JASP: A Framework for Integrating Answer Set Programming with Java

Onofrio Febbraro
DLVSystem s.r.l.
P.zza Vermicelli, Polo Tecnologico,
Rende, Italy

Giovanni Grasso
Oxford University, Department of Computer Science
Parks Road, Oxford, UK

Nicola Leone and **Francesco Ricca**
University of Calabria, Department of Mathematics
Rende, Italy

Abstract

Answer Set Programming (ASP) is a fully-declarative logic programming paradigm, which has been proposed in the area of knowledge representation and non-monotonic reasoning. Nowadays, the formal properties of ASP are well-understood, efficient ASP systems are available, and, recently, ASP has been employed in a few industrial applications. However, ASP technology is not mature for a successful exploitation in industry yet; mainly because ASP technologies are not integrated in the well-assessed development processes and platforms which are tailored for imperative/object-oriented programming languages. In this paper we present a new programming framework blending ASP with Java. The framework is based on $\mathcal{JASP}$, an hybrid language that transparently supports a bilateral interaction between ASP and Java. $\mathcal{JASP}$ specifications are compliant with the JPA standard to perfectly fit extensively-adopted enterprise application technologies. The framework also encompasses an implementation of $\mathcal{JASP}$ as a plug-in for the Eclipse platform, called **JDLV**, which includes a compiler from $\mathcal{JASP}$ to Java. Moreover, we show a real-world application developed with $\mathcal{JASP}$ and **JDLV**, which highlights the effectiveness of our ASP–Java integration framework.

1 Introduction

Answer Set Programming (ASP) (Lifschitz 1999) is a fully-declarative logic programming paradigm, which has been proposed in the area of non-monotonic reasoning and logic programming. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and use a solver to find them (Lifschitz 1999).

After many years of research, the formal properties of ASP are well-understood; notably, ASP is expressive: it can solve problems of complexity beyond NP (Eiter, Gottlob, and Mannila 1997). Moreover, the availability of robust and efficient solvers (Leone et al. 2006; Simons, Niemelä, and Soininen 2002; Lin and Zhao 2002; Babovich and Maratea 2003; Gebser et al. 2007a; Janhunen et al. 2006; Lierler 2005; Drescher et al. 2008; Gebser et al. 2007b; Denecker et al. 2009; Calimeri et al. 2011b) made ASP an effective powerful tool for advanced applications, and stimulated the development of many interesting applications. Among the others, ASP has been applied in the areas of Artificial Intelligence (Gebser et al. 2007b; Denecker et al. 2009; Calimeri et al. 2011b; Balduccini et al. 2001; Baral and Gelfond 2000; Baral and Uyan 2001; Friedrich and Ivanchenko 2008; Franconi et al. 2001; Nogueira et al. 2001; Brewka et al. 2006), Information Integration (Leone et al. 2005; Bertossi, Hunter, and Schaub 2005), and Knowledge Management (Baral 2003; Bardadym 1996; Grasso et al. 2009; 2011). Recently, we have employed ASP for developing some industrial application; in particular, we have used ASP for building systems for workforce management (Ricca et al. 2011b) and e-tourism (Ricca et al. 2010a).

The experience we have gained in developing real-world ASP-based applications, on the one hand has confirmed the viability of the industrial exploitation of ASP; but, on the other hand, it has brought into light some practical obstacles to the development of ASP-based software. The difficulties we faced within practice, have inspired the solutions provided by the development-framework we present in this paper, which is based on our on-the-field experience. We have observed that there is a strong need of integrating ASP technologies (i.e., ASP programs and solvers) in the well-assessed software-development processes and platforms, which are tailored for imperative/object-oriented programming languages. Indeed, the lesson we have learned while building real-world ASP-based applications, confirms that complex business-logic features can be developed in ASP at a lower (implementation) price than in traditional imperative languages, and the employment of ASP brings many advantages from a Software Engineering viewpoint, in flexibility, readability, extensibility, ease of maintenance, etc. However, since ASP is not a full general-purpose language, ASP programs *must be embedded*, at some point, in systems components that are usually built by employing imperative/object-oriented programming languages, e.g., for developing visual user-interfaces.

A first step toward the solution of this problem was the development of an Application Programming Interface (API) (Ricca 2003; Gallucci and Ricca 2007), which offers some methods for interacting with an ASP solver from an embedding Java program. In that work, however, the burden of the integration between ASP and Java is still in the hands of the programmer, who must take care of the (often repetitive and) time-consuming development of ad-hoc

procedures that both control the execution of an external solver and convert the application-data back and forth from logic-based to object-oriented (Java) representations. Note that, programming tools and workbenches basically offer no specific support for this development task and, developers are hindered from adopting a poorly-supported non standard technology. Another issue, which is particularly relevant in the case of enterprise applications (Fowler 2002), is the need of manipulating large databases, which store enterprise information or are often used to make persistent complex object-oriented domain models.

This paper provides some contribution in this setting, to deal with the above mentioned issues. In particular, we present a new programming framework integrating ASP with Java. The framework is based on an hybrid language, called $\mathcal{JASP}$, that transparently supports a bilateral interaction between ASP and Java. The programmer can simply embed ASP code in a Java program without caring about the interaction with the underlying ASP system. The logical ASP program can access Java variables, and the answer sets, resulting from the execution of the ASP code, are automatically stored in Java objects, possibly populating Java collections, transparently. A key ingredient of $\mathcal{JASP}$ is the mapping between (collections of) Java objects and ASP facts. $\mathcal{JASP}$ shares with Object-Relational Mapping (ORM) frameworks, such as Hibernate and TopLink, the structural issues of the *impedance mismatch* (Maier 1990; Keller, Jensen, and Agrawal 1993) problem. Indeed, the data model underlying ASP systems is the relational model (Codd 1970) as for RDBMSs. Thus, Java Objects are mapped to logic facts (and vice versa) by adopting a structural mapping strategy similar to the one employed by ORM tools for retrieving/saving persistent objects from/to relational databases. $\mathcal{JASP}$ supports both a default mapping strategy, which fits the most common programmers' requirements, and custom ORM strategies that can be specified by the programmer for advanced needs. Importantly, custom ORM specifications in $\mathcal{JASP}$ comply with the Java Persistence API (JPA) (Oracle 2009), to perfectly suit enterprise application development standards. Moreover, $\mathcal{JASP}$ supports direct access to data stored in a DBMS, since ORM strategies can be shared with the persistent object management layer, the new language also supports efficient ASP-based reasoning on large repositories of persistent objects.

The framework also encompasses an implementation of $\mathcal{JASP}$ as a plug-in for the Eclipse platform (Eclipse 2001), called **JDLV**. **JDLV** provides an advanced platform for integrating Java with DLV (Leone et al. 2006) – one of the most popular ASP systems – and with DLV^{DB} (Terracina et al. 2008) – the DLV variant working remotely on DBMS data – that can be used when efficient dealing large amounts of data, stored on DBMS, is a compelling requirement. **JDLV** includes a $\mathcal{JASP}$ compiler generating Java code embedding calls to DLV/DLV^{DB}, offering a seamless integration of ASP-based technologies within the most popular development environments for Java.

We assessed our framework on the field, and we report in this paper about an application developed with $\mathcal{JASP}$ and **JDLV**, which highlights the effectiveness of our ASP–Java integration platform.

In short, the contribution of the paper is the following.

- We propose $\mathcal{JASP}$, a language blending ASP with Java.
  - We formally define the syntax (in EBNF) and the rewriting-based semantics of the $\mathcal{JASP}$-core, along with the Object-Relational Mapping, which is transparently applied in JASP-core for the bilateral data-exchange Java-ASP and vice versa.
  - We specify the advanced features of the $\mathcal{JASP}$ language, including the dynamic composition of $\mathcal{JASP}$ modules, the access to databases, the JPA-compliant (Oracle 2009) mechanism for defining advanced mappings between Java objects and ASP facts.
- We design and develop **JDLV** – an implementation of $\mathcal{JASP}$ integrating Java with the ASP system DLV (Leone et al. 2006), that we have incorporated as a plugin in the popular Eclipse platform. (Notaby, any other ASP system can be easily supported.)
  - We specify the advanced features of **JDLV**.
  - We discuss the main issues underlying the **JDLV** implementation, particularly focusing on *Jdlvc*, a compiler from $\mathcal{JASP}$ to Java code embedding calls to DLV, which constitutes a key component of **JDLV**.
- We present a real-world application developed with $\mathcal{JASP}$ and **JDLV**, confirming the usefulness of our framework.

The remainder of this paper is organized as follows: After a brief ASP introduction, we present, in turn, the $\mathcal{JASP}$ language and the **JDLV** system. We then discuss related works, and we finally draw the conclusion after showing an application of our framework.

2 Answer Set Programming

Answer Set Programming (ASP) (Lifschitz 1999) is a truly-declarative programming paradigm proposed in the area of non-monotonic reasoning and logic programming. We refrain from reporting in this paper a formal description of both syntax and semantics of ASP, rather we provide an intuitive account of ASP as a tool for knowledge representation and reasoning by exploiting some examples. We assume the reader familiar with logic programming conventions, and refer to (Gelfond and Lifschitz 1991; Gelfond and Leone 2002; Leone et al. 2006) for a formal description, and to (Baral 2003) and (Gelfond and Leone 2002) for complementary introductory material on ASP.

In ASP, a *rule* is of the form `Head :- Body.`, where `Body` is a logic conjunction possibly involving negation, and `Head` is either an atomic formula or a logic disjunction. A rule without head is usually referred to as an integrity *constraint*. If the body is empty, the rule is called a *fact*. Rules are interpreted according to common sense principles (Gelfond and Lifschitz 1991). Intuitively, a rule can be read "`Head` is true if `Body` is true"; hence, facts represent things that are true (i.e., they model the basic knowledge of a domain); whereas, constraints represent conditions that must not hold in any solution. ASP follows a convention dating back to Prolog (Colmerauer and Roussel 1996), where strings starting with uppercase letters denote logical vari-

```
1 class Graph {
  private Set<Arc> arcs = new HashSet<Arc>();
3 private Set<String> nodes =
                        new HashSet<String>();
5 public void addNode(String id){
    nodes.insert(id); }
7 public void addArc(String from, String to){
    arcs.insert(new Arc(from,to)); }
9 public Set<Colored> compute3Coloring(){
   Set<Colored> res = new HashSet<Colored>();
11  <# in=arcs::arc,nodes::node out=res::col
      col(X,red) v col(X,green)
13                v col(X,blue) :- node(X).
     :- col(X,C), col(Y,C), arc(X,Y).
15  #>
    if_no_answerset { res = null; }
17  return res; }
}
19 public class Arc {
    public String start; public String end;  }
21 public class Colored {
    public String node;  public String color;}
```

Figure 1: A program solving the 3-Colorability problem.

```
NameMap ::= FieldAccess [::( PredName | @ )]
InMapping    ::= in = ( NameMap )+
OutMapping   ::= out = ( NameMap )+
ModuleName   ::= ( Identifier )
SolverOption ::= .( StringLiteral )
BeginModule  ::=
 <# [ModuleName] InMapping OutMapping
EndModule    ::= #> [SolverOption]
AnswerSetHandlers ::=
      [for_each_answerset { Statement }]
      [if_no_answerset { Statement }]
JASPModule   ::= BeginModule
                 ASPProgram
                 EndModule
                 AnswerSetHandlers
BlockStatement ::=
      LocalVariableDeclarationStatement |
      ClassOrInterfaceDeclaration |
      [Identifier:] Statement |
      JASPModule
.
```

Figure 2: EBNF grammar for core JASP Program modules.

ables, while strings starting with lower case letters denote constants.

As an example we present a solution of the *Reachability* problem, which is a classical deductive database application. Given a finite directed graph $G = (V, A)$, the problem is to compute all pairs of nodes $(a, b) \in V \times V$ such that b is reachable from a through a nonempty sequence of arcs in A. We represent G by the binary predicate `arc(X,Y)`, where a fact `arc(a,b)` means that G contains an arc from `a` to `b`, i.e., $(a, b) \in A$. We model the solution as follows:

```
reachable(X,Y) :- arc(X,Y).
reachable(X,Y) :- arc(X,U), reachable(U,Y).
```

The first rule states that node Y is reachable from node X if there is an arc in the graph from X to Y, while the second rule states that node Y is reachable from node X if there exists a node U such that U is directly reachable from X (there is an arc from X to U) and Y is reachable from U.

More complex problems can be solved in a natural way in ASP. Consider the following example, in which we solve an NP-complete problem known as *3-Colorability*. Given a graph $G = (V, A)$, *3-Colorability* amounts to assign to each node of G one of three colors (say, red, green, or blue) such that adjacent nodes always have distinct colors. The input graph G is represented by facts of the form `node(v)` $\forall v \in V$, and `arc(a,b)` $\forall (a, b) \in A$. The solutions is the following ASP program made by only two rules:

```
col(X,red) v col(X,green) v col(X,blue)
                                  :- node(X).
:- arc(X,Y), col(X,C), col(Y,C).
```

The disjunctive rule can be read "if X is a node then it is either colored `red` or `blue` or `green`". The constraint can be read "discard solutions where an arc connects two nodes, namely X and Y, which have both color C". Answer sets have the important property to be minimal w.r.t. subset inclusion, thus each vertex will be associated to precisely one color in any answer set.

The expressivity of ASP and its declarative nature allows one to solve complex problems; in the next section we introduce a language that combines ASP with Java that eases the usage of ASP in real-world systems.

3 The Core Fragment of $\mathcal{JASP}$

In this section we describe the kernel fragment of the $\mathcal{JASP}$ language, called $\mathcal{JASP}$*-core*. $\mathcal{JASP}$*-core* supports the embedding of monolithic blocks of plain ASP code in Java classes. The two-ways interaction with Java variables is based on a specific object-relational mapping strategy, which is able to cope with common usage scenarios. A number of advanced features are described in the next section. Hereafter, we assume familiarity with the Java programming language, and refer the reader to (Gosling, Joy, and Guy L. Steele 2005; Oracle 2011; Alves-Foss 1999) for a formal definition of both syntax and semantics of Java.

An Introductory Example. The $\mathcal{JASP}$ code is very natural and intuitive for a programmer skilled in both ASP and Java; we introduce it by exploiting the example program reported in Figure 1. In detail, we define a Graph class with a method compute3Coloring(), which computes a 3-coloring of the instance at hand. That method is implemented by embedding the ASP program introduced in the previous section. The ASP code is embedded as-it-is in a module statement (lines 11-15) defining the body of method compute3Coloring(). Fields (containing collections of Java objects), such as arcs and nodes, are mapped to corresponding predicates (Line 11) `arc` and `node`, respectively. Moreover, the local variable res is mapped as output variable corresponding to predicate `col` (Line 11). Intuitively, the meaning of this program is the following: when compute3Coloring() is called, the set nodes is transformed in a series of unary facts of the form `node(x)`, one fact for each string x in nodes; similarly, each instance of Arc stored in the variable arcs is transformed in a binary fact.

The generated facts are added to the ASP program which is evaluated (Line 15). In case no 3-coloring exists, variable res is set to **null** (Line 16); else, when the first answer set is computed, for each fact `col` contained in the solution a new object of the class Colored is created and added to res, which, in turn, is returned by the method.

EBNF Syntax. The syntax of $\mathcal{JASP}$ is a direct extension of the syntax of Java where $\mathcal{JASP}$ module statements are allowed in Java *block statements*. Figure 2 reports the EBNF specification of $\mathcal{JASP}$ modules, and the modified BlockStatements production rule to be replaced in the Java language specification defined in (Gosling, Joy, and Guy L. Steele 2005) to obtain the $\mathcal{JASP}$ language specification. (In addition, « <# »,« #> », «for_each_answerset » and «if_no_answerset » are added to the set of language keywords.) The grammar in Figure 2 uses essentially the same BNF-style conventions adopted in (Gosling, Joy, and Guy L. Steele 2005), where white spaces are ignored, non-terminals are denoted in italic, and optional productions occurring at most once are surrounded by square brackets in italic. The only notational difference with the mentioned Java language specification document concerns the way one or more alternative productions for a non-terminal are specified: they are divided by the classical «|» symbol here; they are reported on succeeding lines in (Gosling, Joy, and Guy L. Steele 2005), instead. For any non-terminal that is not defined in Figure 2 we refer the reader to the definition reported in the Java language specification, with the exception of *ASPProgram* and *PredName* specifying the syntax of ASP programs and predicate names, respectively. In principle, any ASP dialect can be plugged in a $\mathcal{JASP}$ core specification provided that variable names (matching *FieldAccess*) that are not explicitly mapped to a predicate name are also syntactically-valid predicate names (i.e. they also match *PredName*). However, we pragmatically chose to be compliant with the latest effort of language standardization in the ASP community, that is the ASP-Core format of the third ASP Competition (Calimeri et al. 2011b). Thus, a definition of both *PredName* and *ASPProgram* can be found in (Calimeri et al. 2011a). In the following, we specify how Java objects and corresponding logic facts are reciprocally created the ones from the others.

Object-Relational Mapping. For the sake of simplicity, we limit our definitions to class names and fields, since other language features (e.g., modifiers, methods) do not play a role in object-relational mappings. We always assume that Java statements are *correct* w.r.t. both syntax and typing rules; and, it is given the set of admissible (Java) *identifiers*. We consider only local scope name conventions, since the definitions introduced in the following can be extended to the general case considering fully qualified names.

A class (schema) $\mathcal{K}$ is a tuple $\mathcal{K} = \langle N, \mathcal{F} \rangle$, where N is the identifier denoting the class name, and $\mathcal{F} = \langle f_1, \ldots, f_m \rangle$ ($m \geq 0$) is the tuple of *declared* fields of N. A field f_i ($0 \leq i \leq m$) is a pair $f_i = (n_i, t_i)$ where n_i is the field name and t_i is the field type. With a little abuse of notation we refer to the type of a variable/field and to the name of the corresponding class interchangeably. We denote by $\mathcal{F}^N$ the tuple of fields of class N, and by $f_i^N = (n_i^N, t_i^N)$ the i-th field of $\mathcal{F}^N$. The set of basic types $\mathcal{B}$ contains String and all Java primitive types and corresponding boxing reference types, e.g., both **int** and Integer are basic types. The set of collection types $\mathcal{C}$ contains all valid capture conversions of Collection<?>, Set<?> and List<?> and the corresponding raw types, e.g., both Set<String> and Set are collection types. The *actual type argument* of a collection $c \in \mathcal{C}$ is denoted by $\mathcal{A}ctual(c)$, e.g., $\mathcal{A}ctual$(Set<String>) is String.

We now introduce the ORM strategy that is transparently applied in $\mathcal{JASP}$*-core*. (This is the default mapping strategy for the full language). Any Java class that is mapped to an ASP representation is required to have no-arguments constructor, and non-recursive type definition (e.g., tree-like structures are not admitted in $\mathcal{JASP}$*-core*); moreover, both array fields and collections fields are not allowed. Otherwise the $\mathcal{JASP}$*-core* program is not *correct* and, an implementation is required to issue an error. (Note that those limitations can be overcome by exploiting JPA annotations in the full language.) The $\mathcal{JASP}$*-core* ORM strategy tries to map one object per logic fact. The number of predicate arguments needed for representing an object of type N, is given by function $\mathcal{A}(\cdot)$ defined as follows:

$$\mathcal{A}(N) = \begin{cases} 1 & \text{if } N \in \mathcal{B} \\ \mathcal{A}(\mathcal{A}ctual(N)) & \text{if } N \in \mathcal{C} \\ 0 & \text{if } N \notin \mathcal{C} \cup \mathcal{B} \wedge |\mathcal{F}^N| = 0 \\ \sum_{f^N = (n^N, t^N)} \mathcal{A}(t^N) & \text{if } |\mathcal{F}^N| > 0. \end{cases}$$

Basically, to represent a field of a basic type we need one argument, to represent a collection we need as many arguments as are needed by representing his actual type, and to represent a field of a non-basic type we need as many arguments as required for representing the basic fields of the included type. Given a Java variable of name V and type T, and a predicate name N the schema mapping function $\mathcal{M}(N, V)$ associates to V a set of predicates containing a predicate of name N and arity $\mathcal{A}(T)$.

For instance, the class Arc in the previous example contains two fields of type String. In line 11 of Figure 1, the variable arcs is mapped to predicate name `arc`; in this case, $\mathcal{M}(arc, arcs)$ is applied associating the binary predicate `arc` to variable arcs. The ASP-Core language, as in ASP, does not support a predicates with variable arguments. In the case in which the predicates employed in the rules of a $\mathcal{JASP}$*-core* module have a different arity w.r.t. the one produced by the mapping then the specification is not correct, and an implementation is expected to issue an error.

ASP facts are created from Java objects by properly filling predicate attributes of the schema defined by $\mathcal{M}(\cdot, \cdot)$. Basic types are mapped to logic terms by exploiting the toString () method, if the resulting string does not match a *symbolic constant* or a *number* of ASP-Core it is surrounded by quotes. Predicate attributes are filled according to the declaration order of fields. The mapping can be inverted to obtain Java objects from ASP facts. Basically, a new Java object is created for each collection of facts matching the schema associated to an output variable mapping. In the cases in which a basic attribute is filled, in the ASP program, by a term that cannot be converted back to the expected Java type, a Java exception is thrown at runtime.

The mapping strategy defined above, in term of the naming conventions for structural ORM patterns defined in (Fowler 2002; Bauer and King 2006), corresponds to mapping classes to relations with a *compound key* made of all class attributes, combined with *embedded value* for one to one associations; The choice of using a compound key (made of all basic attributes) fits the usual way of representing information in ASP, e.g., in the example, one fact models one node. Moreover, it ensures the safe creation of new Java objects without requiring value invention in ASP programs (Calimeri, Cozza, and Ianni 2007). Note that, this strategy has the side effect of discarding both duplicates in a Collection and the object position in a List, since ASP has the "set semantics". Specific ORM strategies are employed by commercial tools to handle such a scenario in relational databases (see (Bauer and King 2006)). Based on our experience, this strategy is sufficient to handle common use cases; nonetheless custom ORM strategies can be specified with JPA annotations in the full language, as described in Section 4. We now provide the semantics of $\mathcal{JASP}$.

Semantics. We first recall that, according to the syntax of $\mathcal{JASP}$-*core*, a $\mathcal{JASP}$ module $\mathcal{J}$ is a tuple $\mathcal{J} = \langle \mathrm{N}, InMapping, OutMapping, ASPProgram, NoAns, Ans \rangle$ where N is the module name, $InMapping$ and $OutMapping$ are sets of mappings, $ASPProgram$ is the embedded ASP program, and $NoAns$ and Ans are the optional Java statements that specify the handling code to be executed either if the ASP program has no answer set or for each answer set, respectively. Conventionally, the default value for N is "module". $NoAns$ and Ans are assumed to be empty if there is no Java code following the keywords if_no_answerset and for_each_answerset, respectively. A *mapping* is of the form $V :: P$ where P is a predicate name and V is a variable/field identifier. Moreover, V is a shortcut for $V :: V$ which is admitted iff V is also a valid predicate name; and, $V :: @$ is a shortcut for $V :: T$ where T is the type of V. The semantics of a $\mathcal{JASP}$-*core* program is given by rewriting into a plain Java program. The rewriting algorithm, which is detailed later on, is based on the class Module defined as follows:

```
class Module {
   StringBuilder program = ...
   void buildFacts(String N, Object V) {
      ...}; // build facts from objects
   boolean nextAnswerSet() {
      ...}; // asp solving process
   Object createObject(String N, Object V) {
      ...}; // create objects from facts
};
```

where nextAnswerSet() implements either an ASP solving algorithm or a suitable call to an external ASP solver. The method nextAnswerSet() stops returning true if an answer set of the ASP specification stored in the program field is found, and false otherwise. Subsequent calls to nextAnswerSet() restart the search for next answer set. nextAnswerSet() is expected to throw a runtime exception in case of problems during the execution of the ASP program. Method buildFacts () (resp. createObject) implements the generation of facts from objects (resp. creation of objects from facts) described previously.

A plain Java program $P^{\mathbb{J}}$ corresponding to the meaning of a $\mathcal{JASP}$-*core* program P is obtained applying the algorithm $\mathbb{J}$ to each $\mathcal{JASP}$-*core* module statement occurring in P. In detail, given a module $\mathcal{J}$ in input the algorithm $\mathbb{J}$ executes the following steps:

- write a declaration statement of a local variable named N of type Module and initialize it as new Module object;
- write a call to N.program.add() function passing the string representation of $ASPProgram$;
- for each $V :: P \in InMapping$ write a call to N. buildFacts (P,V);
- write an if statement checking whether N.nextAnswerSet() returns false;
- write $NoAns$ statements in the then part;
- write a do while code block containing:
  - for each $V :: P \in OutMapping$ a statement assigning to V the result of the call to createObject (P,V);
  - a **break** statement if Ans is empty; or write $NoAns$ statements;
- write a call to nextAnswerSet() in the while condition.

We consider a $\mathcal{JASP}$-*core* specification correct if the program obtained by applying the above algorithm generates a correct Java program according with (Gosling, Joy, and Guy L. Steele 2005).

As an example, the result of applying algorithm $\mathbb{J}$ to the $\mathcal{JASP}$-*core* Module of Figure 1 is:

```
Module module = new Module();
module.program.add("col(X,red) OR
      col(X,green) OR col(X,blue) :- node(X).
    :- col(X,C), col(Y,C), arc(X,Y).");
module.buildFacts("arc",arcs);
module.buildFacts("node",nodes);
if ( ! module.nextAnswerset()) {return null;}
else{ do{
      res =  (Set<Colored>)
         module.createObject("col",res);
      break;
      }while(module.nextAnswerset());
}
```

Note that, if the for_each_answerset block is omitted, as in our example, the $\mathcal{JASP}$-*core* module basically ends when the first model is found, otherwise the statements in the Ans block are iteratively executed for each solution.

4 Advanced Features of the Language

In this section we describe the advanced features of $\mathcal{JASP}$ conceived for easing the development of complex applications, including: syntactic enhancements, incremental programs, database access, and structural mapping with JPA (Oracle 2009) annotations. The $\mathcal{JASP}$-*core* is extended in a natural way to include these additional features.

Named Non-positional Notation. $\mathcal{JASP}$ introduces an alternative compact notation for logic atoms modeling Java objects borrowed from (Ricca and Leone 2007), that can be implemented by rewriting in plain ASP. For instance, class Person has seven fields but we want to select the names of those how are taller than 2 meters, we write the rule

Annotation	Summary
@Entity	Indicates a class with mapping. Class name is the predicated name.
@Table (name="pred-name")	In conjunction with @Entity, to rename the default predicate name.
@Column	Identifies a class member, to be included in the mapping.
@Id	Marks a class member as identifier (key) of the relative table.
@OneToMany @ManyToOne @ManyToMany @OneToOne	On class members to denote associations multiplicity
@JoinTable (name="pred-name")	In conjunction with @OneToMany or @ManyToOne to specify a mapping realized through an associative predicate

Figure 3: Main JPA Annotations.

```
1 public void createTeam(boolean forceMixG){
2   List<Person> people = loadPeople();
    <#+ (m1)  in=people
4      inTeam(X,G) v outTeam(X,G) :-
                    people(name: X, gender:G).
6     :- #count{X: inTeam(X,G)} >5.
    #>
8   if(forceMixG){
      <#+ (m1)
10     :- inTeam(X,GX), not inTeam(Y, GY),
        people(name: Y, gender: GY), GX != GY.
12    #>}
    Set<Team> res = new HashSet<Team >();
14  <# (m1) out=res::inTeam #>
    for_each_answerset {
16  //do something with res
    }}
```

Figure 4: Dynamic Module Composition and Invocation.

```
veryTall(X) :- person(name:X,height:H), H>2.
```

instead of

```
veryTall(X) :- person(X,_,_,_,_,H,_), H>2.
```

This notation improves readability, and is less error-prone.

JPA Mappings. Real-world applications involve complex data models, that may require custom ORM mapping strategies (e.g., to deal with legacy data). Instead of reinventing the wheel, $\mathcal{JASP}$ spouses the work done in the field ORM (Fowler 2002; Bauer and King 2006), and complies with enterprise application development standards for customizing the $\mathcal{JASP}$-*core* mapping strategy. $\mathcal{JASP}$ supports JPA (Oracle 2009) standard Java annotations for defining how Java classes map to relations (logic predicates). Table 3 summarizes the most common JPA annotations employed in the paper. Note that, although ORM frameworks address different behavioral problems w.r.t. $\mathcal{JASP}$ (e.g, object persistence, transaction control, etc.), they are based on a mechanisms to describe/map Java classes into relational data. The full description of JPA's mapping features is out of the scope of this paper (see (Bauer and King 2006; Oracle 2009) for a full account); in the next section, examples and more details on JPA are provided.

An important issue to be considered in JPA mappings is the usage of *surrogate keys* that are *generated values*. Persistence frameworks generate new identifiers according to custom algorithms when persistent objects are saved, whereas $\mathcal{JASP}$ might require to create new ids also when answer sets are transformed in objects. In $\mathcal{JASP}$, it is up to the programmer ensuring that objects have a valid id before being transformed into facts, whereas, for the other direction, there are two possible strategies: (i) embed an ASP dialect supporting value invention (Calimeri, Cozza, and Ianni 2007), so that new ids can be created in the ASP part, e.g., exploiting an id function symbol holding all basic attributes. In this case, the programmer has to be aware of termination problems; (ii) give the programmer the possibility of not specifying a value for the id field by exploiting either non-positional notation or a placeholder term "generatedvalue", so that the buildFacts() procedure becomes in charge of creating new ids. The id generation function can be also shared with the actual persistence framework. The latter is the approach currently supported by our implementation. We also require that the generated id fields cannot be joined in rule bodies (in this case, the system issues a warning), which is a compromise for overcoming the fact that ASP traditionally is a function-free language. Note that, this issue does not occur in the $\mathcal{JASP}$-*core* mapping strategy, since the key is the natural one containing all predicate attributes.

Dynamic Composition of $\mathcal{JASP}$ Modules. $\mathcal{JASP}$-*core* modules are monolithic blocks of ASP rules forming a program, that are executed "in-place". To give more flexibility, we introduced *module increments*, that enable building ASP programs incrementally throughout the application. Syntactically, module increments start by «<+ », and, semantically, correspond to accumulating additional rules and facts to the (possibly new) module at hand, without triggering the solving process. Since modules are interpreted as Java variables/fields, the usual Java scope rules apply to module increments. As an example consider the snippet in Figure 4. Lines 3-7 define incrementally "m1", that generates all teams of at most five people. In line 8 we check a boolean flag that indicates whether teams composed only of people of same gender are allowed, an additional constraint is added to "m1" only in this case (lines 10-12). Finally, in line 14 the module is executed.

Accessing the Host Environment. $\mathcal{JASP}$ allows the programmer to include arbitrary Java expressions in logic rules that are evaluated at runtime. Syntactically $\mathcal{JASP}$ uses the operator ${javaExpr} that is expanded in the string obtained evaluating the Java expression «""+javaExpr » corresponding to a call to the method toString (). For instance,

```
for (int i = 0; i<10; i++)
  <#+ (dyn)
     a(${i},${i+1}). #>
```

dynamically adds ten facts to module "dyn" (i.e., `a(1,2). a(2,3),...a(10,11)`).

Navigating Objects Associations. Real-world applications may involve several associations between domain entities. Navigating associations in ASP rules correspond to

writing several joins. $\mathcal{JASP}$ offers the possibility to navigates associations directly by a dot notation. Consider a class AirCompany associated to a list of flights (Flight); each flight has a crew and list of passengers (both Person), and a person is associated to a Passport. From a variable b of type AirCompany, we navigate to passengers passports as:

```
<#+ (m2) in=b
p(N):-[b.flights.passengers.passport](N).#>
```

$\mathcal{JASP}$ allows to navigate mapped associations at any level of nesting. The advantage of this notation is two-fold. Users can leverage a compact way to write rules and the resulting program is more suitable for optimization at the compiler level. Indeed, a syntactic analysis of the program is sufficient to build facts for only those objects/associations that are actually accessed by the ASP program (e.g., in our example only passengers are actually accessed, and creation of facts corresponding to crew and flight objects can be avoided).

Accessing Databases. $\mathcal{JASP}$ supports data intensive applications enabling reasoning directly on database tables. Those features are handled in our implementation by means of DLV^{DB}, an ASP solver that executes the logic program directly on a DBMS. Database tables are mapped to predicates, by using the mapping statements fromTable=tableName@dburl::predicate (resp. toTable), for read (resp. write) access. For example:

```
<# fromTable=Connect@
   jdbc:myDriver:db-url::arc
   out=reaches
  reachable(X,Y) :- arc(X,Y).
  reachable(X,Y) :- arc(X,U), reachable(U,Y).
  reaches(Y) :- reachable("Rome",Y). #>
```

maps the table Connect to the predicate arc, and queries the database for cities reachable from Rome. The whole computation is expected to be carried out on the database. Note that this query cannot be written in SQL. Moreover, since $\mathcal{JASP}$ can be used in combination with an ORM persistence framework, it allows read-only access to the hosting database tables of persistent objects through a statement like fromDB=**class**−name::predicate. For instance, fromDB=Person::person maps the database table for the annotated class Person to the predicate table.

5 System Description

We have implemented $\mathcal{JASP}$ in a prototype development system, available at http://www.dlvsystem.com/dlvsystem/index.php/JDLV. This system consists of *Jdlvc*, a compiler to generate plain Java classes from $\mathcal{JASP}$ files, complemented with **JDLV**, a plug-in for the popular Eclipse platform (Eclipse 2001). The **JDLV** plugin extends Eclipse with the possibility of editing files in the $\mathcal{JASP}$ syntax in a friendly environment featuring: (i) *automatic completion* for assisted editing logic program rules; (ii) *dynamic code checking and errors highlighting* (producing descriptive error messages and warnings); (iii) *outline view*, a visual representation and indexing $\mathcal{JASP}$ statements, and (iv) *automatic generation of Java code*, by means of our *Jdlvc* compiler.

Given $\mathcal{JASP}$ files as input, the *Jdlvc* compiler produces plain Java classes which manage the generation of logic programs and control statements for the underlying ASP solver. *Jdlvc* is written in Java, and uses Java Reflection to analyze mappings (compile-time) and actual object types (runtime). An enhanced version of the DLV Java Wrapper library (Ricca 2003) is used to implement the solving process trough a call to a the DLV system (Leone et al. 2006). *Jdlvc*'s compilation consists of fours steps: **(1)** input parsing and data structures creation; **(2)** predicate schemas creation and validation; generation of proper Java statements for **(3)** managing logic program creation, and **(4)** for controlling ASP-solver execution and output-handling procedures.

For **(1)** we employ a JavaCC-generated parser for $\mathcal{JASP}$. Our parser is compliant with Java6 and our $\mathcal{JASP}$ implementation supports both the format of the Second ASP Competition (Calimeri et al. 2011b), and the richer DLV dialect. In **(2)** *Jdlvc* analyzes the mapping annotations of all Java classes input of some $\mathcal{JASP}$ module, to compute the relative predicate schemas and input sources (e.g., database tables or in-memory objects). Although JPA enables any complex mapping, currently *Jdlvc* supports the most common mapping annotations reported in Table 3. In **(3)** *Jdlvc* replaces $\mathcal{JASP}$ modules with proper Java statements that build the embedded logic programs. At this stage, module are first validated (e.g., unsafe rules, mismatch predicate-mapping), and rules using non-positional notation are rewritten into plain ASP (on the lines of (Ricca and Leone 2007)). In case of direct access to database tables, *Jdlvc* produces the needed DLV^{DB}-specific mapping files (Terracina et al. 2008). Last, in **(4)** , *Jdlvc* produces Java statements to call the ASP solver along with the corresponding output handling methods.

Performance Evaluation. We have evaluated *Jdlvc* to asses both (a) performance and (b) efficiency of the compiled ASP programs. Regarding (a) we observed that *Jdlvc* compiler is very efficient: it takes < 5s for a huge $\mathcal{JASP}$ file of 188938 lines of code (7.4MB large), and < 70ms for common cases (on a laptop equipped with a 2.40GHz Intel TM Core i5 CPU). Concerning (b) we observed that there is no noticeable difference comparing the execution-time of compiled code with manually-encoded equivalent solutions. (More details are available at *http://www.mat.unical.it/ ricca/downloads/jdlv-kr.zip*).

6 A Real-World Use-Case with $\mathcal{JASP}$

We describe the implementation of a real-world industrial application with $\mathcal{JASP}$. This is a follow-up of a successful ASP-based system described in (Ricca et al. 2011b) and commissioned by the ICO BLG company. ICO BLG operates in the international seaport of Gioia Tauro, which is the largest transshipment terminal of the Mediterranean coast. We have been able to fully integrate our system in a complex scenario of ERPs and decision support systems. An exhaustive description of system requirements and employed ASP encodings is out of the scope of this paper (it can be found in (Ricca et al. 2011b)); rather, we briefly introduce one of the application use-cases and focus on the development steps to showcase $\mathcal{JASP}$'s peculiarities.

```
@Entity public class Employee{
@Id   Integer id;
   String name;
@OneToMany
@JoinTable(name="hasSkill")
   Set<Skill> skills;
}
@Entity
public class AllocationDiary{
@Id   Integer id;
@OneToOne Employee employee;
@OneToOne Skill skill;
   Date date;        }
```

```
@Entity
public class Allocation{
@Id Integer id;
@OneToOne Employee employee;
@OneToOne Shift shift;
@OneToOne Skill skill; }
@Entity public class Calendar{
@Id Integer id;
@OneToOne Employee employee;
   Date date;
   Boolean isAbsent;
   Integer dayHours,weekHours,
         weekOvertime;  }
```

```
@Entity public class Skill{
@Id String name;
   Boolean isCrucial,isHeavy;
}
@Entity public class Shift{
@Id Integer id;
   Date date;
   Integer duration;
@OneToMany
Map<Skill,Integer> neededEmp;
@OneToMany
@JoinTable(name="excluded")
Set<Employee> excluded; }
```

Figure 5: Java Classes and JPA Mappings.

Workforce Management. A crucial management task for ICO BLG is team building: the problem of properly allocating teams of employees for serving cargo ships mooring in the port. To accomplish this task, several constraints have to be satisfied concerning human resources management, such as allocation of the employees with the appropriate skills, fair distribution of the working load, guaranteed coverage of crucial skills, and turnover of the heavy/dangerous roles. Non-optimal resource allocations may cause delays and consequent pecuniary sanctions.

Data Model and JPA Mappings. The first steps of the development are the design of the Java classes modeling the domain, and their object relational mapping for $\mathcal{JASP}$ and the persistence layer. Figure 5 reports the Java classes of the application involved in the team building use-case annotated with JPA mappings. The class Employee, having fields id and name, has a one-to-may association with class Skill; each skill has a name as identifier, and two boolean fields that describe whether the skill is *crucial* (i.e, owned by a few employees) or *heavy* (i.e, dangerous and exposed to risks), respectively. Working shifts are modeled by the homonym class that, besides id, date and duration, contains the number of people needed per skill, and the list of employees to be excluded for this shift for a management decision. Class Calendar models the timesheet of each employee, considering absences, worked hours and overtime for a week. Class AllocationDiary stores the history of allocations of an employee in a given role on a certain date. Finally, class Allocation is used to model the output of the reasoning task, and represents the allocation of an employee to a certain shift in a particular role. The relational schema produced by the JPA mapping is the following: *Employee(id, name), hasSkill(idEmployee, skillName),* for class Employee; *Shift(id, date, duration), neededEmployee(idShift, skillName, value),* and *excluded(idShift, idEmployee)* for Shift class; the remaing classes have no associations and their schema is directly derived from the properties. E.g., for Skill a predicate schema *Skill(name, isCrucial, isHeavy)* is produced.

Encoding in $\mathcal{JASP}$. The implementation of our use-case is reported in Figure 6. It shows the Java method computeTeam(), of the Reasoning class (details are omitted for space reasons), which, given a shift s and a set of employees e, embeds a $\mathcal{JASP}$ module to compute an allocation for s. In our description, we intentionally omit the encoding details, as out of the scope of this paper and fully described in (Ricca et al. 2011b). Instead, we focus on showing how straight and natural is the implementation of this procedure in $\mathcal{JASP}$, and how it perfectly suits the development environment by seamlessly integrating the persistence layer of the application (Hibernate, JPA compliant).

In Figure 6 Line 3, we prepare the teams variable to be filled with the result of the computation. Then we start the teambuilding module, retrieving the Calendar and AllocationDiary relations directly from the DBMS used for object persistence (Line 5-6) with a fromDB mapping (recall that, :: @ indicates to use the class name as predicate name). Then, we map s and e to predicates `employee` and `shift`, respectively. Moreover, we indicate that the output is built from the predicate `assign`, properly mapped to the variable teams (Line 7). Accessing directly the database table, we avoid its materialization in memory, which can be inefficient being calendar and historical allocations very large tables. In Line 9 begins the ASP program solving the problem according to the guess&check programming methodology. The disjunctive rule guesses, in the predicate `assign` (i.e., the output one), the selection of employees that `canBeAssigned` to the shift in appropriate roles. The rule in Line 13 computes those employees who can be assigned to the team, such as those having skills necessary to the shift at hand, but not absent that day and not exceeding the working time limit. Note that, the definition of `canBeAssigned` makes use of $\mathcal{JASP}$'s special syntax to navigate associations (Line 14-15) to access employees' skills and needed skills for the shift, without writing the corresponding joins. Absent employees are computed at line 19, exploiting the calendar table. This rule extensively uses the named notation for predicates, using class properties names.

Employees to be allocated must respect daily and weekly working time limits, as well as a maximum amount of overtime per week. At line 22 we switch to Java, where we use a for loop to dynamically compose the definition of `exceedTime`. Indeed, we use the same "template" rule which is instantiated by the environment access operator (${ max[i]}) to deal with daily, weakly and overtime our. In each iteration we inject only the parametric part (calendar's column and limit), provided as an array of strings.

After specifying the guess part, we write the "checking part" of the encoding introducing constraints to filter out un-

wanted allocations. For each shift, we impose: (i) the correct number of allocated employees per skill, (ii) that an employee has only one role in a team, and (iii) the same employee cannot be allocated on more coincident shifts. Eventually, the last two constraints are specified to: (iv) apply a "round-robin like" policy to equally distribute assignments to heavy roles and to (v) guarantee a fair distribution of the weekly workload among employees These last two constraints ensure that (i) the employee who less recently was assigned to a certain role, must be preferred to the others candidates, and (ii) among two employees to be allocated on a shift, it is preferred the one who has worked less hours during the week, if the gap is greater than a fixed threshold (i.e., maxGap).

7 Related Work and Conclusion

Languages combining together different approaches to solving programming problems are known in the literature as *multi-paradigm languages* (Hailpern 1986; Placer 1991) (See (Spinellis 1994) for a comparative analysis.) Focusing on the combination of declarative and imperative paradigms, the proposals range from API-based approaches, such as JSetL (Rossi, Panegai, and Poleo 2007), ILOG (ILOG 2011), Jess (JESS 2011); to hybrid languages, such as Alma-0 (Apt et al. 1998), DJ (Zhou 1999), Oz (Roy 2005), and PROVA (Kozlenkov et al. 2006). A punctual comparison is made difficult by the number of language-specific features; in general, $\mathcal{JASP}$, like other hybrid languages, offers a more straight and natural approach to programming than the API-based methods, not requiring to learn a new API. Compared with the second family of proposals, a distinctive feature of $\mathcal{JASP}$ is the clean separation between the two integrated programming paradigms interacting through a standard ORM interface. $\mathcal{JASP}$ introduces minimal syntax extensions both to Java and ASP, thus its specifications are both easy to learn by programmers and easier to integrate with other existing Java technologies. The integration issue being one of the drawbacks of existing multi-paradigm proposals as argued in (Rossi, Panegai, and Poleo 2007).

Works focusing on the combination of ASP and Java, are the definition of some ASP APIs (Ricca 2003; Gallucci and Ricca 2007) and the recent proposal of an hybrid language (Oetsch, Puhrer, and Tompits 2011a). The limit of the formers have been already discussed in the Introduction. Concerning the latter, which has been independently developed in parallel to $\mathcal{JASP}$, observe that it is based on a radically different strategy for the interaction with Java, where Java methods (including constructors) can be called by exploiting special atoms in ASP rules. This is very interesting idea that we plan to investigate for future $\mathcal{JASP}$ extensions.

Systems related to **JDLV** were proposed in the field of software engineering for ASP (De Vos and Schaub 2007; 2009)), which include complete IDEs (Febbraro, Reale, and Ricca 2011; Oetsch, Pührer, and Tompits 2011b), and specific programming tools (Brain and De Vos 2005; El-Khatib, Pontelli, and Son 2005; Oetsch et al. 2011; Brain et al. 2007). Comparing **JDLV** with state-of-the-art IDEs for ASP, such as ASPIDE (Febbraro, Reale, and Ricca 2011) and SeaLion (Oetsch, Pührer, and Tompits 2011b), we observe

```
public Allocation computeTeam(Shift s,
      Set<Employee> e, Integer maxGap){
 Set<Allocation> teams = Sets.newHashSet();
<#+(teambuilding)
   fromDB=AllocationDiary::lastAllocation,
         Calendar:@
   in=s::shift, e::employee
   out=teams::assign
  assign(employee:Em, shift:Sh, skill:Sk)
      v nAssign(Em,Sh,Sk) :-
                  canBeAssigned(Em,Sh,Sk).

  canBeAssigned(Em,Sh,Sk) :-
               [e.hasSkill](Em,Sk),
               [s.neededEmp](Sh,Sk,_),
               not exceedTimeLimit(Em,Sh),
   not absent(Em,Sh), not [s.excluded](Sh,Em).

  absent(E,Sh) :-  shift(id:Sh, date:D),
   Calendar(date:D, employee:E, absent:true).
#>
String[] max = {"dHours","12","wHours","36",
           "wOvertime","10"};
for(int i=0;i<max.lenght;i+=2){
  <#+(teambuilding)
  exceedTime(E,Sh) :- shift(Sh,D,Dur),
  Calendar(date:D, employee:E, ${max[i]}:M),
                       Dur+M>${max[i+1]}.
  #>}
<#(teambuilding)
 :- [s.neededEmp](Sh,Sk,EmpNum),
    #count{Em: assign(Em,Sh,Sk)} != EmpNum.

:- assign(_,Em,Sh,Sk1),
        assign(_,Em,Sh,Sk2), Sk1 != Sk2.
:- assign(_,Em,Sh1,_), assign(_,Em,Sh2,_),
                            Sh1 != Sh2.

 prefByTurnover(Em1,Em2,Sh,Sk) :-
  skill(name:Sk,isHeavy:true),
  canBeAssigned(Em1,Sh,Sk),
  canBeAssigned(Em2,Sh,Sk),
  lastAllocation(Em1,Sk,Date1),
  lastAllocation(Em2,Sk,Date2), Date1<Date2.

:- prefByTurnover(Em1,Em2,Sh,Sk),
  assign(_,Em2,Sh,Sk),
  not assign(employee:Em1,shift:Sh,skill:Sk).

 prefByFairness(Em1,Em2,Sh,Sk) :-
  canBeAssigned(Em1,Sh,Sk),
  canBeAssigned(Em2,Sh,Sk), shift(Sh,D,_),
  Calendar(date:D, employee:Em1, wHours:Wh1),
  Calendar(date:D, employee:Em2, wHours:Wh2),
  Wh1 + ${MaxGap} < Wh2.

 :- prefByFairness(Em1,Em2,Sh,Sk),
  assign(_,Em2,Sh,Sk),
  not assign(employee:Em1,shift:Sh,skill:Sk).
#>
if_no_answerset
 {throw new Exception("No allocation found");}
return teams;
```

Figure 6: Team Building with $\mathcal{JASP}$.

that **JDLV** offers less advanced ASP-program editing features. Though, none of the mentioned IDEs support the integration of ASP with imperative/object-oriented languages.

Concluding, this paper presents a framework for integrating Java and ASP. ASP code can be in-lined within Java programs, and the ASP–Java interaction relies on well-assessed ORM strategies. A plugin for the Eclipse platform achieves the seamless integration of ASP with Java development environments. The efficacy of our approach is witnessed by the straight development of a real-world application.

Ongoing work concerns the application of the $\mathcal{JASP}$ approach to other object-oriented languages (such as C++). Moreover, we are studying techniques for efficient compilation of $\mathcal{JASP}$ in Java, and we are extending the editing features of **JDLV** to improve our development platform.

Acknowledgments. This work has been partially supported by the Calabrian Region under PIA (Pacchetti Integrati di Agevolazione industria, artigianato e servizi) project DLVSYSTEM approved in BURC n. 20 parte III del 15/05/2009 - DR n. 7373 del 06/05/2009.

Giovanni Grasso has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement DIADEM, no. 246858.

References

Alves-Foss, J., ed. 1999. *Formal Syntax and Semantics of Java*, LNCS 1523. Springer.

Apt, K. R.; Brunekreef, J.; Partington, V.; and Schaerf, A. 1998. Alma-O: An Imperative Language That Supports Declarative Programming. *ACM TPLS* 20(5):1014–1066.

Babovich, Y., and Maratea, M. 2003. Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. http : *//www.cs.utexas.edu/users/tag/cmodels.html*.

Balduccini, M.; Gelfond, M.; Watson, R.; and Nogeira, M. 2001. The USA-Advisor: A Case Study in Answer Set Planning. In *LPNMR-01*, LNCS 2173, 439–442. Springer.

Baral, C., and Gelfond, M. 2000. Reasoning Agents in Dynamic Domains. In *Logic-Based Artificial Intelligence*. Kluwer. 257–279.

Baral, C., and Uyan, C. 2001. Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming. In *LPNMR-01*, LNCS 2173, 186–199. Springer.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Bardadym, V. A. 1996. Computer-Aided School and University Timetabling: The New Wave. In *PTAT'95*, LNCS 1153, 22–45. Springer.

Bauer, C., and King, G., eds. 2006. *Java Persistence with Hibernate*. Manning.

Bertossi, L. E.; Hunter, A.; and Schaub, T., eds. 2005. *Inconsistency Tolerance*, LNCS 3300. Springer.

Brain, M., and De Vos, M. 2005. Debugging Logic Programs under the Answer Set Semantics. In *ASP'05*. CEUR.

Brain, M.; Gebser, M.; Pührer, J.; Schaub, T.; Tompits, H.; and Woltran, S. 2007. Debugging asp programs by means of asp. In *LPNMR'07*, LNCS 4483, 31–43. Springer.

Brewka, G.; Coradeschi, S.; Perini, A.; and Traverso, P., eds. 2006. *ECAI 2006, Including PAIS 2006*, volume 141 of *FAIS*. IOS Press.

Calimeri, F.; Ianni, G.; Ricca, F.; and 3rd ASP Competition Organizing Committee, T. 2011a. Third ASP Competition File and language formats. TR, University of Calabria. www.mat.unical.it/aspcomp2011/files/latestSpec.pdf.

Calimeri, F.; Ianni, G.; Ricca, F.; Alviano, M.; Bria, A.; Catalano, G.; Cozza, S.; Faber, W.; Febbraro, O.; Leone, N.; Manna, M.; Martello, A.; Panetta, C.; Perri, S.; Reale, K.; Santoro, M. C.; Sirianni, M.; Terracina, G.; and Veltri, P. 2011b. The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In *LPNMR*, LNCS 6645, 388–403. Springer.

Calimeri, F.; Cozza, S.; and Ianni, G. 2007. External sources of knowledge and value invention in logic programming. *AMAI* 50(3–4):333–361. Elsevier.

Codd, E. F. 1970. A relational model of data for large shared data banks. *Communications of the ACM* 13(6):377–387.

Colmerauer, A., and Roussel, P. 1996. *The Birth of Prolog*. New York, USA. ACM.

De Vos, M., and Schaub, T., eds. 2007. *SEA'07: Software Engineering for Answer Set Programming*, volume 281. CEUR. Online at http : *//CEUR−WS.org/Vol−281/*.

De Vos, M., and Schaub, T., eds. 2009. *SEA'09: Software Engineering for Answer Set Programming*, volume 546. CEUR. Online at http : *//CEUR−WS.org/Vol−546/*.

Denecker, M.; Vennekens, J.; Bond, S.; Gebser, M.; and Truszczynski, M. 2009. The Second Answer Set Programming Competition. In *LPNMR*, LNCS 5753, 637–654.

Drescher, C.; Gebser, M.; Grote, T.; Kaufmann, B.; König, A.; Ostrowski, M.; and Schaub, T. 2008. Conflict-Driven Disjunctive Answer Set Solving. In *KR 2008*, 422–432. AAAI Press.

Eclipse. from 2001. Eclipse. http : *//www.eclipse.org/*.

Eiter, T.; Gottlob, G.; and Mannila, H. 1997. Disjunctive Datalog. *ACM TODS* 22(3):364–418.

El-Khatib, O.; Pontelli, E.; and Son, T. C. 2005. Justification and debugging of answer set programs in ASP. In *Proc. of Automated Debugging*. California, USA. ACM.

Febbraro, O.; Reale, K.; and Ricca, F. 2011. Aspide: Integrated development environment for answer set programming. In *LPNMR 2011*, LNCS 6645, 317–330. Springer.

Fowler, M. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Franconi, E.; Palma, A. L.; Leone, N.; Perri, S.; and Scarcello, F. 2001. Census Data Repair: a Challenging Application of Disjunctive Logic Programming. In *LPAR 2001*, LNCS 2250, 561–578. Springer.

Friedrich, G., and Ivanchenko, V. 2008. Diagnosis from first principles for workflow executions. TR, Alpen Adria University, Applied Informatics, Austria.

Gallucci, L., and Ricca, F. 2007. Visual Querying and Application Programming Interface for an ASP-based Ontology Language. In *SEA 07*, 56–70. CEUR.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007a. Conflict-driven answer set solving. In *IJCAI 2007*, 386–392.

Gebser, M.; Liu, L.; Namasivayam, G.; Neumann, A.; Schaub, T.; and Truszczyński, M. 2007b. The first answer set programming system competition. In *LPNMR'07*, LNCS 4483, 3–17. Springer.

Gelfond, M., and Leone, N. 2002. Logic Programming and Knowledge Representation – the A-Prolog perspective . *AI* 138(1–2):3–38.

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

Gosling, J.; Joy, B.; and Guy L. Steele, G. B., eds. 2005. *The Java Language Specification, Third Edition*. Addison-Wesley.

Grasso, G.; Iiritano, S.; Leone, N.; and Ricca, F. 2009. Some DLV Applications for Knowledge Management. In *LPNMR 2009*, LNCS 5753, 591–597. Springer.

Grasso, G.; Leone, N.; Manna, M.; and Ricca, F. 2011. ASP at work: spin-off and applications of the DLV system. In *LPNMR 2011*, LNCS 6565, 432–451. Springer.

Hailpern, B. 1986. Multiparadigm Languages and Environments - Editor's Introduction. *IEEE Software* 3(1):6–9.

ILOG. 2011. ILOG. http : *// ilog . com/products/*.

Janhunen, T.; Niemelä, I.; Seipel, D.; Simons, P.; and You, J.-H. 2006. Unfolding Partiality and Disjunctions in Stable Model Semantics. *ACM TOCL* 7(1):1–37.

JESS. 2011. JESS. http : *// www.jessrules . com/*.

Keller, A. M.; Jensen, R.; and Agrawal, S. 1993. Persistence software: Bridging object-oriented programming and relational databases. In *Proc. of ACM SIGMOD 1993*, 523–528. ACM.

Kozlenkov, A.; Peñaloza, R.; Nigam, V.; Royer, L.; Dawelbait, G.; and Schroeder, M. 2006. Prova: Rule-Based Java Scripting for Distributed Web Applications: A Case Study in Bioinformatics. In *EDBT*, LNCS 4254, 899–908. Springer.

Leone, N.; Gottlob, G.; Rosati, R.; Eiter, T.; Faber, W.; Fink, M.; Greco, G.; Ianni, G.; Kałka, E.; Lembo, D.; Lenzerini, M.; Lio, V.; Nowicki, B.; Ruzzi, M.; Staniszkis, W.; and Terracina, G. 2005. The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In *SIGMOD 2005*, 915–917. ACM.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL* 7(3):499–562.

Lierler, Y. 2005. Disjunctive Answer Set Programming via Satisfiability. In *LPNMR'05*, LNCS 3662, 447–451. Springer.

Lifschitz, V. 1999. Answer Set Planning. *ICLP'99*, 23–37.

Lin, F., and Zhao, Y. 2002. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *AAAI-2002*. AAAI Press / MIT Press.

Maier, D. 1990. Representing database programs as objects. In *Advances in database programming languages*. ACM. 377–386.

Nogueira, M.; Balduccini, M.; Gelfond, M.; Watson, R.; and Barry, M. 2001. An A-Prolog Decision Support System for the Space Shuttle. In *PADL 2001*, LNCS 1990, 169–183. Springer.

Oetsch, J.; Pührer, J.; Seidl, M.; Tompits, H.; and Zwickl, P. 2011. Videas: A development tool for answer-set programs based on model-driven engineering technology. In *LPNMR 2011*, LNCS 6645, 382–387. Springer.

Oetsch, J.; Puhrer, J.; and Tompits, H. 2011a. Extending Object-Oriented Languages by Declarative Specifications of Complex Objects using Answer-Set Programming. TR cs.AI/1112.0922, arXiv.org.

Oetsch, J.; Pührer, J.; and Tompits, H. 2011b. The SeaLion has Landed: An IDE for Answer-Set Programming. TR, Technische Universität Wien, Austria.

Oracle. 2009. JSR 317: JavaTM Persistence 2.0. http : *// jcp . org/en/ jsr / detail ?id=317*.

Oracle. 2011. JSR 901: JavaTM Language Specification. http : *// jcp . org/en/ jsr / detail ?id=901*.

Placer, J. 1991. Multiparadigm research: a new direction of language design. *SIGPLAN Not.* 26(3):9–17.

Ricca, F., and Leone, N. 2007. Disjunctive Logic Programming with types and objects: The DLV$^+$ System. *JAL* 5(3):545–573. Elsevier.

Ricca, F.; Dimasi, A.; Grasso, G.; Ielpa, S. M.; Iiritano, S.; Manna, M.; and Leone, N. 2010a. A Logic-Based System for e-Tourism. *Fundamenta Informaticae* 105((1–2)):35–55.

Ricca, F.; Grasso, G.; Alviano, M.; Manna, M.; Lio, V.; Iiritano, S.; and Leone, N. 2011b. Team-building with Answer Set Programming in the Gioia-Tauro Seaport. *TPLP*.

Ricca, F. 2003. The DLV Java Wrapper. In *ASP'03*, 305–316. Online at http : *// CEUR−WS.org/Vol−78/*.

Rossi, G.; Panegai, E.; and Poleo, E. 2007. JSetL: a Java library for supporting declarative programming in Java. *Softw., Pract. Exper.* 37(2):115–149.

Roy, P. V., ed. 2005. *Multiparadigm Programming in Mozart/Oz, Second International Conference*, LNCS 3389. Springer.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and Implementing the Stable Model Semantics. *AI* 138:181–234.

Spinellis, D. D. 1994. Programming Paradigms as Object Classes: A Structuring Mechanism. Master's thesis, Imperial College of Science, Tech. and Med. University of UK.

Terracina, G.; Leone, N.; Lio, V.; and Panetta, C. 2008. Experimenting with recursive queries in database and logic programming systems. *TPLP* 8:129–165.

Zhou, N.-F. 1999. Building Java Applets by Using DJ - A Java-based Constraint Languag. In *COMPSAC '99, 27-19 1999, Phoenix, AZ, USA*, 442–447. IEEE.