A Most Specific Method Finding Algorithm for Reflection Based Dynamic Prolog-to-Java Interfaces

Satyam Tyagi and Paul Tarau

Department of Computer Science
University of North Texas
P.O. Box 311366
Denton, Texas 76203
E-mail: {*tyagi,tarau*}*@cs.unt.edu*

Abstract. In the context of direct and reflection based extension mechanisms for the Jinni 2000 Java based Prolog system, we discuss the design and the implementation of a reflection based Prolog to Java interface. While the presence of dynamic type information on both the Prolog and the Java sides allows us to automate data conversion between method parameters, the presence of subtyping and method overloading makes finding the most specific method corresponding to a Prolog call pattern fairly difficult. We describe a run-time algorithm which closely mimics Java's own compile-time method dispatching mechanism and provides accurate handling of overloaded methods beyond the reflection package's limitations. As an application of our interfacing technique, a complete GUI library is built in Prolog using only 10 lines of application specific Java code.

Keywords: *Java based Language Implementation, Prolog to Java Interface, Method Signatures, Dynamic Types, Method Overloading, Most Specific Method, Reflection*

1 Introduction

In this paper, we discuss the extension of the Jinni 2000 Java based Prolog system [6, 8] with a reflection based generic Java Interface. After overviewing the Jinni architecture we describe the Prolog API (Application Programmer Interface) used to invoke Java code and the mapping mechanism between Prolog terms and their Java representations. We next discuss the implementation of a reflection based Prolog-to-Java interface. We will overcome some key limitations of Java's Reflection package (a Java API which accesses Java objects and classes dynamically). The main problem comes from the fact that reflection does its work at run time. Although the called classes have been compiled - the invoking code needs to determine a (possibly overloaded) method's signature dynamically - something that Java itself does through fairly extensive compile time analysis.

First, we discuss the desired functionality provided by Java at compile time and explain it through a simple example. Subsequently, we provide the algorithm behind our implementation, which achieves at runtime the functionality that Java provides at compile time. We also show some nice properties of our algorithm such as low computational complexity. Finally we describe an example application (a GUI for Jinni) developed almost completely in Prolog using the reflection API.

The Method Signature Problem Most modern languages support method overloading (the practice of having more than one method with same name). In Java this also interacts with the possibility of having some methods located in super classes on the inheritance chain. On a call to an overloaded method, the resolution of which method is to be invoked is based on the method signature. Method signature is defined as the name of the method, its parameter types and its return type[1].

The problem initially seems simple: just look for the methods with the same name as call, number and type of parameters as the arguments in the call and pick that method.

The actual problem arises because Java allows *method invocation type conversion*. In other words this means that we are not looking for an exact match in the type of a parameter and the corresponding argument, but we say it is a match if the type of argument can be converted to the type of a corresponding parameter by method invocation conversion [2]. Apparently, this also does not seem to be very complicated: we just check if the argument type converts to the corresponding parameter type or not. The problem arises because we may find several such matches and we have to search among these matches the **most specific method** - as Java does through compile time analysis. If such a method exists, then that is the one we invoke. However, should this search fail, an error has to be reported stating that no single method can be classified as the most specific method.

This paper will propose a comprehensive solution to this problem, in the context of the automation of type conversions in Jinni 2000's bidirectional Prolog to Java interface.

2 The Jinni Architecture

Jinni 2000 consists of a combination of fast WAM based Prolog engines using integer arrays for WAM data areas together with a Java class based term hierarchy - on which runs a lightweight Java based Interpreter, interoperating with the internal tagged integer based WAM representation, through an automated bidirectional data conversion mechanism.

Jinni's Java interface is more flexible and uses programmer- friendly Java class hierarchy of its interpreted Kernel Prolog [8] engines, instead of the high

[1] In resolving the method call Java ignores the return type.

performance but fairly complex integer based term representations of its WAM based BinProlog-style engines [9].

In this section we will explain the mapping from Prolog term types to Java classes.

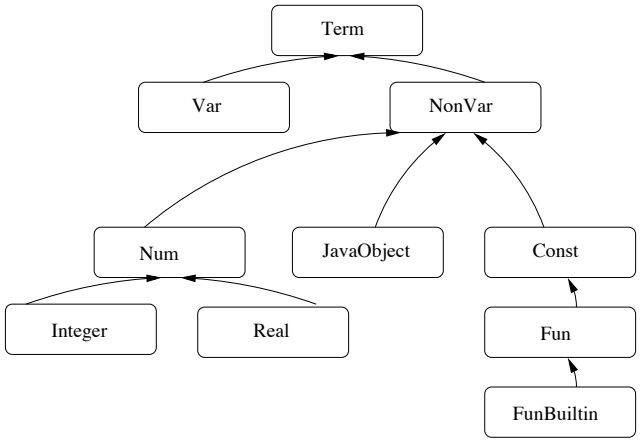

Fig. 1. Java Classes of Prolog Term Hierarchy

2.1 The Term Hierarchy

The base class is **Term** which has two subclasses: **Var** and **NonVar**. The **NonVar** class is in turn extended by **Num**, **JavaObject** and **Const**. Num is extended by **Integer** and **Real**. **Term** represents the generic Prolog term which is a finite tree with unification operation distributed across data types - in a truly object oriented style [6]. The **Var** class represents a Prolog variable. The **Integer** and **Real** are the Prolog Numbers. **Const** represents all symbolic Prolog constants, with the compound term (called *functor* in Prolog) constructor class **Fun** designed as an extension of **Const**.

JavaObject is also a subclass of **Const** which unifies only with itself[2] and is used like a wrapper around Objects in Java to represent Prolog predicates.

2.2 The Builtin Registration Mechanism

Jinni's Builtins class is a specialized subclass of Java's Hashtable class. Every new component we add to Jinni 2000 can provide its own builtin predicates as a subclass of the Builtins class. Each added component will have many predicates, which are to be stored in this Hashtable mapping their Prolog representation to their Java code, for fast lookup. Let us assume the Prolog predicate's name

[2] Modulo Java's *equals* relation.

is *prologName* and the corresponding Java classes name is *javaName*. We make a class called *javaName* which extends FunBuiltin (a descendant of Term with which we represent a Prolog functor (compound term). It accepts a string (the functor's name) and an integer in its constructor (arity). When we call the **register** method of the appropriate descendant of the Builtins class, a new Hashtable entry is generated with the supplied *prologName* and the arity as key and *javaName* as its value. Whenever the Prolog predicate *prologName* with appropriate arity is called, we can look up in constant time which class (*javaName*) actually implements the **exec** method of the builtin predicate in Java. Each component extending the Builtins class will bring new such predicates and they will be added to the inherited Hashtable with the mechanism described above - and therefore will be usable as Prolog builtins as if they were part of the Jinni 2000 kernel.

2.3 The Builtin Execution Mechanism

The descendents of the **FunBuiltin** class implement builtins which pass parameters, while the descendents of the **ConstBuiltin** class implment parameterless builtins. Both **FunBuiltin** and **ConstBuiltin** have an abstract method called **exec** to be be implemented by the descendent *javaName* class. This is the method that is actually mapped to the Prolog builtin predicate with *prologName* and gets invoked on execution of the predicate. The **exec** method implemented by the *javaName* class will get arguments (**Term** and its subclasses) from the predicate instance using **getArg** methods and will descover their dynamic through a specialized method. Once we have the arguments and know their types we can do the required processing. The **putArg** method, used to return or check values, uses the **unify** method of Terms and its subclasses to communicate with the actual (possible variable) predicate arguments. On success this method returns **1**. If putArg does not fail for any argument the exec method returns **1**, which is interpreted as a **success** by Prolog. If at least one unification fails we return **0**, which is interpreted as a **failure** by Prolog. We call this mapping a *conventional builtin* as this looks like a builtin from Prolog side, which is known at compile time and can be seen as part of the Prolog kernel.

3 The reflection based Jinni 2000 Java Interface API

Our reflection based Jinni 2000 Java Interface API is provided through a surprisingly small number of conventional Jinni builtins. This property is shared with the JIPL [3] interface from C-based Prologs to Java. The similarity comes ultimately from the fact that Java's reflection package exhibits to Java the same view provided to C functions by JNI - the Java Native Interface:

new_java_class(+'ClassName',-Class).

This takes in as first argument the name of the class as a constant. If a class with that name is found it loads the class and a handle to this class is returned in the second argument wrapped inside our *JavaObject*. Now this handle can be used to instantiate objects.

new_java_obj(+Class,-Obj):-new_java_obj(Class,new,Obj).
new_java_obj(+Class,+new(parameters),-Obj).

This takes in as the first argument a Java **class** wrapped inside our *JavaObject*. In the case of a *constructor* with parameters, the second argument consists of **new** and parameters (Prolog numeric or string constants or other objects wrapped as *JavaObjects*) for the constructor. As with ordinary methods, the (most specific constructor) corresponding to the argument types is searched and invoked. This returns a handle to the new object thus created again wrapped in *JavaObject* in the last argument of the predicate. If the second parameter is missing then the void constructor is invoked. The handle returned can be used to invoke methods: **invoke_java_method(+Object, +methodname(parameters), -ReturnValue).**

This takes in as the first argument a Java class's instantiated object (wrapped in *JavaObject*), and the method name with parameters (these can again be numerical or, string constants or objects wrapped as *JavaObjects*) in the second argument. If we find such an (accesible and unambigously most specific) method for the given object, then that method is invoked and the return value is put in the last argument. If the return value is a number or a string constant it is returned as a Prolog number or constant else it is returned wrapped as a *JavaObject*.

If we wish to invoke static methods the first argument needs to be a class wrapped in JavaObject - otherwise the calling mechanis is the same

The mapping of datatypes between Prolog and Java looks like this:

Java	**Prolog**
int maybe (short,long)	Integer
double maybe (float)	Real
java.lang.String	Const
any other Object	JavaObject is a bound variable, which unifies only with itself

Table 1. Data Conversion

4 The details of implementation

4.1 Creating a class

The reflection package uses the Java reflection API to load Java classes at runtime, instantiate their objects and invoke methods of both classes and objects. The Java Reflection *Class.forName("classname")* method is used to create a class at runtime. In case an exception occurs, an error message stating the exception is printed out and a **0** is returned, which is interpreted as a **failure** by Prolog. The error message printing can be switched on/off by using a flag.

This is interfaced with Prolog using the conventional Builtin extension mechanism getting the first argument passed as a Prolog constant seen by Java as a String. After this, the Java side processing is done and the handle to the required class is obtained. Finally this handle wrapped as a *JavaObject* is returned in the second argument.

Example:
new_java_class('java.lang.String',S)
Output:
S=JavaObject(java.lang.Class_623467)

4.2 Instantiating an Object

First of all, the arguments of a constructor are converted into a list, then parsed in Prolog and provided to Java as *JavaObjects*. Then each one is extracted individually. If the parameter list is empty then a special token is passed instead of the JavaObject, which tells the program, that a void constructor is to be used to instantiate a new object from the class. This is done by invoking the given class' *newInstance()* method, which returns the required object.

If the argument list is not empty, the class (dynamic type) of the objects on the argument list is determined using the *getClass()* method and stored in an array. This array is used to search the required constructor for the given class using the *getConstructor(parameterTypes)* method. Once the constructor is obtained, its *newInstance(parameterList)* method is invoked to obtain the required object. The exception mechanism is exactly the same as for creating a new class as explained above.

This also uses the conventional Builtin extension mechanism to interface with Java, therefore Objects are wrapped as *JavaObjects*. Prolog **Integers** are mapped to Java **int** and Prolog's **Real** type becomes Java **double**. The reverse mapping from Java is slightly different as **long**, **int**, **short** are mapped to Prolog's **Int**, which holds its data in a **long** field and the **float** and **double** types are mapped to Prolog's Real (which holds is data in adouble field). Java Strings are mapped to Prolog constants and vice versa (this is symmetric).

Example:
new_java_obj(S,new(hello),Mystring)

Output:
MyString=JavaObject(java.lang.String_924598)

4.3 Invoking a method

The method invoking mechanism is very similar to the object instantiation mechanism. The mapping of datatypes remains the same. The exception mechanism is also exactly same as that of constructing objects and classes.

First we determine the class of the given object. The getConstructor method is replaced by *getMethod(methodName, parameterTypes)* except that it takes in as the first argument a method name. Once the method is determined, its return type is determined using the *getReturnType().getName()* for the mapping of Prolog and Java datatypes following the convention described earlier. If the return type is void the value returned to Prolog will be the constant 'void'. To invoke the required method (*the method we wish to invoke*) we call the obtained method's *invoke.(Object, parameterList)* method and will return after conversion the return value for the given method.

To invoke static methods, first we determine whether the object passed as the first argument is an instance of the class **Class**. If so, this is taken to be the class whose method is to be searched, and the call to invoke looks like *invoke.(null, parameterList)*

Example
invoke_java_method(Mystring,length,R)
Output:
R=5
Example
invoke_java_method(Mystring,toString,NewR)
Output:
NewR=hello

5 Limitations of Reflection

An important limitation of the reflection mechanism is that when we are searching for a method or a constructor for a given class using the given parameter types. The reflection package looks for exact matches. That means if we have an object of class Sub and we pass it to a method, which accepts as argument an object of class Super, which is Sub's super-class, we are able to invoke such a method in normal Java, but in case of reflection our search for such a method would fail and we would be unable to invoke this method. The same situation occurs in the case for an accepting an **interface** method, which actually means accepting all objects implementing the **interface**. The problem arises in the first place because method could be overloaded and Java decides, which method to call amongst overloaded methods at compile-time and not at runtime. We discuss in the next section how the Java compiler decides, which method to call at compile time.

6 Java Compile Time Solution

The steps involved in the determination of which method to call once we supply the object whose method is to be invoked and the argument types.

6.1 Finding the Applicable Methods

The methods that are applicable have the following two properties:

- The name of the method is same as the call and the number of parameters is same as the arguments in the method call.
- The type of each argument can be converted to the type of corresponding parameter by method invocation conversion.

This broadly means that either the parameter's class is the same as the corresponding argument's Class, or that it is on the inheritance chain built from the argument's class upto **Object**. If parameter is an **interface**, the argument implements that interface. We refer to [2] for a detailed description of this mechanism.

6.2 Finding the Most Specific Method

Informally, **method1** is more specific than **method2** if any invocation handled by **method1** can also be handled by **method2**.

More precisely, if the parameter types of method1 are M_{11} to M_{1n} and parameter types of method2 are M_{21} to M_{2n} method1 is more specific then method2 if M_{1j} can be converted to M_{2j} for all**j** from **1** to **n** by method invocation conversion.

6.3 Overloading Ambiguity

In case no method is found to be most specific then method invocation is ambiguous and a compile time error occurs.

Example:
Consider class A superclass of B and two methods with name m.

m(A,B)
m(B,A)

Now an invocation which can cause the ambiguity is.

m(instance of B, instance of B)

In this case both method are applicable but neither is the most specific as *m(instance of A,instance of B)* can be handled only by first one while *m(instance of B,instance of A)* can be handled only by second one i.e. either of the method's all parameters can not be converted to other's by method invocation conversion.

6.4 Example: Method Resolution at Compile time

Method resolution takes place at compile time in Java and is dependent on the code which, calls the method. This becomes clear from the following example.

Consider two classes Super and Sub where Super is superclass of Sub. Also consider class A with a method m and class Test with a method test, the code for the classes looks like this:

Super.java

```
public class Super {}
```

Sub.java

```
public class Sub extends Super {}
```

A.java

```
public class A {
  public void m(Super s) { System.out.println("super");}
}
```

Test.java

```
public class Test {
  public static void test(){
    A a=new A();
    a.m(new Sub());
  }
}
```

On invocation of method *test()* of class **Test**, method *m(Super)* of class **A** is invoked and **super** is printed out. Let's assume that we change the definition of the class **A** and overload the method *m(Super)* with method *m(Sub)* such that **A** looks like this:

A.java

```
public class A {
  public void m(Super s) {System.out.println("super");}
  public void m(Sub s) {System.out.println("sub");}
}
```

If we recompile, and run our test method again, we expect **sub** to be printed out since *m(Sub)* is more specific than *m(Super)* but actually **super** is printed out. The fact is method resolution is done when we are compiling the file containing the method call and when we compiled the class Test we had the older version of class A and Java had done resolution based on that class A. We can get the expected output by recompiling class Test, which now views the newer version of class A and does the resolution according to that, and hence we get the expected output **sub**.

7 Finding a Most Specific Method at Runtime

We will follow a simple algorithm. Let's assume that the number of methods which are accessible and have same name and number of parameters as the call is **M** (small constant) and the number of arguments in the call is **A** (a small constant). Let us assume that the maximum inheritance depth of the class of an argument from Object down to itself in the class hierarchy tree is **D** (a small constant) It can be trivially shown that the complexity of our algorithm is bounded by **O(M * A * D)**. Our algorithm mimics exactly the functionality of Java and the following example would run exactly the same on both Java and our interface, the only difference being that since Java does the resolution at compile time, in case of an ambiguous call Java would report a compile time error while we do the same thing at runtime and hence, throw an exception with appropriate error message. So if class A looks like this:

A.java

```
public class A {
  public void m(Super s1,Sub s2) {System.out.println("super");}
  public void m(Sub s1, Super s2) {System.out.println("sub");}
}
```

and the class Test looks like this: **Test.java**

```
public class Test {
  public static void test(){
    A a=new A();
    a.m(new Sub(),new Sub());
  }
}
```

then Java will not compile class **Test** and give an error message. In our case there is no such thing as the class **Test**, but the equivalent of the above code would look like follows:

```
new_java_class('A',Aclass),
new_java_obj(Aclass,Aobject),
new_java_class('Sub',Subclass),
new_java_obj(Subclass,Subobject1),
new_java_obj(Subclass,Subobject2),
invoke_java_method(Aobject,m(Subobject1,Subobject2),Return).
```

The result will be an ambiguous exception message and the goal failing with **no**.

Our Algorithm We will now describe our algorithm in detail:

```
CorrectMethodCaller(methodName, Arguments[ ] /*Size A*/)
1.  Method = get_exact_match_reflection(methodName,Arguments[ ])
```

```
2.  If Method != null
3.    return invoke(Method,Arguments[ ])
4.  MethodArray[ ] = get_methods(methodName,A) /*Size M*/
5.  MethodParameterDistanceArray[ ][ ] = {infinity} /*Size M*A*/
6.  For m = 0 to M
7.    MethodParameterArray[m][ ] =
        MethodArray[m].get_method_parameters() /*Size M*A*/
/*Finds distances of method parameters from the arguments
  and stores in the array*/
8.  For a = 0 to A do
9.    DistnceCounter = 0
10.   While Arguments[a].type != null do /*Loops over D*/
11.     For methods m = 0 to M do
12.       If MethodParameterArray[m][a] == the Arguments[a].type
13.         MethodParameterDistanceArray[m][a] = DistanceCounter
14.       Arguments[a].type = Super(Arguments[a].type)
15.       DistanceCounter = DistanceCounter + 1.
/*Find the method with minimum distance of parameters from arguments*/
16. Minimum = infinity
17. For m = 0 to M do
18.   Sum = 0
19.   For a = 0 to A do
20.     Sum = Sum + MethodParameterDistanceArray[m][a]
21.   If Sum < Minimum
22.     mChosen = m
/*Check if our selection is correct*/
23. For m = 0 to M do
24.   If m == mChosen
25.     continue
/*Skip those methods in which atleast one parameter never occurs
  in the inheritance hierarchy from the argument to Object*/
26.   For a = 0 to A do
27.     If MethodParameterDistanceArray[m][a] == infinity break
28.   If a < A cotinue
/*Check if "most specific method condition" is violated by mChosen*/
29.   For a = 0 to A do
30.     If MethodParameterDistanceArray[m][a] <
          MethodParameterDistanceArray[mChosen][a]
31.       Throw ambiguous exception
32. return invoke(MethodArray[mChosen],Arguments[ ])
```

8 An example application/GUI using reflection API

This GUI has almost completely been implemented in Prolog using the reflection API. A special builtin which, allows us to redirect output to a string is used to interface default Prolog i/o to textfield/textarea etc. The total Java code is less than 10 lines. Jinni provides, on the Java side, a simple mechanism to call Prolog *Init.jinni("Prolog command")*. Since we do not have references to

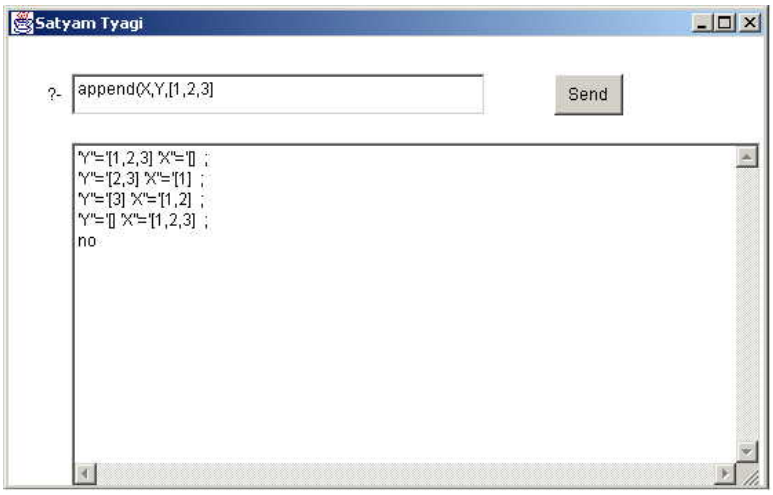

Fig. 2. Screenshot of Prolog IDE written in Prolog

different objects in the Java code, but everything is in the Prolog code, we need a mechanism to communicate between Java's action-listener and Prolog. Jinni's Linda blackboards have been used for this purpose [5, 7]. Whenever an action takes place the Java side calls Jinni and does a out with a number for type of action on the blackboard by calling something like *Init.jinni("out(a(1))")*. On the Prolog side we have a thread waiting on the blackboard for input by doing an *in(a(X))*. After the out variable **X** gets unified with **1** and depending on this value, Prolog takes the appropriate action and again waits for a new input. Hence, we can make action events such as button clicks communicate with Prolog. The code for button "send" in the Appendix B shows exactly how this is done.

9 Related Work

Here we discuss a few other approaches followed for interfacing Prolog to Java using reflection or the Java Native Interface (JNI), and also a Scheme interface to Java using reflection. First, Kprolog's JIPL package provides an interesting API from Java to a C-based Prolog and has a more extensive API for getting and setting fields. It also maps C-arrays to lists. The Kprolog's JIPL has dynamic type inference for objects, but the problem of method signature determination and overloading has not been considered in the package [3].

SICStus Prolog actually provides two interfaces for calling Java from Prolog. One is the JASPER interface which uses JNI to call Java from a C-based Prolog. To obtain a method handle from the Java Native Interface requires to specify the signature of the method explicitly. So JASPER requires the user to specify as a string constant the signature of the method that the user wishes to call. This transfers the burden of finding the correct method to the user [4], who

therefore needs to know how to specify (sometimes intricate) method signatures as Strings.

SICStus Prolog also has another interesting interface for calling Java from Prolog as a Foreign Resource. When using this interface the user is required to first declare the method which he wants to call and only then can the user invoke it. Declaring a method requires the user to explicitly state the ClassName, MethodName, Flags, and its Return Type and Argument Types and map it to a Prolog predicate. Now the Prolog predicate can be used directly. This feature makes the Java method call look exactly like a Prolog builtin predicate at runtime - which keeps the underlying Java interface transparent to, for instance, a user of a library. (*This is very much similar to our old Builtin Registration and Execution mechanism, with one difference: here registration or declaration is on the Prolog side, while we were doing the same on Java side - for catching all errors at compile time.*) The interface still requires the programmer to explicitly specify types and other details as the exact method signature [4].

Kawa Scheme also uses Java reflection to call Java from Scheme. To invoke a method in Kawa Scheme one needs to specify the class, method, return type and argument types. This gives a handle to call the method. Now the user can supply arguments and can call this method. Again, the burden of selecting the method is left to the user as he specifies the method signature [1].

In our case, like JIPL and unlike other interfaces, we infer Java types from Prolog's dynamic types. But unlike JIPL, and like with approaches explicitely specifying signatures, we are able to call methods where the argument type is not exactly same as the parameter type. Hence, our approach mimics Java exactly. The functionality is complete and the burden of specifying argument types is taken away from the user.

10 Future Work

Future work includes extending our API, as currently we do not support getting and setting fields and arrays. Another interesting direction which is a consequence of the development of a reflection based API, is the ability to quickly integrate Java applications. We have shown the power of the API with the simple GUI application. Such applications can be built either completely in Java with an API based on methods to be called from Prolog, or almost completely in Prolog using only the standard packages of Java.

Jinni 2000 has support for *plugins* such as different Network Layers (TCP-IP and multicast sockets, RMI, CORBA) and a number applications such as Teleteaching, Java3D animation tolkit developped with its conventional builtin interface. New applications and plugins can now be added by writing everything in Prolog while using various Java libraries. Arguably, the main advantage of such an interface is that it requires a minimal learning effort from the programmer.

11 Conclusion

We have described a new reflection based Prolog to Java interface which takes advantage of implicit dynamic type information on both the Prolog and the Java sides. Our interface has allowed to automate data conversion between overloaded method parameters, through a new algorithm which finds the most specific method corresponding to a Prolog call. The resulting run-time reflective method dispatching mechanism provides accurate handling of overloaded methods beyond the reflection package's limitations, and is powerful enough to support building a complete GUI library mostly in Prolog, with only a few lines of application specific Java code.

The ideas behind our interfacing technique are not specific to Jinni 2000 - they can be reused in improving C-based Prolog-to-Java interfaces like JIPL or Jasper or even Kawa's Scheme interface. Actually our work is reusable for any languages with dynamic types, interfacing to Java, as our work can be seen as just making Java's own Reflection package more powerful.

References

1. P. Bothner. Kawa, the Java based Scheme system. Technical report, 1999. Available from http://www.delorie.com/gnu/docs/kawa/kawa_toc.html.
2. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification, Second Edition*. Java Series. Addison-Wesley, 1999. ISBN:0-201-31008-2.
3. N. Kino. Plc.java: JIPL class source. Technical report, KLS Research Labs, 1997. Available from http://www.kprolog.com/jipl/index_e.html.
4. SICStus Support. Manual document of SICStus Prolog. Tecnical report, Swedish Institute of Computer Science, May 2000. Available from http::://www.sics.se/isl/sicstus/docs/latest/html/docs.html.
5. P. Tarau. A Logic Programming Based Software Architecture for Reactive Intelligent Mobile Agents. In P. Van Roy and P. Tarau, editors, *Proceedings of DIPLCL'99*, Las Cruces, NM, Nov. 1999. http://www.binnetcorp.com/wshops/ICLP99DistInetWshop.html.
6. P. Tarau. Inference and Computation Mobility with Jinni. In K. Apt, V. Marek, and M. Truszczynski, editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48. Springer, 1999. ISBN 3-540-65463-1.
7. P. Tarau. Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 109–123, London, U.K., 1999.
8. P. Tarau. Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In J. Lloyd, editor, *Proceedings of CL'2000*, London, July 2000. to appear at Springer-Verlag.
9. P. Tarau, K. De Bosschere, and B. Demoen. Partial Translation: Towards a Portable and Efficient Prolog Implementation Technology. *Journal of Logic Programming*, 29(1–3):65–83, Nov. 1996.

Appendix A: Example of Prolog code for the Reflection Based Jinni GUI

```
/* builds a simple IDE with GUI components*/
jinni_ide:-
  new_java_class('JinniTyagiFrame',JTF),
  new_java_obj(JTF,new('Satyam Tyagi'),NF),
  invoke_java_method(NF,show,_Ans),
  new_java_class('java.awt.Label',L),
  new_java_obj(L,new('?-'),NL),
  invoke_java_method(NL,setBounds(30,50,20,30),_A4),
  invoke_java_method(NF,add(NL),_C4),
  new_java_class('java.awt.TextField',TF),
  new_java_obj(TF,new('type Prolog query here'),NTF),
  invoke_java_method(NTF,setBounds(50,50,300,30),_A1),
  invoke_java_method(NF,add(NTF),_C1),
  new_java_class('java.awt.Button',B),
  new_java_obj(B,new('Send'),NB),
  invoke_java_method(NB,setBounds(400,50,50,30),_A3),
  invoke_java_method(NF,add(NB),_C3),
  invoke_java_method(NB,addActionListener(NF),_B4),
  new_java_class('java.awt.TextArea',TA),
  new_java_obj(TA,new('results displayed here'),NTA),
  invoke_java_method(NTA,setBounds(50,100,500,250),_A2),
  invoke_java_method(NF,add(NTA),_C2),
  bg(the_loop(NTA,NTF)). % code not shown for the_loop/2
```

Appendix B: Java code for the Send Button in the Reflection Based Jinni GUI

```
import java.awt.*;
import java.awt.event.*;
import tarau.jinni.*;

public class  JinniTyagiFrame extends Frame implements ActionListener{
        public JinniTyagiFrame(String name) {
                super(name);
                setLayout(null);
                resize(600,400);
        }

        public void actionPerformed(ActionEvent ae){
                String click=ae.getActionCommand();
                if(click.equals("Send")){
                        Init.askJinni("out(a(1))");
                }
        }

}
```