UNIVERSIDADE DE ÉVORA

GNU Prolog to Java

A study on how to connect the two programming environments

Dissertação de

Mestrado em Inteligência Artificial Aplicada

Orientador: Prof. Salvador Pinto de Abreu

Por: David José Murteira Mendes

"Esta dissertação não inclui as críticas e sugestões feitas pelo júri"

Évora, Abril de 2004

UNIVERSIDADE DE ÉVORA

GNU Prolog to Java

A study on how to connect the two programming environments

Dissertação de

Mestrado em Inteligência Artificial Aplicada

Orientador: Prof. Salvador Pinto de Abreu

Por: David José Murteira Mendes

"Esta dissertação não inclui as críticas e sugestões feitas pelo júri"

Évora, Abril de 2004

Resumo

Neste trabalho pretende-se efectuar o estudo e a implementação de uma interface bidireccional entre o GNU-Prolog e a linguagem Java. O objectivo desta ferramenta é dar a possibilidade de utilizar o poder da programação em lógica dentro de um ambiente multi-plataforma. Pretende-se possibilitar a escrita de programas Prolog que invocam métodos Java e programas Java que chamam predicados Prolog. Java e Prolog são um par ideal para disponibilizar úteis "aplicações inteligentes", com interfaces actuais, disponibilizadas para diferentes sistemas operativos existentes. Misturada com o Java esta "inteligência" beneficia das características desta linguagem como independência de plataformas, segurança, tratamento de excepções e segurança de tipos entre outras. Uma ligação destas possibilita a criação de ferramentas tais como servidores para diagnóstico de problemas, aplicações robot que se deslocam automatizadamente na Internet, agentes inteligentes móveis que atendem pedidos de outros agentes com capacidade de raciocínio sejam humanos ou não. Um dos objectivos primários, no entanto, que se pretende atingir é a integração de uma implementação Prolog compatível ISO e open source com todos os ambientes de desenvolvimento e ferramentas Java disponíveis actualmente. Como tal a ligação pretendida é feita a nível muito baixo para poder apresentar características de alta performance, flexibilidade e portabilidade.

Abstract

This work is intended to study and put up a bidirectional interface between GNU-Prolog and the Java language.The purpose of this tool is the possibility to use the power of logic programming within a cross platform environment.This meaning to write prolog programs invoking java methods and Java programs calling prolog predicates. Java and Prolog are an ideal pair for delivering useful intelligent applications with state-of-the-art user interfaces deployed over several operating systems and media. Mixed with Java this "intelligence" benefits from all of the design characteristics of this language like platform independence, security, type safety, exception handling, and so on, to create such tools as servers for diagnosing problems, spider and robot applications that transparently wander the net, mobile intelligent agents attending requests from other reasoning agents, human or not. One of the primary objectives, though, intended to be achieved is to integrate a full blown, ISO Prolog compliant, open source Prolog with the many IDEs and tools in the Java momentum.

Contents

List of Figures

List of Tables

Chapter 1

Acknowledgments

For the whole first part of this work Miguel Calejo has to be acknowledged for his page served has a foundation for all the Internet search about Prolog and Java mingled systems. My whole family has to be blessed for putting up with my bad temper for all these lengthy months. Special acknowledgment addressed to Fasoft, Lda. for funding one entire year of absence. Thanks should also go DONALD KNUTH and LESLIE LAMPORT, who developed the wonderful TEX and LATEX typesetting packages. Last but not least my Professor Salvador Pinto Abreu that showed me the appropriate directions for the work to be done and helped a lot when a hit into some wall appeared.

Chapter 2

Introduction

The purpose of this work is to create a link between GNU-Prolog and the Java language. The reasons for using GNU Prolog are summarized in the following features taken from the Free Software Foundation's GNU-Prolog page[1]

- GNU Prolog is a free Prolog compiler with constraint solving over finite domains.
- GNU Prolog accepts a Prolog+constraint program and produces a native binary (like gcc does from a C source).
- The obtained executable is then stand-alone.
- The size of this executable can be quite small since GNU Prolog can avoid to link the code of most unused built-in predicates.
- The performances of GNU Prolog are very encouraging (comparable to commercial systems).
- Beside the native-code compilation, GNU Prolog offers a classical interactive interpreter (top-level).
- The Prolog part conforms to the ISO standard for Prolog with many extensions very useful in practice (global variables, OS interface, sockets,...).
- GNU Prolog also includes an efficient constraint solver over Finite Domains (FD). This opens contraint logic programming to the user combining the power of constraint programming to the declarativity of logic programming.

Mainly aimed at prolog developers, it will allow Java method calling and field accessing, both class and instance fields and methods from GNU-Prolog and simultaneously predicate calling and term accessing from Java. All the appealing features of Java like the for instance:

- Object Oriented behavior

[1] http://www.gnu.org/software/prolog

- Platform Independence
- Internet protocols friendliness
- Easy UI development

shall be taken into account and be implemented as far as possible. The work development has been divided in three different phases:

- **State-of-the-art overview**
  - Presentation of the existing implementations
  - Study of the different technologies involved
- **Specification of the interface**
  - GNU-prolog APIs and the foreign interface as defined by Daniel Diaz [2]
  - Java Native Interface (JNI)
  - Interface prototype with an interpreter with access to java methods
- **Implementation**
  - Fully implement the prototype with both way communication
  - Some toy examples to demonstrate the usability

Chapter 3

Overview of previous work

There are mainly four alternatives to the implementation of a Java-Prolog interface:

- Native interface
- Object serialization interface
- JNI-based interface
- Prolog engines in Java

The previous work will be presented here ordered by the way they are implemented using one of these alternatives. Other systems are not considered as java-prolog interfaces for they are not full prolog (Edinburgh, ISO,...) but rather some prolog like languages or inference engines. These were not minimally studied and they are only referenced below as hybrid systems for those really interested. The following have different levels of approaching whether because it was somehow not interesting or not enough interesting information was found.

3.1 Native interface

Native interfaces are obviously those showing best performance when the underlying prolog machines are still good performers. Overhead from integrating different levels in architecture is minimized.These implementations suffer, however, from a design problem that aparts their interest from a major concerning factor that rules the present work: openness and platform independence.

3.1.1 Amzi

Built upon a technology named a 'Logic Server' that allows Prolog components to be easily integrated with other applications/environments. It is a library written in C/C++ that can be linked to any application that can call a dynamic library. The Logic Server includes the ability to reason over databases and data

from clients/servers, plus it can be easily extended to provide any additional interfaces or capabilities. The Java implementation includes a main 'LogicServer' class that encapsulates a Prolog engine and its API, and an 'LSException' class used for error handling. They are both included in a Java package, 'amzi.ls'. The LogicServer class includes all the methods that give the developer full control over the Prolog engine. These include methods to:

- setup, initialize, reset and close the Prolog engine
- load and/or consult Prolog programs
- issue Prolog queries
- assert/retract Prolog terms
- convert between Java strings and Prolog terms
- get values into Java variables from Prolog atoms, strings and numbers
- build and decompose Prolog lists
- build and decompose Prolog structures
- return information about errors

There is the LSException class that has no methods and is simply used to signal and catch errors. To use the LogicServer class you import the amzi.ls package into your Java program. From there you can either instantiate a new LogicServer object and invoke its methods, or you can define a new class that extends the LogicServer class adding new methods and variables. The Amzi! Logic Server provides tools that let you implement your own extended predicates. These allow you to write Prolog code that directly accesses anything Java can access. The Java methods that implement extended predicates, must be declared as returning type boolean, and as public. They can be added one at a time using the API function AddPred, which adds a single predicate at a time. If your extended predicate is in a package, then the package name must be included in the class name, delimited by forward slashes, to AddPred as follows:

ls.AddPred("extpred", 1, "javapkg/jprolog", "extpred", this);

3.1.2 New Features

Amzi! 6.2 is the final release so far. Some of the new release characteristics are:

- Robust Server Architecture
- Easier Development and Deployment
- ISO-Standard

- High Performance Logic Base
- Comprehensive Internet Support for .NET, ASP, JSP and Java Servlets
- New Libraries and Utilities
- Full Internationalization
- Increased System Limits
- Improved Documentation
- Advanced Mathematics

With the following details:

- ISO module support
- Indexed and sorted storage options for dynamic clauses
- New predicates for examining logic base
- COM object for .NET support with samples
- ASP, JSP and Java Servlet samples
- Cross-reference and syntax checking utility
- XML library
- Date/time library
- Performance probe utility
- Debugging predicate ?/1 built-in
- ISO stream support
- New Prolog flags, many settable by directives
- loadlsx/1 allows dynamic loading of LSXs
- ensure_loaded and include directives
- double_quote_strings, upper_case_atoms alternate syntaxes
- Locale-specific multi-byte characters supported
- Either slash (or
) can be used in file paths
- tilt_slashes/2 adjusts path delimiters for a platform

- copy_term/2 built-in for fast copies
- load/4 allows loading .plm images from memory
- Unlimited atom table
- Maximum variables in a clause up to 4095
- Maximum arity of a clause up to 4095
- No limit on the number of clauses in a compiled predicate
- Infinite precision real numbers optionally available
- Mathematical tools
- New internal garbage collector
- Chez Ray samples, a compendium of advanced Prolog tricks

3.2 Object serialization interface

Serialization is the standard way of passing objects between different virtual machines in Java. Any class that implements the java.io.Serializable interface is a candidate for serialization. This technique is present since the JDK 1.1 and is very well understood and used exhaustively by Java programmers. Should there be good ways to perform some kind of "serialization" in Prolog and this forms a well-founded technique to achieve inter-language communication. Definite Clause Grammars is a widely known methodology in the Prolog community and is thus a strong candidate for the serialization technique for a grammar can parse/construct some kind of object/method representation. With a good mechanism to pass the serialized objects (marshal and unmarshaling of objects and method calling) between both the Prolog and Java machines it appears to be an efficient way of communication between languages.

3.2.1 InterProlog (XSB)

InterProlog is implemented as a set of standard Java classes and Prolog predicates. It provides a simple facelift to Prolog, by running it under a separate process and redirecting its STDIO to a Java window. It also provides Prolog with the ability to call any Java method, and for Java to invoke Prolog goals, by using standard TCP/IP sockets to pass object/term data among both processes. All the interface is done through the serialization and reflection libraries of java. The evolution of Java as a dynamic language lead to the articulation of Java Serialization with Prolog Definite Clause Grammars. InterProlog currently has support for XSB prolog, XSB uses an emulator written in C and integrates partial evaluation techniques for specializing partly instantiated calls. When using InterProlog, 2 windows pop-up, one for the normal standart output and the other for standart input from XSB running in a separate process. All Prolog built-ins continue available, such as file I/O, etc. InterProlog provides a regular mechanism to specify Java objects in Prolog. It is based on the use of standard Serialization on the Java side, and on a Definite Clause Grammar on the Prolog side. This grammar is able to parse a sequence of bytes representing a serialized object into a Prolog term representing/specifying it, and vice-versa: given an object specification term, it is able to produce a sequence of bytes such that the standard Java Serialization process can recreate the object. For specifying java objects in Prolog there is a template based facility implemented through 2 alternative predicates, which are generated based on (serialized object) examples sent from the Java side, and which should be used to specify, on the Prolog side, objects of the respective classes. There is some special InterProlog Java code, together with a set of "special" object specifications, for passing Java basic type arguments back and forth to Javamessage(). Javamessage() is the main InterProlog primitive for synchronous communication with java.

InterProlog allows a Java program to use Prolog encapsulated in a Java object, a PrologEngine instance.

PrologEngine may be used at different levels of sophistication, by resorting to simple textual communication or to structured objects. For simple applications it is sufficient to communicate with Prolog using plain text. If a Java application wishes to get some data back from Prolog in a controlled manner, and/or specific Java code is written to support a Prolog project, then in addition to the previous PrologEngine methods others may become relevant: teachMoreObjects, registerJavaObject, isAvailable, deterministicGoal.

3.2.2 Prolog IV

Prolog IV is the latest product from PrologIA, a company based in Marseille. All the line of products come from the initial investigation by Alain Colemarauer in the LIM (Laboratoire d'Informatique de Marseille) since the invention of Prolog. It has, by far , the most appealing user interface seen in a CLP geared tool. Unfortunately no information could be gathered about the way java applets/applications are implemented nor about the communication between Java and Prolog IV. I assume that it is sockets based following Miguel Calejo's suggestion.

3.3 JNI-based interface

Most of the systems that achieved actual implementation are based in JNI. As it will become evident in the appropriate section where JNI benefits will be presented "see section 4.2 on page 23" it's perhaps the most appropriate technology to realize this kind of work. After the presentation of all the previous work it will also be presented a matrix "see section 4.3 on page 24" with the relevant advantages/disadvantages of the different technologies

3.3.1 Jasper (Sicstus)

Jasper is a bi-directional interface between Java and SICStus. The Java-side of the interface consists of a Java package (se.sics.jasper) containing classes representing the SICStus run-timesystem (SICStus, SPTerm, etc). The Prolog part is designed as a library module (library(jasper)) and an extension to the foreign language interface. The library module library(jasper) (see Jasper) provides functionality for controlling the loading and unloading the JVM (Java Virtual Machine), meta-call functionality (jasper_call/4), and predicates for managing global and local object references. These are provided in order to make it easy to call Java methods on-the-fly from Prolog without having to create a foreign resource first. The foreign language interface extensions enables Java-methods to be called as Prolog predicates using foreign/3 declarations, note that, all functionality of the foreign language interface is available through the use of the meta call facility in library(jasper). Jasper can be used in two modes, depending on which system acts as Parent Application. If Java is the parent application, the SICStus runtime kernel will be loaded into the JVM using the System.loadLibrary() method (this is done indirectly when instantiating the SICStus object). In this mode, SICStus is loaded as a runtime system (see Runtime Systems). If SICStus is the parent application, Java will be loaded as a foreign resource using the query use_module(library(jasper)). The Java engine is initialized using jasper_initialize/[1-2].

3.3.1.1 Calling Java from Prolog

Java methods can be called much in the same way as C functions are called, by creating a foreign resource. When loaded, this resource installs a set of predicates which are mapped onto Java-methods such that invoking a Java method looks like any other Prolog predicate call. Such methods are sometimes called direct mapped. In fact, a foreign resource is not language specific itself. The language is instead specified in the second argument to the foreign/3 fact and it is possible to mix foreign C functions with foreign Java methods. Java-methods are declared similarly to C-functions. There are two major differences. The first is how methods are identified. It is not enough to simply use an atom as the C interface does.

Instead, a term method/3 is used:

method(+ClassName,+MethodName,+Flags)

Used as first argument to foreign/3 when declaring Java methods and when calling a Java method through the meta call facility jasper_call/4. The first argument is an atom containing the Fully Qualified Classname of the class (for example, java/lang/String) The second argument is the method name. The third argument is a list of flags. Possible flags are instance or static, indicating whether or not the method is static or non-static. Non-static methods must have an object-reference as their first argument. This is a reference to the object on which the method will be invoked.
This term is then used to identify the method in the foreign_resource/2 predicate. So, to define a foreign resource exporting the non-static Java method getFactors in the class PrimeNumber in the package numbers, the method/3 term would look like

method('numbers/PrimeNumber','getFactors',[instance])

The syntax for foreign/3 is basically the same as for C-functions:
foreign(+MethodIdentifier, java, +Predicate) [Hook]

Specifies the Prolog interface to a Java method. MethodIdentifier is method/3 term as described above. Predicate specifies the name of the Prolog predicate that will be used to call MethodIdentifier. Predicate also specifies how the predicate arguments are to be translated into the corresponding Java arguments.

3.3.1.2 Calling Prolog from Java

Calling Prolog from Java is done by using the Java package jasper. This package contains a set of Java classes which can be used to create and manipulate terms, ask queries and request one or more solutions. The functionality provided by this set of classes is basically the same as the functionality provided by the C-Prolog interface of Sicstus.

Before any predicates can be called, the SICStus run-time system must be initialized. This is done by instantiating the SICStus class.

NOTE: This class must only be instantiated once per Java process. Multiple SICStus-objects are not supported.

- The next step is to load the Prolog code. This is done by the method restore.
- Now, everything is set up to start making queries.
- It is now time to create the arguments for the query.

The arguments are placed in an array which is passed to a suitable method to make the query. The arguments consist of objects of the class SPTerm. there are three ways of making a query, either to produce a single solution (SICStus.query,...), for side-effect only (SICStus.queryCutFail,...) or to produce several solutions through backtracking (SICStus.openQuery,...).

The openQuery methods returns a reference to the query, an object of the SPQuery class. To obtain solutions, the method nextSolution is called with no arguments. nextSolution returns true as long as there are more solutions.

3.3.2 JPL (for SWI)

JPL is an open source package providing a bridge between Java and Prolog. SWI is one of the, so far, fastest prolog available with extensible built-in predicates.

3.3.3 JIPL (K-Prolog, B-prolog)

This package, named JIPL, gives interface between Java and Prolog through JNI. With this package, you can

- call Prolog predicates from any Java application/applet.
- call Java methods, access fields of objects from Prolog programs.

There are, as far as known, two systems developed with this underlying package

3.3.4 K-Prolog

Being K-Prolog a commercial product, the amount of available information is far less than that for B-Prolog, since all the interface is based on the same package it was decided to bundle the full detailed instructions which appeared to be very similar on a single description found next.

3.3.5 B-Prolog

B-Prolog is a system for running Prolog and CLP(FD) programs. Like most other systems, it includes an interpreter and provides an interactive interface through which the user can consult, list,compile, load, debug and run programs. It not only runs ISO Prolog programs, but also supports delaying (or corouting) and constraint solving over finite domains and Booleans. It also provides interfaces through which external languages (currently C and Java) and Prolog can call each other. Some functionalities including module systems and garbage collection are not available now and are to be implemented in the future. Tabling is implemented in B-Prolog. An application that uses the Java interface usually works as follows: The Java part invokes a Prolog predicate and passes it a Java object together with other arguments; the Prolog predicate performs necessary computation and invokes the methods or directly manipulates the fields of the Java object.

3.3.5.1 Calling Java from Prolog

javaMethod(+ClassOrInstance, +Method, -Return)
Invoke a Java method, where

- ClassOrInstance is either an atom that represents a Java class's name, or a term $addr(I1,I2) that represents a Java object. Java objects are passed to Prolog from Java. It is meaningless to construct an object term by any other means.
- Method: is an atom or a structure in the form f(t1,...,tn) where f is the method name, and t1,...,tn are arguments.
- Return: is a variable that will be bound to the returned object by the method.

javaMethod(+ClassOrInstance, +Method)
The same as javaMethod/3 but does not require a return value.
javaGetField(+ClassOrInstance, +Field, -Value)
Get the value of Field of ClassOrInstance and bind it to Value. A field must be must be an atom.
javaSetField(+ClassOrInstance, +Field, +Value)
Set Field of ClassOrInstance to be Value.

3.3.5.2 Calling Prolog from Java

A Prolog call is an instance of the class bprolog.plc.Plc. It is convenient to import the class first:
import bprolog.plc.Plc;
The class Plc contains the following constructor and methods:

- public Plc(String functor, Object args[]) It constructs a prolog call where functor is the predicate name, and args is the sequence of arguments of the call. If a call does not carry any argument, then just give the second argument an empty array new Object[].
- public static void startPlc(String args[]) Initialize the B-Prolog emulator, where args are parameter-value pairs given to B-Prolog. Possible parameter-value pairs include:

  "-b" TRAIL : words allocated to the trail stack
  "-s" STACK : words allocated to the local and the heap
  "-p" PAREA : words allocated to the program code area
  "-t" TABLE : words allocated to the table area

  where TRAIL, STACK, PAREA and TABLE must all be strings of integers. After the B-Prolog emulator is initialized, it will be waiting for calls from Java. Initialization needs to be done only once. Further calls to startPlc have no effect at all.

- public static native boolean exec(String command) Execute a Prolog call as represented by the string command. This method is static, and thus can be executed without creating any Plc object. To call a predicate in a file, say xxx.pl, it is necessary to first have the Prolog program loaded into the system. To do so, just execute the method exec("load(xxx)") or exec("consult(xxx)").
- public boolean call() Execute the Prolog call as represented by the Plc object that owns this method. The return value is true if the Prolog call succeeds or false if the call fails.

3.3.5.3 Data conversion between Java and B-Prolog

The following table converts data from Java to Prolog and vice versa:

Java	*Prolog*
Integer	*integer*
Double	*real*
Long	*integer*
BigInteger	*integer*
String	*atom*
Object array	*list*
Object	*$addr(I1,I2)*

Table 3.1: Data conversion table in B-Prolog

Since primitive data types in Java cannot be converted into Prolog, the conversion between arrays and lists needs further explanation. A Java array of some type is converted into a list of elements of the corresponding converted type. For instance, an Integer array is converted into a list of integers. In contrast, a Prolog list, whatever type whose elements belong to, is converted into an array of Object type. When an array element is used as a specific type, it must be casted to the type.

3.3.6 yajxb (XSB)

Made by Stefan Decker in Stanford University, yajxb realizes a connection from Java to XSB. In contrast to InterProlog, using Yajxb causes Java to call XSB directly via Java's native interface mechanism. Yajxb for the moment does not support calling java from XSB. Some caveats are still pending. Java loads the XSB system as a shared library. During initial loading, when using XSB, Java sometimes crashes nondeterministically (4-10% of loading tries).Once it is loaded, everything works fine. Being a recent development (the downloadable sources are dated 10/03/2001) new developments are expected.

3.3.7 BinProlog

This very complete system ,by BinNet, has an interface through C. It is based on the binarization of clauses, which roughly consists of making the continuations explicit. The WAM is specialized and the abstract code is emulated by an emulator written in C. The BinProlog runtime emulator combined with the C-ified compiler are packaged into a dynamic library (jbp.dll or libjbp.so). A stub jBinPro.c based on BinProlog's C interface implements a call_bp_main C function which is declared as a native Java method in file JavaLog.java. A method similar to this one appears to be the most promising way to achieve the job proposed in this work.

3.4 Prolog engines in Java

3.4.1 BirdLand

Although often referred in the internet in every page that cares about this subject, the Web page related to this work could not be found.

3.4.2 DGKS Prolog

Written from the ground up in Java, basically only lacking the IO predicates usually found in Prolog implementations.

3.4.3 JavaLog

JavaLog is a Prolog interpreter written in Java designed to allow easy integration between Java and Prolog. JavaLog was developed at the ISISTAN Research Institute [1] and is currently used in several research projects related to artificial intelligence supported by Object Oriented concepts for Software Development. Namely software agents developed under the Brainstorm project that aims the building of multi-agent systems through software architectures supporting the usage of both the object-oriented and logic paradigms.

JavaLog has the following features:

- It enables the creation and usage of Java objects in Prolog programs, mixing Logic/OO paradigms.
- It preprocess Java methods with embedded Prolog enabling the common use of local variables in both paradigms.
- It supports the manage of common knowledge of several instances of Javalog's Prolog interpreter by means of a blackboard architecture.
- It allows the physical distribution of Prolog interpreters using Java RMI and logic module mobility.

JavaLog should run on any Java(tm) 1.1.x or 1.2.x virtual machine and has been tested on Linux 2.0.x, Win9X, WinNT, Solaris, IRIX and OSF1.

3.4.4 NetProlog

NetProlog is a logic programming system that generates a binary code, executable in the Java Virtual Machine (JVM). It follows almost completely the syntax traditionally used in the ISO Prolog implementations. For each logic predicate is

[1] http://www.exa.unicen.edu.ar/isistan/

generated a corresponding Java class, which can be used as a regular code generated for the JVM. It has a user graphical interface, where the programs, source and object, can be edited, compiled, printed, etc.

The NetProlog system has been developed using the language Visual Prolog as the working language. The code can be compiled for either Windows or Linux. The code generated by NetProlog is CGI (Common Gateway Interface) independent. This characteristic is very important for programs developed for the Internet, such as intelligent agents for instance. The object code is generated for the JVM pattern, which eliminates the necessity of the execution of the CGI programs to process HTML forms.

3.4.5 JIP Java Internet Prolog

JIP - Java Internet Prolog is a cross-platform PureJava 100% prolog interpreter developed in JDK 1.1 (also working in Java 2 Platform) and supporting the prolog Edinburgh syntax. It is compatible with other famous prolog engine (such as LPA, Quintus, SWI, etc.) and can be run by any browser supporting JDK 1.1. JIProlog has an easy-to-use API by which, you can invoke the prolog interpreter in any Java applet/application without dealing hard with native code (JNI or RNI/JDirect) and without requiring signed applet (as required by other prolog interpreters written in C/C++). By the API you can invoke the prolog interpreter in your Java classes in a very simple way calling your prolog predicates in the same way you call a Java? method and, vice versa, you can invoke Java classes in your prolog code as you call predicates.

JIP allow to extend the set of built-in predicates implementing them in the same way you write Java classes. So you can add custom features, such as custom dialogs or custom algorithms, and you can speed up the computation transforming your prolog predicates (defined in your prolog code) in built-in predicates implementing them in a Java classes derived from a special Built-In class exported by the API. Besides, JIP support external database of clauses. JIProlog allow you to use such an external database of facts as it is a predicate stored in the prolog memory. The prolog engine is based on an ASM (Abstract Syntax Machine) implemented using a LISP-like depth-first search. It is composed by a very complex hierarchy of Java? Classes implementing the ASM with typical prolog heuristic (such as backtracking, cutting mechanisms, etc.) a manager for built-in predicates, the parser for prolog language and so on. It also implements a mechanism to call prolog predicates from any Java classes and vice versa to invoke Java classes from your prolog code. JIProlog is a prolog interpreter to run your prolog code as a "stand-alone" application without an external Java? applet/application. In other words you can write your program entirely in prolog (eventually using your dialogs or classes extensions) and run it as a "stand-alone" application. Only you must define in your prolog program the predicate main/0 that is the entry point predicate that will be called by the interpreter when it will start.

3.4.6 JProlog

This was the first prolog to java translator. It is based on continuation passing style compilation, called binarization transformation [6]. See all the subsequent work that was based on this first breakthrough: BinProlog, JINNI, K-prolog, B-prolog, Prolog Café

3.4.7 MINERVA

Compiles ISO-13211-1 Prolog into it's own virtual machine which is executed in java. MINERVA is a commercial programming system geared for intelligent client-server applications on the internet.

3.4.8 JINNI

A Prolog Interpreter in Java for Intelligent Mobile Agent Scripting and Internet Programming. Jinni compiler integrates a high performance pure Java based Prolog system, a GUI development library, Multiple Databases and an Object Oriented Layer. It supports remote predicate calls and multiple network transport layers (HTTP, TCP/IP, UDP, multicast, Corba, RMI etc.). Web pages or components of ZIP files are handled just as if they were ordinary Prolog files. As a multi-threaded Java program providing portable networking and Prolog style rule based reasoning, Jinni is an ideal tool for building Intelligent Agents. It is based on the same line of research of it's author Paul Tarau and it complements the BinProlog commercial product from BinNet which as stated in the corresponding section is not suitable for applet development for it is JNI based.

3.5 Hybrid systems

These are systems that implement sub or supersets of prolog namely linear logic programming languages.

3.5.1 LLPj

First implementation of the LLP -> Java. See Prolog Café below.

3.5.2 Prolog Café

Prolog Café is a Prolog-to-Java source-to-source translator system [1].Is is based on the first prolog to java translator jProlog developed by B.Demoen and Paul Tarau (see jProlog above). This system translates LLP into java via the LLPAM [5]. LLPAM is a extension of the standart WAM [7] for LLP. LLP is a superset of prolog. There was a so-called 'first implementation' that was based on the original LLPAM. The main difference from the first implementation is resource compilation. Resource formulas are compiled into closures which consist of a reference of compiled code and a set of biddings for free variables. Calling these resources is integrated with the ordinary predicate invocation. It's 2,2 times faster compared with jProlog.

3.5.3 W-Prolog

Written before all the other stuff presented here, W-Prolog 1.0 was the first Prolog interpreter written in Java. Had Initial release in October 1996 and it was followed fairly soon by Demoen and Tarau's compiler. W-Prolog is an interpreter for a Prolog like language implemented in Java. The implementation is extremely portable and can be run as an applet under Java-capable web browsers. W-Prolog has a nicer user interface than most Prolog systems (which typically provide a command line interface). It provides simple tracing and has an (optional) occur check. The W-Prolog system is small and comparatively simple. However it is not particularly fast. The language is given by the following simple grammar:

Program ::= Rule | Rule Program
Query ::= Term
Rule ::= Term. | Term :- Terms.
Terms ::= Term | Term , Terms
Term ::= Number | Variable | AtomName | AtomName(Terms)
| [] | [Terms] | [Terms | Term]
| print(Term) | nl | eq(Term , Term)
| if(Term , Term , Term) | or(Term , Term) | not(Term) | call(Term) | once(Term)
Number ::= Digit | Digit Number
Digit ::= 0 |... | 9
AtomName ::= LowerCase NameChars
NameChars ::= NameChar | NtameChar NameChars

NameChar ::= a |... | z | A |... | Z | Digit

W-Prolog can be run as an applet or as a standalone application. It can also be embedded and called from another program. To run W-Prolog as an applet construct an HTML file containing the tag:

```
<center>
<h1>W-Prolog</h1>
<applet code=WProlog.class width=120 height=65>
</applet>
</center>
```

and use netscape (or internet explorer) to view this file.
The W-Prolog inference engine can be called from Java code. It is recommended that W-Prolog be run as a standalone application. Earlier versions of W-Prolog supported loading files in the applet version.
This worked flawlessly when run with Sun's appletviewer but refused to work under netscape.
Another example of undesired behavior under netscape concerns threads. Each query in W-Prolog runs in its own thread. This means that a long running (or non terminating) query won't freeze the interface (although it will slow subsequent queries down). This works, but not under netscape.

3.5.4 Kiev

Kiev is a full-featured language targeted to creation of complex applications. Kiev is backward compatible with Java and Kiev compiler generates code for JVM. Kiev language has an embedded AI engine that inherits many features and power from Prolog language. Like Prolog it allows both check possible solutions for rules or find out possible a solution (or some/all solutions) that satisfy the rule.

Chapter 4

Interfacing Java with GNU-Prolog

The motivation to use GNU Prolog as well as some of it's characteristics that matter for this work are presented.Second the JNI architecture is also presented and explained why it's considered to be the best choice. A comparison matrix is shown that summarizes the different advantages and disadvantages for each technique. Then the job requirements and the (serious) limitations encountered are explained in order to frame and explain the work done. Finally one last section in this chapter will present the intended API design.

4.1 GNU Prolog

GNU Prolog development was started in the of computing science department at the university of Paris 1 by Daniel Diaz[1]. It is currently an open source project with the following features, taken from it's manual:

- **Prolog system:**

  Conforms to the ISO standard for Prolog (floating point numbers, streams, dynamic code,. . .). a lot of extensions: global variables, definite clause grammars (DCG), sockets interface, operating system interface,. . .

  More than 300 Prolog built-in predicates.

  Prolog debugger and a low-level WAM debugger.

  Line editing facility under the interactive interpreter with completion on atoms.

  powerful bidirectional interface between Prolog and C.

- **Compiler:**

[1] http://crinfo.univ-paris1.fr

Native-code compiler producing stand alone executables.

Simple command-line compiler accepting a wide variety of files: Prolog files, C files, WAM files,. . .

Direct generation of assembly code 15 times faster than wamcc + gcc.

Most of unused built-in predicates are not linked (to reduce the size of the executables).

Compiled predicates (native-code) as fast as wamcmcc on average.

Consulted predicates (byte-code) 5 times faster than wamcc.

It is also appealing for the use in this work the fact that it only uses standard tools like the GNU linux linker or the assembler. It does not, however, allows for dynamic linking for it has to be fully statically compiled. Important also is the fact that a very efficient Finite Domain (FD) solver is embedded in GNU Prolog and, therefore, the presented tool opens the possibility to integrate Java OO with Constraint Programming. As seen before a two way communication tool is to be made between GNU-Prolog and Java. GNU-Prolog is equipped with powerful ways for the integration development. These tools are based in a strong API between Prolog and the C language herein referred as the **foreign** interface. The **foreign** interface is clearly explained in the GNU Prolog manual [2]. Provided such a proper tool to work with it was decided to use it heavily. This interface allows a Prolog predicate to call a C function. Both simple and complex C routines depending whether the routine arguments are simple C types or complex structures which have to be represented in Prolog as some kind of complex structures like, for instance, compound terms. Also made possible through the use of the **foreign** interface is the ability to create non-deterministic code in C and, therefore, in Java. The naming of this API is taken from the **foreign/2** directive that declares a C function interface. The general form is `foreign(Template, Options)` which defines an interface predicate whose prototype is `Template` according to the options given by `Options`. `Template` is a callable term specifying the type/mode of each argument of the associated Prolog predicate. Each argument of `Template` specifies the foreign mode and type of the corresponding argument. This information is used to check the type of effective arguments at run-time and to perform Prolog <-> C data conversions. `Options` is a list of foreign options. Possible options are:

`fct_name(F)`	*F is an atom representing the name of the C function to call*
`return(boolean/none/jump)`	*Specifies the value returned by the C function*
`bip_name(Name, Arity)`	*Initializes the error context with Name and Arity*
`choice_size(N)`	*Specifies that the function implements a non-deterministic code. N is an integer specifying the size needed by the non-deterministic C function*

Table 4.1: Possible options in the GNU Prolog **foreign** API

In the opposite direction, the **foreign** interface enables the creation of fully statically compiled C functions that can, to a great, extent, manage the GNU Prolog environment. Has can be seen in the GNU Prolog manual there are functions to:

- Start the prolog environment
- Open a query
- Create terms in the heap
- Compute te first solution
- Eventually compute next solutions
- Close the query
- Stop the prolog environment

It is even possible to define an alternative main function to the embeded caller of a top-level prolog in C.

4.2 What is JNI

First introduced in JDK 1.1, JNI is touted as the final and correct way to mesh portable code with platform code. JNI is a portable specification by definition. Namely:

- JNI is portable across JVM implementations. The same binaries should work with any JVM on any particular platform.
- JNI handles data in a portable manner: Rather than passing raw Java data structures directly to native code, access to data structures is indirect. JNI defines native calls through which the native code can read and write class and object members.
- JNI is friendly to garbage collection, providing new techniques for managing dynamic objects that do not interfere with advanced garbage collection techniques. It accommodates the underlying JVM garbage collection facilities without placing any burden on the native programmer. JNI does not expose JVM memory to the native code as anything other than object references. Since these references provide a level of indirection between the JVM and the native code, the native programmer is protected from garbage collection activity.

Being aimed at a truly portable environment through both Operating Systems and JVMs the natural choice for the integration architecture seems to be JNI. To connect Java methods to native functions one has to define individual methods as native, then there is a corresponding C/C++ entry point whose mangled name "see section 5.2.4 on page 38" reflects the C naming convention and the corresponding Java signature. The well documented tool that generates header files and derive the entry point names is `javah`.
All native side access to Java classes, objects and members is indirect. The native code must jump through a few hoops to touch the class contents. The JVM can be started up from a native application through the invocation API, which launches the JVM, provided as a shared library, through a simple sequence of C/C++ calls.

Currently for developing Java based interfaces for Prolog systems these are probably the most portable for they are defined as a standard for Java communication natively with the underlying operating system facilities. Obviously, when is available an interface for a Prolog system based on some relatively low-level language like C/C++ is much easier to implement a wrapper, using Prolog, for that interface than to develop a completely new one. However, use of native methods in Java is of limited interest due to security concerns that avoid the creation of applets with these kind of mechanisms. Certain *"limitations"* of Java must be overcome by some native interface whether implemented directly through the Prolog system or through another third interface invocation. The following must be included:

- Interaction with the environment
- Device control
- Interprocess interaction
- Interaction with the window manager

Of particular interest is the use of the reflection API within JNI. With the capability to introspect through the class hierarchy and within a particular class through its members it's possible to build a representation at a highly effective C tier to be used later mapped into the Prolog level.

4.3 Comparison matrix between technologies

In the previous chapters both advantages and disadvantages of each technology for java-prolog integration have been mentioned. Since the different levels of approaching made to each product led to various levels of detail when considering the benefits or weaknesses of any particular issue a detailed and thourough summary seems to be in order now. Some of the characteristics explained below are an advantage in some technologies and a disadvantage in others.

- Performance
  Apparent response speed of the integrated system relative to its underlying systems (a JVM and a Prolog machine). It may be considered a measure of the performance penalty in which the particular technique incurs. Normally if the communication between both systems is maintained by a thin layer of data structures and interlayer communication the performance penalty is not noticeable. Obviously the native interfaces should show best results in this particular aspect.

- Standards adherence
  When there are standards defined in the underlying technologies the integration technique should not rely on a particular state of that standards at any point in time. Being GNU Prolog and specially Java subject of fast evolving standards, those should be able to develop independently and not interfere with the integration. This integration can not be compromised by the normal evolution o any of the standards in any of the layers.

- Applet development and deployment
  An applet runs in a JVM integrated in an HTTP browser. The possibility of running the integrated system in those JVMs relies normally on security issues that are usually addressed in the Java platform but not in the Prolog platform. Its responsability of each implementation to make these issues addressed in their solution and it does not seem to be possible to consider this point as an advantage or drawback of each technique.

- Platform independence
  A serious deficiency might be considered when the interfaces depend on some characteristics of the systems where the underlying levels are running. One of the often stated objectives of Java is platform independence both operating system and hardware and it would be a serious drawback if the interface would rely on on particular aspects of some operating system for instance. Of course the portable way of doing this kind of integration in Java, JNI, has the strongest point in this question.

- Coupling level
  Whether the interface implies tightly coupled systems with shared data-structures and synchronous process communication or they can be loosely coupled, communicating asynchronously in a local or distributed manner. The less coupling inferred by an interface the better for as much independence as possible is the correct way to behave in the present distributed asynchronous world.

Sometimes, due to the underlying systems, an optimal solution attending all of these characteristics is hard to encounter. The following matrix shows a comprehensive view of every of those issues that arose at any of the points mentioned in the previous chapters.

Advantages	Disadvantages
Native interface	
Performance	*High development complexity*
Applet development	*Standards adherence*
	Platform independence
	Coupling level
JNI based interface	
Performance	*Coupling level*
Standards adherence	*Platform independence*
Smooth development complexity	*Applet development*
Prolog engines in java	
Applet development	*High development complexity*
Performance	*Standards adherence*
Platform independence	*Coupling level*

Table 4.2: Integration methods relative advantages

4.4 Requirements and limitations

In its current incarnation, the GNU-Prolog compiler has a drastic drawback that inhibits the complete fulfillment of the initial objective of this work. As it will be seen later the JNI technology relies heavily on dynamic linking of native methods. ***GNU-Prolog does not currently allows dynamic linking***. It generates native code in a position dependent way so that no shared objects can be created which reference natively the GNU Prolog code. Browsing through this work an interested reader should now wonder why then bother to show a full grown implementation of the Java to Prolog API if it is not possible to make it work. The reason steems from the fact that this limitation is likely to be overcomed in some point in the future by change of at least one of the following two realities

- JNI depends on shared object creation for native functionality implementation
- GNU-Prolog is not statically linked and it would then be possible to create shared objects based on it

The first reality is more likely to be surpassed and even currently it remains to be checked if some, not off-the-shelf JVMs, like the gcj compiler, imposes this kind of limitation. Ahead "see section 6 on page 45" it will be shown the possibility to achieve this kind of integration with this tool. This work shows a very efficient way of achieving the intended purpose and it can always be used as a proposal foundation. In the subsequent chapters the implementation is explained thoroughly

and it is completely implemented allthough the native functionality of the Java to GNU-Prolog API was not, for the moment, possible to be done.

4.5 API Design

Using the tools presented in the two previous sections a specification for the API structure can be drafted. A two layer API shall be used.

1. A low-level API entirely developed in C for the connection to the JNI. This is mainly a C middle tier designed to exhibit very high performance, this layer will be called **gpl_java_C**. For the gpl_java_C an interface from the JNI invocation API to GNU-prolog through the `foreign/2` primitive will be developed.

2. A higher-level interface designed to be more friendly for both the Prolog and Java programmers to use. It will be called **gpl_java_PL**.

The positioning is thus:

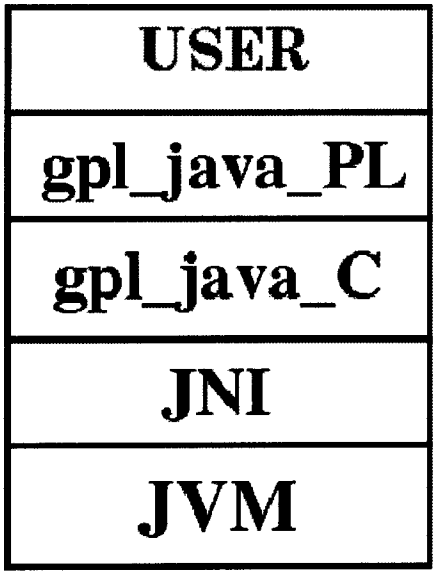

Figure 4.1: API Design

The typing granularity in GNU-Prolog **foreign** API is smaller than that of java so the conversion between Java and GNU-prolog types should not be performed based on the JNI native types but instead on the different **foreign** types. However,

since the JNI functions are appropriate for all the Java typing range it was decided to use them as foundation for the interface development. The conversion table used for simple types was thus the following:

Java	*JNI Native*	*Foreign*
boolean	*jboolean*	*boolean*
byte	*jbyte*	*byte*
char	*jchar*	*char*
short	*jshort*	*integer*
int	*jint*	*integer*
long	*jlong*	*long*
float	*jfloat*	*float*
double	*jdouble*	*float*
void	*void*	*void*

Table 4.3: Conversion table GNU Prolog <-> Java

The reverse table mapping from GNU-Prolog types to JNI is as follows:

GNU-Prolog	*C type*	*JNI Native*	*Description of the C type*
boolean	*boolean*	*jboolean*	*unsigned 8 bits*
byte	*byte*	*jbyte*	*signed 8 bits*
char	*char*	*jchar*	*unsigned 16 bits*
short	*integer*	*jshort*	*signed 16 bits*
int	*integer*	*jint*	*signed 32 bits*
long	*long*	*jlong*	*signed 64 bits*
float	*float*	*jfloat*	*32 bits*
double	*float*	*jdouble*	*64 bits*
void	*void*	*void*	*N/A*

Table 4.4: Reverse mapping table GNU Prolog <-> Java

With these decisions in mind a complete architecture has to be devised and the chosen implementation is presented next.

Chapter 5

Implementation

Several new functions, to manipulate the JVM, were developed in C and linked with a GNU prolog interpreter. This kind of integration between GNU prolog and the lower layers are the most effective for general availability and fast performance. The work was divided in two different sections according to the direction of the intended communication both are explained in the following two sub-sections.

5.1 Prolog to Java

The API that allows the Java calls from an arbitrary language is the invocation C based API.

To run prolog code in Java there has to be the possibility to:

- invoke one or several JVMs (Java Virtual Machine) from GNU Prolog.
- load class methods.
- traverse the class hierarchy loading methods.
- create java objects.
- call both static or instance methods with the appropriate return value.
- build prolog glue predicates from the loaded methods.
- call prolog predicates with the appropriate returned value from the underlying method.

Several functions were developed in the C language, linked to a GNU Prolog interpreter to provide new predicates for Java manipulation. All these predicates are named consistently in the form java_XXX to be easily distinguishable from other Prolog predicates. The convention of using input parameters first and output parameters last is followed. Several predicates behave non-deterministically, e.g. java_MethodID or java_SuperClass. With an architecture that is intended to be as

lightweight as possible, apart from the direct access that the JNI facilitates, some structures were added at an intermediate level (C level). These were created to avoid repetitive JNI method calling by maintaining a certain amount of information locally cached at the C level. The different structures will be explained attached with their purpose.

5.1.1 single vs. multithreading

From the strict JNI point of view it should be rather simple to implement Java multi-threading in GNU Prolog. The structure used is the following:

```
typedef struct {
    JNIEnv *env;
    JavaVM *jvm;
} Java_VM;
```

This structure is used to keep a link between a JNIEnv environment and a JVM. In the implementation presented here, an array of Java_VM structures is used. All the function calls in the JNI API use an opaque structure pointed to by a JNIEnv* pointer. This pointer is fetched when the JVM is created by use of the JNI_CreateJavaVM function. Originally it was meant to support multithreading by the use of the AttachCurrentThreadfunction but for the sake of simplicity and to avoid incurring in problems with multithreading in prolog, it was decided to use only one JVM and a single thread of execution. The JVM initialization is done by use of java_CreateJavaVM/1 (see section A.2.0.7 on page 49) which mimics the behavior of JNI_CreateJavaVM of the JNI API. The multithreading enabling function java_GetJavaVM/2 (see section A.2.0.8 on page 49) although implemented was deprecated since it was chosen not to support multiple threads.

5.1.2 Class loading

Right after initializing the JVM, Java classes can be loaded through the use of java_GetClass/3 (see section A.2.0.10 on page 50). As usually found from now on the output parameter of the java_CreateJavaVM/1 predicate will be used as input parameter for the different functions to provide the link to the adequate JNIEnv pointer. One can find this unnecessary since only one such pointer is in use but the structure is enabled for future use whenever multithread synchronization is addressed. The java_GetClass/3 uses the JNI function `FindClass` to load the class named in java_GetClass second parameter. This function proceeds recursively filling a local array of classes until all the relevant parts of the class hierarchy have been visited. The array that supports this local class information is made up of elements with the following structure:

```
typedef struct {
    int jvm; //index to the jvms
```

```
    char *clazz; //class name
    jclass classID;
    jclass superClassID;
    int superClassIndex; //index to the superClass;
                         //if java.lang.Object then -1
} Java_Class;
```

This array will allows us not to repeatedly call JNI functions to get the ClassID pointer by using only one level of indirection to reach the class. It can be easily understood that the first field points to a java_VM slot, the second and third is information pertaining to the visited class and the last ones are used for hierarchy maintenance. The last parameter of java_GetClass/3 is input/output, it may be unified or not and it provides the array address where the class is located or checks if that class is in the specified slot or not returning a boolean value. Another important predicate to handle classes is java_SuperClass/2 (see section A.2.0.11 on page 50). It's the first predicate presented that behaves non-deterministically. If both arguments are ground it returns a boolean value indicating wether the second class is superClass of the first, otherwise it provides class hierarchy traversing facilities by providing the different available solutions.

5.1.3 Method loading

To keep the method information locally the predicate to use is java_GetMethods/1 (see section A.2.0.12 on page 50). Instead of having all the visited classes methods loaded, only those methods brought in by java_GetMethods/1 are loaded to the associated local structure:

```
typedef struct {
    int jvm; //index to the jvms
    int classIndex; //index to the class
    char *method; //method name
    char *signature; //method signature
    int ctIndex; //index to the call table
    int nParams; //Number of parameters
    jboolean isStatic; //is static flag
    jmethodID methodID;
} Java_Method;
```

In relation to the loading of methods there are two types of class visibility:

1. visited classes are those whose methods are loaded through java_GetMethods/1.
2. traversed classes that belong to the hierarchy but whose methods are not loaded.

The reason to have classes with different levels of information present relates with an intelligent resource management. All the final class direct genealogy should have their methods loaded. All the information presented here is collected through Java reflection at load time and kept locally for performance purposes. Reflection, or introspection, is one of the major improvements in language design brought in by the Object Oriented paradigm. Through it, the inners of a class are exposed. Both its methods and its members properties can be reached. Several wrappers to the Java reflection API are defined in the JNI and used here thoroughly. The java_Method struct first two fields are indexes into the Java_VM and Java_Class arrays. The third and fourth are for the method's name and Java signature. The ctIndex field points to the index in the callback tables referred in the appropriate section (see section 5.1.4 on page 32). The nParams field could be used for run-time checking of parameter accordance between definition and call. Actually no checking is currently done for the higher level API does, in Prolog, all the predicate building taking appropriate care of both argument number and typing. The flag isStatic is crucial for knowing what callback table to use and, thus, the appropriate method calling function. Finally the methodID gathers a method pointer that can be used later to perform immediate invocation.

5.1.4 Reflection and callback tables

The invocation API is used to embed a Java virtual machine into a Prolog interpreter. Through this interface using the GNU prolog foreign interface [2] there is the possibility to use the JNI to access the Java refection classes: you can pull a Java class apart, learn the names and types of its data fields, the names and signatures of its methods and its place in the inheritance hierarchy. The signatures include enough information about the return and parameter types. An approach to translate between Java signatures and 'Prolog signatures' is presented (see section 5.1.7 on page 34). they are more adequate for Prolog programmers. There must be the possibility for a Java enabled Prolog to be able to invoke any method, *static* or *non-static* that may return an arbitrary Java result and also have an arbitrary number of Java arguments. Some JNI coding is needed to mimic the operations provided by the reflection API. Depending on whether it is a *static* or *non-static* method, and on its return type, there are different method invocation functions in JNI. Class methods are invoked using *Call<type>StaticMethodA* and *non-static* methods using *Call<type>MethodA*. An effective solution was adapted from [3]: two tables are set up with the addresses of the appropriate JNI call back functions. As seen in the previous section, when calling the java_GetMethods/1 (see section A.2.0.12 on page 50) the Java_Method array is filled with all the method information, namely their names, signatures, the callback table index for later reference when eventually invoking, the number of parameters for easier verification of parameter accordance between call and signature and whether it's a *static* or instance method for using the appropriate entry in one of the callback tables:

	Static Callback Table	Instance Callback Table
0	*CallStaticObjectMethodA*	*CallObjectMethodA*
1	*CallStaticBooleanMethodA*	*CallBooleanMethodA*
2	*CallStaticByteMethodA*	*CallByteMethodA*
3	*CallStaticCharMethodA*	*CallCharMethodA*
4	*CallStaticShortMethodA*	*CallShortMethodA*
5	*CallStaticIntMethodA*	*CallIntMethodA*
6	*CallStaticLongMethodA*	*CallLongMethodA*
7	*CallStaticFloatMethodA*	*CallFloatMethodA*
8	*CallStaticDoubleMethodA*	*CallDoubleMethodA*
9	*CallStaticVoidMethodA*	*CallVoidMethodA*

Table 5.1: Method calling callback table

The Call<type>MethodA and CallStatic<type>MethodA are one of the possibilities of using JNI to do method calling. In this style of invocation the method parameters are provided as a jvalue array of values as the function third parameter. Other possibilities were Call<type>MethodV which take as its third argument a variable argument list as defined by the ANSI C header file <stdargs.h> and CallTypeMethod where arguments are simply listed in the function call; These are not used in the GPL/JNI interface due to the inability to verify the correct argument typing and numbering using these kind of argument passing.

The jvalue structure is a union with the following definition:

```
typedef union jvalue {
    jboolean    z;
    jbyte       b;
    jchar       c;
    jshort      s;
    jint        i;
    jlong       j;
    jfloat      f;
    jdouble     d;
    jobject     l;
} jvalue;
```

So it can hold any kind of JNI native type as a value. To perform a method invocation, after filling it's parameters, only the already known callback function as to be used, as it will be shown further on section 5.1.8 (on page 36).

5.1.5 Object representation in Prolog

An object ID in JNI is represented as a long integer represented using 32 bits. A compound term in Prolog does not have such a well known implementation. A

fixed structure was chosen that has an arbitrary predicate name (w for historical reasons) and a Upper Half and Lower Half number of bits as arguments. These are built given a jObject object ID that is divided into two halves. One shifted 16 bits right and both masked with 0xFFFF. Then both sets of bits are turned into valid prolog integers and a compound term is built as can be seen in the java_New function.

5.1.6 Object instance creation

To create an object using JNI there are three different functions available differing in the way how the constructor arguments are bundled to the JNI function, much in the same way as the method calling functions that we will see in section 5.1.8 on page 36. Currently under the gpl_java environment a three step approach has been followed:

```
1. Allocate space for the object constructor arguments
2. Fill the arguments
3. Call the java_New predicate
```

Another approach could be the one presented in the future work section (see section 6.2.6.3 on page 46). To allocate space in java the interface predicate java_Create_Args/2 (see section A.2.0.14 on page 51) is used with the number of arguments the constructor uses. The family of functions java_PutArg_XXX (see section A.2.0.15 on page 51)where XXX may be int, float, String or obj are then used to fill in the arguments. Finally the java_New function has to be called with the built args and the correct constructor signature. A Prolog term is returned as seen above.

5.1.7 Converting between Java and Prolog signatures

For convenience to the prolog programmer a Prolog representation of class signatures was introduced. A Java methods signature is made up of the methods parameters types indicated through a character in the case of primitive types and the *Fully Qualified Name* of a class and a ; after the FQN in the case of a generic class. A [precedes the type indication in case it is an array. The methods parameters are enclosed in parentheses and the return type follows them. The following table has a description of all the cases:

Signature Encoding	
B	*byte*
C	*char*
D	*double*
F	*float*
I	*int*

J	*long*
S	*short*
V	*void*
L<fully-qualified-class>;	*fully-qualified-class*
[<sigtype>	*Array of <sigtype>*
(<sigtype-list>)<return-sigtype>	*Method signature*

Table 5.2: Conversion table between Java and "GNU Prolog signatures"

Summarizing a Java signature is **(<sigtype-list>)<return-sigtype>**. For instance a method accepting as input an array of integer and a string and returning a boolean has `([ILjava.lang.String;)Z` for signature.

The method used to build a Prolog signature is based on list notation to represent argument sequences with the following self-explanatory named elements to indicate the different types:

```
char
byte
int
long
short
float
double
boolean
array(<type>)
object(<path>)
```

where <type> is either one of the primitive types or a path of an object representation and <path> is in the form of a list representing the hierarchy of classes in the original signature but reversed. For instance the same signature presented above would be represented in gpl_java as
`[array(int),object(['String',lang,java])]-boolean`
As you noticed the order of the terms in the Prolog signature list that represents a FQN is inverted with relation to the Java hierarchy. This was made for it is easier, for instance, to reach the final class name using only the list header and the rest of the list can then be processed through the normal prolog list processing primitives but then, this is seldom used. There could be portability problems with different integer sizes the representation of char, byte, int, long and short is the atom `int` with a companion sub-type numeral ranging from 0 to 5 respectively. The same technique was used to handle the different float sizes, both the float and double Java types map to a Prolog float with a sub-type respectively 0 and 1. The predicate type/2 (see section A.2.1.2 on page 55) converts between a Prolog signature in the

first argument and a Java signature in the second so it can be used after retrieving a method's Java signature through java_MethodID/4 and piping it through name/2 to obtain a valid Prolog signature.The predicate type/2 is implemented with a DCG to perform the type conversions. For instance in the toy example referred in the appendixes with the loading of the `testClass` class that contains among others the `sumIntegers` method the java signature can be converted to the prolog signature by use of the following prolog pipeline:

```
j_mid(sumIntegers, JSig, M), name(JSig, Sig), type(PSig, Sig).
M = 2 JSig = '(II)I' PSig = [int,int]-int Sig = [40,73,73,41,73] ?
```

In the utilization examples in the file `test_java.pl` present in the appendixes "see section C.2 on page 62" the convertion is fully illustrated. Another method could be to convert when calling java_GetMethods/1 and put immediately in the Java_Method struct in a field with the Prolog signature but this was not done.

5.1.8 Predicates to call Java methods

There are 3 steps that need to be performed to do a Java method call.

1. Initialize the space needed for arguments and return value
2. Fill in an array of parameters
3. Actually call the method

For the methods space to be initialized the function java_Create_Args/2 (see section A.2.0.14 on page 51) has to be called. This new function allocates space to hold the methods arguments. Since all the arguments are held in a jvalue union, in order to use the Call<type>MethodA template function, the number passed of jvalues are malloced and they will hold later any kind of intended java value. Particular care has to be taken to adequately fill the jvalue array of parameters and the appropriate java_PutArg_<type> has to be used. A call for each argument has to be done and none more. Finally, according to the return type, the appropriate java_CallMethod_Go_<type> should be used, returning the expected value as its last argument.

5.1.9 Predicate building from Java methods

After initializing the JVM, loading one class and getting its methods, it's possible to produce a callable clause, a Prolog predicate, for a loaded method. This is ultimately accomplished by the Prolog written predicate java_Compile/5 (see section A.2.1.1 on page 55). java_Compile creates a clause with the methods name that can be asserted or pretty printed with portray_clause/1. To easily call the java_Compile predicate, without using the helper predicates presented in section C.1 on page 61 such as j_comp/2, the method signature have to be fetched by using

java_MethodID/3, converted to a Prolog signature by use of type/2 before calling java_Compile/5, whose output parameter is the callable clause corresponding to the method provided. j_comp/2 does all this work plus asserting the resulting clause. The java_Compile/5 predicate can be found in the jni_compiler.pl file. It uses a DCG to do the all the conversion between a Java prototype and a callable clause in GNU prolog. It uses all the aspects of the Java/Prolog signature to create a clause that accepts the correct number and type of arguments and returns the correct type in the return value.

5.2 Java to GNU Prolog

The API is this direction is presented. First the way how it has to be done in the JNI architecture. Then the way it was done in the present API development. Finally some conclusions about the usability of the proposed framework.

5.2.1 How to do it with JNI

In the following subsections the steps needed to accomplish the integration of GNU Prolog into Java are presented. A native method in Java is identified using the `native` keyword to modify the method declaration. No body is then defined for that method within the Java class where that method is declared. Instead, the body of the `native` method is defined in a separate C source file. All the `native` methods have a corresponding C function. The `javah` tool takes has input a Java Class file and generates a function prototype for each `native` method declaration. This prototype constrains how the native function is written by defining its input and output. A library was developed that with methods that wrap the **foreign** interface.Let's look at the steps involved in writing Java code to use our particular native wrappers.

5.2.2 Identify native functionality

We have existing code, the GNU Prolog to C (**foreign**) API, which we want to deploy to Java through some wrappers. `javah` assumes a C calling interface between Java and native code. All that there is to be done is to get the call stack correct when calling the API from Java.

5.2.3 Describing the interface to the native code

The existing interface is mapped directly onto Java native method declarations.

5.2.4 Writing the java code

Declaring a `native` method within a Java class is as simple as the use of the `native` attribute keyword. It's method names are simply preceded with the key-

word and no body of the method is supplied. From a syntactical perspective, the abstract and the native keywords are identical. They both defer the method implementation. In the case of an `abstract` method a subclass defines the method. In the case of a `native` method the implementation has to be defined in a C source file. The tool `javac` is then, as usual, used to compile the class files for the next tool `javah` to process for the include files, with the function prototypes, generation. Once the Java class files with native methods are created `javah` takes them as input to produce a C header file with a function prototype for each `native` method declared in the input class file. When using `javah` to produce the header files, care must be taken to use the option `-jni` to produce a JNI style function prototypes as opposed to the old-style (JDK 1.0) function prototypes. The name mangling used by the JNI 1.1 specification states that for a class `Clazz` with a native method `NativeMethodz` a prototype with the mangled name `Java_Clazz_NativeMethodz` is created. The generated header file is then available for inclusion.

5.2.5 Writing the native code

The native code has the implementation of all the methods declared has `native` in the class file. It follows strictly the C argument passing convention and, as seen in the header files generated, for each method in the class a function is defined with two extra arguments.

- a JNIEnv pointer as its first argument
- an object reference as its second

All the types of the input arguments and the return value are defined by the JNI[1]. The C source files include the header files generated in the above step and proceeds with the native implementation of the desired functionality. In our intended line of work only the wrapping of the GNU Prolog to C API functions has to be done to provide to Java the functionality of the `foreign` API.

5.2.6 Building the library

The library is built as a `shared object library` with the special suffix `.so`. To create the gprologJava.o object file needed to build the shared library a compilation made with the aid of the GNU Prolog compiler (`gplc`) is used, to include the GNU Prolog machinery. In the example presented here the following line is used:

```
gplc -C -O -C -D_REENTRANT -C -fpic -C \
 -I$(JAVA_HOME)/include \
-C -I$(JAVA_HOME)/include/linux -c gprologJava.c
```

[1]Look at `jni.h` in the `include` directory under the Java home directory

where JAVA_HOME is the Java home directory. For the gcc and gplc compilers to find the required include files (jni.h and jni_md.h) when linking it is perhaps necessary to provide a -I directive in the compilation line. In the above mentioned example the line used was:

```
gcc -shared -o libgprologJava.so gprologJava.o \
 $(PROLOG_HOME)/lib/obj_begin.o \
 $(PROLOG_HOME)/lib/obj_end.o \
 $(PROLOG_HOME)/lib/libbips_pl.a \
 $(PROLOG_HOME)/lib/libengine_pl.a \
 $(PROLOG_HOME)/lib/liblinedit.a $(C_INCLUDES)
```

Where PROLOG_HOME is the GNU Prolog home directory and C_INCLUDES are the necessary directories for Java shared object assembling. It can be checked in the appendix Makefile that the definition of C_INCLUDES is:

```
C_INCLUDES= -I$(JAVA_HOME)/include \
            -I$(JAVA_HOME)/include/linux \
            -I$(JAVA_HOME)/src/launcher \
            -L -L$(JAVA_HOME)/jre/lib/i386/classic \
            -L -ljvm \
            -L -L$(JAVA_HOME)/jre/lib/i386/native_threads \
            -L -lhpi \
            -L -L$(JAVA_HOME)/jre/lib/i386 -L -lverify \
            -L -L.
```

So, a libgprologJava.so shared library is created and this is the object file that has to be linked when using native GNU Prolog functionality in Java.

5.2.7 Loading and invoking the `native` methods

According to the *Java Language Specification*, code within a static initialization block gets executed when the class is initialized. A class is initialized at its first active use. An active use includes the invocation of a method declared in the class. This would include a native method invocation and, therefore, you can be guaranteed that the library containing the code that implements the native methods will be loaded if the loading is done within the class declaring the native methods. The class method System.loadLibrary must then be invoked. A way to do this is by placing a static initializer block within the class that declares the native methods. This is the way it was used in the attached example gprologJava.java.

5.2.8 The whole picture

Summarizing, there are a number of steps that need to be accomplished to use native functionality within Java:

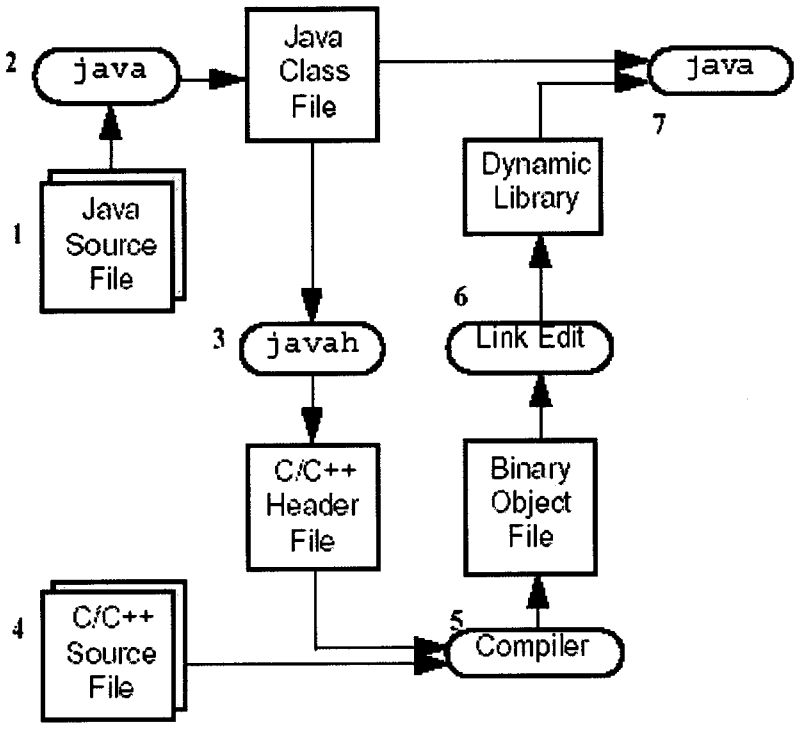

Figure 5.1: Steps in JNI compiling

1. Write Java source file
2. Run `javac` to produce a class file
3. Run `javah` with the previous class file to produce the function prototypes
4. Write C source to functions prototypes generated by `javah`
5. Compile C file to produce object files
6. Run linker to produce a dynamically loadable library
7. Run java on class file produced before

5.2.9 The present solution

Since there was a very well defined API for C connection to GNU Prolog why not to emulate the C API with wrappers written in Java that accomplish the same functionality. These wrappers were defined in a library that is to be called with

the `System.loadLibrary` class method. A library `libgprologJava.so` provides the JNI functions. It can be located in both the client and server and has to be accessible to the java environment. The Java methods were named with the exact same names of the wrapped functions in the GNU Prolog API. So, for instance, we have `Pl_Query_Begin` as a method name to wrap the `Pl_Query_Begin` function in the GNU Prolog **foreign** API.

5.2.10 Loading the library, and running a Java application

As seen above in the invoking methods section (see 5.2.7 on page 39) the library gprologJava can be referenced in a static block initializer. Then when invoking one of the native methods it is actually loaded. The library gprologJava.so must then be present in the defined path for shared objects inclusion, normally defined through the LD_LIBRARY_PATH environment variable.

5.3 Building the solution

The creation of the executables with the aid of the `make` utility has been mentioned in the respective sections. Appart from the creation, it is also necessary to define the appropriate environment for execution, specially for the dynamic linking necessary in Java. The most confortable way of gathering the correct directories for library finding is by use of the `ldconfig` utility. Since several libraries of Java are needed (`libjvm.so`, `libhpi.so`, ...) their locations should be included in the configuration file `ld.so.conf` and the `ldconfig` utility invoked. The locations of these libraries are the `JAVA_HOME` directory itself as well as both the `$JAVA_HOME/classic` and `$JAVA_HOME/native_threads` subdirectories.

Chapter 6

Conclusions and future work

6.1 Conclusions

As devised from the previous two chapters this implementation was made to be a thin, efficient way of communication between the two environments. Some facilities can now be built upon the proposed ones to embellish the usage mainly from the Prolog programmers point of view. This is due to options taken explicitly at start and also due to the time for completion depletion that the time taken to accomplish this work was incurring.

6.2 Future work

6.2.1 Exception handling

One aspect that assumes great importance in the modern languages is a structured, uniform way to handle unexpected behaviors from a program. This has been handled thoroughly in GNU Prolog but mostly in Java where a very complete and standartized exception handling is defined. The integration of both methods of handling exceptions with passing of signals from one environment to the other has not been addressed. The development level of the work at this stage has much more important TODOs than this.

6.2.2 Manipulating Java variables

JNI includes in its definition inumerable methods to manipulate Java variables. These methods differ about the type. For setting an instance field the methods Set<type>Field are used being <type> the usual names for every of the distinct JNI types similar to the functions for method invocation (see 5.1.4 on page 32). For static (class) fields the equivalent SetStatic<type>Field are used. This methods should be facilitated to the native environment to enable GNU Prolog to handle variables in Java.

6.2.3 Multiple JVM handling and Java multithreading with different JVMs

It sounds a good idea to make use of the multithreading capabilities of Java in which JNI has a friendly architecture. However multithreading synchronizing in Prolog is still a pioneering subject and so it is left as a probable development using this proposed framework.

6.2.4 Making it work with dynamic linking of the native Prolog functionality

This is a development where it is particularly difficult to measure the possibility to be achieved. Currently GNU Prolog is an open source project developed under GNU (who could imagine) that maintains a maillist `users-prolog@gnu.org`. This list is intended to serve as a communication tool for developers involved. However, a low activity is present in this list currently. No answers are given to questions about future development so short term developments will probably not be held.

6.2.5 Making it work without having to link dynamically in the JNI

In this work, at first, not aware of the dinamic linking nature of JNI and the fact that GNU Prolog is compiled in non-PIC mode the method picked to implement was the one-to-one mapping sugested in [4]. The advantage of one-to-one mapping is that it is typically more efficient in converting the data types that are transferred between the Java virtual machine and native code. This one-to-one mapping is used since top performance is intended and portability matters. This, although possible is not very simple to acomplish in a short term. It was searched mainly if different Java VMs could allow directly the use of JNI without dinamic loading. Generally this is not possible and all the JVMs checked always use the same standardized method of providing the native functionality to Java (the `System.loadLybrary` method that issues a `dlopen`). When searching mainly the gcj implementation [1] a possibility was found. First some clarifications about terms should be introduced[2]:

- Static library

  Is simply a collection of ordinary object files. These object files are linked into the executable at compile time.In theory, code in static ELF libraries that is linked into an executable should run slightly faster (by 1-5

- Shared library

  Shared libraries are libraries that are loaded by programs when they start. They have to be found by the program so, normally, they reside in a

[1] http://gcc.gnu.org/java/

[2] http://www.dwheeler.com/program-library/Program-Library-HOWTO/index.html

directory that figures in a list of searchable directories by the programs in a specific environment. They are built by compiling the source in position independent mode (-fpic or -fPIC so that the references are relocateble and linked into libraries with special names (.so in UNIX-like systems or .DLL in Windows systems). For performance reasons the library placement usually uses a cacheing system that involves the "installation" of libraries prior to use. All these different steps have to be taken in order to install the shared libraries needed for Java to issue a `System.loadLibrary` method call.

- Dinamically Loaded library

  Dynamically loaded (DL) libraries are libraries that are loaded at times other than during the startup of a program. DL libraries aren't really a different kind of library format (both static and shared libraries can be used as DL libraries); instead, the difference is in how DL libraries are used by programmers[8], there is an API for opening a library, looking up symbols, handling errors, and closing the library. Since this API is rather different for many flavors of Operating Systems a wrapper library should be used when trying to achieve greater portability. This is, of course, one of the reasons why it was not addressed in this work.

How does a static library fills the need for the invocation of `System.loadLibrary` ? The answer lies on the standard class loader itself. Native libraries are located by class loaders. Class loaders have many uses in the Java virtual machine including, for example, loading class files, defining classes and interfaces, providing namespace separation among software components, resolving symbolic references among different classes and interfaces, and finally, locating native libraries. Class loaders provide the namespace separation needed to run multiple components (such as the applets downloaded from different web sites) inside an instance of the same virtual machine. A class loader maintains a separate namespace by mapping class or interface names to actual class or interface types represented as objects in the Java virtual machine. Each class or interface type is associated with its defining loader, the loader that initially reads the class file and defines the class or interface object. Two class or interface types are the same only when they have the same name and the same defining loader.The virtual machine does not allow a given JNI native library to be loaded by more than one class loader. Attempting to load the same native library by multiple class loaders causes an `UnsatisfiedLinkError` to be thrown. The purpose of this restriction is to make sure that namespace separation based on class loaders is preserved in native libraries. Without this restriction, it becomes much easier to mistakenly intermix classes and interfaces from different class loaders through native methods. The virtual machine unloads a native library after it garbage collects the class loader associated with the native library. Because classes refer to their defining loaders, this implies that the virtual machine has also unloaded the class whose static initializer called `System.loadLibrary` and loaded the native library. The virtual

machine attempts to link each native method before invoking it for the first time. It is possible to call the JNI function RegisterNatives to register the native methods associated with a class, this is the way how it has to be done when using statically linked functions. The programmer has to manually link native methods by registering a function pointer with a class reference, method name, and method descriptor. With these points clear is evident that what is needed is a DL library no matter wether it is Static or Shared library. The point now is how to produce static linking of JNI code. With the gcj JVM it seems possible to be done and the main issues are:

- All the application shall be built with the *-static* option. This option normally is not recommended for it has some unsolved issues in several architectures.
- The above has to be done in a system with a gcc-toolchain well parameterized. It has to be configured from the ground up with *–disable-shared.* This is not a feasible thing in a regular development system since it compromises the shared architecture in which most of the applications are based currently so it can only, normally, be done in a special purpose system. This point merely states that what is needed is a static libgcj.
- A thorough check has to be done if the resulting executable is not linking to any other DSO apart from the system libraries (libc, libm and libpthread). And this stretches the interest of our system since it means that it can't be linked with most of the interesting libraries that exist nowadays (AWT, SWING, GTK2, ...). There are, nonetheless, very interesting statically linked libraries that, due to their interest, were ported to static libraries like the SWT library that forms the basis of eclise IDE for instance.

Having all this setup the only thing left is to pack all the JNI archives into a whole archive wrapper. The linker cannot know how any method in the archive is going to be used and that the methods regarding to JNI are registered native so they will all have to be wrapped up.

6.2.6 Minor arrangements

6.2.6.1 Setting a different CLASSPATH

So far there is no support for class loading from different locations. The class files have to be in the directory where both gpl_java and gprologjava are. By simple handling of the proper environment variables (CLASSPATH of course) it is easy to include classes from different locations.

6.2.6.2 Maintaining a Prolog signature in the methods struct

Although a simple feature to implement it could lead to a considerable optimization since by its use, the build a rebuild of Prolog signatures back and forth dont have to be done.

6.2.6.3 An easier way for creating Java objects in Prolog

The multi step manner of creating an Object that is currently implemented does not benefit of the Prolog like signature that was invented to pick the intended constructor. A Prolog like way of building an object could be provided in the same way that jni_compile/4 is provided. With this facility it should be easier for the Prolog programmer to call a sole predicate that given a Prolog signature of the constructor and a compound term with the arguments would pick the correct constructor and build the object. Such predicate can be easily implemented now in gpl_java using Prolog. This new predicate would convert the Prolog signature using type. Scan the arguments provided in the compound term to build the correct structure for the NewObjectX function and call it. It could use whatever of the three NewObjectX JNI functions available depending on how it was easier to convert between the compound term and a Java list of values, an array of jvalues or an arbitrary list of arguments. This new predicate would make all the process transparent to the Prolog programmer.

6.2.6.4 Unneeded creation of objects when calling static methods

As can be seen by the usage of gpl_java, there is some incongruency in compulsively creating objects which are really not needed since the only purpose is to call a static (class) method. Actually the objects parameters don't even need to be filled since when calling the method through the associated predicate the methods arguments are provided at that moment. A better way for the interface to be presented would be to let the java_Compile predicate to decide whether it it is needed to accept an object as parameter in case it is an instance method or not. This slightly different approach would avoid the unnecessary creation of objects just for argument accordance in the predicate calling.

Bibliography

[1] Mutsunori Banbara and Naoyuki Tamura. Translating a linear logic programming language into java. In *Proceedings of ICLP'99 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 19–39, 1999.

[2] Daniel Diaz. *GNU PROLOG A Native Prolog Compiler with Constraint Solver over Finite Domains*. INRIA, 1.6 edition, Jun 2001. for GNU Prolog version 1.2.14.

[3] Rob Gordon. *Essential JNI*. Prentice Hall, Inc., 1998.

[4] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification, The*. Addison Wesley, June 1999.

[5] N. Tamura and Y. Kaneda. Extension of WAM for a linear logic programming language. In A. Ohori T. Ida and M. Takeichi, editors, *Second Fuji International Workshop on Functional and Logic Programming*, Nov 1996.

[6] Paul Tarau and M. Boyer. *Elementary logic programms*. Number 456 in Lecture notes in computer science. Springer, procedings of programming language implementation and logic programming edition, Aug 1990.

[7] David Warren. An abstract prolog instruction set. Technical Report 309, SRI International, Oct 1983.

[8] David A. Wheeler. Program library howto. Technical report, The Linux Documentation Project, April 2003. Revision 1.2.

Appendix A

Reference

Below is the reference for all the API functions. First the low-level (GPL_java_C) API and then the higher level API (GPL_java_PL) predicates are introduced. Presentation is based on the GNU Prolog manual way with a templates and description sections for each API function. Later on the interface from Java to Prolog (gprologJava) is presented. The class is presented using the Javadoc usual way

A.1 Prolog to Java

The interface when considered this way allows the calling of Java from GNU-Prolog. The usual way of doing it is by:

- Create a JVM using java_CreateJavaVM
- Load a class using java_GetClass
- Load the classes methods using java_GetMethods
- Pick a method using java_MethodID
- Build an object using java_New
- Create space for arguments with java_Create_Args
- Load the different parameters with java_PutArg_XXX
- Convert the method's signature between Java and GNU-Prolog using type
- Call the method with the appropriate java_CallMethod_Go_XXX
- Compile the method into Prolog using Java_Compile
- Assert the resulting clause to the Prolog database
- Call the method in Prolog
- Deallocate the space with java_CallMethod_Free

A.2 GPL_java_C

In this API functions that are deterministic always return a boolean with the success or failure values. Non deterministic functions return a choice point.

A.2.0.5 java_CallMethod/4

Templates
java_CallMethod(+object_term, +methods_index, +args_list, -result_term), [return(boolean)]
Description
Calls an instance method belonging to the object named in the first arg. The method called is loaded in the methods array in the position given in the second arg. The arguments are provided as a list in the third arg. The result is positioned as an object compound term in the final arg.
Notes
Deprecated manner of calling a method without allocating memory explicitly. It lacks the precise definition of the result as it is returned as a compound term.

A.2.0.6 java_GetVersion/3

Templates
java_GetVersion(+JVM_index, -VersionMajor, -VersionMinor), [return(boolean)]
Description
Given the first argument with the JVM serial number (beginning with 0), returns the major digit on the second argument (always 1) and the minor digit on the third argument (1 or 2) of the JVM version number
Notes
It should work properly with different JVMs loaded/created allthough currently only one JVM can be loaded with java_CreateJavaVM.

A.2.0.7 java_CreateJavaVM/1

Templates
java_CreateJavaVM(-JVM_index), [return(boolean)]
Description
Creates a JVM, loads it into the JVMs array and returns its index in the array.
Notes
Currently only one JVM can be loaded/created so it always goes into position 0.

A.2.0.8 java_GetJavaVM/2

Templates
java_GetJavaVM(+JVM_index, -JVM), [return(none)]
Description
Returns a JVM (pointer) of the index mentioned in the first argument. **Notes**

Unused since the helper classes table provides this information locally and deprecated since it was chosen not to multithread.

A.2.0.9 java_DestroyJavaVM/1

Templates
java_DestroyJavaVM(+JVM_index), [return(none)]
Description
Destroys the JVM in the named index and deallocates the corresponding memory
Notes

A.2.0.10 java_GetClass/3

Templates
java_GetClass(+JVM_index, +Class, ?ClassIndex), [return(none)]
Description
It functions according to the third argument. If it is an output argument it loads in the JVM named in the first argument the class whose fully qualified name is the second argument. If it is an input argument it checks if the class loaded in the position named is that mentioned in the second argument.
Notes
It may be called recursively until the top class java.lang.Object is reached and so it may be used, with the aid of the following java_SuperClass/2 function, cleanly for a class browser implementation.

A.2.0.11 java_SuperClass/2

Templates
java_SuperClass(?ClassIndex, ?SuperClassIndex), [choice_size(1)]
Description
Fully qualified names on both arguments of a class, in the first, and its relative superclass, in the second.
Notes
Non-deterministic if at least one of the two arguments is variable.

A.2.0.12 java_GetMethods/1

Templates
java_GetMethods(+ClassIndex), [return(none)]
Description
Given a class index in the argument loads its methods into the methods helper table.
Notes
Does not load any method in the class hierarchy but the named class methods.

A.2.0.13 java_MethodID/4

Templates
java_MethodID(+ClassIndex, ?MethodName, -MethodSIG, -MethodID), [choice_size(1)]
Description
Given the class index in the first argument it functions non-deterministically according to the second argument. In the third argument the Java signature of the method is always returned. In the fourth argument the index of the method in the methods helper table is returned. If the second argument is variable the name of the methods in the helper table are successively returned. If it is unified the signature and position of the named method are returned deterministically.
Notes

A.2.0.14 java_Create_Args/2

Templates
java_Create_Args(+NoArgs, -ArgsArray), [return(boolean)]
Description
Allocates space and returns a pointer where to insert ulterior arguments. The first argument being the number of arguments to allocate space for and the second a pointer to the space created.
Notes
The function allocates a number of jvalue (see section 5.1.4 on page 33) slots where different sizes of arguments from the GNU-Prolog point of view can surely be stored.

A.2.0.15 java_PutArg_int/4

Templates
java_PutArg_int(+ArgsArray, +ArgNo, +ArgValue, +SubType), [return(boolean)]
Description
Insert into the arguments array, previously allocated with java_Create_Args (see section A.2.0.14 on page 51), into the position named in the second argument the value given in the third. The particular kind of integer has to be referred in the last argument and it has to be: 0 for char, 1 for byte, 2 for int, 3 for long, 4 for boolean and 5 for a short.
Notes
All the java_PutArg_XXX functions named from now on are used to fill arguments and have to be called prior to method calling with the appropriate java_CallMethod_Go_XXX or object creation with java_New.

A.2.0.16 java_PutArg_float/3

Templates
java_PutArg_float(+ArgsArray, +ArgNo, +ArgValue, +SubType), [return(boolean)]
Description
Insert into the arguments array, previously allocated with java_Create_Args (see section A.2.0.14 on page 51), into the position named in the second argument the value given in the third. The particular kind of real number has to be referred in the last argument and it has to be: 0 for float and 1 for double.
Notes

A.2.0.17 java_PutArg_String/3

Templates
java_PutArg_String(+ArgsArray, +ArgNo, +ArgValue), [return(boolean)]
Description
Insert into the arguments array, previously allocated with java_Create_Args (see section A.2.0.14 on page 51), into the position named in the second argument the value given in the third.
Notes

A.2.0.18 java_PutArg_obj/3

Templates
java_PutArg_obj(+ArgsArray, +ArgNo, +ArgValue), [return(boolean)]
Description
Insert into the arguments array, previously allocated with java_Create_Args (see section A.2.0.14 on page 51), into the position named in the second argument the value given in the third.
Notes

A.2.0.19 java_CallMethod_Go_int/4

Templates
java_CallMethod_Go_int(+Object, +MethodID, +ArgsArray, -ReturnValue)
Description
Given an object in the first argument, invokes the method whose ID is the second parameter with the arguments present in the third and returns an integer in the last parameter.
Notes
This predicate should only be called with methods that return integer values, with a signature with template (XXX)I.

A.2.0.20 java_CallMethod_Go_float/4

Templates
java_CallMethod_Go_float(+Object, +MethodID, +ArgsArray, -ReturnValue)
Description
Given an object in the first argument, invokes the method whose ID is the second parameter with the arguments present in the third and returns a float in the last parameter.
Notes
This predicate should only be called with methods that return integer values, with a signature with template (XXX)F.

A.2.0.21 java_CallMethod_Go_void/3

Templates
java_CallMethod_Go_void(+Object, +MethodID, +ArgsArray)
Description
Given an object in the first argument, invokes the method whose ID is the second parameter with the arguments present in the third.
Notes
This predicate should only be called with methods that return nothing (return void in Java), with a signature with template (XXX)V.

A.2.0.22 java_CallMethod_Go_obj/4

Templates
java_CallMethod_Go_obj(+Object, +MethodID, +ArgsArray, -ReturnValue)
Description
Given an object in the first argument, invokes the method whose ID is the second parameter with the arguments present in the third and returns an object in the last parameter.
Notes
This predicate should only be called with methods that return integer values, with a signature with template (XXX)L<Fully-qualified-name>; .

A.2.0.23 java_CallMethod_Go_String/4

Templates
java_CallMethod_Go_String(+Object, +MethodID, +ArgsArray, -ReturnValue)
Description
Given an object in the first argument, invokes the method whose ID is the second parameter with the arguments present in the third and returns a java String in the last parameter.
Notes

This predicate should only be called with methods that return nothing (return void in Java), with a signature with template (XXX)Ljava.lang.String;.

A.2.0.24 java_CallMethod_Free/1

Templates
java_CallMethod_Free(+ArgsTerm)
Description
Rebuilds the java reference given in the Prolog term in the parameter and deallocates the memory referenced.
Notes

A.2.0.25 java_CallStaticVoidMethod/3

Templates
java_CallStaticVoidmethod(+JVM_index, +ClassName, +MethodName)
Description
Invokes the void method named in the third parameter belonging to the class named in the second. **Notes**
This predicate is deprecated since the using of the the callback tables which provide a better way to assure type conformity in the arguments and the returning value. It can only be called, as the name suggests, with void methods. It was only kept alive for it is a point of proof that the static methods don't need an object created just for the purpose of calling.

A.2.0.26 java_New/4

Templates
java_New(+JVM_index, +ClassIndex, +ArgsArray, -Object)
Description
In the JVM in the first argument and the class in the second argument, picks a constructor and applies the parameters in the third argument to create an object that is returned in the last argument.
Notes
Since JNI allows, using reflection, to pick a constructor correctly is is not needed to name it explicitly.

A.2.1 GPL_java_PL

The following predicates are built in Prolog so the parameter direction does not make much sense. It is provided here only as an indication of the most common use for better prolog usability.

A.2.1.1 java_Compile/5

Templates
java_Compile(+JVM, +CLASS, +METHOD, +PLTYPE, -CLAUSE)
Description
Succeeds if in the JVM, in CLASS, there is a METHOD with prolog signature PLTYPE and creates the CLAUSE.
Notes
The clause obtained from this predicate is suitable for prolog calling so its most common use should be assertion or pretty printing

A.2.1.2 type/2

Templates
type(?PLTYPE, ?JTYPE)
Description
Utility for converting between Prolog (PLTYPE) and Java (JTYPE) signatures
Notes

A.3 Java to GNU Prolog

In this section the Java methods created to wrap the **foreign C** interface are presented. The aspect is based in the Javadoc standard where applicable. The following, with exception of Main_Wrapper, are all created as native implementations and defined in the class **gprologJava**.

A.3.1 Main_Wrapper

Templates
public native int Main_Wrapper(int argc, String args[])
Description
This wrapper only functions as a debug aid where different invocations of the defined methods can be tested.
Notes

A.3.2 Pl_Query_Begin

Templates
public void Pl_Query_Begin(boolean recoverable)
Description
This is the wrapper for the function Pl_Query_Begin. It is used to initialize a query.
Notes

The recoverable parameter shall be set to true if the user wants to recover, at the end of the query, the memory space consumed by the query.

A.3.3 Pl_Query_Call

Templates
public int Pl_Query_Call(int functor, int arity, java.lang.Object[] arg)
Description
This is the wrapper for the function Pl_Query_Call. The function Pl_Query_Call(functor, arity, arg) calls a predicate passing arguments.
Notes
It is used to compute the first solution.

A.3.4 Pl_Query_Next_Solution

Templates
public int Pl_Query_Next_Solution()
Description
This is the wrapper for the function Pl_Query_Next_Solution. It is used to compute a new solution.
Notes
It must be only used if the result of the previous solution was PL_SUCCESS (i.e. TRUE).

A.3.5 Pl_Query_End

Templates
public void Pl_Query_End(int op)
Description
This is the wrapper for the function Pl_Query_End. It is used to finish a query.
Notes
This function mainly manages the remaining alternatives of the query. However, even if the query has no alternatives this function must be used to correctly finish the query.

A.3.6 Pl_Get_Exception

Templates
public java.lang.Object Pl_Get_Exception()
Description
This is the wrapper for the function Pl_Get_Exception. It can be used to obtain the exceptional term raised by throw/1. **Notes**

A.3.7 Pl_Exec_Continuation

Templates
public void Pl_Exec_Continuation(int functor, int arity, java.lang.Object[] arg)
Description
This is the wrapper for the function Pl_Exec_Continuation. It replaces the current calculus by the execution of the specified predicate.
Notes

Appendix B

Compiling and running

The compile/link lines can be checked in the following Makefile:

```
# Makefile for gprolog <-> Java interface
JAVA_HOME=/opt/j2sdk1.3
PROLOG_HOME=/usr/local/gprolog-1.2.14
INCLUDES=-C -I$(JAVA_HOME)/include \
            -C -I$(JAVA_HOME)/include/linux \
            -C -I$(JAVA_HOME)/src/launcher \
            -L -L$(JAVA_HOME)/jre/lib/i386/classic  -L -ljvm \
            -L -L$(JAVA_HOME)/jre/lib/i386/native_threads \
            -L -lhpi \
            -L -L$(JAVA_HOME)/jre/lib/i386 -L -lverify \
            -L -L.

C_INCLUDES= -I$(JAVA_HOME)/include \
            -I$(JAVA_HOME)/include/linux \
            -I$(JAVA_HOME)/src/launcher \
            -L -L$(JAVA_HOME)/jre/lib/i386/classic  -L -ljvm \
            -L -L$(JAVA_HOME)/jre/lib/i386/native_threads \
            -L -lhpi \
            -L -L$(JAVA_HOME)/jre/lib/i386 -L -lverify \
            -L -L.

all : gpl_java gprologJava.so gprologJava.class

clean :
    rm gpl_java

gpl_java : gpl_java.pl gpl_java.h gpl_java.c types.pl Makefile \
           jni_compiler.pl helper.pl
    gplc -v -o gpl_java -C -g gpl_java.pl  gpl_java.c helper.pl \
```

```
            types.pl jni_compiler.pl $(INCLUDES)

#Experimenting gcj
libgcjSample.so : gcjSample.o
    gcc -shared -o libgcjSample.so gcjSample.o

gcjSample.o : gcjSample.c gcjSample.h
    gcc -c gcjSample.c

gcjSample.class : gcjSample.java
    gcj -C gcjSample.java

gcjSample.h : gcjSample.class
    gcjh -jni gcjSample

gcjSample : gcjSample.class
    gcj -fjni -o gcjSample gcjSample.class --main=gcjSample
#End experimenting gcj

#Experimenting gcj -> GNU Prolog
libgcj_test.so : gcj_test.o
    gplc -v -L -shared -L -Wl,-soname,libgcj_test.so \
 -o libgcj_test.so gcj_test.o
#The following is the linking line directly invoking gcc
#   gcc -o libgcj_test.so gcj_test.o \
# /usr/local/gprolog-1.2.14/lib/obj_begin.o -shared \
# -Wl,-soname,libgcj_test.so \
# -L/usr/local/gprolog-1.2.14/lib \
# /usr/local/gprolog-1.2.14/lib/obj_end.o -lengine_pl -llinedit

gcj_test.o : gcj_test.c gcj_test.h
    gplc -v -C -Wall -C -fPIC -L -g -C -I$(JAVA_HOME)/include \
-C -I$(JAVA_HOME)/include/linux -c gcj_test.c

gcj_test.class : gcj_test.java
    gcj -C gcj_test.java

gcj_test.h : gcj_test.class
    gcjh -jni gcj_test

gcj_test : gcj_test.class gcj_test.o
    gcj -fjni -o gcj_test gcj_test.class --main=gcj_test \
 -L. -lgcj_test
```

```
#Java -> GNU Prolog
libgprologJava.so : gprologJava.o
    gcc -shared -o libgprologJava.so gprologJava.o \
 /usr/local/gprolog-1.2.14/lib/libengine_pl.a

gprologJava.o : gprologJava.c gprologJava.h
    gplc -C -O -C -I$(JAVA_HOME)/include \
-C -I$(JAVA_HOME)/include/linux -c gprologJava.c

gprologJava.class : gprologJava.java
    gcj -C gprologJava.java

gprologJava.h : gprologJava.class
    gcjh -jni gprologJava

gprologJava : gprologJava.class
    gcj -fjni -o gprologJava gprologJava.class \
 --main=gprologJava

#gprologJava.so : gprologJava.c gprologJava.h
#   gplc -C -O -C -D_REENTRANT -C -fpic -C -I$(JAVA_HOME)/include \
#-C -I$(JAVA_HOME)/include/linux -c gprologJava.c
#   gplc -C -shared -o libgprologJava.so gprologJava.o $(INCLUDES)

gprologJava.so : gprologJava.c gprologJava.h
    gplc -C -O -C -D_REENTRANT -C -fpic -C -I$(JAVA_HOME)/include \
-C -I$(JAVA_HOME)/include/linux -c gprologJava.c
    gcc -shared -o libgprologJava.so gprologJava.o \
 $(PROLOG_HOME)/lib/obj_begin.o \
 $(PROLOG_HOME)/lib/obj_end.o \
 $(PROLOG_HOME)/lib/libbips_pl.a \
 $(PROLOG_HOME)/lib/libengine_pl.a \
 $(PROLOG_HOME)/lib/liblinedit.a $(C_INCLUDES)
```

Appendix C

Usage examples

C.1 Helper predicates in helper.pl

Some predicates which are usually just shortage of names based upon the assumption that actually only one JVM is used are presented in the file helper.pl. They are also usage examples that can be both apprehended, as a kind of learn by example tool, and used by the Prolog programmer. Such examples/predicates are, for instance, java_Init/1 that initializes a JVM with the class passed as argument loaded and j_comp/2 that compiles the method name given as first argument belonging to the class in the second argument. These predicates make use of j_c_jvm/1 that is java_CreateJavaVM/1, j_gc/3 that is java_GetClass/3, j_gm/1 that is java_GetMethods/1, j_mid/3 that is java_MethodID/4 and j_new/2 that is java_New/4 all these later predicates assume only one JVM and the class intended to be used is the first one loaded.

```
j_c_jvm(JVM) :- java_CreateJavaVM(JVM).
j_gc(JVM,C,Class) :- java_GetClass(JVM, C, Class).
j_gm(C) :- java_GetMethods(C).
j_mid(MName,MSig,M) :- java_MethodID(0, MName, MSig, M).
j_new(MParms, Object) :- java_New(0, 0, MParms, Object).

java_Init(C) :- j_c_jvm(JVM), j_gc(JVM,C,Class), j_gm(Class).
j_comp(MName, CName) :- j_mid(MName,MSig,M), name(MSig, Sig),
  type(PSig, Sig), java_Compile(0, CName, MName,  PSig, Clause),
  assertz(Clause).
```

C.2 Various examples in test_java.pl

As the gpl_java tool was being built, the new features were being tested through examples gathered in this file. So it both shows the usage enhancement and broader capabilities that were being achieved and it can be used as a source of examples of the way gpl_java is supposed to be used. Some examples or, better said, their usage are now deprecated and are no longer of interest but they were kept for the sake of the work development understanding.

```
% The following tests evolve for greater completion:

java_CreateJavaVM(JVM), java_GetVersion(JVM,Major,Minor)

java_CreateJavaVM(JVM), java_GetClass(JVM, testClass, Class),
  java_CallStaticVoidMethod(JVM, Class, printJavaString).

java_CreateJavaVM(JVM), java_GetClass(JVM, testClass, Class),
  java_GetMethods(Class),java_MethodID(Class,
    printHelloWorld, MethodSig, Method),
  java_New(JVM, Class, '()V', '', Object),
  java_CallMethod(Object, Method, ['']).

java_CreateJavaVM(JVM), java_GetClass(JVM, testClass, Class),
  java_New(JVM, Class, '()V', '', Object).

java_CreateJavaVM(JVM), java_GetClass(JVM, testClass, Class),
  java_New(JVM, Class, '(Ljava/lang/String;)V',
    'Testing after hours', Object).

java_CreateJavaVM(JVM), java_GetClass(JVM, testClass, Class),
  java_GetMethods(Class),
  java_MethodID(Class, sumInstanceIntegers, MethodSig, Method),
  java_New(JVM, Class, '()V', '', Object),
  java_CallMethod(Object, Method, [2,4], Res).

java_CreateJavaVM(JVM), java_GetClass(JVM, testClass, Class),
  java_GetMethods(Class),
  java_MethodID(Class, MethodName, MethodSig, Method),
  java_New(JVM, Class, '()V', '', Object),
  java_CallMethod(Object, Method, [2,4], Res).

java_CreateJavaVM(JVM), java_GetClass(JVM, testClass, Class),
  java_GetMethods(Class),
  java_MethodID(Class, sumIntegers, MethodSig, Method),
```

```
  java_New(JVM, Class, '()V', '', Object),
  java_CallMethod(Object, Method, [2,4], Res).

%Now both initializers work:
java_CreateJavaVM(JVM), java_GetClass(JVM, testClass, Class),
  java_GetMethods(Class),
  java_MethodID(Class, printJavaString, MethodSig, Method),
  java_New(JVM, Class, '()V', '', Object),
  java_CallMethod(Object, Method, ['Testing after hours'],Res).

java_CreateJavaVM(JVM), java_GetClass(JVM, testClass, Class),
  java_GetMethods(Class),
  java_MethodID(Class,printJavaString, MethodSig, Method),
  java_New(JVM, Class, '(Ljava/lang/String;)V',
    'Testing even after', Object),
  java_CallMethod(Object, Method,['Testing'], Res).

%For the hierarchy testing
java_CreateJavaVM(JVM), java_GetClass(JVM, testSubClass, Class),
  java_GetMethods(Class),
  java_MethodID(Class,printHelloWorld, MethodSig, Method),
  java_New(JVM, Class, '', Object),
  java_CallMethod(Object, Method,[], Res).

%For the new interface testing
java_CreateJavaVM(JVM), java_GetClass(JVM, testClass, Class),
  java_GetMethods(Class),
  java_MethodID(Class,sumIntegers, MethodSig, Method),
  java_New(JVM, Class, '', Object),
  java_Create_Args(2,Args), java_PutArg_int(Args, 0, 13, 2),
  java_PutArg_int(Args, 1, 14, 2),
  java_CallMethod_Go_int(Object, Method, Args, Res),
  java_CallMethod_Free(Args).


%The following will do
%provided a java_Init(testClass) in helper.pl.

java_Init(testClass).

java_MethodID(0, sumIntegers, MethodSig, Method),
  java_New(0, 0, '()V', Object), java_Create_Args(2, Args),
  java_PutArg_int(Args, 0, 134, 2),
  java_PutArg_int(Args, 1, 453, 2),
```

```
  java_CallMethod_Go_int(Object, Method, Args, Res).

j_mid(printHelloWorld, MSig, M), j_new(MSig, Object),
  java_Create_Args(0, Args),
  java_CallMethod_Go_void(Object, M, Args).

j_mid(printJavaString, MSig, M), j_new(MSig, Object),
  java_Create_Args(1, Args),
  java_PutArg_String(Args, 0, 'Testing String Args'),
  java_CallMethod_Go_void(Object, M, Args).

j_mid(printJavaString, MSig, M), j_new(MSig, Object),
  java_CallMethod(Object, M,
   ['Testing java_CallMethod with void return'], Res).

j_mid(sumIntegers, MSig, M), j_new(MSig, Object),
  java_CallMethod(Object, M, [123, 532], Res).

%From now on using jni_compiler
j_mid(sumIntegers, MSig, M), name(MSig, Sig), type(PSig, Sig),
  java_Compile(0, testClass, sumIntegers, PSig, Clause),
  assertz(Clause),j_new(MSig,Object),
  sumIntegers(Object, 23, 37, Soma).

j_mid(printHelloWorld, MSig, M), name(MSig, Sig),
  type(PSig, Sig),
  java_Compile(0, testClass, printHelloWorld,  PSig, Clause),
  assertz(Clause),j_new(MSig,Object),
  printHelloWorld(Object, _).

%Working
j_comp(printHelloWorld,testClass),  j_new('()V',Obj),
  printHelloWorld(Obj,_).

j_comp(sumIntegers), j_new('()V',Obj),
  sumIntegers(Obj,23,69,Soma).

j_comp(multiplyFloats), j_new('()V',Obj),
  multiplyFloats(Obj,2.3,6.9,Mult).

%Not Working due to different float-double implementations
java_Compile(0, testClass, multiplyFloats,
    [float,float]-float, Clause),
  assertz(Clause), j_new('(FF)F', Obj),
```

```
  multiplyFloats(Obj, 2.3, 3.4, Mult).

%New definition of java_New
java_Create_Args(2, Args), java_PutArg_int(Args, 0, 23, 2),
java_PutArg_int(Args, 1, 37, 2), java_New(0,0,Args, Object).
```