress();
```

These four abstract operations will be implemented in *Server_Socket* and *Client_Socket*.

Once defined the needed typed terms I have proceeded with the actual library implementation.

4.2 Server Side

The server side main functionalities are:

- `tcp_socket_server_open(+Address, -Socket, +Options)`
- `tcp_socket_server_accept(+ServerSock, -Client_Addr, -Client_Slave_Socket)`
- `tcp_socket_server_close(+ServerSocket)`

To create a new Server_Socket the user has to call the predicate *tcp_socket_server_open/3* and must specify a binding address in the form *Address:Port*. If the socket has been succesfully created, the second argument passed to the predicate will be unified with the new *Server_Socket* Term created; for this reason it has to be unbound.

Actually three parameters can be passed to the creation predicate. The last one is a list of Options. By now the only option recognized is the maximum queue length for incoming connection, that is the backlog number. This parameter can be passed in the options list through *backlog(n)*. If no backlog is specified, it is set by default to 0, which means the queue length is infinite.

The predicate *tcp_socket_server_accept/3* is used to accept a connection. This is a blocking predicate and when it succeeds it unifies *Client_Addr* with the client address in the form *Address:Port* and *Client_Slave_Socket* with the new socket created to communicate with that client.

If some problem happens while using the previous two predicates they fail, so an appropriate behaviour should be executed in the prolog program.

The third predicate is used to close a Server_Socket and the Term *Server-Socket* passed has to be bound with a Server_Socket.

4.3 Client Side

On the client side a user can open a socket that automatically connect to the specified address, where a server socket should be listening.

The predicate used to open a new Client_Socket is:

- `tcp_socket_client_open(+Address, -Socket)`

The argument *Address* has to be in the form *Address:Port* and it is the address of the server listening for connection requests.

If the connection succeeds the variable *Socket* is unified with the new *Client_Socket* created.

4.4 Communication

Once a connection is established two sockets can communicate using the following predicates:

- `write_to_socket(+Socket, -Msg)`
- `read_from_socket(+Socket, -Msg, +Options)`
- `aread_from_socket(+Socket, +Options)`

The first predicate is used to write a message in the OutputStream associated to the *Socket* passed (it has to be a Client_Socket). If the operation succeeds the predicate is valued as true, otherwise it fails.

The second predicate is used to read a message from a *Socket* that should be bound to a *Client_Socket*. If it succeeds the received message is unified with variable *Msg*. This is a blocking predicate, but the user can specify a timeout. If nothing is read until the timeout expires, the predicate fails. To set a timeout an option *timeout(millis)* has to be passed through the list *Options*.

The third predicate implements the asynchronous read from the specified socket that, as in the previous predicate, has to be bound to a *Client_Socket*.

When something is read it is asserted in the Prolog theory using the method *assertA* of the object *TheoryManager* whose reference is get through the engine method *getTheoryManager()*.
The user can specify two options for this predicate: *timeout(millis)* and *assertZ*. If the timeout expires and nothing is read, nothing is asserted. If the *assertZ* option is specified, what is read from the socket will be asserted through the method *assertZ*, instead of *assertA*.

The implementation of the predicate *aread_from_socket/3* involves the creation of a thread associated to each socket. The class that implements this thread is included in the SocketLib and it is called *Reader*. When a Prolog program executes a predicate *aread_from_socket/3*, a new Reader is started or notified (if it was already created and started before). It then waits for a new message and assert it, if no problems happen.
In this way the main control flow is uncoupled from the one waiting for a new message, avoinding useless loss of time.

If a *read_from_socket/3* is executed while a Reader is still waiting to read data from the same socket, it fails because it makes no sense to contemporary read from the same socket. The only way to cope with this behaviour is to use timeouts.

4.5 Thread Safety in Theory Management

The implementation of asynchronous read in SocketLib involves some Thread Safety issues; in fact, as shown before, the predicate *aread_from_socket/3* uses the methods *assertA* and *assertZ* of the class TheoryManager. These methods are not thread safe because they are not syncronized and clauses are stored in two maps, both of the type *ClauseDatabase*. This class extends *HashMap* and, with all its features, it inherits its weaknesses. In fact, as written in the Java 7 Documentation, its implementation is not synchronized.

To grant that no race conditions happen, every method of the class *TheoryManager* has to be synchronized, so it can provide mutual exclusion in theory access.

The access to the theory is always done using *TheoryManager* methods, so if they are declared synchronized, thread safety can be granted. During the solving of a goal, the main engine executes each predicate associating it to a predefined *State* that can be of several types. Specifically, during the rule selection in the class *StateRuleSelection*, the method *find* implemented in *EngineManager* is called to select next rule to apply. This method is not

synchronized, but it calls the method *find* of the class *TheoryManager* that is synchronized, and this ensures that a consistent clause is read.

4.6 Basic Multi-Threading support

During the implementation of the Socket Library I implemented a basic support to multi-threading using a new engine to solve a goal passed by the user. When the new engine is created, its theory is set to the theory taken from the main engine. It is like a C-style fork where the code of the two processes is the same, but they do not share anything. Of course this is only a basic support because a lot of work has to be done in this field.

The only predicate I have implemented is:

```
create_thread(+Goal, -Id, +Options)
```

Where Goal is the goal to solve.

4.7 Changes in tuProlog source code

There are some changes I have made in *tuProlog* source code.

In the previous paragraph I have described what I changed in the class *TheoryManager* to grant thread safety.

Another change is in the method *unify* of the class *Var*. In this method there is a check on the type of the Term passed for the unification process. If the term is an *AbstractSocket* it has to succeed, so the block:

```
}else if (!(t instanceof Number)) {
    return false;
}
```

has been changed into:

```
}else if (!(t instanceof Number) && !(t instanceof AbstractSocket)) {
    return false;
}
```

Since new threads are created when asynchronous read are used, it is necessary to destroy them when the execution of a goal is finished. For this reason in *SocketLib* the method *onSolveEnd()* has been implemented and its task is to stop every waiting *Reader* and close all the *Server_Socket*s and

*Client_Socket*s still opened. The method *onSolveEnd()* is inherited from the class *Library*.

When a user stops the engine while solving a goal, the *EngineManager* calls the method *Engine.mustStop()* to stop it. This method sets to true a boolean variable declared in the class *Engine* and called *mustStop*. The method *onSolveEnd()* is not executed, so the threads started are not stopped and all the sockets still opened are not closed. To cope with this behaviour I have added to the class *LibraryManager* a method called *onSolveHalt()* and to the class *Library* a method with the same name *onSolveHalt()*. This last method is then overridden in the library *SocketLib* and it simply calls *onSolveEnd()* to close all sockets and stop all threads.
So now the *EngineManager* calls the method *Engine.mustStop()*, as seen before, and then it calls the method *LibraryManager.onSolveHalt()* that calls the method *onSolveHalt()* on every loaded library. *LibraryManager.onSolveHalt()* is a new method of the class *LibraryManager* implemented only for this purpose.
So the method *solveHalt()* in the *EngineManager* becomes:

```
public void solveHalt() {
    env.mustStop();
    libraryManager.onSolveHalt();
}
```

while before it was:

```
public void solveHalt() {
    env.mustStop();
}
```

Figure 2 shows what happens when the Stop button is pressed, starting from the listener of the Stop button events.

To support multi-threading I had to change the method *getTheory()* of the class Prolog. In fact, when a new thread is created, it has to get the theory from the main engine that started it, using the method *getTheory*. This method calls *TheoryManager.getTheory()* and is declared synchronized. This means that when a thread tries to get the theory from the main engine, it remains blocked until the main engine completed its execution (return from method solve). If this method is not declared synchronized the newly started thread can immediately read its theory and thread safety is assured, given that *TheoryManager.getTheory()* remains synchronized.

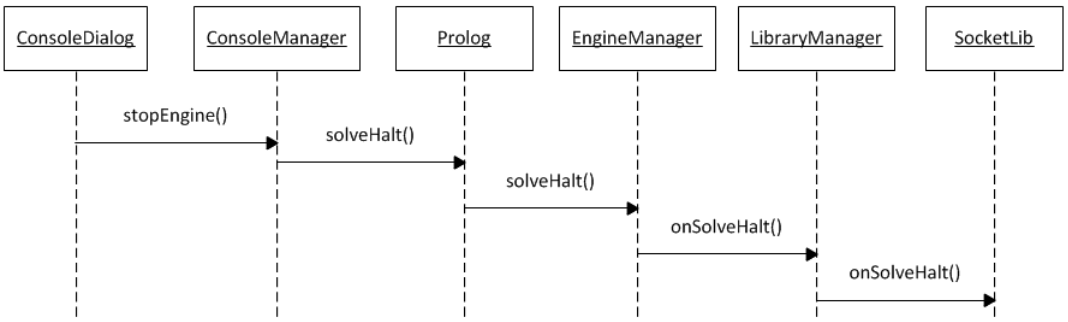

Figure 2: Halt Event Sequence Diagram

4.8 Tests and Examples

To test the correctness of the implemented library I have run several tests during the library implementation and I found that everything is working as expected.

To show some examples and contemporary test the library I have implemented a couple of Prolog programs.

The first one simply tests synchronous message exchange between a client and a server. The client sends three messages ("test1","test2","test3") and waits for the server response to each message ("echo(test1)","echo(test2)","echo(test2)"). This is the client code:

```
client(X,Y,Z):-tcp_socket_client_open('127.0.0.1:4444',Sock),
               write_to_socket(Sock,test1),
               read_from_socket(Sock,X,[]),
               write_to_socket(Sock,test2),
               read_from_socket(Sock,Y,[]),
               write_to_socket(Sock,test3),
               read_from_socket(Sock,Z,[]).
```

While the server code is:

```
server(X,Y,Z):- tcp_socket_server_open('127.0.0.1:4444', Sock,[]),
                tcp_socket_server_accept(Sock,ClientAddr,Slave),
                read_from_socket(Slave,X,[]),
                write_to_socket(Slave,echo(X)),
                read_from_socket(Slave,Y,[]),
                write_to_socket(Slave,echo(Y)),
```

```
                read_from_socket(Slave,Z,[]),
                write_to_socket(Slave,echo(Z)),
                tcp_socket_server_close(Sock).
```

As you can see the client starts a connection and then begins to exchange messages with the server. The server opens a ServerSocket, waits for a connection and starts exchanging messages with the client.

The second example is a *Superserver* that iteratively waits for a connection and, when received, starts a new thread to manage a service request (computation of a factorial). I have used the library *ThreadLib* that provides a really basic support to multi-threading.

The *Superserver* code is:

```
start:- tcp_socket_server_open('127.0.0.1:4444', Sock,[]),
        superserver(Sock).

superserver(Sock):- tcp_socket_server_accept(Sock,ClientAddr,Slave),
                    create_thread(server(ClientAddr,Slave),_,[]),
                    superserver(Sock).

fact(0,1).
fact(1,1).

factorial(X,Y):- fact(X,Y),!.
factorial(X,Y):- X1 is X-1, factorial(X1,Y1),Y is X*Y1.

server(ClientAddr,Slave):- read_from_socket(Slave,Msg,[]),
                           factorial(Msg,Res),
                           write_to_socket(Slave,Res),
                           checkNewReq(Slave).

checkNewReq(Slave):- end,retract(end),!.
checkNewReq(Slave):- newReq(N), retract(newReq(N)),
                     factorial(N,Res),
                     write_to_socket(Slave,Res),
                     aread_from_socket(Slave,[]),
                     checkNewReq(Slave).
checkNewReq(Slave):- aread_from_socket(Slave,[]),
                     checkNewReq(Slave).
```

When a connection is received a new thread is created with the goal *server(ClientAddr,Slave)*. This predicate first waits for a synchronous factorial request and, when the request is received and served, it calls the predicate *checkNewReq(Slave)* that iteratively waits other asynchronous requests checking if the predicate *newReq(N)* has been asserted, until the fact *end* is asserted.

An example of client that ask for the calculation of the factorial of 4 and 6 is:

```
client(Msg,Msg1):- tcp_socket_client_open('127.0.0.1:4444',Sock),
                   write_to_socket(Sock,6),
                   read_from_socket(Sock,Msg,[]),
                   write_to_socket(Sock,newReq(4)),
                   read_from_socket(Sock,Msg1,[]),
                   write_to_socket(Sock,end).
```

The last example is only a proof of concept; of course multi-threading support has to be expanded.

5 Future Work

Some future work that can be done on this library can be:

- Implementation of UDP socket communication.
- Extend Multi-Threading Support. For example it may be interesting to develop a multi-threading library that runs different threads sharing the same engine or to react to some kind of events using different threads. By now it is not even possible to see the output of a thread.

References

[1] GNUProlog Manual, *http://www.gprolog.org/manual/gprolog.html*.

[2] SWI-Prolog Documentation *http://www.swi-prolog.org/pldoc/index.html*

[3] SICStus Prolog Documentation *http://www.sics.se/sicstus/docs/latest4/html/sicstus.html/*

[4] CIAO Prolog Documentation *http://ciaohome.org/docs/branches/1.14/13646/CiaoDE-1.14.2-13646_ciao.html/ciao.html*