A Java Wrapper for DLV

Francesco Ricca

Department of Mathematics
University of Calabria,
87030 Rende (CS), Italy
ricca@mat.unical.it

Abstract. Disjunctive logic programs are a powerful tool in knowledge representation and commonsense reasoning. The recent development of an efficient disjunctive logic programming engine, named DLV, allows to exploit disjunctive logic programs for solving complex problems. However, disjunctive logic programming systems are currently missing any interface supporting the integration between commonly used software development languages (like Java or C++) and disjunctive logic programs. This paper focuses on the DLV Wrapper, a library, actually implemented in Java, that "wraps" the DLV system inside an external application, allowing to embed disjunctive logic programs inside Object-Oriented source code.

1 Introduction

Nowadays the need for representing and manipulate complex knowledge arises in different areas of computer science including, especially, Artificial Intelligence, Knowledge Management. Logic based formalisms for Knowledge Representation and Reasoning and among them Disjunctive Logic Programming (DLP) become interesting tools to satisfy this need. Disjunctive logic programs are logic programs where disjunction is allowed in the head of the rules and negation may occur in the body of the rules. Such programs are now widely recognized as valuable tool for knowledge representation and common sense reasoning [Gelfond and Lifschitz1991] [Lobo *et al.*1992]. In the last few years, much effort has been spent in this field for theoretical studies. In particular, much research has been done on the semantics of DLP. Today, the most accepted semantic is the *answer set semantics* [Gelfond and Lifschitz1991,Leone *et al.*1997]. Disjunctive logic programs with answer sets semantics allow to express very complex problems up to Σ_2^P complexity class.

The hardness of the evaluation of DLP programs discouraged the implementation of DLP engines. In 1997 appears the first solid implementation of a DLP system, called DLV (i.e. DataLog with Vel) [Eiter *et al.*2000,Faber *et al.*2001]. The DLV core language, which is disjunctive datalog with answer set semantics, has been enriched in several ways [Buccafurri *et al.*2000] and the DLV system has been improved incorporating several optimization techniques. Today, the DLV system is recognized to be the state-of-art implementation of DLP.

From a technical point of view, DLV is a highly portable program, written in ISO C++, available in binary form for various platforms (see [Faber and Pfeifer since 1996]).

The availability of a system supporting such an expressive language in an efficient way is stimulating people to use logic-based systems for the development of their applications.

Currently, the DLV system is used for educational purpose both in European and American Universities in AI and database courses. Its applicability for Knowledge Management and Information Integration is under investigation in the EU project INFOMIX.

On the other hand, today, a large number of software applications is developed by using object oriented languages like C++ and Java and the need of integrating such type of application with logic-based systems is arising. However, DLP systems do not support any type of integration with current software development tools. In particular, the DLV system cannot be easily integrated in an external application.

In this paper, we try to overcome the above problem. In particular, we describe an API, currently implemented in Java and named DLV Wrapper, which allows to embed disjunctive logic programs inside object-oriented source code.

The DLV Wrapper is an Object-Oriented library that "wraps" up the DLV system in a Java program. In other words, the DLV Wrapper acts as an interface between Java programs and the DLV system. By using a suitable hierarchy of Java classes, the DLV Wrapper allows to combine Java code with disjunctive logic programs.

We can summarize a DLV invocation by the following steps:

1. Setup input and invocation parameters.
2. Run DLV.
3. Handle DLV output.

The DLV Wrapper gives us full control on DLV execution. Note that DLV could spend much time in computing answer sets, because disjunctive logic programs can encode hard problems (they allow us to express every property that is decidable in deterministic polynomial time with an oracle in NP). But, as soon as a new model is computed, DLV outputs it. In order to handle this situation, we provided three modes of invocation: *synchronous, model synchronous* and *asynchronous*.

If we run DLV in *synchronous* mode, the Java thread calling DLV is blocked until DLV ends computation. The Java thread calling DLV can only access DLV output when DLV execution terminates. If we run DLV in *model synchronous* mode or in *asynchronous* mode, the Java thread calling DLV can access models as soon as they are computed. The DLV Wrapper gives us a method that tests if a new model is available. If we run DLV in *model synchronous* mode, this method blocks the Java thread calling DLV until a new model is computed or DLV ends. If we run DLV in *asynchronous* mode this method never blocks the Java thread calling DLV.

The DLV Wrapper provides flexible interfaces for input and output. We handle input and output of DLV by using Java objects. This feature allow us to fully embed disjunctive logic programs inside Object-Oriented source code. Moreover, input programs can be composed by several text files and in memory Java objects and output can be redirected specifying the storage device (main memory, hard disk, etc.), for each ground predicate in a model.

Importantly, the DLV Wrapper also helps data integration by providing a database connectivity mechanism that allows to import data in DLP programs and exports the result of a DLV computation by using JDBC.

In the rest of the paper, we focus on the description and usage of the DLV Wrapper API. In Section 2, we illustrate the DLV system, recalling its architecture, and the core language. In Section 3, we outline the structure of the DLV Wrapper and its internal working principles. In Section 4, we show, by a running example, how to use the DLV Wrapper. In Section 5 we draw our conclusions.

2 The DLV system

DLV is an efficient Answer Set Programming (ASP) system implementing the consistent answer set semantics [Gelfond and Lifschitz1991] with various language enhancements like support for logic programming with inheritance and queries, integer arithmetics and various built-in predicates. It is a highly portable program[1], available in binary form for various platforms (sparc-sun-solaris2.6, alpha-dec-osf4.0, i386-linux-elf-gnulibc2, ppc-apple-darwin, i386-unknown-freebsd4.2 etc.) and it is easy to build DLV on further platforms[2].

2.1 Kernel Language

The kernel language of DLV is disjunctive datalog extended with strong negation under the answer set semantics [Eiter *et al.*1997,Gelfond and Lifschitz1991].

Syntax Strings starting with uppercase letters denote variables, while those starting with lower case letters denote constants. A *term* is either a variable or a constant. An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and t_1,...,t_n are terms. A *literal* l is either an atom a (in this case, it is *positive*), or a negated atom $\neg a$ (in this case, it is *negative*).

Given a literal l, its *complementary* literal is defined as $\neg a$ if $l = a$ and a if $l = \neg a$. A set L of literals is said to be *consistent* if for every literal $l \in L$, its complementary literal is not contained in L.

In addition to literals as defined above, DLV also supports built-ins, like `#int`, `#succ`, $<$, `+`, and `*`.

For details, we refer to our full manual [Faber and Pfeifer since 1996].

A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1 \ \vee \ \cdots \ \vee \ a_n \ \texttt{:-} \ b_1, \cdots, b_k, \ \text{not} \ b_{k+1}, \cdots, \ \text{not} \ b_m.$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are literals, $n \geq 0$, $m \geq k \geq 0$, and not represents *negation-as-failure* (or *default negation*). The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r,

[1] Including all frontends, DLV consists of around 40000 lines of ISO C++.

[2] For up-to-date information on the system and a full manual please refer to the project homepage [Faber and Pfeifer since 1996].

while the conjunction $b_1, ..., b_k, \text{ not } b_{k+1}, ..., \text{ not } b_m$ is the *body* of r. A rule without head literals (i.e. $n = 0$) is usually referred to as *integrity constraint*. If the body is empty (i.e. $k = m = 0$), we usually omit the " :- " sign.

We denote by $H(r)$ the set of literals in the head, and by $B(r) = B^+(r) \cup B^-(r)$ the set of the body literals, where $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$ are the sets of positive and negative body literals, respectively.

A *disjunctive datalog program* $\mathcal{P}$ is a finite set of rules.

Semantics DLV implements the consistent answer sets semantics which has originally been defined in [Gelfond and Lifschitz1991].[3]

Before we are going to define this semantics, we need a few prerequisites. As usual, given a program $\mathcal{P}$, $U_\mathcal{P}$ (the *Herbrand Universe*) is the set of all constants appearing in$\mathcal{P}$ and B_π (the *Herbrand Base*) is the set of all possible combinations of predicate symbols appearing in $\mathcal{P}$ with constants of $U_\mathcal{P}$ possibly preceded by $\neg$, in other words, the set of ground literals constructible from the symbols in $\mathcal{P}$.

Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_\mathcal{P}$; $Ground(r)$ is also called the *Ground Instantiation* of r. In a similar way, given a program $\mathcal{P}$, $Ground(\mathcal{P})$ denotes the set $\bigcup_{r \in \mathcal{P}} Ground(r)$. For programs not containing variables $\mathcal{P} = Ground(\mathcal{P})$ holds.

For every program $\mathcal{P}$, we define its *answer sets* in two steps using its ground instantiation $Ground(\mathcal{P})$, following [Lifschitz1996]: First we define the answer sets of positive programs, then we give a reduction of general programs to positive ones and use this reduction to define answer sets of general programs.

An interpretation I is a set of literals. A consistent interpretation $I \subseteq B_\pi$ is called *closed under* a positive, i.e. not-free, program $\mathcal{P}$, if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap I \neq \emptyset$ whenever $B(r) \subseteq I$. I is an *answer set* for a positive program $\mathcal{P}$ if it is minimal w.r.t. set inclusion and closed under $\mathcal{P}$.

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program $\mathcal{P}$ w.r.t. a set $X \subseteq B_\pi$ is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap X \neq \emptyset$ holds, and deleting the negative body from the remaining rules.

An answer set of a general program $\mathcal{P}$ is a set $X \subseteq B_\pi$ such that X is an answer set of $Ground(\mathcal{P})^X$.

The core language of DLV can be used to encode problems of high computational complexity (up to Σ_2^P complexity class), in a highly declarative fashion, following "**Guess&Check**" paradigm [Buccafurri *et al.*2000], but we do not pursue this issue any further here.

3 The DLV Wrapper

In this section we describe the wrapper that we have implemented to make DLV usable from Java applications.

[3] Note that we only consider *consistent answer sets*, while in [Lifschitz1996] also the inconsistent set of all possible literals is a valid answer set.

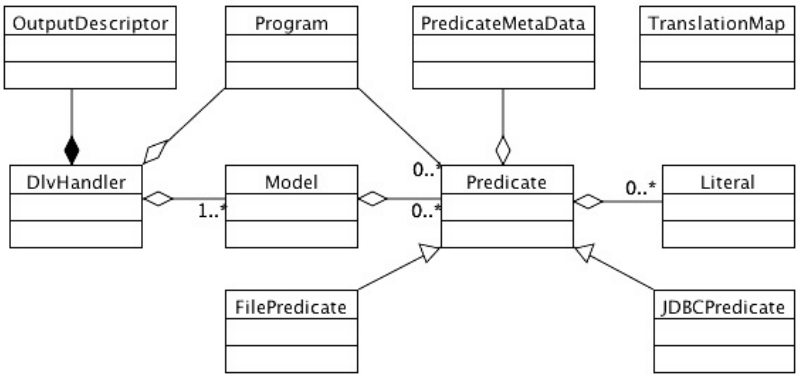

Fig. 1. The *DLV* package UML class diagram.

The DLV Wrapper is an Object-Oriented library, implemented in Java, allowing to embed a disjunctive logic program inside a Java program. The whole library is depicted in Figure 1 by an UML (Unified Modeling Language) class diagram. Next we describe classes that allow to model input and output of DLV, then we talk about the core class *DlvHandler*, outlining its most important features.

3.1 Data representation: The classes *Predicate, Literal, Model* and *Program*

In Section 2, we described DLP syntax and semantic. We briefly recall that disjunctive logic programs (DLV input) are finite sets of rules; models, representing program solutions (DLV output), are sets of ground literals.

We now describe how input and output of DLV are dealt with an Object-Oriented library. First of all, we introduce *Literal* and *Predicate* classes, that model ground predicates and its "basic bricks" (ground literals). Then, we describe *Model* class and *Program* class, that respectively model output and input of DLV.

Please note that, ground literals can also be part of a logic program. We use *Literal* and *Predicate* classes to friendly manipulate both DLV input[4] and DLV output.

The Literal class The *Literal* inner class models ground literals. It provides methods to access and modify terms, to verify if a literal is positive and to create its complement.

Literal class is an inner class of *Predicate* class (described in the following subsection). In this way, every *Literal* object is closely related to the enclosing *Predicate* instance.

The Predicate class As previously pointed out, the *Predicate* class models ground predicates.

The *Predicate* class is a powerful tool, since it allows a Java programmer to handle data to be used as DLV input and to manipulate DLV output. It provides full access to

[4] In this case we refer to the ground part of an input program.

predicate features like predicate name, arity and size[5]. Moreover, it provides two useful interfaces to access ground literals. The first one is inspired to the well-known interface *java.util.Enumeration*, the second one is based on the interface *java.sql.ResultSet* taken from JDBC.

Enumeration-like and *ResultSet-like* interfaces give full control on the underlying ground predicate implementation. The former is useful if we want to directly access *Literal* objects. The latter can help Java programmers, expert on database programming and JDBC, to handle ground predicates in a well-known way[6].

Despite columns in database tables have name and type, such kind of information does not exist for arguments in a predicate. In order to fully implement *ResultSet-like* interface we provided the *PredicateMetaData* class.

The *PredicateMetaData* class maps a name and a data type to each argument in a predicate.

We implemented, by using such a type of information, methods like *int getInt(String name)* which retrieves information, from the current row (current literal) on the column (argument) named "name", and automatically transforms it in an integer value. A *PredicateMetaData* map is only required to enable some special method in the *ResultSet-like* interface, and it is not mandatory in general.

It is worth to point out that *Predicate* class instances store ground literals in memory. This can be a serious limitation if we deal with large amount of data. To solve this problem, we actually provided two *Predicate* subclasses: *FilePredicate* and *JDBCPredicate*. The first one stores data in text files (in datalog format) and the second one stores data in relational databases.

Later, we better describe the *JDBCPredicate* class and database access.

The Model class The *Model* class represents models (i.e. answer sets).

Since models are sets of ground predicates, the *Model* class implements a collection of *Predicate* objects.

We can retrieve *Predicate* instances inside a *Model*, either specifying its name (by means of the *getPredicate* method) or in a sequential way. In the latter case, we can choose between two set of methods. The first one is inspired to the *java.util.Enumeration* interface, the second to the *java.sql.ResultSet* interface.

The static constant "NO_Model" allows to represent a program which has not models. We can test if a Model instance m is "no model" calling *m.isNoModel()* method or testing $m == Model.NO\_Model$.

The Program class The *Program* class models logic programs.

DLV has a flexible mechanism to specify input programs. It allows to divide a logic program into several text files. The *Program* class extends this mechanism. In fact, it also allows to specify input by using *String* and *Predicate* objects.

[5] The size of a predicate is the number of literal that compose it.

[6] We recall that ground predicates corresponds to tables in relational databases.

This powerful extension fully embeds logic programs inside Java programs. This way, DLV input can directly be handled by using Java objects. Moreover, we can import data from relational databases supporting JDBC, by using the *JDBCPredicate* class[7].

3.2 The DLV Wrapper architecture: the *DlvHandler* class

In this section we outline the overall dlv wrapper architecture and the *DlvHandler* class which implements kernel features of the DLV wrapper.

How previously pointed out (see Section 2) DLV is shipped as binary program. We can run it from a command line specifying invocation parameters and input programs. DLV outputs answer sets in text format.

The DLV Wrapper executes DLV, in an external native process, acting as a command line user.

Every Java application has a single instance of the class *java.lang.Runtime* which allows the application to interface itself with the environment the application is running in.

The *Runtime.exec()* method creates a native process and return an instance of a subclass of *java.lang.Process* that can be used to control the process and obtain information about it. The *java.lang.Process* class provides methods for performing input and output to a process, waiting for a process to complete, checking the exit status, and destroying (killing) a process. The *DlvHandler* class manages a DLV instance, which is a native process, by using the *Runtime.exec()* method and the *java.lang.Process* class. All DLV standard I/O operations will be redirected to the *DlvHandler* instance (inside the Java Virtual Machine) through a system pipe. This way the *DlvHandler* class feeds input to and gets output from DLV.

Please note that DLV invocation parameters and text files which contain input programs are specified as command line parameters by a string array. Data saved in memory (by using *Predicate* instances) or in databases are redirected to DLV through the system pipe.

The *DlvHandler* instance collects DLV output and parses it, building a collection of *Model* objects.

The whole process is depicted in Figure 2.

All classes the DLV Wrapper is made of are contained in a Java package named *DLV*. To embed disjunctive logic programs inside Java code we must include the *DLV* package and perform the following steps:

1. Setup input and invocation parameters.
2. Run DLV.
3. Handle DLV output.

We setup input by using a *Program* object. We use a *DlvHandler* object to set invocation parameters, run DLV process instances and handle DLV output. The *DlvHandler* class gives us full control on DLV execution. The *DlvHandler* object calls DLV, feeding input and retrieving output. As soon as DLV outputs a new model, the *DlvHandler* object parses it and build a *Model* object. The *DlvHandler* object stores each

[7] We describe this feature later.

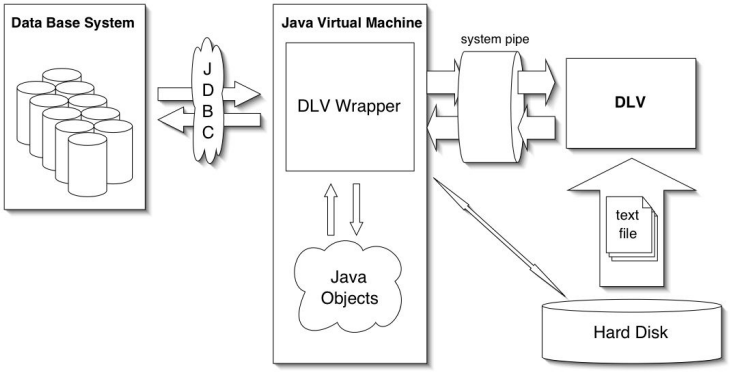

Fig. 2. The DLV invocation process.

Model object in a collection. We can handle this collection of *Model* objects by suitable methods. The *DlvHandler* class implements an interface inspired to the well-known java.util.Enumeration interface for accessing *Model* objects.

Moreover, each *DlvHandler* object has got an *OutputDescriptor* object (see Figure 1). The *OutputDescriptor* class describes how to parse DLV output. In particular, the *OutputDescriptor* describes how to build *Predicate* objects to be inserted in *Model* objects. The *DLV* package provides three types of *Predicate* objects (see Figure 1):

- the *Predicate* class;
- the *FilePredicate* class;
- the *JDBCPredicate* class.

The first one stores data in main memory; the second one stores data in a text file (in datalog format); the last one stores data in a relational database table. We specify, by using the *OutputDescriptor* class, the predicate class to be used for each ground predicate. This way, we are able to choose the storage device (main memory, hard disk, etc.) for each ground predicate[8].

How previously pointed out, DLV could spend much time in computing answer sets, because disjunctive logic programs can encode hard problems (see Section 2). But, as soon as a new model is computed, DLV outputs it. In order to handle this situation, we provided three modes of invocation: *synchronous*, *model synchronous* and *asynchronous*.

If we run DLV in *synchronous* mode, the Java thread calling DLV is blocked until DLV ends the computation. The Java thread calling DLV can only access DLV output when DLV execution terminates. If we run DLV in *model synchronous* mode or in *asynchronous* mode, the Java thread calling DLV can access models as soon as they are computed.

The DLV Wrapper provides a method that tests if a new model is available. If we run DLV in *model synchronous* mode, this method blocks the Java thread calling DLV

[8] The default storage device is main memory

until a new model is computed or DLV ends. If we run DLV in *asynchronous* mode this method never blocks the Java thread calling DLV.

Database access How previously pointed out the DLV Wrapper also helps data integration by providing a database connectivity mechanism allowing to import data in DLP programs and exporting the result of a DLV computation by using JDBC. This important feature is implemented through the *JDBCPredicate* class. The *JDBCPredicate* class wraps a javax.sql.ResultSet object inside a *Predicate* object. The *JDBCPredicate* class automatically performs a mapping from table columns to predicate arguments by a *PredicateMetadata* object[9]. Note that we easily integrate database data and logic programs by using *JDBCPredicate* objects. In fact, we always work with "*Predicate*" objects, because the *JDBCPredicate* class hide implementation details.

We can build a *JDBCPredicate* from an SQL query or a javax.sql.ResultSet object. We import data from a database including a *JDBCPredicate* in a *Program* object. We export DLV output to database tables specifying suitable mappings for ground predicates in *OutputDescriptor* objects[10]. In particular we have to map ground predicates to *JDBCPredicate* objects[11].

4 The DLV Wrapper "at work"

In this section, we show how to call DLV through the DLV Wrapper by a running example.

As previously pointed out, you must perform some step to invoke DLV. Now we show the complete list of steps you have to implement to invoke DLV:

1. Build a *Program* object and setup input.
2. Build a *DlvHandler* object.
3. Set input program and invocation parameters.
4. Run DLV.
5. Handle DLV output by using *Model*, *Predicate* and/or *Literal* classes.

In the following, we implement a Java program which outputs two possible solutions of a given Graph 3-colorability problem instance. We informally recall the Graph 3-colorability problem definition.

> Given a graph G in the input, assign each node one of three colors (say, red, green, or blue) such that adjacent nodes always have different colors.

Graph 3-colorability is a hard (NP-complete) problem.

We represent nodes and arcs with a set of facts by using $node$ (unary) and arc (binary) predicates. We can solve the problem using the following disjunctive logic

[9] The JDBCPredicate also allows to customize this mapping by using *TranslationMap* objects. We do no describe here this feature

[10] We described this feature in previous subsection.

[11] Note that, in this case, we must set the DLV invocation parameter "maximum number of models" to one.

program:

r_1 : $color(X, red) \vee color(X, blue) \vee color(X, green)$ `:-` $node(X)$
c_1 : `:-` $arc(X, Y), color(X, C), color(Y, C)$

The disjunctive rule r_1 guesses solution candidates and the c_1 constraint checks solution admissibility.

We partition DLV input in several sources. We save the disjunctive rule r_1 in the file *guessIDB.dl* and the following set of nodes in the file *nodeEDB.dl*.

```
node(minnesota). node(wisconsin). node(illinois). node(iowa). node(indiana).
node(michigan). node(ohio).
```

We write the constraint c_1 in a *String* object and we use a *Predicate* object to represent arc predicate.

```
String check=‘‘ :- arc(X, Y), color(X, C), color(Y, C).’’;
Predicate p=new Predicate(‘‘arc’’,2);
p.addLiteral(p.new Literal(new String[]{"minnesota", "wisconsin"}));
p.addLiteral(p.new Literal(new String[]{"illinois", "iowa"}))
p.addLiteral(p.new Literal(new String[]{"illinois", "michigan"}))
p.addLiteral(p.new Literal(new String[]{"illinois", "wisconsin"}))
p.addLiteral(p.new Literal(new String[]{"illinois", "indiana"}))
p.addLiteral(p.new Literal(new String[]{"indiana", "ohio"}))
p.addLiteral(p.new Literal(new String[]{"michigan", "indiana"}))
p.addLiteral(p.new Literal(new String[]{"michigan", "ohio"}))
p.addLiteral(p.new Literal(new String[]{"michigan", "wisconsin"}))
p.addLiteral(p.new Literal(new String[]{"minnesota", "iowa"}))
p.addLiteral(p.new Literal(new String[]{"wisconsin", "iowa"}))
p.addLiteral(p.new Literal(new String[]{"minnesota", "michigan"}))
```

We can now implement the first step.

```
// build a Program object and setup input
 Program pr=new Program();
//set input
 pr.addProgramFile("guessIDB.dl"); // adds disjunctive rule r1
 pr.addString("ckeck’’); // adds integrity constraint c2
 pr.addProgramFile("nodeEDB.dl"); // adds node predicate
 pr.addPredicate(p) // adds arc predicate
```

In the following code we implement second and third step:

```
// build a DlvHandler object
 DlvHandler dlv=new DlvHandler("dl.exe");
```

```
// set input program
 dlv.setProgram(pr);
// set invocation parameters
 dlv.setNumberOfModels(2); // computes no more than two solutions
 dlv.setIncludeFacts(false);
```

In the last code fragment we implement forth and fifth step:

```
try
{
// run DLV by using model synchronous method of invocation
 dlv.run(Dlv.MODEL_SYNCHRONOUS);

// DLV output handling
 while(dlv.hasMoreModels()) // for each model, wait until DLV find a new model
 {
   Model m=dlv.nextModel(); // gets next model
   if(!m.isNoModel())
   {
    while(m.hasMorePredicates()) // for each predicate in m
    {
      Predicate p=m.nextPredicate(); // gets next predicate
      System.out.println(p.toString()); // print out p
    }
    System.out.println(''--- END Model'');
   } else System.out.println(''I cannot find a model'');
 }
}
catch(DLVException d) { d.printStackTrace(); }
catch(DLVExceptionUncheked du) { du.printStackTrace(); }
finally
{
 System.err.println(dlv.getWarnings()); // print out errors
}
```

5 Conclusions and future work

We have presented the DLV Wrapper, an Object-Oriented library, currently implemented in Java, that allows to embed disjunctive logic programs inside Object-Oriented programs. Basically, the DLV Wrapper executes, in an external native process, the DLV system, feeding input and getting output through a system pipe. Moreover, by using suitable hierarchy of classes, the DLV Wrapper allows to handle DLV Input and DLV output by using Java objects and to import and export data from/to commercial database systems implementing JDBC drivers. In this way, the DLV Wrapper achieves a tight coupling between disjunctive logic programs and Java programs.

We believe that the DLV Wrapper will speed up the development of software applications, both in academia and in industry, employing the highly expressive power of DLP, by using the DLV system.
As for as current and future work is concerned, we are testing the DLV Wrapper library in real application contexts.
Moreover, we are optimizing both internal data structures and control structures in order to improve efficiency. Finally, we plan to enrich the current DLV Wrapper implementation by: (i) adding support for both client-server invocation mechanisms and XML data source handling; (ii) developing a C++ implementation of the library.

6 Acknowledgments

I thank Nicola Leone and Giovambattista Ianni for the support offered in the realization of such work.
This work was supported by the European Commission under project INFOMIX, project no. IST-2001-33570, and under project WASP, project no. IST-2001-37004.

References

[Buccafurri *et al.*2000] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[Eiter *et al.*1997] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[Eiter *et al.*2000] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[Faber and Pfeifer since 1996] Wolfgang Faber and Gerald Pfeifer. `DLV` homepage, since 1996. `http://www.dlvsystem.com/`.

[Faber *et al.*2001] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 635–640, Seattle, WA, USA, August 2001. Morgan Kaufmann Publishers.

[Gelfond and Lifschitz1991] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[Leone *et al.*1997] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation*, 135(2):69–112, June 1997.

[Lifschitz1996] Vladimir Lifschitz. Foundations of Logic Programming. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, Stanford, 1996.

[Lobo *et al.*1992] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.