LogicObjects : A Linguistic Symbiosis Approach to Bring the Declarative Power of Prolog to Java

Sergio Castro
RELEASeD lab
ICTEAM Institute
Université catholique de Louvain, Belgium
sergio.castro@uclouvain.be

Kim Mens
RELEASeD lab
ICTEAM Institute
Université catholique de Louvain, Belgium
kim.mens@uclouvain.be

Paulo Moura
Dept. of Computer Science
University of Beira Interior
& CRACS, INESC–TEC
Portugal
pmoura@di.ubi.pt

ABSTRACT

Logic programming is well suited for declaratively solving computational problems that require knowledge representation and reasoning. Object-oriented languages, on the other hand, are well suited for modeling real-world concepts and profit from rich ecosystems developed around them, which are often missing from logic languages. For applications that require both the declarative power of logic programming and the rich modeling expressiveness and development environments offered by object-oriented languages, there is a need for reconciling both worlds. *LogicObjects* is our linguistic symbiosis framework for integrating Prolog within the Java language. It extends Java with annotations that allow Java programs to interact transparently and automatically with Prolog programs.

Keywords

Linguistic Symbiosis, Object-Oriented Programming, Logic Programming, Multi-paradigm programming

1. INTRODUCTION

Object-oriented programming languages have demonstrated their usefulness for modeling real-world concepts. In addition, the availability of continuously growing software ecosystems around them, including advanced IDEs and extensive libraries, has contributed to their success. Declarative languages like Prolog, however, are more convenient for expressing problems of declarative nature, such as rule-based systems [7, 14].

Linguistic symbiosis [11] has been used in the past to solve the problem of integrating programs written in different languages. In particular, the *SOUL* [9] language implements advanced language symbiosis features between Smalltalk and Prolog [7, 10]. Some limits and issues with *SOUL* have been discussed in [9]. This work is a first step towards a framework that overcomes most of these limitations.

RAM-SE'12, June 13, 2012, Beijing, China

Although there exists some other work that focusses on facilitating the interaction from an object-oriented language to a logic language, to the best of our knowledge this is the first approach that provides a truly transparent, automatic and customizable integration from the perspective of the object-oriented language.

This paper is structured as follows: Section 2 presents a problem of declarative nature and its implementation in a logic language. This will be referenced in the next sections. Section 3 presents our framework and shows how it enables a transparent access from Java to our implementation in logic. Section 4 discuss relevant related work and section 5 presents our conclusions and future work.

2. CASE STUDY: *THE LONDON UNDERGROUND*

This case addresses a typical problem that can be implemented easily with a logic programming language: a querying system about subway lines and stations in a big city (e.g., to query the number of intermediate stations from one station to another). Since most public transport systems require a user-friendly interface, which can be developed more easily with an object-oriented language, a good solution would be to implement the declarative part of the application in Prolog and the user interface in Java.

The first stage of the problem consists in expressing our knowledge about the London Underground in terms of logic statements. Most of the code in this section has been taken "as is" from [8]. However, we did introduce an interesting variation in the original code. Instead of implementing it in plain Prolog, we used Logtalk [12], a portable object-oriented layer on top of Prolog, thus allowing us to benefit from the symbiosis features it offers while at the same time facilitating the mapping that needs to be made between objects belonging to the two worlds.

In our universe of discourse, stations are *connected* to other stations through underground lines. A station is *nearby* another one if there is at most one station in between them. Finally, a station A is *reachable* from another station B if there exists a list of stations L that form a path going from B to A.

Listing 1 shows the Logtalk definition of the *metro* object (we omitted some predicate clauses to save space). Note that, although Logtalk also supports classes, we are not using the word *class* here, since we are just defining a *prototype* object and to avoid confusion with Java classes. The *metro* object encapsulates the knowledge about how stations are connected (lines 5–8), plus the rules for the logic predicates

nearby/2 (lines 10–11) and *reachable/3* (lines 13–14). The messages (queries) that the *metro* object can answer are specified using the *public/1* directive (line 2).

```
1  :- object(metro).
2
3    :- public([connected/3, nearby/2, reachable/3]).
4
5    connected(station(bond_street),
         station(oxford_circus),line(central)).
6    connected(station(oxford_circus),
         station(tottenham_court_road),line(central)).
7    connected(station(bond_street),
         station(green_park),line(jubilee)).
8    ...
9
10   nearby(X,Y):-connected(X,Y,L).
11   nearby(X,Y):-connected(X,Z,L),connected(Z,Y,L).
12
13   reachable(X,Y,[]):-connected(X,Y,L).
14   reachable(X,Y,[Z|R]):-connected(X,Z,L),reachable(Z,Y,R).
15
16 :- end_object.
```

Listing 1: The *metro* object in Logtalk

Messages are sent in Logtalk using the *::/2* operator. For example, to find which stations are connected to the station *Bond Street* we can use the query shown in listing 2.

```
metro::connected(station(bond_street), Station, Line).
```

Listing 2: Invoking a Logtalk method

Logtalk and Prolog both support non-deterministic queries, which allows retrieving all existing solutions for a query using backtracking, and meta-programming, which allows e.g. to construct a list with all solutions to a query. For example, to get a list of all stations connected to *Bond Street* we could write the query shown in listing 3.

```
findall(
  Station,
  metro::connected(station(bond_street), Station, Line),
  Stations
).
```

Listing 3: Invoking a Logtalk method using the findall/3 meta-predicate

Listing 4 shows the definition of a parametric object [13] *line/1*.

```
1  :- object(line(_Name)).
2
3    :- public([name/1, connects/2]).
4
5    name(Name) :- parameter(1, Name).
6
7    connects(Station1, Station2) :- self(Self),
         metro::connected(Station1, Station2, Self).
8
9  :- end_object.
```

Listing 4: The *line* object in Logtalk

The object's sole parameter (line 1) can be retrieved with the method *name/1* (line 5). This object also defines a *connects/2* method (line 7) that answers stations directly connected by the *line* represented by the receiver object. This method implementation is delegated to the *metro* object.

Our last object is the *station* object (Listing 5). As for the *line* object, it is also a parametric object having as sole parameter the name of a station (line 1). It defines a method *connected/1* (line 7) that answers if the station is connected with another station (or answers the stations that are connected if the parameter is an unbound variable). *connected/2* (line 9) takes as a second parameter the underground line that connects the stations. The method *nearby/1* (line 11) answers if the station is nearby another station received as a parameter. The method *reachable/1* (line 13) answers if the station received as parameter is reachable from the receiver station object (or answers the reachable stations if the parameter is an unbound variable). The method *reachable/2* (line 15) does the same, but includes in its second argument a list with all the intermediate stations. All these methods are delegated to the *metro* object.

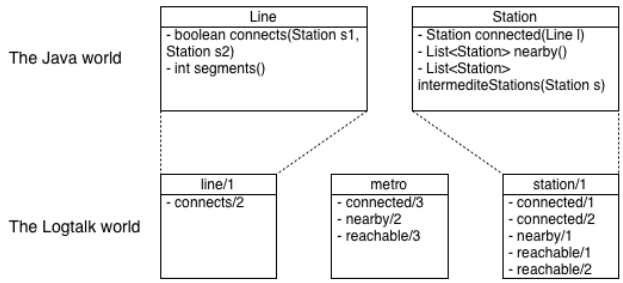

Figure 1: Objects from two different worlds living in symbiosis.

```
1  :- object(station(_Name)).
2
3    :- public([name/1, connected/1, connected/2, nearby/1,
         reachable/1, reachable/2]).
4
5    name(Name) :- parameter(1, Name).
6
7    connected(Station) :- connected(Station, _).
8
9    connected(Station, L) :- self(Self),
         metro::connected(Self, Station, L).
10
11   nearby(Station) :- self(Self), metro::nearby(Self,
         Station).
12
13   reachable(Station) :- reachable(Station, _).
14
15   reachable(Station, IntermediateStations) :- self(Self),
         metro::reachable(Self, Station,
         IntermediateStations).
16
17 :- end_object.
```

Listing 5: The *station* object in Logtalk

Finally, we show the *loader* file of our library. A loader file defines the collection of objects that should be loaded (line 2) when requiring a Logtalk library. We further discuss these objects in the next section.

```
1  :- initialization(
2    logtalk_load([metro, station, line])
3  ).
```

Listing 6: The *load_all.lgt* loader file

3. LOGICOBJECTS

In the previous section we have shown the implementation of the declarative part of our case study using Logtalk objects. In this section, we add Java counterparts to these objects and show how objects from the Java world can symbiotically interact with objects in Logtalk. All these objects are illustrated in figure 1.

In the remainder of this section we describe the linguistic symbiosis techniques employed by our *LogicObjects* framework. Our current implementation focusses on a symbiosis from the Java point of view. We decided to start the design and implementation of our symbiosis from the object-

oriented language since this direction has been reported [6, 7, 9] as the most difficult to achieve transparently.

3.1 Linguistic symbiosis

Linguistic symbiosis [11] is the ability of a program to transparently invoke routines defined in another language as if they were defined in its own language [9]. In addition, Wuyts and Ducasse [16] add that to achieve real symbiosis, objects from each language must be understood in the other. In our particular context, these generic symbiosis requirements could be rephrased as being able to:

- Map Java methods to logic queries.
- Translate Java objects to logic terms, and back.

Several additional problems specific to symbiosis between an object-oriented language and a logic language are presented in [7, 9]. We repeat some of them below:

Unbound variables: Most object-oriented languages cannot work with unbound variables. However, it is common in logic to call a predicate with unbound variables.

Return values: In object-oriented languages, methods often return objects as a result of their execution. In logic, there are no such return values: results are returned by binding values to unbound variables. More than one value can be returned in this way.

Managing multiplicity: In object-oriented languages there is a difference (e.g., return type) between methods that return a single value or a collection of values. Logic languages make no distinction between predicates that produce a single solution or many solutions.

Let us now discuss how *LogicObjects* deals with these issues.

3.2 Translating objects to logic terms

In the context of symbiosis between Java and Prolog, Java objects should have a representation as logic terms and logic terms should be manipulatable as Java objects [3]. Hence, we need a mechanism for mapping logic terms to objects and vice-versa.

Brichau et al. [3] defined such a mapping for the specific problem of transforming Java objects representing parse tree nodes to logic terms and vice-versa. In their work, there is an implicit direct mapping between a logic predicate name and a class name. The arguments of logic predicates are mapped to the children of the parse tree nodes by means of the same recursive algorithm.

We generalize their mapping solution by providing, using Java annotations, a general mapping between logic predicate names and Java classes and between predicate arguments and Java object properties.

To illustrate our technique, let's consider the implementation of the *Line* class in Java, shown in listing 7. This class is the Java counterpart of the *line* Logtalk object defined in listing 4. The annotation *LObject* on line 1 identifies this class as (partially) implemented in logic. We refer to these classes as *symbiotic classes*. The name of the Logtalk object on the logic side providing this implementation is derived from the class name *Line*. The default mapping is basically a transformation from Java camel-case naming convention to Prolog names with lowercase tokens separated by uncerscores. E.g., the Java class *FooBar* would be translated to the Logtalk object *foo_bar*. This default mapping can be overridden easily, however, since the *LObject* annotation can include a *name* attribute indicating explicitly the name of the object on the logic side implementing the logic methods of this Java class. When provided, this name will be the one used, instead of deriving it from the class name. When the object on the logic side is a parametric object, its parameters need to be declared on the Java side by means of a *params* attribute in the *LObject* annotation. In the *Line* class example, the single parameter of the parametric object on the logic side is mapped to the instance variable *name*. An instance of Java class *Line* with its name set to *"central"*, is thus automatically translated to the logic term *line(central)*.

In this simple example the transformation of the object property *name* to a term is straightforward, since it is just a string of characters and no additional mapping information is required. If the property would have been a symbiotic object (in case where somewhere in the class hierarchy, the field's declaration includes an *LObject* annotation) the transformation process will continue recursively, given that the property object could also have properties that are symbiotic objects and so on.

```
1  @LObject(params = {"name"})
2  public abstract class Line {
3    private String name;
4
5    public String getName() { return name; }
6    public void setName(String name) { this.name = name; }
7
8    @LMethod
9    public abstract boolean connects(Station s1, Station
         s2);
10
11   @LMethod(name = "connects", params = {"_", "_"})
12   public abstract int segments();
13 }
```

Listing 7: The *Line* object in Java

Translating a term to an object is the inverse process but in this case we need a *translation context*, which encapsulates the translation objective and environment. With this context, we can answers questions such as: Is the translated object going to be assigned to a field? Is it the result of a symbiotic method (a Java method implemented in Prolog)? Are there relevant annotations in the context (e.g., a field or method) that should influence the translation? And so on.

The procedure of transforming a logic term into a Java object starts by attempting to find a symbiotic class in the system which name and number of parameters correspond with the logic predicate's name and its number of arguments. Once we found such a symbiotic class, the translation context plays an active role in how the Java object will be instantiated. If, according to the context, the expected type is an ancestor class of the found symbiotic class (or they are the same), then we just need to ask the framework for an instance of the symbiotic class, as is shown in section 3.7. However, if the found symbiotic class is an ancestor of the expected type, according to the translation context, then it is an instance of the expected type that should be created, and its properties should be set according to the arguments to the logic predicate. For translating each of these arguments, a new translation context will be created according to the field declaration of the matching property, and the conversion algorithm is applied recursively.

In case where no symbiotic class can be found, the translation needs to be guided entirely by the translation context. For example, if the expected type is a list, a Java list will be instantiated. This list will be filled with the translation

to Java objects of each member of the corresponding Prolog list. If additional information on the expected type of these list members is provided (e.g., when the Java list is a parameterized type), then a new translation context is created and the list elements are translated according to this context. For an example of such translation see section 3.6.

3.3 Mapping methods to logic queries

As in [9], by default methods are mapped to logic predicates with the same name as the method name, and the number of method arguments as the predicate cardinality. An example of this default mapping can be found in listing 7. The method *connects* on line 9 has two arguments. It is recognized by the framework as a symbiotic method since it includes the *LMethod* annotation (line 8). It is mapped by default to the Logtalk method *connects/2* in listing 4, line 7. Note that this particular Logtalk method will be executed in the context of the Logtalk object corresponding to the Java object receiving the message.

Although this default mapping is often useful, a programmer can always customize it by adding attributes to the *LMethod* annotation. An example of such a customization is the Java method *segments* on lines 11–12. As specified by the *name* and *params* annotation attributes, this method will be mapped to the logic predicate *connects/2*.

With this technique we are thus able to map one single Logtalk predicate to different Java methods according to our needs and give each of these mappings a distinct semantics.

3.4 Dealing with unbound variables

In Prolog, it is common to write queries with unbound variables. In Java, however, all variables must be bound to a value (either an explicit assigned value, or a default initialization value). As mentioned before, the *segments* method of class *Line* is mapped, through the *LMethod* annotation, to the predicate *connects* with arity two. Parameters specified in the *params* attribute are interpreted as Prolog terms. In this case, both parameters are the symbol "_", which is interpreted as an anonymous logic variable by Prolog.

Before evaluating a symbol as a term certain macro substitutions will happen, however. For example, the symbol *$n* (where $n \in \mathbb{N}_0$) is interpreted as the object received as n^{th} parameter by the method. The class *Station* (listing 8) provides examples of methods including these symbols in their *params* attribute (lines 9 and 17). The current available macros are detailed in section 3.9.

```
1  @LObject(params = {"name"})
2  public abstract class Station {
3    private String name;
4
5    public String getName() { return name; }
6    public void setName(String name) { this.name = name; }
7
8    @LSolution("S")
9    @LMethod(params = {"S", "$1"})
10   public abstract Station connected(Line line);
11
12   @LWrapper @LSolution("S")
13   @LMethod(params = {"S"})
14   public abstract List<Station> nearby();
15
16   @LSolution("IntermediateStations")
17   @LMethod(name = "reachable", params = {"$1",
         "IntermediateStations"})
18   public abstract List<Station>
         intermediateStations(Station station);
19 }
```

Listing 8: The *Station* object in Java

Parameters can even contain Java expressions that, upon evaluation, will be converted to a logic term. An example of this will be given in section 3.9.

3.5 Returning results

The value returned by symbiotic methods depends on the context of the method declaration. For example, when nothing else is specified and the method returns a boolean value (as for the *connects* method in class *Line*), the framework is smart enough to query the corresponding predicate in Prolog and answer whether the query has at least one solution.

If the method instead returns a numeric value (as for the *segments* method in class *Line*) and nothing else is specified, the return value will be the number of results of the query. In our example, the number of segments an underground line has corresponds to the number of stations it connects. This is the total number of answers of the logic method *connects/2* in the *line* Logtalk object.

For customizing method results, the framework offers an annotation *LSolution*. Its *value* attribute specifies a logic term that will be returned by the method (after conversion to a Java object), according to the variable bindings of a solution. To illustrate this, consider the implementation of the *Station* class (listing 8). The method *connected* (lines 8–10) answers another station that is connected to the receiver on the line sent as a parameter. Since the *name* attribute is not specified in the annotation, the Java method name (after being translated to Prolog conventions) is used as the predicate to query. The first parameter is a Prolog logic variable *S*; the second parameter is the term representation of the first parameter to the Java method. The result returned by the Java method will be the term specified by the *value* attribute of its *LSolution* annotation. Any unbound variables in that term will be bound according to the bindings of the variables present in the arguments of the Logtalk method. In our example, the term to return is the Prolog logic variable *S* that will be bound, upon execution, to a compound term of the form *station(nameStation)*. This compound term will be transformed automatically to an instance of the *Station* class according to the algorithm described in section 3.2.

Although in our example the term to return was just a logic variable, it can be a logic term of arbitrary complexity.

3.6 Managing multiplicity

Given that a logic query can have multiple solutions, we need a way to differentiate when the programmer wants just one solution or all of them. Initially we tried to infer this from the method return type. For example, if the method returns a collection class, then with certain probability its intention is returning a collection of results instead of just one. This assumption is not always valid however.

Consider the method *intermediateStations* in the *Station* class (listing 8, lines 16–18). This method is mapped to the *reachable/2* predicate in the *station* Logtalk object (listing 5, line 15). The *params* attribute in the *LMethod* annotation indicates that the first parameter of the Logtalk predicate will be the logic term representation of the first parameter of the Java method (indicated by the macro *$1*). The second parameter is the Prolog variable *IntermediateStations*.

The *LSolution* annotation specifies that the return value of the method is the variable *IntermediateStations*. As explained in section 2, this variable is bound to a list with all the intermediate stations between the receiver station object, and the station object sent as first parameter of the method. Then, the Java method is returning a list of ob-

jects that corresponds to the binding of one variable in *one* solution (the first) answered by the Logtalk predicate. This is thus an example where a method returning a collection of objects is not intending to answer a group of solutions, but just one single solution interpreted as a collection.

In order to resolve ambiguities, we offer the *LWrapper* annotation. If a method does not include the *LWrapper* annotation, it will return just one result if the *LSolution* annotation is present. If the *LSolution* is also absent, the result will try to be inferred from the return type of the method, as explained in section 3.5.

The method *nearby* (listing 8, lines 12–14) is an example of the usage of the *LWrapper* annotation (line 12). One answer to this method is a station that is nearby the receiver station object. Given that the *LWrapper* annotation is present, the framework considers the type of the method (a *List* class) as a container of all its solutions. The expected type of each solution is given by the first (an only) type parameter in the parameterized return type. In our example, the return type is *List<Station>*. Thus the type for each solution is *Station*. As explained in section 3.2, this type will guide the process of transforming the logic term in the variable *IntermediateStations* to an object. If the container type is not parameterized, then the type of each solution would be considered *Object* and the framework will try to infer the right transformation based only on the returned logic term.

3.7 Instantiating a symbiotic class

We have seen in the previous examples that symbiotic classes and their symbiotic methods are declared as abstract. In order to obtain an instance of symbiotic classes the framework provides the *LogicObjectFactory* class. By means of its *create* method, new instances of symbiotic classes can be created. This is illustrated in listing 9, line 1.

```
1  Line line =
       LogicObjectFactory.getDefault().create(Line.class);
2  line.setName("central");
3  System.out.println("Number of segments: " +
       line.segments());
```

Listing 9: Instantiating a symbiotic class

The output of the code snippet in listing 9 is: *Number of segments: 2*. This corresponds to the number of segments in the line *central*, as specified on the Prolog side (listing 1).

3.8 Auto-loading Logtalk objects

Usually, in Prolog we need to load a file containing rules and facts before we are able to query and reason about them. Likewise, in Logtalk, objects are usually defined in their own file. Normally, there is a loader file that loads all the application objects, as exemplified in listing 6.

Our framework provides several mechanisms for transparently loading Logtalk objects before they can be used. When instantiating a symbiotic class, if the framework detects that a Logtalk file with an *equivalent* name (taking into account the mapping between naming conventions) exists in the same package as the class, it will transparently load it. This happens only the first time when the class is instantiated. If the Logtalk files are in a different directory, or if they must be loaded in a specific order, the loader can make use of a *logicobjects.properties* configuration file. This file should be located in the same package as the symbiotic classes. In our example, this configuration file contains just the line:

imports=logic_lib.example.metro.load_all.

It is possible to include additional files by separating them with a comma. It is also possible to specify a file as a compound term. For instance, the following line is equivalent to the previous example:

imports=logic_lib(example(metro(load_all))).

If none of these techniques is enough for defining the Logtalk dependencies of a symbiotic class, the programmer could use the *imports* attribute in the *LObject* annotation described in section 3.2. This attribute defines the Logtalk files that should be loaded before the symbiotic class can be instantiated.

3.9 Other features

Macros

Term expressions are typically used when adapting method parameters or when defining a method's return value. Previous examples have shown that these expressions often need to include the representation as terms of certain Java objects. To facilitate this task, following macros can be used:

$$ A comma separated list of the original arguments of the method, converted into terms.

$0 The object receiver ('this') of the method.

$n The n^{th} parameter of the method as term.

Symbiosis terms

Terms between */{* and */}* delimiters are *symbiosis terms* [16] and contain Java expressions that will be transformed into logic terms before evaluation by the logic engine. As in SOUL, such terms can be used as a logic condition or as an argument to a logic condition.

In the current implementation, logic variables cannot be included in the expression. However, it is possible to include method parameters or other values using the same constants introduced by Javassist [4] to ease the (re-)definition of instrumented methods at runtime. These constants are evaluated in the context of the symbiotic method. Note that the macro symbols mentioned in the previous section are a subset of the Javassist constants that can be used in a symbiosis term (we refer to the Javassist documentation for detailed information and examples about the available symbols). As an example, consider the symbiotic term shown in listing 10.

```
my_term(/{myJavaMethod($1)/}, $0)
```

Listing 10: A symbiotic term example

In this term, the first argument of the predicate *my_term* is the converted return value (a term) of sending the *myJavaMethod* message to the object using this symbiotic term. The *myJavaMethod* method takes as parameter the first parameter of the original Java method where this symbiosis term is used. The second parameter is the object receiver of the method (also a term).

Integration with Prolog

Our framework also offers an *LQuery* annotation for symbiotic interactions with logic libraries. This annotation uses a

value attribute that can include any arbitrary Prolog query. E.g. an *LQuery(predicate1($0), predicate2($1))* method annotation will be interpreted as querying the logical conjunction of the predicate *predicate1* having as argument the receiver of the method, and the predicate *predicate2* having as argument the first parameter of the Java method. All the other annotations (e.g., *LSolution*, *LWrapper*) are, of course, still available and with the same semantics.

4. RELATED WORK

In addition to the SOUL language, there are a number of works attempting to provide a symbiotic integration between object-oriented languages and a logic language. Most of them are focused on interactions, from the logic language, with objects from an object-oriented language. Examples include Java, Scala, and Python implementations of Prolog (e.g. [2, 1, 15]). These implementations usually provide a set of built-in Prolog predicates that allow easy access to the implementation language libraries.

A technique for instrumenting annotated abstract methods as logic queries is proposed in [5]. Types participating in a method declaration explicitly represent distinct types of Prolog terms, so there is not a real transparent interaction between objects in the two worlds.

For a more extensive survey of other systems integrating logic reasoning and object-oriented programming we refer to [6].

5. CONCLUSIONS AND FUTURE WORK

The starting point of our work were the problems and issues [9] found while designing and using the symbiotic language SOUL. As these problems were mainly localised in the object-oriented side, we focussed our current effort on this aspect of the symbiosis. Our technique improves on each of the reported problems, proposing an elegant solution based on annotations, and including many possibilities for customization.

Our future work will focus on implementing a full two-way symbiosis. We plan to use the reflective mechanisms of Logtalk for transparently and automatically referring to Java objects and invoking their methods in a similar way as has already been accomplished from the Java side. In addition, we will explore techniques for establishing a causal connection between objects belonging to our two different worlds.

Acknowledgements. This work is partially supported by the LEAP project (PTDC/EIA-CCO/112158/2009), the ERDF/COMPETE Program and by FCT project FCOMP-01-0124-FEDER-022701.

6. REFERENCES

[1] C. F. Bolz. Pyrolog: A Prolog interpreter written in Python using the PyPy translator toolchain. `https://bitbucket.org/cfbolz/pyrolog/`.

[2] D. Boulanger and U. Geske. Using Logic Programming in Java Environment (Extended Abstract). Technical Report 10, Knowledge-Based Systems Group, Vienna University of Technology, Austria, 1998.

[3] J. Brichau, C. De Roover, and K. Mens. Open unification for program query languages. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, 2007.

[4] S. Chiba. Load-time structural reflection in java. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, pages 313–336, London, UK, UK, 2000. Springer-Verlag.

[5] M. Cimadamore and M. Viroli. Integrating java and prolog through generic methods and type inference. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 198–205, New York, NY, USA, 2008. ACM.

[6] D'Hondt, Maja. A survey of systems that integrate logic reasoning and object-oriented programming. Technical report, Vrije Universiteit Brussel, 2003.

[7] D'Hondt, Maja and Gybels, Kris and Jonckers, Viviane. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, pages 1328–1335, New York, NY, USA, 2004. ACM.

[8] P. Flach. *Simply logical: intelligent reasoning by example*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[9] K. Gybels. SOUL and Smalltalk - Just Married: Evolution of the Interaction Between a Logic and an Object-Oriented Language Towards Symbiosis. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, 2003.

[10] K. Gybels, R. Wuyts, S. Ducasse, and M. D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Comput. Lang. Syst. Struct.*, 32(2-3):109–124, July 2006.

[11] Y. Ichisugi, S. Matsuoka, and A. Yonezawa. Rbcl: A reflective object-oriented concurrent language without a run-time kernel. In *International Workshop on New Models for Software Architecture (IMSA): Reflection And Meta-Level Architecture*, pages 24–35, 1992.

[12] P. Moura. *Logtalk - Design of an Object-Oriented Logic Programming Language*. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal, Sept. 2003.

[13] P. Moura. Programming Patterns for Logtalk Parametric Objects. In S. Abreu and D. Seipel, editors, *Applications of Declarative Programming and Knowledge Management*, volume 6547 of *Lecture Notes in Artificial Intelligence*, pages 52–69. Springer-Verlag, Berlin Heidelberg, Apr. 2011.

[14] S. Russel and P. Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 1995.

[15] P. Tarau. Styla: a lightweight Scala-based Prolog interpreter based on a pure object oriented term hierarchy. `http://code.google.com/p/styla/`.

[16] R. Wuyts and S. Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. *International Workshop on MultiParadigm Programming with Object-Oriented Languages*, 2001.