LogicObjects: Enabling Logic Programming in Java Through Linguistic Symbiosis *

Sergio Castro[1], Kim Mens[1], and Paulo Moura[2]

[1] ICTEAM Institute, Université catholique de Louvain, Belgium
{sergio.castro,kim.mens}@uclouvain.be
[2] Center for Research in Advanced Computing Systems, INESC–TEC, Portugal
pmoura@inescporto.pt

Abstract. While object-oriented programming languages are good at modelling real-world concepts and benefit from rich libraries and developer tools, logic programming languages are well suited for declaratively solving computational problems that require knowledge reasoning. Non-trivial declarative applications could take advantage of the modelling features of object-oriented programming and of the rich software ecosystems surrounding them. Linguistic symbiosis is a common approach to enable complementary use of languages of different paradigms. However, the problem of concepts *leaking* from one paradigm to another often hinders the applicability of such approaches. This issue has mainly been reported for object-oriented languages participating in a symbiotic relation with a logic language. To address this issue, we present *LogicObjects*, a linguistic symbiosis framework for transparently and (semi-) automatically enabling logic programming in Java, that aims to solve most of the problems of *paradigm leaking* reported in other works.

Keywords: Linguistic Symbiosis, Object-Oriented Programming, Logic Programming, Multi-Paradigm Programming

1 Introduction

Object-oriented languages like Java have demonstrated their usefulness for modelling 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 are more convenient for expressing problems of declarative nature, such as expert systems [1,2]. Linguistic symbiosis [3] has been used in the past to solve the problem of integrating programs written in different languages [1]. Some limitations and issues when implementing such symbiosis, mainly from the point of view of the object-oriented language, have been highlighted in [4] and referred to as *paradigm leaking*. Building upon an earlier position paper [5], this work

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

presents a framework that overcomes most of these limitations, while providing a transparent, (semi-)automatic and customisable integration from the perspective of the object-oriented language. New in this paper are the introduction of improved mechanisms for automatic adaptation of logic routine results in the object-oriented world, a context dependent mapping of Java objects to multiple representations in Prolog, and a general mechanism for expressing Java objects in a convenient logic representation even in pure object-oriented programs. We validate our technique by comparing it to the well-known JPL library [6] for invoking logic routines from Java and illustrate the reduction in programming effort.

This paper is structured as follows. Section 2 presents our running example and a logic programming solution. Sections 3 and 4 present our framework and how it enables a transparent and automated access from Java to our implementation in logic. Section 5 discusses related work and Section 6 concludes and presents future work.

2 Case Study: *The London Underground*

Our running example addresses a typical problem that can be implemented easily with a logic language: a querying system about subway lines and stations. But public transportation systems also require a user-friendly interface, which can be developed more easily in an object-oriented language. Therefore, this is a typical case where we can profit from a symbiotic integration between Prolog and Java. In this section, we present a straightforward implementation of our example application in a logic language, discuss how common approaches typically would integrate its logic routines in an object-oriented language, and give an intuitive introduction to our approach and its advantages over current techniques.

Implementation in logic The first stage of the problem consists in expressing our knowledge about the London Underground as a set of logic statements. Most of the code in this section has been adapted from [7], with an interesting variation: instead of implementing it in plain Prolog, we use Logtalk [8], a portable object-oriented layer on top of Prolog, facilitating in this way the mapping that needs to be made between objects belonging to each of the two worlds.

```
1  :- object(metro).
2    :- public([connected/3, nearby/2, reachable/3, line/1]).
3
4    connected(station(green_park), station(charing_cross), line(jubilee)).
5    connected(station(bond_street), station(green_park), line(jubilee)).
6    connected(station(bond_street), station(oxford_circus), line(central)).
7    ...
8
9    nearby(S1, S2) :- connected(S1, S2, _).
10   nearby(S1, S2) :- connected(S1, S3, L), connected(S3, S2, L).
11
12   reachable(S1, S2, []) :- connected(S1, S2, _).
13   reachable(S1, S2, [S3| Ss]) :- connected(S1, S3, L), reachable(S3, S2, Ss).
14
15   line(Name) :- setof(L, S1^S2^connected(S1,S2,L), Ls), list::member(line(Name), Ls).
16 :- end_object.
```

Listing 2.1: The `metro` object in Logtalk

In our example, stations are *connected* to other stations by underground lines. A station is *nearby* another one if there is at most one station in between them. 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 2.1 shows the Logtalk definition of the `metro` object.[3] The `metro` object encapsulates the knowledge about how stations are connected, plus the rules for the logic predicates `nearby/2`, `reachable/3` and `line/1`. The messages (queries) that the `metro` object can respond to are specified by the `public/1` directive in line 2. Messages in Logtalk are sent using the `::/2` operator, as illustrated on line 15 for the `member/2` method.

```
1 :- object(line(_Name)).
2   :- public([connects/2]).
3
4   connects(S1, S2) :- self(Self), metro::connected(S1, S2, Self).
5 :- end_object.
```

Listing 2.2: The `line` object in Logtalk

Listing 2.2 shows the definition of a *parametric object* [9], `line/1`, which encapsulates the operations of an object representing an underground line. The object parameter denotes the name of the *line*. A `connects/2` predicate (line 4) answers stations directly connected by the *line* object receiving the message. The method implementation is delegated to the `metro` prototype object.

```
1 :- object(station(_Name)).
2   :- public([connected/2, nearby/1, reachable/2]).
3
4   connected(S, L) :- self(Self), metro::connected(Self, S, L).
5
6   nearby(S) :- self(Self), metro::nearby(Self, S).
7
8   reachable(S, IStations) :- self(Self), metro::reachable(Self, S, IStations).
9 :- end_object.
```

Listing 2.3: The `station` object in Logtalk

Our last object is the `station` object (Listing 2.3). As for the `line` object it is also a parametric object having as sole parameter the name of a station. It defines a method `connected/2` that unifies its first parameter with a station that is connected to this `station` object, through the underground line unified with the second parameter. The method `nearby/1` answers if this station is nearby another station received as a parameter. The method `reachable/2` unifies its first parameter with a station that is reachable from this station object, through a list of intermediate stations unified with the second parameter. As with the `line` object, methods in this `station` object delegate to the `metro` object.

Integration of logic routines in an object-oriented language Most approaches for integrating logic routines in an object-oriented language rely on an explicit mapping between the artefacts of the two worlds. Notions such as a *logic engine*, *logic terms*, *queries*, and *query results* are explicitly represented in the

[3] Note that we are defining a *prototype* instead of a *class* as we would do in Java. Although Logtalk also supports classes, using a prototype is simpler in this case.

object-oriented programs. In the best case, the mappings of these artefacts are simple to implement, but tend to clutter the object-oriented applications that use them with significant boilerplate code that is not related to the core functionality of the application, obscuring in this way its understanding and further evolution. As a representative example of such approaches, we show how a logic routine can be invoked from within Java using the JPL library.

Listing 2.4 shows a partial implementation of a Java class `Station` that uses this library. We include the `connected(Line)` method (lines 14–27) together with required mapping methods. This Java method delegates to the `connected/2` method of the `station/1` Logtalk object (Listing 2.3, line 4). For brevity, we do not discuss here all the details of this JPL-based implementation but we highlight that it contains no less than 14 lines of code just for dealing with mapping tasks. Furthermore, these mapping tasks rely on the existence of auxiliary adapter methods like `asTerm()` and `create(Term)` that are required everywhere we need to adapt a Java object to the logic world and back. In more complex examples, the required boilerplate code can be even more significant.

```
1  public class Station {
2      String name;
3      ...
4      //mapping an instance of Station to a logic Term
5      public Term asTerm() {
6          return new Compound("station", new Term[] {new Atom(name)});
7      }
8      //mapping a logic Term to an instance of Station
9      public static Station create(Term stationTerm) {
10         String lineName = ((Compound)stationTerm).arg(1).name();
11         return new Station(lineName);
12     }
13     //mapping a Java method to a Logtalk method
14     public Station connected(Line line) {
15         Station connectedStation = null;
16         String stationVarName = "Station";
17         Term[] arguments = new Term[]{new Variable(stationVarName),  line.asTerm()};
18         Term message = new Compound("connected", arguments);
19         Term objectMessage = new Compound("::", new Term[] {asTerm(), message});
20         Query query = new Query(objectMessage);
21         Hashtable<String, Term> solution = query.oneSolution();
22         if(solution != null) {
23             Term connectedStationTerm = solution.get(stationVarName);
24             connectedStation = create(connectedStationTerm);
25         }
26         return connectedStation;
27     }
28     ...//other methods mapped to logic routines
29 }
```

Listing 2.4: The `Station` class in Java using JPL

Towards a conceptual mapping with LogicObjects Our framework provides an alternative to avoid such explicit boilerplate mapping code. As a first example, lines 7–8 of Listing 2.5 show how the `connected(Line)` Java method gets reduced to two lines of code: the method declaration and one annotation.

```
1  @LObject(args = {"name"})
2  public abstract class Station {
3    String name;
4    public Station(String name) { this.name = name; }
5
6    //answers a station connected to this station by means of a line
7    @LMethod(args = {"LSolution", "$1"})
8    public abstract Station connected(Line line);
9
10   //answers the list of nearby stations
```

```
11    @LComposition @LMethod(args = {"LSolution"})
12    public abstract List<Station> nearby();
13
14    //answers the list of intermediate stations between this and another station
15    @LMethod(name = "reachable", args = {"$1", "LSolution"})
16    public abstract List<Station> intermediateStations(Station station);
17  }
```

Listing 2.5: The `Station` class in Java using LogicObjects

The `Station` class is the Java counterpart of the `station/1` Logtalk object defined in Listing 2.3. It declares a `name` member variable (line 3) denoting the name of the underground station. The `Line` class (Listing 2.6) is the Java counterpart of the `line/1` Logtalk object defined in Listing 2.2. It declares a `name` member variable denoting the name of the underground line.

```
1  @LObject(args = {"name"})
2  public abstract class Line {
3    String name;
4    public Line(String name) { this.name = name; }
5
6    //answers if two stations are connected by this line
7    public abstract boolean connects(Station s1, Station s2);
8
9    //answers the number of stations connected by this line
10   @LMethod(name = "connects", args = {"_", "_"})
11   public abstract int segments();
12 }
```

Listing 2.6: The `Line` class in Java using LogicObjects

Finally, the `Metro` class (Listing 2.7) is the Java counterpart of the `metro` Logtalk object defined in Listing 2.1.

```
1  public abstract class Metro {
2    //answers a list with all lines
3    @LComposition @LMethod(name="line", args={"L"})
4    public abstract List<Line> lines();
5
6    //answers an existing line with a given name
7    public abstract Line line(String s);
8  }
```

Listing 2.7: The `Metro` class in Java using LogicObjects

3 LogicObjects

In this section we describe the linguistic symbiosis techniques employed by our *LogicObjects* framework. Figure 1 lists all the annotations currently supported. Our current implementation focusses on a symbiosis from the Java point of view.

ANNOTATION	DESCRIPTION
@LObject	Maps Java objects to Logtalk objects
@LMethod	Maps a Java method to a Logtalk method
@LQuery	Maps a Java method to a Prolog query
@LSolution	Maps one logic solution to a Java object
@LComposition	Maps a set of logic solutions to a Java object
@LExpression	Defines a method return value as an object expressed as a logic term
@LDelegationObject	Maps Java objects to Logtalk objects (in the context of a method invocation)

Fig. 1: Annotations currently supported by LogicObjects.

We decided to design and implement our symbiosis from the perspective of the object-oriented language, since this is the direction that has been reported [1,4] as the most difficult to achieve transparently and automatically. We start our discussion by describing the linguistic symbiosis problems we are going to solve in the remainder of this section.

3.1 Linguistic Symbiosis

Linguistic symbiosis [3] is the ability of a program to transparently invoke routines defined in another language as if they were defined in its own language [4]. Wuyts and Ducasse [10] add that, to achieve real symbiosis, objects from one language must be understood in the other. In our particular context, these generic symbiosis requirements could be rephrased as being able to:

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

In addition, several problems specific to symbiosis between object-oriented and logic programming languages have been presented in [1,4]. We repeat the most significative from the object-oriented language perspective below:

Unbound variables: Unlike most object-oriented languages, it is common in logic programming 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 programming, 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.

The expression *"paradigm leak"* [4] has been used in the past to refer to such mapping problems, suggesting a *leakage* of concepts from one paradigm to another. Let us now discuss how our framework deals with these issues.

3.2 Translating Java 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 [10,11]. Since in our technique the first step to map an object to a logic term is to find a mapping between its class and a predicate name, we start by explaining how our framework achieves such a mapping.

Mapping Class Names to Predicate Names Brichau et al. [11] defined a mapping between class names and predicate names for the specific problem of transforming 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 customizable mapping between logic predicate names and Java classes and between predicate arguments and Java object properties.

To illustrate our technique, let us consider the implementation of the `Line` class in Java, shown in Listing 2.6. We refer to this class as a *symbiotic class* since part of its implementation is transparently managed by an object on the logic side. The `@LObject` annotation on line 1 provides custom mapping data for our framework. For example, its optional `name` attribute maps instances of this class to a Logtalk object implementing on the logic side the symbiotic methods of the class. In this case, given that no predicate name is explicitly specified, the name of the corresponding Logtalk object is automatically derived from the class name `Line`. This default mapping is a transformation from Java camel-case naming convention to Prolog names with lowercase tokens separated by underscores. E.g., the Java class `FooBar` would be translated to the Logtalk object `foo_bar`.

This is an example of how, by providing smart default mappings, we reach a complete automation in common cases. At the same time, a programmer can opt for explicitly specifying custom mappings when the defaults are not convenient.

Mapping Java Objects to Logic Terms When the object on the logic side is a parametric object, its parameters need to be declared on the Java side by means of an `args` attribute in the `@LObject` annotation. In the `Line` class example, this attribute is present in the `@LObject` annotation. It maps the instance variable `name` to the single parameter of the parametric object `line` on the logic side. An instance of the Java class `Line` with its name set to *"central"* is thus automatically translated to the logic term `line(central)`. In this example, the transformation of the object property `name` to a term is straightforward, as it is just a string. If the property had been a symbiotic object (e.g., when its class or a superclass includes an `@LObject` annotation) the transformation process would continue recursively, as the property object could also have properties that are symbiotic objects. When the object on the logic side is not a parametric object, the `@LObject` annotation can be omitted (e.g., the `Metro` class in Listing 2.7).

Mapping Logic Terms to Java Objects Translating a logic term to an object is the inverse process. However, in this case we need to consider the *translation context*, which encapsulates the translation objective and environment. With this context, we can answer questions such as: Is the translated object going to be assigned to a field? Or 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? What is the expected type of the object in the Java world?

The procedure of transforming a logic term into a Java object starts by attempting to find a symbiotic class whose name and number of parameters correspond to the logic predicate's name and arity. Once we have located and instan-

tiated the logic class, the conversion algorithm continues recursively for mapping each of the term arguments to the object properties. The context provides valuable guidance to choose the right class to instantiate. For example, different Java types could be mapped to a Prolog list representation (e.g., classes implementing the `List` or `Map` interfaces). Therefore, when translating a list term to a Java object, the expected type will influence the selection of the best mapping (e.g., symbiotic classes incompatible with the expected type will be ignored). If many symbiotic classes are compatible with the expected type, by default the framework returns the first match. This can be customized by means of a `preferedClass` attribute in the `LSolution` annotation.

3.3 Mapping Java Methods to Logic Queries

As in [4], methods are mapped by default to logic predicates with the same name and arity. An example of this mapping is found in the `connects(Station, Station)` method (Listing 2.6, line 7). Since this Java method has two parameters, it is mapped to the Logtalk method `connects/2` in Listing 2.2, line 4.

However, a programmer can always customize this mapping by adding a `@LMethod` annotation. The Java method `segments()` illustrates this (Listing 2.6, lines 10–11). As specified by the `name` and `args` annotation attributes, this method will also be mapped to the logic predicate `connects/2`. With this technique, we are thus able to map a single Logtalk predicate, `connects/2`, to different Java methods: `int segments()` and `boolean connects(Station, Station)`, according to our needs. The semantics of these mappings is explained in section 3.6.

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. Consider the `segments()` method mentioned before. Its arguments are explicitly specified by means of the `args` attribute of the `@LMethod` annotation. These arguments are interpreted as Prolog terms. In this case, both parameters are the symbol "`_`", which is interpreted as an anonymous logic variable.The class `Station` (Listing 2.5) provides examples of methods having as arguments non-anonymous variables. For instance, the predicate to which the method `connected` (lines 7–8) is mapped, takes as first argument a logic variable *LSolution* and as second argument the first parameter received by the Java method (referred to with the macro expression `$1`).

3.5 Return Values

The result of a logic query can be seen as a set of frames binding logic variables to terms, where each frame corresponds to one logic solution. The solution of a symbiotic method is a transformation from this set of frames to a Java object. By default, a Java object representation of the first logic solution (the first frame) is considered by our framework as the symbiotic method return value. This section discusses techniques for instantiating such Java object from a single logic solution. The composition of a set of solutions is discussed in Section 3.6.

Inferring Return Values from a Logic Variable Name Our first heuristic is based on a naming convention: If one of the logic variables in a query has as name *LSolution*, its binding in the frame of the first solution will be considered as the term representation of the Java object to return. As an example, reconsider the implementation of the method `connected(Line)` in the `Station` class (Listing 2.5, lines 7–8). This method is mapped to the Logtalk method `connected/2` (Listing 2.3, line 4). As specified by the `args` attribute of the `@LMethod` annotation, the query's first parameter is a Prolog variable *LSolution* and the second parameter is the term representation of the first parameter of the Java method. Upon evaluation of the query, the *LSolution* variable will be bound to a compound term of the form `station(nameStation)`. Given the convention introduced above, the return value of the symbiotic method will be the transformation of this term to a Java object according to the algorithm discussed in section 3.2.

Inferring Return Values from Method Signatures If no variable with name *LSolution* is found in the query, the framework will attempt to infer its return value from its signature. The term representation of this value has as name the method name (adapted to Prolog naming conventions) and as arguments the parameters of the method. The implementation of the `Metro` class illustrates this. The `line` method (Listing 2.7, line 7) is mapped to a method with the same name on the logic side. In case that the Logtalk method succeeds, the framework will consider as the solution to the method the logic term `line` having as argument the only string parameter of the method. This term will be converted to an instance of the `Line` class according to the algorithm discussed in Section 3.2. In case a line with the name given as a parameter of the Java method does not exist in the logic world, the method will return `null`.

Explicit Specification of Return Values The previous heuristics reduce the amount of explicit mappings that need to be specified by a programmer. However, we do provide a `@LSolution` annotation to let a programmer specify explicitly the term representation of the Java object to return, overriding the heuristics presented above. This term can be of arbitrary complexity and refer to as many logic variables as required. For instance, if we had wanted to encode explicitly the heuristics for returning the logic variable *LSolution* as the return value of the `connected(Line)` method (Listing 2.5, lines 7–8), we could have done so by annotating it with `@LSolution("LSolution")`. Since this is the default mapping for the solution, it can be omitted, but if an alternative or more complex solution is desired, this can be defined explicitly with the `@LSolution` annotation as well. An example of this is shown in listing 3.1, line 4.

Inferring Return Values from Non-Symbiotic Methods The previous techniques for specifying the return value of a method from a term representation of its result can be generalized to non-symbiotic methods. Methods that should not be mapped to logic routines, but that still want to express their return

value as a term expression, can do this by means of the `@LExpression` annotation. For example, Listing 3.1 shows the implementation of a factory class. It provides methods to instantiate certain symbiotic objects that are part of our problem domain. The first method (lines 4–5) creates a new *Station* object by specifying the term representation of its return value with a `@LSolution` annotation. This logic term has the form `station($1)`, where `$1` gets substituted by the first parameter of the method. The second factory method (lines 8–9) does something equivalent to the first one. In this case, no explicit return value is specified with a `@LSolution` annotation, implying that the framework will infer its result from the method signature as discussed before. The term representation of the value to return will be a functor with the same name as the method and having as arguments the method parameters (i.e., `line(String)`).

```
1  @LObject
2  public abstract class MetroFactory {
3      //creates a station with a given name
4      @LExpression @LSolution("station($1)")
5      public abstract Station station(String name);
6
7      //creates a line with a given name
8      @LExpression
9      public abstract Line line(String name);
10 }
```

Listing 3.1: The `MetroFactory` class in Java

3.6 Managing Multiplicity

The previous section illustrated how the framework infers the return value of a symbiotic method from the first solution of a logic routine. This section discusses how to compose a value from multiple solutions, or from properties of the logic solution set.

It is not trivial to infer that a method should return a composition of multiple solutions (e.g., as a list) instead of a single solution. 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 to return the collection of results rather than a single result. This assumption is not always valid, however. Consider, for example, the method `intermediateStations(Station)` in the `Station` class (Listing 2.5, lines 15–16). This method is mapped to the predicate `reachable/2` in the `station` Logtalk object (Listing 2.3, line 8). The `args` attribute in the `@LMethod` annotation indicates that the first parameter of the Logtalk method will be the logic term representation of the first parameter of the Java method (indicated by the macro-expression `$1`). The second parameter is the Prolog variable *LSolution*. As explained in Section 2, upon execution of the Logtalk method, the *LSolution* variable is bound to a list with the intermediate stations between the receiver station object and the station object passed as first parameter. The method return value is the value bound to that variable in the first solution, according to the heuristics discussed in section 3.5. The Java method thus returns a list of objects that corresponds to the binding of one variable in *one* solution (the first) answered by the Logtalk query. This

is an example where a method returning a collection of objects is not intended to answer a single collection of different solutions, but rather a single solution consisting of a collection of objects. In order to resolve ambiguities between both ways of interpreting collections, `LogicObjects` provides the `@LComposition` annotation. The Java method `nearby()` (Listing 2.5, lines 11–12) in class `Station` is an example of the usage of the `@LComposition` annotation (line 11). This method is mapped to the Logtalk method `nearby/1` which takes as argument an unbound logic variable *LSolution*. On the logic side, the unbound variable passed as argument will be bound to a station nearby the receiver station object. On the Java side, as in the previous example, a binding of the *LSolution* variable corresponds to the term representation of an individual solution. Given the `@LComposition` annotation, the framework considers the type of the method (a `List` class) as a container of all its solutions.

Another example is the `lines()` method in the `Metro` class (Listing 2.7, lines 3–4). In this case, the arguments of the method do not include a *LSolution* variable, neither a `@LSolution` annotation. Therefore, the term representation of each solution is given by the name and arguments of the Logtalk method (given explicitly by the `name` and `args` attributes of the `@LMethod` annotation). As in the previous example, the `@LComposition` annotation will instruct the framework to collect all these individual results in a collection. In both cases, the framework will choose a collection class implementing the Java `List` interface, given that this is the return type of the method.

Finally, the return value of a method could be inferred from properties of the complete logic solution set. For example, in case when none of the heuristics discussed in this section can be applied, the framework will inspect the return type of the method. If this is a numeric type, the return value will be the number of results of the query (e.g., the `segments()` method in class `Line`). If it is a boolean, the method answers whether the query produces at least one solution (e.g., the `connects(Station, Station)` method in class `Line`).

3.7 Delegation Objects

We have found cases where the logic representation of a Java object depends on the context where such logic representation is required. To illustrate this, consider a list in Prolog, which is represented as a comma separated list of members as in this example: `[a,b,c]`. In order for the query `[a,b,c]::length(X)` to be valid, two Logtalk objects are required (one object for the empty list, which is an atom, and a parametric object for the non-empty lists, which are compound terms). To maintain a one-to-one mapping, the list methods can be encapsulated in a `list` Logtalk object instead. This allows us to write e.g. `list::length([a,b,c], L)`. On the Java side the best logic representation for a list of objects (e.g., an implementation of `Iterable`) is a logic list term (e.g., `[a,b,c]`). Then this is the default mapping assumed by the framework if nothing else is specified. However, the Logtalk object that knows how to deal with list operations does not correspond to this default logic representation, but rather to the Logtalk object `list`. Therefore, it can be convenient to use this representation in the context

of a method invocation. To deal with this kind of situations, our framework provides a `@LDelegationObject` annotation. This annotation allows a programmer to specify mapping data that will be considered in the context of a logic method invocation, and will be ignored in any other context. The class `MyList` (Listing 3.2) shows an example. This class extends the `ArrayList` class, which is translated by default to a logic list term (as all implementations of `Iterable`).

```
1 @LDelegationObject(name="list", imports="library(types_loader)")
2 public abstract class MyList extends ArrayList<String> {
3     @LMethod(args={"$0", "LSolution"})
4     public abstract int length();
5 }
```

Listing 3.2: A list class declaring a delegation object

The `@LDelegationObject` annotation has the same attributes, with equivalent semantics, as the `@LObject` annotation. In our example, the `name` attribute specifies that the `list` object on the logic side will receive the logic messages sent to instances of this Java class. As we mentioned before, this will not affect the default logic representation of objects that are instances of this class.

The `length()` Java method is mapped to a Logtalk method with the same name. Its first argument is the term representation of an instance of `MyList` receiving the message (referred by the macro `$0`). To build this representation, the framework ignores the `@LDelegationObject` annotation and prefers the default logic representation for lists. The second argument is an unbound logic variable *LSolution*. Upon execution of the logic method, the value bound to this variable in the first solution to the query will become the Java method return value. Given the `@LDelegationObject` annotation, the receiver of the method on the logic side will be the `list` Logtalk object, which provides the method `length/2`, which will bind the second parameter to the length of the list sent as the first parameter.

3.8 Instantiating Symbiotic Classes

To use our framework, a programmer simply needs to instantiate logic classes using a provided factory method. Everything else, including the transparent import of dependencies on the logic side, is automatically managed using runtime code generation and byte code instrumentation techniques. As an example, Listing 3.3 shows an instantiation of a logic class and the invocation of a logic method. The first argument of the factory method corresponds to the logic class to instantiate. The other arguments correspond to the logic class constructor parameters. In this code snippet, the output corresponds to the number of segments on the line *central*, as specified on the logic side (Listing 2.1).

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

Listing 3.3: Instantiating a symbiotic class

4 Validation

To validate our approach, we re-implemented with LogicObjects the JPL example of Section 2 (Listing 2.4). This implementation required a significant amount of boilerplate code. Figure 2 shows the notable reduction in code size, and thus in programming effort, that can be gained by using our LogicObjects framework.

The figure also compares the result of a stress test. We show the difference in execution time required by each pair of corresponding methods in the two implementations.[4] Since currently LogicObjects employs JPL to invoke logic routines, the differences in processing time can serve as a measure of the *adaptation effort* (i.e., a measure of the complexity of adaptation heuristics in different scenarios).

There are many factors that influence such an effort. For example, methods that do not require an adaptation of their parameters (i.e., not including an `args` attribute in a `@LObject` annotation) are the ones with less impact on execution time (e.g., the `connects` method in class `Line` and the `line` method in class `Metro`). On the other hand, methods using macro expressions are among the ones with greater increase in execution time (e.g., the `connected` and `intermediateStations` methods in class `Station`). In addition, the adaptation effort is greater in methods manipulating collection of objects (e.g., the `connected` method in class `Station` and the `lines` method in class `Metro`), since it grows proportional to the amount of objects (also requiring adaptation) in such collections.

In spite of the reduction in program size, the increase in execution time is considerable. However, we regard these results as promising, since there are many optimisation paths to follow in order to reach an acceptable performance in a production setting, such as caching certain mappings so they do not have to be calculated every time, or the usage of a Prolog engine embedded in the JVM. In addition, our framework does not impose any overhead in the execution of a logic routine per se, which is often the real bottleneck performance wise, but on the adaptation of its arguments and the interpretation of its results as objects. For this reason we have preferred to avoid any premature optimisation.

	Line			Station				Metro		
		connects	segments		connected	nearby	istations		lines	line
#LOC in JPL	30	7	7	64	14	14	19	40	14	10
#LOC in LogicObjects	10	1	2	14	2	2	2	7	2	1
time JPL *		1.9	1.7		1.9	2.5	1.7		3.0	1.9
time LogicObjects*		12.9	13.6		43.8	38.4	44.3		38.4	11.6
Δt ≈ adaptation effort		11.0	11.9		41.9	35.9	42.6		35.4	9.7

* *time in seconds, 50000 executions.*

Fig. 2: A comparison between LogicObjects and JPL.

5 Related Work

Several aspects of this work are inspired by SOUL [12], a Prolog dialect that is implemented in and symbiotic with the object-oriented language Smalltalk. Particularly, we improve on the open questions and limitations reported in experiments implementing symbiosis from the object-oriented language perspective. [4]

[4] Tests accomplished with a 2.8 GHz Intel Core 2 Duo processor and 4 GB of RAM.

E.g., the reported approach in SOUL for dealing with the problem of returning multiple vs. just a single solution to a query, consists of always returning a collection. When a query has just one solution, its solution is wrapped in a collection wrapper. In order to provide an automatic adaptation, such a wrapper delegates to its wrapped object any message it cannot understand. Unfortunately, this can create subtle problems if the wrapped object is also a collection, as explained in Section 3.6. We have therefore preferred the choice of making logic methods return by default their first solution, and to explicitly use a `@LComposition` annotation whenever the expected return value should be a composition of solutions instead, thus sacrificing a bit on automation to gain on soundness. Concerning how to return values from methods implemented as logic predicates, SOUL limits this answer to the value of a logic variable or an expression written in the object-oriented language. Our approach supports this and in addition allows a programmer to express the value to return as a logic term of arbitrary complexity. We also consider that our technique for mapping method names and their arguments to their logic counterparts is as automatic as the SOUL technique (when relying on the default mappings offered by our framework), without excluding the possibility for other customizations when the defaults do not fit one's needs. Furthermore, our technique for dealing with unbound variables (expressed as annotation arguments) is simpler than the proposal of SOUL of extending the syntax and semantics of the object-oriented language to support the notion of unbound variables on the object-oriented side.

In addition to SOUL, other techniques exist that use advanced linguistic symbiosis for analyzing object-oriented programs (e.g., [13,14]). However, the focus of these techniques is on querying (or transforming) object-oriented programming artefacts from the logic side, rather than achieving an automatic and transparent linguistic symbiosis from the object-oriented language perspective.

There are a number of other works attempting to provide a symbiotic integration between object-oriented languages and Prolog. Most of them do this from the perspective of the logic language, mainly by offering a set of built-in Prolog predicates that enable easy access to the object-oriented language [15,16,17]. In the best cases, libraries for communication from the object-oriented language back to the logic world are provided, but they fail to abstract the programmer from low level mappings, requiring an explicit representation of logic concepts (logic engine, queries, logic terms) in the object-oriented program (as was the case with the JPL example). The same problem occurs for rule engines embedded in Java, like [18], that use a declarative language other than Prolog.

An interesting mapping technique from methods to logic predicates using method type parameters and annotations is presented in [19]. The main shortcoming of this approach is that the types participating in the declaration of symbiotic methods have to be logic term types. Therefore, there is no implicit mapping between objects and their term representations, but term objects must be explicitly created every time a method is invoked.

Another interesting approach that integrates Java with a logic constraint solver is presented in [20]. That work relies on a symbolic virtual machine and

the syntax of Java programs is left unmodified. Methods evaluated as logic computations are identified with an annotation. Logic variables are also identified with annotations and are limited to Java primitive types. A limitation is the lack of adaptation of the result of a logic method as in our approach; instead all logic methods must return an object instance of class `Solutions`.

6 Conclusions and Future Work

In this work we presented a framework based on linguistic symbiosis to facilitate the invocation of logic routines from an object-oriented program. The framework focusses in particular on providing a solution from the object-oriented language perspective, since for this direction many difficulties and issues have been reported in the past [4]. Our framework proposes an elegant and customizable solution to each of these previously reported problems. In essence our approach relies on extending the Java language with annotations expressing the declarative nature of certain artefacts, having a counterpart in the logic world.

Although our current implementation is based on Java, most of the ideas presented in this work can be extrapolated to other object-oriented languages. However, we have found that a statically-typed language on the object-oriented side offers considerable advantages for attaining an automatic and transparent symbiosis with a logic language, since valuable information can be extracted from the types of objects belonging to each of the two worlds. This additional type information helps to guide the automatic and transparent conversion of objects from or to the object-oriented world. Nevertheless, in a dynamically-typed language our approach could be reproduced by annotating relevant artefacts (e.g., symbiotic classes and methods) with type data.

Finally, we believe that achieving a complete automatic symbiosis is only possible under the assumption that there is a single valid mapping between artefacts belonging to two different languages. As we have demonstrated, this is not always the case and it is often desirable to let a programmer specify explicitly the desired mapping between artefacts. We have thus opted to provide a "good enough" automation that provides typical default mappings for the most common cases (thus providing a high degree of automation), while at the same time leaving enough flexibility to the programmer to provide more information on the nature and semantics of the desired mapping when required by the problem.

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 expressions 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, thus completing a full two-way symbiosis framework.

References

1. D'Hondt, M., Gybels, K., Jonckers, V.: Seamless Integration of Rule-Based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis. In: Proceedings

of the 2004 Symposium on Applied computing (SAC), ACM (2004) 1328–1335
2. Russel, S., Norvig, P. In: Artificial Intelligence, A Modern Approach. Prentice Hall (1995)
3. Ichisugi, Y., Matsuoka, S., Yonezawa, A.: 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. (1992) 24–35
4. Gybels, K.: 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)
5. Castro, S., Mens, K., Moura, P.: LogicObjects : A Linguistic Symbiosis Approach to Bring the Declarative Power of Prolog to Java. In: Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE). (June 2012)
6. Singleton, P.: JPL: A Java Interface to Prolog. http://www.swi-prolog.org/packages/jpl/java_api/index.html (September 2012)
7. Flach, P.: Simply Logical: Intelligent Reasoning by Example. John Wiley & Sons, Inc., New York, NY, USA (1994)
8. Moura, P.: Logtalk - Design of an Object-Oriented Logic Programming Language. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal (September 2003)
9. Moura, P.: Programming Patterns for Logtalk Parametric Objects. In: Applications of Declarative Programming and Knowledge Management. Volume 6547 of Lecture Notes in Artificial Intelligence. Springer-Verlag (April 2011) 52–69
10. Wuyts, R., Ducasse, S.: Symbiotic Reflection between an Object-Oriented and a Logic Programming Language. International Workshop on MultiParadigm Programming with Object-Oriented Languages (2001)
11. Brichau, J., De Roover, C., Mens, K.: Open Unification for Program Query Languages. In: Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007). (2007)
12. Wuyts, R.: Declarative Reasoning about the Structure of Object-Oriented Systems. In: Proceedings of the TOOLS USA '98 Conference, IEEE Computer Society Press (1998) 112–124
13. De Volder, K.: JQuery: A generic code browser with a declarative configuration language. In: PADL. (2006) 88–102
14. Semmle Ltd.: SemmleCode. http://semmle.com/ (2010)
15. Boulanger, D., Geske, U.: Using Logic Programming in Java Environment (Extended Abstract). Technical Report 10, Knowledge-Based Systems Group, Vienna University of Technology, Austria (1998)
16. Friedrich Bolz, C.: Pyrolog: A Prolog interpreter written in Python using the PyPy translator toolchain. `https://bitbucket.org/cfbolz/pyrolog/`
17. Paul Tarau, P.: Styla: a lightweight Scala-based Prolog interpreter based on a pure object oriented term hierarchy. `http://code.google.com/p/styla/`
18. Friedman-Hill, E.: Jess in Action: Java Rule-based Systems. Manning, Greenwich, CT (2003)
19. Cimadamore, M., Viroli, M.: Integrating Java and Prolog Through Generic Methods and Type Inference. In: Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), ACM (2008) 198–205
20. Majchrzak, T.A., Kuchen, H.: Logic java: combining object-oriented and logic programming. In: Proceedings of the 20th international conference on Functional and constraint logic programming. WFLP'11, Springer-Verlag (2011) 122–137