UNIVERSITÀ DELLA CALABRIA

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Matematica e Informatica

Dottorato di Ricerca in Matematica e Informatica

XXX Ciclo

Tools and Techniques for Easing the Application of Answer Set Programming

Settore Scientifico Disciplinare INF/01 INFORMATICA

Coordinatore: Ch.mo Prof. Nicola Leone

Supervisori: Prof. Francesco Calimeri

Prof.ssa Simona Perri

Dottorando: Dott. Davide Fuscà

Dedicated to my parents

Sommario

L'Answer Set Programming (ASP) è un paradigma dichiarativo per la risoluzione di problemi complessi caratterizzato dalla sua alta espressività e dalla possibilità di rappresentare conoscenza incompleta. Per tali motivi l'ASP è ampiamente utilizzato in IA nonché adoperato come strumento per la rappresentazione della conoscenza (KRR). Grazie al suo potere espressivo e alla disponibilità di numerosi sistemi ASP, l'Answer Set Programming ha recentemente guadagnato popolarità e viene utilizzato in diversi domini applicativi. Questo ha reso chiaro la necessità di apposite procedure e sistemi per facilitare lo sviluppo di applicazioni basate su ASP. Inoltre, la sua diffusione da un ambito strettamente teorico ad un ambito più pratico, richiede delle funzionalità aggiuntive per facilitare l'interoperabilità e l'integrazione con sistemi esterni. Migliorare le prestazioni degli attuali sistemi ASP è un aspetto cruciale per consentirne l'utilizzo in nuovi contesti applicativi. I contributi di questa tesi hanno l'obbiettivo di affrontare proprio tali sfide, introducendo nuovi sistemi e tecniche per facilitare l'utilizzo di ASP. In particolare è stato presentato EMBASP: un'architettura per l'integrazione dell'Answer Set Programming in sistemi esterni per generiche applicazioni con differenti piattaforme e risolutori ASP. EMBASP permette un meccanismo esplicito per la traduzione bilaterale di stringhe riconosciute dai sistemi ASP e oggetti nel linguaggio di programmazione. Nel presente lavoro sono anche stati definiti alcuni strumenti per gestire la computazione esterna in programmi ASP, implementando un'infrastruttura per garantire l'esecuzione di script Python tramite atomi esterni nel nuovo grounder ASP *I-DLV*. È stato inoltre individuato e implementato, all'interno dello stesso sistema, un'architettura addizionale per la creazione di direttive al fine di garantire

l'interoperabilità, importante per fornire direttive predisposte per connessioni a database relazionali e a grafi. Infine, per migliorare le prestazioni computazionali dei sistemi attuali ASP, sono stati presentati i sistemi *DLV2* e *I-DLV+MS*. *DLV2* aggiorna il sistema *DLV* tramite nuove tecniche di valutazione, combinando *I-DLV* con il moderno solver *wasp*, mentre *I-DLV+MS* è un nuovo sistema ASP che integra *I-DLV* con un selettore automatico il quale sceglie induttivamente il migliore solver ASP, in base alle caratteristiche intrinseche dell'istanziazione prodotta da *I-DLV*.

Abstract

Answer Set Programming (ASP) is a well-established declarative problem solving paradigm; it features high expressiveness and the ability to deal with incomplete knowledge, so it became widely used in AI and it is now recognized as a powerful tool for knowledge representation and reasoning (KRR).

Thanks to the expressive language and the availability of diverse robust systems, Answer Set Programming has recently gained popularity and has been applied fruitfully to a wide range of domains. This made clear the need for proper tools and interoperability mechanisms that ease the development of ASP-based applications. Also, the spreading of ASP from a strictly theoretical ambit to more practical aspects requires additional features for easing the interoperability and integration with other software; furthermore, improving the performance of actual ASP system is crucial for allowing the use of the potential of ASP in new practical contexts.

The contribution of this thesis aims at addressing such challenges; we introduce new tools and techniques for easing the application of ASP. In particular, we present EMBASP: a framework for the integration of ASP in external systems for general applications to different platforms and ASP reasoners. The framework features explicit mechanisms for two-way translations between strings recognisable by ASP solvers and objects in the programming language.

Furthermore, we define proper means for handling external computations in ASP programs, and implement a proper framework for explicit calls to Python scripts via external atoms into the ASP grounder *I-DLV*. We also define and implement, into the same system, an additional framework for creating ad-hoc directives for interoperability and make use of it for providing some ready-made

ones for the connection with relational and graph databases.

Eventually, we work at improving the ASP computation, and present two new ASP systems: *DLV2* and *I-DLV+MS*. *DLV2* updates *DLV* with modern evaluation techniques, combining *I-DLV* with the solver *wasp*, while *I-DLV+MS* is a new ASP system that integrates *I-DLV*, with an automatic solver selector for inductively choose the best solver, depending on some inherent features of the instantiation produced by *I-DLV*.

Contents

1 Introduction **1**

I Context **5**

2 Answer Set Programming **6**
2.1 Syntax 6
2.2 Semantics 10

3 ASP Computation **14**
3.1 Computation Flow 14
3.2 Systems 15

4 ASP at Work: Achievements and Challenges **18**
4.1 Knowledge Representation and Reasoning 18
4.1.1 Deductive Database Applications 19
4.1.2 The GCO Declarative Programming Methodology . . . 20
4.2 Applications and Challenges 23

II Integrating ASP Systems into external applications **28**

5 Improving ASP-based Software Development: State of the Art and Motivations **29**

6 **A Novel General Approach for Embedding Declarative Reasoning Modules: EMBASP** **31**
6.1 Abstract Architecture 32
6.2 Implementing EMBASP 34
6.3 Specializing the Framework 37

7 **Embedding ASP Programs** **39**
7.1 Generalizing the Framework 42

8 **EMBASP on the Field: some actual ASP-Based Applications** **44**

III Easing Interoperability of ASP Systems **47**

9 **Improving Interoperability: State of the Art and Motivations** **48**

10 **External Sources of Computations** **51**
10.1 External Atoms 51
10.2 External Atoms Mapping Policy 54
10.3 Taking Advantage from External Computations for KRR 56
10.4 Assessing External Computation Machinery 58

11 **Exchanging Data with External Sources of Knowledge** **62**
11.1 Connecting with Relational and Graph Databases 62
11.2 Experimentally Assessing the Interoperability Mechanisms . . . 66

IV Boosting Performance **69**

12 **Improving Performance: the Need for Speed** **70**

13 **A New Efficient ASP System** **72**
13.1 The *I-DLV* Grounder 73
13.2 The ASP System *DLV2* 76
13.2.1 *DLV2* Overview 76
13.3 Experiments 79

14 Automating Solver Selection **82**
14.1 *I-DLV+MS* Overview 83
14.2 Experimental Evaluation 86
14.2.1 Impact of Solver Selection 86
14.2.2 Comparison to the State of the Art 87

15 Conclusion **88**

Appendix A Embed ASP in Android Application **91**
A.1 The Cell Class 91
A.2 The Main Activity 92

Appendix B External Computation and Sources of Knowledge Encodings **96**
B.1 External Computations Benchmarks 96
B.2 Interoperability Benchmarks 101

Appendix C External Propagators in *DLV2* **104**

Chapter 1

Introduction

Answer Set Programming (ASP) is a purely declarative formalism for knowledge representation and reasoning developed in the field of logic programming and nonmonotonic reasoning. Unlike the traditional programming languages, ASP allows representation of a given computational problem by the means of a logic program specifying a description of the desired solution. The problem is encoded using logic rules, allowing for both disjunctions in rule heads and nonmonotonic negation in the body, such that its solution can be computed as models, called *answer sets*; hence, an answer set solver can be used in order to actually find such solutions [75].

Unlike formalisms like *Prolog* that have strong procedural elements, the answer set semantics is fully declarative, therefore neither the order of rules nor the order of the literals affects the result and program termination. The answer set semantics is an extension of the *stable model semantics* [61] that has been enriched and generalized. Such work on the language definition has been carried out by the scientific community, and several extensions have been studied and proposed over the years until the ASP-Core-2 [17] standard language became the official language of the ASP Competition series.

After more than twenty years of scientific research, the theoretical properties of ASP are well understood as witnessed by the availability of a number of robust and efficient systems, including *DLV* [72], *wasp* [3], *Cmodels* [74], *Smodels* [81], *IDP* [100], *lp2sat* [67], and Potassco suite, *clingo*, *clasp* and *gringo* [53, 52, 54]. The availability of such systems has enabled ASP to be employed

in many different domains for practical applications for the development of industrial and enterprise applications [31, 45, 11]. Notably, this spreading of ASP from a strictly theoretical ambit to more practical aspects make clear the need for proper tools and interoperability mechanisms that ease the development of ASP-based applications. Also, the increasing employment of ASP in many different domains requires additional features for easing the interoperability and integration with other software by accommodating external source of computation and value invention within ASP programs.

The "traditional" approach to the evaluation of ASP programs relies on a grounding module (*grounder*), that generates a propositional theory semantically equivalent to the input program, coupled with a subsequent module (*solver*) that applies propositional techniques for generating its answer sets. Many ASP tools are focused on one of the two processes, due to the complexity of implementing a *monolithic* full ASP system. However, monolithic systems offer more control over the entire process enabling new features for improving the performance due to the coupling of the grounding and solving system. Moreover, the current ASP solvers feature several different optimization techniques, thus causing them to outperform each other, depending on the domain at hand. This is due to many reasons, such as different data structures, input simplifications and heuristics that might work better or worse, depending on the specific domain. Therefore, one might think of obtaining consistently good performance over different problems by means of proper machine learning techniques that inductively choose the "best" solver according to input features that increase the performance of the actual ASP systems.

Contributions

The present work has the goal of proposing solutions for properly addressing several challenges arising from the practical application of ASP in real-world domains.

In particular, the main contributions of the work of this thesis summarised in the following:

- We present EMBASP: a framework for the integration of ASP in external systems for general applications along with ready-made specializations

to different platforms and ASP reasoners. The framework features explicit mechanisms for two-way translations between strings recognisable by ASP solvers and objects in the programming language, giving the developer the possibility to work separately on ASP-based modules and on the applications that make use of them. In order to illustrate the use of the framework, we present an actual Java implementation and several specialized libraries for the state-of-the-art ASP systems, on mobile and desktop platforms, respectively, showing some applications developed that prove the effectiveness of the framework.

- In order to facilitate the integration of ASP with external systems, as a second contribution, we extend the ASP language with the capability of handling external computations with explicit calls to Python scripts via external atoms, and with interoperability mechanisms for the connection with relational and graph databases via explicit directives for importing/-exporting data. Similar features have been already proposed in the literature; however, we propose them here mainly for two reasons. First, we wanted to enrich *I-DLV*, the new grounding module of the ASP system *DLV*[1], with such capabilites; furthermore, we wanted to guarantee optimal performance in all scenarios, even at the cost of lowering the expressivity of the extensions. It is worth noting that *I-DLV* has been conceived as a flexible tool for experimenting with ASP and its applications and as a system explicitly adapt for encompassin extensions and new features. We report the results of experiments assessing the performance of *I-DLV* while making use of its novel features.

- A third contribution lays in the field of performance optimization. To this end, we present two new ASP systems: *DLV2* and *I-DLV+MS*. *DLV2* updates *DLV* with modern evaluation techniques, combining *I-DLV* with the solver *wasp*. Also, *DLV2* extends the core modules by application-oriented features, using constructs, like directives, which customise the heuristics of the system and improve its solving capabilities.

  I-DLV+MS is a new ASP system that integrates *I-DLV* with an auto-

[1]It is worth noting that we were actively involved in the *I-DLV* project since the very beginning, being part of the core team during our PhD program

matic solver selector: machine-learning techniques are applied to inductively choose the best solver among a set of available ones, depending on the inherent features of the instantiation produced by *I-DLV*. We define a specific set of features, and then carry out an experimental analysis for computing them over the instantiations obtained from the instances of benchmarks submitted to the 6th ASP competition. Furthermore, we test *I-DLV+MS* performance both against the state-of-the-art ASP systems and the best-established multi-engine ASP system ME-ASP, proving that *I-DLV+MS*, even though still at a prototypical stage, already shows good performance.

Organization

The remainder of this work is structured as follows.

The first part introduces the syntax and the semantics of Answer Set Programming and provides examples of knowledge representation and reasoning and practical ASP applications. The second Part introduces the EMBASP framework: in particular, the main reasons for improving ASP-based software and the generalized framework EMBASP with the actual implementation are introduced. After that, the external atoms and the explicit directives implemented in *I-DLV* are introduced in Part three, to improve the interoperability of ASP systems. Afterwards, *DLV2* and *I-DLV+MS* are presented in Part four.

Part I

Context

Chapter 2

Answer Set Programming

Answer Set Programming (ASP) [13, 37, 40, 62, 79, 80] is a declarative formalism that has become widely used in Artificial Intelligence and recognized as a powerful tool for Knowledge Representation and Reasoning (KRR). In this chapter, we recall the syntax and semantics of ASP, according to the ASP-Core-2 standard [17], the official language of ASP Competition series [26, 57].

2.1 Syntax

A *term* is either a *variable*, a *constant*, an *arithmetic terms* or *functional term*.

By convention, strings starting with upper case letter refers to *variables* otherwise are *constants*. Constants can be either *symbolic constants* (strings starting with some lowercase letter), *string constants* (quoted strings) or integers.

An *arithmetic terms* has the form

- $-(t)$
- $(t \diamond u)$

where t and u are terms and $\diamond \in \{+, -, *, /\}$; parentheses can optionally be omitted in which case standard operator precedences apply.

Given a *functor* f (the function name) and terms $t_1, \ldots, t_k$ the expression $f(t_1, \ldots, t_k)$ is a functional term if $k > 0$, whereas $f()$ is a synonym for the symbolic constant f.

Example 2.1. *Example of constants, variables ,arithmetic terms, and functional terms are:*

- *Constants: 10, 1990, word, "WORD"*
- *Variables: X , Y, W, Word*
- *Function Terms: f(10), f(word), f(X), f(W,10)*
- *Arithmetic Terms: X+1, Y*(R+T),* $-Y$

A *classical atom* is $p(t_1,\ldots,t_n)$ where:

- p is a predicate of arity n;
- $t_1,\ldots,t_n$ are terms;
- $n \geq 0$.

If the arity of p is $n = 0$, parenthesis are omitted and the simpler notation p is used. An a classical atom $p(t_1,\ldots,t_n)$ is *ground* if all its terms are constants. A *literal* is either a positive literal or negative literal. A positive literal is an atom $p(t_1,\ldots,t_n)$, while a negative literal is an atom preceded by the *negation as failure* symbol `not` .

A *built-in* atom has form

$$t \succ u$$

where:

- $\succ \in \{<, \leq, >, \geq, =, \neq\}$;
- t and u are terms.

An *aggregate element* has the form

$$t_1,\ldots,t_m : l_1,\ldots,l_n$$

where:

- $t_1, \ldots, t_m$ are terms;
- $l_1, \ldots, l_n$ are literals;
- $m \geq 0$ and $n \geq 0$.

An *aggregate atom* has the form

$$\#agg\{e_1; \ldots; e_n\} \succ u$$

where:

- $e_1; \ldots; e_n$ are aggregate elements;
- $\#agg \in \{\#count, \#sum, \#min, \#max\}$;
- $\succ \in \{<, \leq, >, \geq, =, \neq\}$;
- u is a term.

Given an aggregate atom a, the expressions a and `not` a are *aggregate literals*. In the following, we write *atom* without further qualification to refer to some classical, built-in or aggregate atom.

A *disjunctive rule* or *rule* r is a formula of the form

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

where:

- $a_1 \ \ldots \ a_n$ are classical atoms;
- $b_1 \ \ldots \ b_m$ are atoms;
- $n \geq 0, m \geq k \geq 0$.

The disjunction

$$a_1 \mid \ldots \mid a_n$$

is the *head* of r, while

$$b_1, \ldots, b_k, \texttt{ not } b_{k+1}, \ldots, \texttt{ not } b_m$$

is the *body* of r. A rule without head atoms (i.e. $n = 0$) is usually referred to as an *integrity constraint*. A rule has precisely one head atom (i.e. $n = 1$) is called *normal rule* and if it have empty body (i.e. $k = m = 0$), it is called a *fact*.

We denote by $H(r)$ the set of the head atoms, and by $B(r)$ the set of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., negatively) in $B(r)$. Hence, the following sets are associated with a rule r of the form:

- $H(r) = \{a_1, \dots a_n\}$;
- $B(r) = \{b_1, \dots, b_k, \texttt{not } b_{k+1}, \dots, \texttt{not } b_m\}$;
- $B^+(r) = \{b_1, \dots, b_k\}$;
- $B^-(r) = \{\texttt{not } b_{k+1}, \dots, \texttt{not } b_m\}$.

For a literal L, $var(L)$ denotes the set of variables occurring in L. For a conjunction (or a set) of literals C, $var(C)$ denotes the set of variables occurring in the literals in C, and, for a rule r, $var(r) = var(H(r)) \cup var(B(r))$. A rule r is *safe* if each variable appearing in r appears also in some positive body literals of r, i.e. $var(r) = var(B^+(r))$.

Example 2.2. *Consider the following rules:*

- $r_1 : a(X) \text{:-}\ b(X, Y + X).$
- $r_2 : a(Y + 1) \text{:-}\ \texttt{not}\ b(Y).$
- $r_3 : a(X, f(Y)) \text{:-}\ b(X, Z).$

Rule r_1 is safe because all the variables appearing in r_1 also appear in the positive body literals $b(X,Y)$, however r_2 and r_3 are not safe because of variables Y.

A *weak constraint* has the form

$$:\sim\ b_1, \dots, b_n. [w@l, t_1, \dots, t_m]$$

where:

- $b_1, \ldots, b_n$ are literals;
- $t_1, \ldots, t_m$ are terms;
- $n > 0$ and $m \geq 0$;
- w standing for weight;
- l standing for level.

Writing the part "$@l$" can optionally be omitted if $l = 0$; that is, a weak constraint has level 0 unless specified otherwise.

A program P is a finite set of rules and weak constraints. A program is ground if no variables appear in it.

A predicate p is an *Extensional Database* (EDB) predicate if for each rule r with $p \in H(r)$, r is a fact; all other predicates are referred to as *Intensional Database* (IDB) predicates.

2.2 Semantics

Given a disjunctive logic program P:

- **The Herbrand Universe** of P, denoted as U_P, is the set of all constants appearing in P;
- the **Herbrand Base** of P, denoted as B_P, is the set of all ground atoms constructible from the predicate symbols appearing in P and the constants of U_P.

A substitution σ is a mapping from a set V of variables to the Herbrand universe U_P of a given program P. For some object O (rule, weak constraint, literal, aggregate element, etc.), we denote by $\sigma(O)$ the object obtained by replacing each occurrence of a variable $v \in V$ by $\sigma(v)$ in O.

Given a collection $e_1; \ldots; e_n$ of aggregate elements, the *instantiation* of $\{e_1; \ldots; e_n\}$ is the following set of aggregate elements

$$inst(\{e_1; \ldots; e_n\}) = \bigcup_{1 \leq i \leq n} \{\sigma(e_i) | \sigma \text{ is well-formed substitution for } e_i\}$$

A substitution σ is *well-formed* if the arithmetic evaluation, performed in a standard way, of any arithmetic subterm is well-defined.

A *ground instance* of a rule or weak constraint r denoted as $ground(r)$ is a well-formed substitution σ for r and for every aggregate atom appearing in $\sigma(r)$, $\{e_1;\ldots;e_n\}$ is replaced by $inst(\{e_1;\ldots;e_n\})$.

Therefore we denote by $ground(P)$ the set of all the ground instances of the rules occurring in P

$$ground(P) = \bigcup_{r \in rules(P)} ground(r)$$

An *interpretation* I of P is a set of ground atoms that is a subset of B_P. A ground positive literal A is *true* with respect to an interpretation I if and only if $A \in I$, otherwise A is *false*. A ground negative literal `not` A is *true* respect to an interpretation I if and only if A is *false*, otherwise is `not` A is *false*.

An interpretation I *satisfies* a ground rule r if at least one atom in $H(r)$ is *true*, i.e. $H(r) \cap I \neq \emptyset$, whenever all body literals of r are *true*, i.e. $B^+(r) \subset I$ and $B^-(r) \cap I = \emptyset$. Finally, an interpretation M is a *model* of P if M satisfies all rules in $ground(P)$. A model M for P is *minimal* if no model N for P exists such that N is a proper subset of M, i.e. $M \subset N$. The set of all minimal models for P is denoted by $MM(P)$

Given a ground program P and an interpretation I, the *reduct* of P respect I, denoted by P^I, is obtained from $ground(P)$:

- deleting rules with false negative literals, i.e. $\forall r \in ground(P)$ such that $B^-(r) \cap I \neq \emptyset$;
- removing all *true* negative literals such that $\forall r' \in ground(P)^I$, $H(r') = H(r)$ and $B(r') = B(r)^+$.

Definition 1. *[83, 62] Let I be an interpretation for a program P. I is an* ***answer set*** *for P if $I \in MM(P^I)$ (i.e., I is a minimal model for the program P^I).The set of all answer sets for P is denoted by $AS(P)$.*

Example 2.3. *Given the general logic program P_1 :*

$$p_2 \mid p_3 \text{ :- } p_1.$$

$$p_3 \text{ :- } \texttt{not } p_2, \texttt{ not } p_1.$$

$$p_2 \mid p_1 \text{ :- } \texttt{not } p_3.$$

and the interpretation $I = \{p_3\}$, the reduct P_1^I consists of $\{p_3 \text{ :- } \texttt{not } p_2, \texttt{not } p_1.\}$.

It is easy to see that I is a minimal model of P_1^I, and for this reason it is also an answer set of P_1. Now, consider $J = \{p_2\}$. The reduct P_1^J is $\{p_2 \mid p_1 \text{ :- } \texttt{not } p_3.\}$ and it can be easily verified that J is an answer set of P_1. If, on the other hand, we take $K = \{p_1\}$, the reduct P_1^K is $\{p_2 \mid p_3 \text{ :- } p_1., \ p_2 \mid p_1 \text{ :- } \texttt{not } p_3.\}$ and K is not an answer set of P_1^K: the rule $\{p_2 \mid p_3 \text{ :- } p_1.\}$, is not true w.r.t K and hence K is not a model for P_1^K. Indeed, it can be verified that I and J are the only answer sets of P_1.

Optimal Answer Set To select the optimal answer sets of $AS(P)$, we map an interpretation I to P as follow:

weak(P,I) = $\{(w@l, t_1, \ldots, t_m)|$
:∼ $b_1, \ldots, b_n.[w@l, t_1, \ldots, t_m]$ occurs in $ground(P)$ and
$b_1, \ldots, b_n$ are true respect to I $\}$

For any integer l, let

$$P_l^I = \Sigma_{(w@l, t_1, \ldots, t_m) \in weak(P,I)} w$$

denote the sum of integers w over tuples with $w@l$ in $weak(P,I)$. Then, an answer set $I \in AS(P)$ is *dominated* by $I' \in AS(P)$ if there is some integer l such that $P_l^{I'} < P_l^I$ and $P_{l'}^{I'} = P_{l'}^I$ for all $l' > l$.

Definition 2. *An answer set $I \in AS(P)$ is* ***optimal*** *if there is no $I' \in AS(P)$ such that I is dominated by I'*

Example 2.4. *Consider the following program P:*

$$a \mid b.$$

$$b \mid c.$$

$$d \mid e \ :-\ a,\ c.$$

$$:\sim\ b.[1@1]$$

$$:\sim\ a,\ e.[4@1]$$

$$:\sim\ c,\ d.[3@1]$$

The $AS(P)$ are: $A1 = \{a,c,d\}$, $A2 = \{a,c,e\}$, $A3 = \{b\}$. Then, we have: $P_1^{A1} = 3$, $P_1^{A2} = 4$, $P_1^{A3} = 1$. Thus, the unique (optimal) answer set is $A1 = \{b\}$ with weight 1 at level 1.

Chapter 3

ASP Computation

The traditional approach for a computation of an answer set is characterized by two phases, namely *instantiation* (*grounding*) and *answer set search*, as depicted in Figure 3.1. The first step transforms the input program into a semantically equivalent one, i.e. a program without variables; the second phase applies a propositional algorithm for finding answer sets.

3.1 Computation Flow

The instantiation phase is much more than a simple replacement of a variable by using all possible ground terms. Indeed, grounding solves a complex problem which is in general EXPTIME-hard [33]. Hence, instantiation has a significant impact on the performance of the whole ASP system, because its output is the input for an ASP solver which then, in the worst case, takes exponential time in the size of the input [9, 10]. Therefore, an ASP grounder efficiently produces a ground program which has precisely the same answer sets as the full one but is much smaller in general.

The solving phase is divided into two modules: *Model Generator* and *Answer Set Checker*. The Model Generator takes as input a propositional ASP program produced by the grounder and returns as output answer set candidates. Meanwhile, the goal of the Answer Set Checker is to verify if a model is an answer set for an input program. This task is computationally expensive in general, because checking the stability of a model is well-known to be co-NP-complete

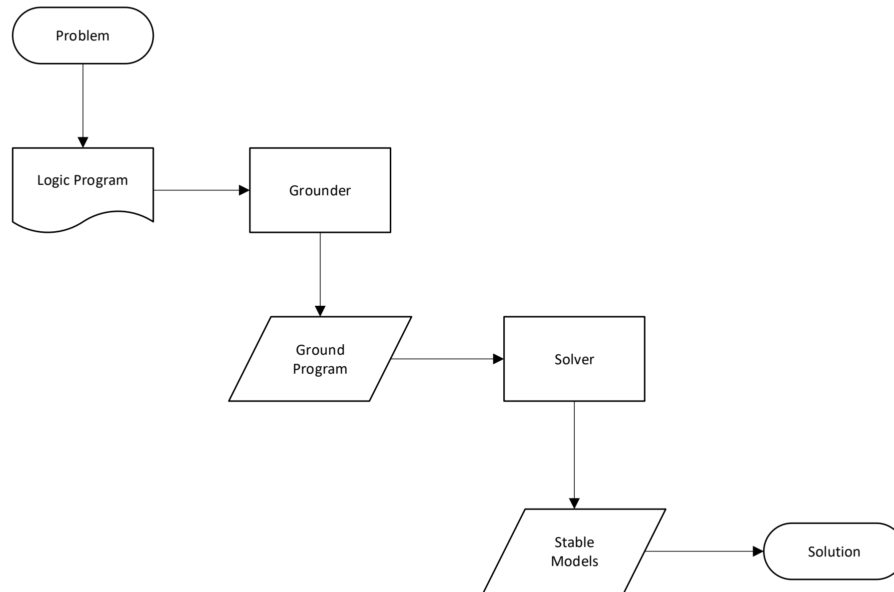

Figure 3.1: The ASP traditional computation

.

in the worst case [39]. In the case of hard problems, this check can be carried out by translating the program into a SAT formula and checking whether it is unsatisfiable.

3.2 Systems

The two first stable grounders were *lparse* [93] and the *DLV* instantiator. They accept different classes of programs, and follow different strategies for the computation. The first binds non-global variables by domain predicates, to enforce ω-restrictedness [93], and instantiates a rule r scanning the extensions of the domain predicates occurring in the body of r, generating ground instances accordingly. On the other hand, the only restriction on *DLV* input is safety. Furthermore, the *DLV* instantiation strategy is based on semi-naive database techniques [97] for avoiding duplication, and domains are built dynamically. Over

the years, a new efficient grounder has been released, namely *gringo* [54]. The first versions accepted only domain restricted programs with an extended notion of domain literal in order to support λ-restrictedness [59]; starting form version 3.0, *gringo* removed domain restrictions and instead requires programs to be safe as in *DLV* and evaluates them by relying on semi-naive techniques as well. Another grounder system is *I-DLV*: the new intelligent instantiator of *DLV*. The new grounder has been redesigned and re-engineered with the aim of building a renovated ASP grounder for improving the performance and native support the ASP-Core-2 standard language [17]. *I-DLV* is more than an ASP grounder, also resulting in a complete and efficient deductive database system. Moreover, it has been extended with a number of optimization techniques that have been explicitly designed by contextualizing it in the setting of an ASP grounder [23].

ASP solvers can be classified into *based on translation* and *native* according to the evaluation strategies employed. Solvers based on translation, rewrite an ASP program to other formula and then use specific solvers as black box, while native solvers use explicit algorithms and data structures for dealing with ASP programs.

ASSAT [76] was one of the first translation based solvers that rewrite an ASP program into propositional formulas and then call an external SAT solver. Comparable techniques were also adopted by *CMODELS*[6] and more recently by the *lp2sat* [67] family of solvers.

Among the first native solver, we mention *DLV* [71] and *SMODELS* [81]. *DLV* implement a backtracking technique with look-head heuristic supporting cautious and brave reasoning. *SMODELS* implements a DPLL-like algorithm based on source pointers as introduced in [92]. The system supports normal logic programs, while disjunctive programs are supported by its extension called *GNT* [68]. However, more recent ASP native solvers are *clasp* [52] and *wasp* [3] systems. Both systems use source pointers, backjumping, learning, restarts, and look-back heuristics. Nevertheless, the two systems differ on data structures and input simplification. Moreover, *wasp* system have an enhancement version of the algorithm of *DLV* concerning the computation of unfounded set. Instead, *clasp* is based on nogoods [55] for the unfounded set check.

Among the monolithic ASP systems that incorporate both grounder and solver systems we mention *clingo* [53] and *DLV* [72]. *clingo* is the union of *gringo* and

clasp systems for the grounding and solving modules. Furthermore, after version 4.0, the system is able to exercise a form of control over the computational tasks, with the main purpose of supporting dynamic and incremental reasoning, thanks to the integration of *gringo* and *clasp* in a single system.

DLV has been one of the first robust and reliable ASP systems, and its project dates back a few years after the first definition of answer set semantics [61, 62]. Moreover, *DLV2* combines *I-DLV* grounder, with the well-assessed solver *wasp*, allowing annotations and directives that customize heuristics of the system and extend its solving capabilities, as will be seen see in Chapter 13.

Chapter 4

ASP at Work: Achievements and Challenges

Thanks to the expressive language and the availability of diverse efficient systems, Answer Set Programming has recently gained popularity and has been applied fruitfully to a wide range of domains, both in academia and in industry.

In this chapter, we introduce the use of ASP as a tool for knowledge representation and reasoning and show how its fully declarative nature allows us to encode a large number of problems using straightforward and elegant logic programs. After that, we present with a brief overview and a proper reference some of these ASP applications, in particular, in Artificial Intelligence, Robotics, Bioinformatics, and industrial applications, also discussing current challenges raised by these applications that ASP has to overcome.

4.1 Knowledge Representation and Reasoning

In this section we introduce the use of ASP as a tool for knowledge representation and reasoning. We first illustrate how to encode a classical problem of the deductive database application; next we present a general ASP programming methodology, and then test it on different computationally hard problems.

4.1.1 Deductive Database Applications

In the following, we show how to encode a classical Deductive Database problems *Reachability* and *Same Generation* via a straightforward and elegant logic program.

Reachability The problem amounts to computing all pairs of reachable nodes in a graph G determining the transitive closure of the relation storing the edges. Then given a directed graph $G = (V,E)$, we shall compute all pairs of nodes $(v1,v2) \in V \times V$ such that $v2$ is reachable from $v1$ through a non-empty sequence of arcs in E.

In the following ASP encoding, the relation $edge(X,Y)$ represents E, where a fact $edge(v1,v2)$ means that G contains an arc from $v1$ to $v2$, i.e., $(v1,v2) \in E$; the set of nodes V is not explicitly represented, since the facts implicitly describe the nodes. Hence, the following program computes a relation $reachable(X,Y)$ containing all facts $reachable(v1,v2)$ such that $v1$ is reachable from $v2$ through the arcs of the input graph G:

$$reachable(X,Y) \text{ :- } edge(X,Y).$$
$$reachable(X,Y) \text{ :- } edge(X,Z),\ reachable(Z,Y).$$

Same Generation Given a parent-child relationship, represented by acyclic directed graph, we shall find all pairs of persons belonging to the same generation. Two persons are of the same generation if they are either siblings, or children of two persons of the same generation. If input is encoded by a relation $parent(X,Y)$, where a fact $parent(a,b)$ states that a is a parent of b, the solution can be encoded by the following program, which computes a relation $same_generation(X,Y)$ containing all facts such that X is of the same generation as Y :

$$same_generation(X,Y)$$
$$\text{:- } parent(P,X),\ parent(P,Y).$$

$$same_generation(X,Y) \text{ :- } parent(P1,X),$$
$$parent(P2,Y),\ same_generation(P1,P2).$$

4.1.2 The GCO Declarative Programming Methodology

The "Guess&Check" (*GC*) paradigm [37] is one of the most common ASP programming methodologies. A *GC* program features 2 modules:

- **Guessing Part**, that defines the search space (usually by means of disjunctive rules);
- **Checking Part**, that checks solution admissibility (usually by means of integrity constraints).

When dealing with optimization problems, the methodology can be further extended to match a "Guess/Check/Optimize"[14] (*GCO*) paradigm with a third module:

- **Optimizing Part** (optional), that specifies preference criteria usually by means of *weak constraints*.

We introduce here the application of *GCO* paradigm on a classical computationally hard problems.

3-COL The problem 3-colorability (3-COL) is a classical NP-complete problem, and consists of the assignment of three colors to the nodes of a graph in such a way that adjacent nodes always have different colors.

The following program computes the admissible ways of coloring the provided graph G given a set of facts F: *node* and *arc*, that represent, respectively, the nodes and the arcs of G.

$$\begin{array}{ll} r_1: & color(X,r) \mid color(X,y) \mid color(X,g) \text{:-} node(X). \\ r_2: & \text{:-} arc(X,Y),\ color(X,C),\ color(Y,C). \end{array}$$

Rule r_1 (*guess*) states that every node of the graph must be colored as **r**ed or **y**ellow or **g**reen indeed r_2 (*check*) forbids the assignment of the same color to any couple of adjacent nodes. The minimality of answer sets guarantees that every node is assigned only one color. Thus, there is a unique correspondence between the solutions of the 3-coloring problem and the answer sets of $F \cup \{r_1, r_2\}$ and therefore the graph represented by F is 3-colorable if and only if $F \cup \{r_1, r_2\}$ has some answer set.

EXAM SCHEDULING Consider the problem of scheduling the exams for several universities and each course should be assigned exactly in one of these three time slots: t_1, t_2, t_3. Specific instance of the problem are provided with a set of facts F specifying the exams to be scheduled. The predicate *exam* has four arguments representing, respectively, the identifier of the exam, the professor who is responsible for the exam, the curriculum to which the exam belongs, and the year in which the exam has to be allocated in the curriculum.

Several exams can be assigned to the same time slot but the same professor in the same time slot cannot run two different exams. Furthermore, exams of the same curriculum should be assigned to different time slots and if it is not possible all exams of a curriculum C should minimize first of all the overlap between exams of the same year of C and afterward between exams of different years of C.

The problem can be encoded by the following program P:

$$r_1: \ assign(Id,t_1) \mid assign(Id,t_2) \mid assign(Id,t_3) :- exam(Id,P,C,Y).$$

$$r_2: \ :- assign(Id,T),\ assign(Id',T),\\ Id \neq Id',\ exam(Id,P,C,Y),\ exam(Id',P,C',Y').$$

$$r_3: \ :\sim assign(Id,T),\ assign(Id',T),\\ exam(Id,P,C,Y),\ exam(Id',P',C,Y),\ Id \neq Id'.[1@2]$$

$$r_4: \ :\sim assign(Id,T),\ assign(Id',T),\\ exam(Id,P,C,Y),\ exam(Id',P',C,Y'),\ Y \neq Y'.[1@1]$$

The guessing part has a single disjunctive rule (r_1) defining the search space. It is evident that the $AS(r_1 \cup F)$ are the possible assignments of exams to time slots.

The checking part consists of one constraint (r_2), discarding the assignments of the same time slot to two exams of the same professor and the $AS(r_1 \cup r_2 \cup F)$ correspond precisely to the admissible solutions.

Finally, the optimization part consists of two weak constraints (r_3, r_4). Both weak constraints state that exams of the same curriculum should, if possible, not

be assigned to the same time slot. However r_3 which has higher priority (level 2), states this desire for the exams of the curriculum of the same year, while r_4, which has lower priority (level 1), states it for the exams of the curriculum of different years. Thus, the $AS(P) \cup F$ correspond precisely to the desired schedules.

SUDOKU Sudoku is a logic-based combinatorial puzzle. The objective of the game is to fill a 9×9 grid with numbers so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called "blocks") contains all of the digits from 1 to 9 and initially the puzzle provides a partially completed grid.

The set of facts F given, representing the schema to be completed, are:

- a binary predicate *pos* encodes possible position coordinates;
- *symbol* is a unary predicate encoding possible symbols (numbers);
- facts of the form *sameblock(x1,y1,x2,y2)* state that two positions $(x1,y1)$ and $(x2,y2)$ are within the same block;
- facts of the form *cell(x,y,n)* represent that a position (x,y) is filled with symbol n.

The following ASP program computes the solutions of the Sudoku schema at hand.

$$
\begin{array}{ll}
r_1: & cell(X,Y,N) \mid nocell(X,Y,N) \text{ :- } pos(X), \\
& \quad pos(Y),\ symbol(N). \\
r_2: & \text{:- } cell(X,Y,N),\ cell(X,Y,N1),\ N1 \neq N. \\
r_3: & assigned(X,Y) \text{ :- } cell(X,Y,N). \\
r_4: & \text{:- } pos(X),\ pos(Y),\ \texttt{not}\ assigned(X,Y). \\
\\
r_5: & \text{:- } cell(X,Y1,Z),\ cell(X,Y2,Z),\ Y1 \neq Y2. \\
r_6: & \text{:- } cell(X1,Y,Z),\ cell(X2,Y,Z),\ X1 \neq X2.
\end{array}
$$

$$r_7: \ :- \ cell(X1,Y1,Z),\ cell(X2,Y2,Z),\ Y1 \neq Y2,$$
$$sameblock(X1,Y1,X2,Y2).$$
$$r_8: \ :- \ cell(X1,Y1,Z),\ cell(X2,Y2,Z),\ X1 \neq X2,$$
$$sameblock(X1,Y1,X2,Y2).$$

Rules $r_1 - r_4$ ensure that each cell is filled with exactly one number (*symbol*), guessing the value for each cell. Meanwhile, rules r_5 and r_6 check that a number does not occur more than once in the same row or column and rules r_7 and r_8 ensure that in the same block two different cells do not have the same number.

4.2 Applications and Challenges

The main advantage of ASP is the high expressive power of its language allowing the user to represent problems that belong to the complexity class Σ_2^P ,i.e. NP^{NP}. Moreover, ASP is fully declarative and the ASP program is straightforward, concise and elegant as shown in Section 4.1. Due to the continuous improvements of ASP solvers and the language extensions, as shown in Chapter 3.2, ASP has been used in many different domains [45, 11]. It has been used in the areas of Artificial Intelligence, Robotics, Bioinformatics, and also for industrial applications.

In the Robotics field, ASP has been used in various applications, such as assembly planning, mobile manipulation, geometric rearrangement, multi-robot path finding, coordination, and planning. For instance [44] present an application of ASP to housekeeping robotics for planning actions of multiple PR2 robots. In another work [46], the authors use ASP to study the problem of finding optimal plans for multiple teams of robots through a mediator, to manufacture a given number of orders within a given time. In [102] the authors use ASP to describe objects and relations between them and to improve the localization of objects in an indoor environment. The authors have also implemented it in a wheeled robot navigating in an office building.

In order to meet requirements of different application domains, ASP has been extended, to support higher-order atoms as well as external atoms. These extensions of ASP, called *HEX-Programs* [41], allow one to embed external sources

of computation in a logic program, as will be shown in Chapter 9. Thus, HEX-programs are useful for various tasks, including meta-reasoning, data type manipulations, and reasoning on top of Description Logics. As an example, in [19] they propose an AI agent for the computer game *Angry Birds* [1]. The agent is based on HEX-programs and guesses possible targets and estimates the damage for each object. However, this estimation requires physics and external atoms are used to interface with a physics simulator.

ASP has been applied in several biology and bioinformatics applications, providing a declarative problem solving framework for combinatorial search problems and knowledge-intensive reasoning tasks. For instance, in [96] the authors propose an action language based framework for hypothesis formation for signaling networks. They model a biological signaling network as an action description in ASP and show that the hypothesis formation problem can be translated into an abduction problem. This translation facilitates the complexity analysis and illustrates the applicability with an example of hypothesis formation in the signaling network of the p53 protein. In [60] they introduce an approach to detecting inconsistencies in large biological networks by using ASP. In particular, the authors propose a methodology to provide explanations for inconsistencies by determining minimal representations of conflicts and they compare the yeast regulatory network with the genetic profile data of SNF2 knock-outs and find the data to be inconsistent with the network. Moreover, ASP has been used to study a variation of the protein structure prediction problem [34]. They present experimental comparisons between the declarative encodings of various computationally hard problems in both ASP and Constraint Logic Programming over finite domains (CLP(FD)) and investigate how the solvers in the two domains respond to different problems. A related line of research is the 2D HP-protein structure prediction problem, in which, given a protein sequence, the goal is to find a folding of the sequence in the 2D square lattice space such that most HH pairs are neighboring.

The availability of efficient ASP solvers has facilitated the implementation of many advanced ASP applications, not only in academia but also in the industry. As an example, in [89] a system based on ASP has been developed to automatically produce an optimal allocation of the available personnel of the

[1] https://www.angrybirds.com

international port of Gioia Tauro. The system can build new teams satisfying several constraints or complete the allocation automatically when the roles of some key employees are fixed manually in a pure declarative manner, allowing for the fine tuning of both problem specifications and ASP programs while interacting with the stakeholders. Furthermore, ASP has been successfully applied in applications in the tourism industry. For instance, ASP-based application has been integrated into an E-tourism portal that implements a smart advisor for the selection of the most promising offers for customers of a travel agency [66]. In this application ASP has been used for developing different search modules with the aim of selecting the holiday packages that best fit the customer's needs. The system enhances the business of the travel agency by reducing the time needed to choose and sell the holiday offers, and suggests the offers which match the user profile, thereby increasing the level of customer satisfaction. ASP has also been successfully employed in in the health field. In particular, a multi-source data cleaning system, whose goal is to detect and automatically correct both syntactic and semantic anomalies in medical knowledge bases [73].The system has been applied to clean up the data stored in the tumor registries of the Calabria Region, integrating information from several neighborhood healthcare centers. Notably, thanks to ASP a simplified specification of the logic of the data cleaning task can be obtained.

We discussed above several successful applications of ASP, in different domains. However, such widened application of the formalism to practical domains addressed some important challenges concerning integration, interoperability and scalability. Therefore, in the below paragraphs we describe these three challenges raised by real ASP applications.

Integration. For instance, from a technical point of view, a large number of software applications is developed by using object-oriented languages and the need for integrating such type of applications with logic-based systems has arisen. Moreover, the worldwide commercial, consumer and industrial scenario has significantly changed recent years; smartphones or "smart"/wearable are constantly gaining popularity as their computational power and features increase. Furthermore, as shown in Section 3.2, the availability of a number of robust and efficient systems is growing. In this context, there is still a lack of tools for the

integration of ASP in external systems for generic applications, which ease the development for different platforms and ASP reasoners. For example, in the context of computer games, the use of ASP with the purpose of building an AI agent can be intuitive and simple due to the fully declarative nature of ASP [48]. These applications rely on a *AI module* based on ASP, which provides the correct action to be performed during the game. However, the *core module*, i.e. the layer that manages the application, is implemented using object-oriented languages and, without external tools for easing the integration of ASP in an external system, a user has to deal with the communications and the translation of input/output with the specific ASP reasoner. Furthermore, the growing popularity of smartphone devices increases the development of smartphone applications, including games app. In this scenario, embedding the capabilities of ASP reasoner that can natively run on mobile devices gives rise to the potential use of ASP also for these platforms.

Interoperability. Extending the ASP language with the aim of easing the interoperability and integration with external systems are other important challenges addressed by ASP. For instance, *zLog* system is a platform for customer profiling for phone call routing based on ASP [2]. The key idea is to classify customer profiles and try to anticipate their actual needs for creating a personalised experience of customer care service. Then, the operator of the call-center defines the customer categories. Once a new category has been defined, zLog automatically generates an ASP program which provides its logical encoding and can be executed by *DLV*. The definition of customer categories is carried out via a user-friendly visual interface that allows one to specify and modify categories. When a customer calls the call-center, he/she is automatically assigned to a category (based on his/her profile) and then routed to an appropriate human operator or automatic responder. The zLog platform has been deployed in a production system handling Telecom Italia call-centers, and it is in actual use. Every day, over one million telephone calls for diagnostic services reach the call-centers of Telecom Italia. To manage all the calls, the system has been parallelised using a multiprocessor architecture to cope with the high workload. In this type of applications, the performance of ASP system is crucial and, without using parallel

[2]http://www.exeura.eu/en/solution/customer-profiling

architecture simplifies the development of the entire system. Moreover, the zLog platform executes *DLV* over a database system implementing an ad-hoc module for the connection between *DLV* and the relational database. In this context, easing the interoperability and integration of ASP with external system simplifies the use of the language, especially in the application in which data are stored in a relational database. Extension of the ASP language that allows the user to import from or export data to a different source of knowledge, not only obviates the need for the development of specific middleware to handle connections with those sources but also grants additional performance due to the integration into the ASP solver.

Scalability. Another challenge is the grounding blow-up of the program that makes the usage of plain ASP unviable. For instance a recent application of ASP to abduction in Natural Language Understanding [91] shows that plain ASP solvers are not effective. In particular, this work studied the abduction in First Order Horn logic theories where all atoms can be abduced, i.e. aims to find a set of explanatory atoms that make a set of goal atoms true with respect to a background theory. Also, preferred solutions among possible abductive explanations have been chosen concerning three objective functions: cardinality minimality, coherence, and weighted abduction. Then, they represent this reasoning problem in ASP, in order to obtain a flexible framework for experimenting with global constraints and objective functions, and to test the boundaries of what is possible with ASP. However, realizing this problem in ASP is challenging as it requires value invention and equivalence between certain constants and, in particular, it has been shown in [91] that the grounding of all constraints makes the solving step too hard for state-of-the-art solvers. Hence, the evaluation of these problems through identifying the set of rules that cause the blow-up and instantiates lazily during the solving is another important challenge that requires a coupling of grounding and solving systems in a *monolitic* ASP system.

Part II

Integrating ASP Systems into external applications

Chapter 5

Improving ASP-based Software Development: State of the Art and Motivations

The theoretical properties of ASP, after more than twenty years of research, are well understood and the solving technology, as evidenced by the availability of many robust and efficient systems [26], is mature for practical applications, as we show in Chapter 4. Notably, ASP teaching is growing in universities worldwide, and, significantly, is switching its focus from narrowly theoretical to more practical aspects.

However, a significant number of software applications is being developed by using object-oriented languages, hence the need for integrating this type of application with logic-based systems. In recent years embedding ASP reasoning modules into external systems has been investigated in the literature. For instance, the *DLVJava Wrapper* [88] is a library implemented in Java, that "wraps" the *DLV* system inside an external application acting as an interface between Java programs and the *DLV* system and handle input and output of *DLV* by using Java objects. Another framework for integrating *DLV* in an external system is *JDLV*, that acts as a "wrapper" as the *Java Wrapper*, but also provides an advanced platform for integrating Java with DLV using JPA annotations for defining how Java classes map to ASP program, similarly to ORM frameworks.

Work in [87] introduces a formal language for defining mappings of in-

put/output of an ASP program in the form of objects intended to be handled by various programming languages. A Python library, namely *PY-ASPIO*, is in charge of interfacing the statements embedded in the ASP program with the selected object-oriented language guided by custom annotations on the programming language code.

Concerning generic logic-embedding tools Tweety [94] is an open source framework for experimenting with logical aspects of artificial intelligence. It consists of a set of Java libraries that allow a user to use different knowledge representation systems supporting different logic formalisms, ranging from classical logics, over logic programming and computational models for argumentation, to probabilistic modeling approaches, including ASP.

Despite the various frameworks described above, none of them is able to overcome the challenge introduced in Section 4.2 and there is still a lack of tools for taking advantage of the knowledge representation capabilities of ASP in the widest range of contexts of the mobile setting. Moreover, most of them are specifically bound to a single or specific solver, like Java Wrapper or JDLV, and the connection between the logic-based aspects and the object-oriented are very tight. The benefits of a flexible translation between Java Object and ASP language gives developers the possibility to work separately on ASP-based modules and on applications that make use of them, and keep things simple when developing complex applications. Let us think, for instance, of a scenario in which different figures are involved, such as Android/Java developers and KRR experts. Both figures can take advantage of the fact that the knowledge base and the reasoning modules can be designed and developed independently from the rest of the Java-based application.

In the following, we present EMBASP a framework for the integration of ASP in external systems for generic applications. The work starts with the goal to ease the development of mobile applications natively using logic-based reasoners and to our knowledge, it represented the first attempt reported in the literature for ASP [20, 21]. Nevertheless, the framework has been extended [47] for fostering the use of ASP within real-world and industrial contexts, where it gained popularity; the framework has been made more abstract, and independent from the running platform.

Chapter 6

A Novel General Approach for Embedding Declarative Reasoning Modules: EMBASP

In this chapter we present EMBASP: a framework for the integration of ASP in external systems for generic applications. It consists of an abstract architecture, implementable in a programming language of choice. The work starts with the goal to allow the usage of the ASP reasoners on mobile platforms and it represented the first attempt reported in the literature [20, 21]. Nevertheless, the framework has been extended [47] to easily allows proper specializations to different platforms and ASP reasoners. In the following section, we present the EMBASP architecture and then, propose a Java implementation.

The general architecture of EMBASP is depicted in Figure 6.1. It defines an abstract framework to be implemented in some object-oriented programming language. Due to its abstract nature, Figure 6.1 just reports the general dependencies of the main modules. Moreover, each concrete implementation might require specific dependencies from the inner components of each module, as can be observed in Figure 6.2, which is related to a concrete Java implementation and will be discussed hereafter.

The aim of the framework design is intended to ease and guide the generation of suitable libraries for the use of specific solvers on particular platforms; resulting applications manage ASP solvers as “black boxes”. Therefore, the re-

sulting libraries can be used in order to effectively embed ASP reasoning modules within any kind of application developed for the targeted platforms, handled by the ASP system(s) at hand. In addition, as already discussed above, the framework is meant to give developers the possibility to work separately on ASP-based modules and on the applications that makes use of them, thus keeping things simple when developing complex applications. Additional specific advantages/disadvantages might arise depending on the programming language chosen for deploying libraries and on the target platform; special features, in fact, can make implementation, and in turn extensions and usage, easier or harder, to varying degrees.

6.1 Abstract Architecture

The framework architecture has been designed by means of four modules: *Core*, *Platforms*, *Languages*, and *Systems*, whose indented behavior is described next.

Core Module

The *Core* module defines the basic components of the *Framework*.

The *Handler* component mediates the communication between the *Framework* and the user who can provide it with the input program(s) via the component *Input Program*, along with any desired solver's option(s) via the component *Option Descriptor*. A *Service* component manages the chosen solver executions.

Two different execution modes can be made available: synchronous or asynchronous. While in the synchronous mode any call to the execution of the solver is *blocking* (i.e., the caller waits until the reasoning task is completed), in asynchronous mode the call is non-blocking: a *Callback* component notifies the caller once the reasoning task is completed. The result of the execution (i.e., the output of the logic system) is handled by the *Output* component, in both modes.

Platforms Module

The *Platforms* module contains what is platform-dependent; in particular, the *Handler* and *Service* components from the *Core* module that should be adapted according to the platform at hand, due to their role in launching solvers.

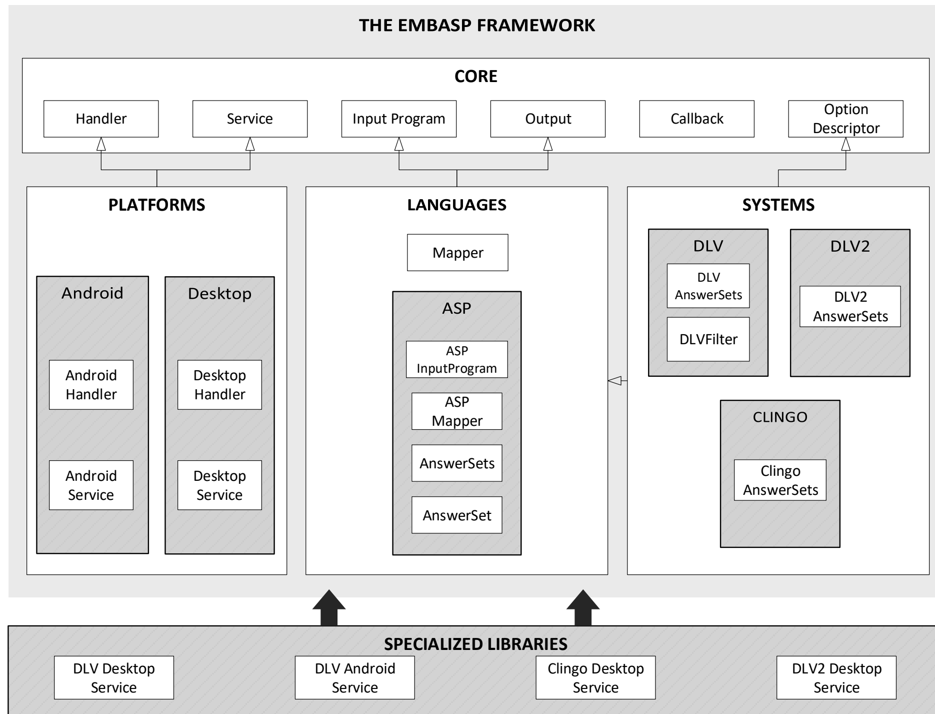

Figure 6.1: A visual overview of EMBASP: the abstract *Framework* with the general dependencies of the main modules represented by empty arrows, and the proposed *Specialized Libraries*. Darker blocks are related to the proposed specializations.

Languages Module

The *Languages* module defines specific facilities for each supported logic language.

The generic *Mapper* component is conceived as a utility for managing input and output via objects, if the programming language at hand permits it.

The sub-module ASP comprises components such as *ASPInputProgram* that adapts *Input Program* to the ASP case, while *AnswerSet* and *AnswerSets* represent the *Output* for ASP. In details, *AnswerSets* represents the output of an ASP system, i.e. a set of Answer Set, and *AnswerSet* describe a single Answer Set. Moreover the *ASPMapper* allow management of ASP input facts and answer sets via objects.

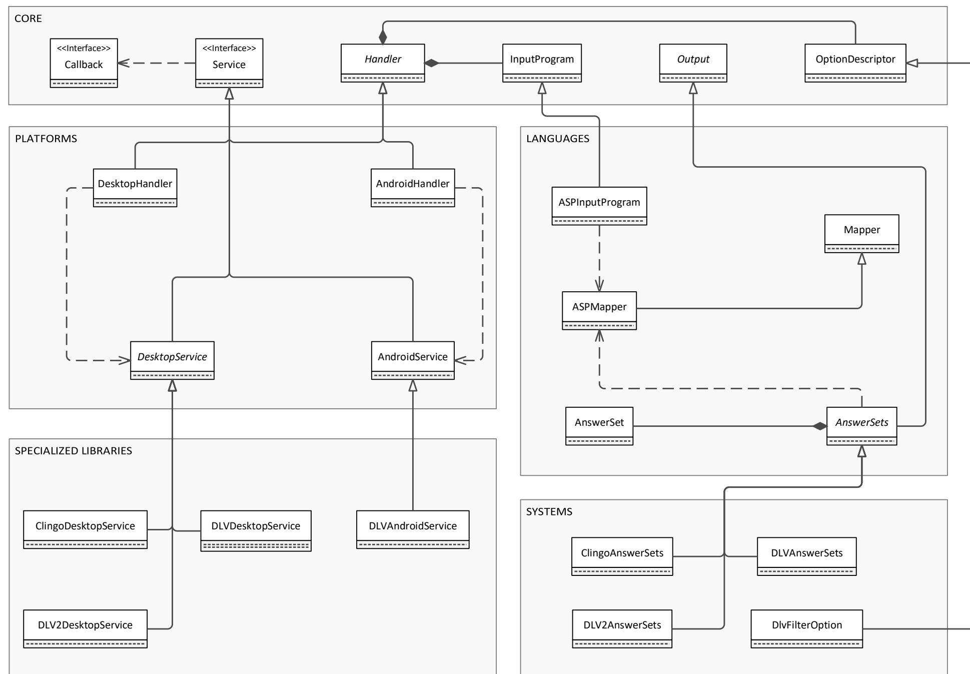

Figure 6.2: Simplified class diagram of the provided Java implementation of EMBASP.

Systems Module

The *Systems* module defines what is system-dependent. In particular, the *Input Program*, *Output* and *Option Descriptor* components from the *Core* module should be adapted in order to effectively interact with the solver at hand.

6.2 Implementing EMBASP

In the following, we propose a Java[1] implementation of the architecture described above. Besides the implementation of the framework itself, proper specialized libraries have been implemented.

In particular we implemented the main modules by means of classes or interfaces, and four specialized libraries that permit the use of *DLV* (ver. 12-17-2011)

[1]https://www.oracle.com/java

on Android[2] and the use of *clingo* on desktop. In addition, the recently released *DLV2*, a completely renewed version of *DLV*, has been embedded which combines the ASP grounder *I-DLV* [23] and the *wasp* solver [1].

Figure 6.2 provides details about classes and interfaces of the implementation. In order to better outline correspondences with the abstract architecture of Figure 6.1, classes belonging to a module have been grouped together. The complete UML class diagram is available online at [43].

Core module implementation

Each component in the *Core* module has been implemented by means of an homonymous class or interface.

In particular, the `Handler` class collects `InputProgram` and `OptionDescriptor` objects communicated by the user.

As far as the asynchronous mode is concerned, the class `Service` depends on the interface `Callback`, since once the reasoning service has been terminated, the result of the computation is returned back via a class `Callback`.

Platforms module implementation

In order to support a new platform, the *Handler* and *Service* components must be adapted.

As for the Android platform, we developed an `AndroidHandler`: this handles the execution of an `AndroidService`, which provides facilities for running the execution of an ASP reasoner on the Android platform.

Similarly, for the desktop platform we developed a `DesktopHandler` and a `DesktopService`, which generalizes the usage of an ASP reasoner on the desktop platform, allowing both synchronous and asynchronous execution modes.

While both synchronous and asynchronous modes are provided in the desktop setting, we stick to the asynchronous one on Android: indeed, mobile users are familiar with apps featuring constantly reactive graphic interfaces, and according to this native asynchronous execution policy, we want to discourage a blocking execution.

[2]http://developer.android.com

Languages module implementation

This module includes specific classes for the management of input and output to ASP solvers.

The *Mapper* component of the *Languages* module is implemented via a `Mapper` class, that translate input and output into Java objects. Such translations are guided by Java Annotations[3], a form of metadata that mark Java code and provide information that is not part of the program itself: they have no direct effect on the operation of the code they annotate.

In this contex, we make use of such a feature so that it is possible to translate input and output into strings and vice-versa via two custom annotations, defined according to the following syntax:

- *@Id (string_name)*: the target must be a class;
- *@Param (integer_position)*: the target must be a field of a class annotated via @Id.

In particular, for ASP *@Id* represents the predicate name of a ground atom that can appear as input (fact) or output (within the returned answer set(s)), while fields annotated with *@Param* define the terms and their positions in such atoms.

By means of the Java Reflection mechanisms, annotations are examined at runtime, and taken into account to define the translation properly.

The user has to register all annotated classes to the `Mapper`, although the classes involved in input translation are automatically detected. If the classes intended to translate are not annotated or not correctly annotated, an exception is raised. Other problems might occur if, once that the solver output is returned, the user asks for a translation into objects of non annotated classes: in this case a warning is raised and the request is ignored.

Notably, this feature designed to allow developers the possibility to work separately on the logic-based modules and on the Java side. The mapper acts as a middle-ware that enables the communication among the modules, and eases the burden of developers by means of an explicit, ready-made mapping between Java objects and the logic modules.

Further insights on this feature are illustrated in the next chapter by means of some EMBASP use cases.

[3]https://docs.oracle.com/javase/tutorial/java/annotations/

Moreover, the `ASPMapper` class, which acts like a translator, provides the means for a two-way translation between strings recognizable by the ASP solver at hand and Java objects directly employable within the application. Its aim is to translate ASP input and output from and to objects: thus we find a dependency between `ASPInputProgram` and `AnswerSets` classes. `ASPInputProgram` extends `InputProgram` to the ASP case. In addition, since the "result" of an ASP solver execution consists of answer sets, the *Output* class has been extended by the `AnswerSets` class composed of a set of `AnswerSet` objects.

Systems Module Implementation

The classes `DLVAnswerSets`, `ClingoAnswerSets` and `DLV2AnswerSets` implement specific extensions of the `AnswerSets` class, in charge of manipulating the output of the respective solvers.

Moreover, this module contain classes extending `OptionDescriptor` to implement specific options of the solver at hand. For instance, the class `DLVFilter` is a utility class representing the filter option of DLV.

6.3 Specializing the Framework

The implemented library derived from EMBASP allows the embedding of ASP reasoning module in external systems for generic applications. These computations are handled by *DLV* from within Android and Desktop applications, and by *DLV2* and *clingo* inside standalone Desktop applications.

The classes `DLVDesktopService`, `ClingoDesktopService` and `DLV2DesktopService` are in charge of offering ASP reasoning on Desktop platform while `DLVAndroidService` offer the same support on Android.

`DLVAndroidService` is a specific version of `AndroidService` for the execution of *DLV* on Android. *DLV* was not available for Android because it is natively implemented in C++, while the standard development process on Android is based on Java. To this end, *DLV* has been purpose rebuilt using the NDK (Native Development Kit)[4], and linked to the Java code using the JNI (Java Native

[4]https://developer.android.com/tools/sdk/ndk

Interface)[5]. This grants access to the APIs provided by the Android NDK, and in turn accedes to the *DLV* exposed functionalities directly from the Java code of an Android application. The previously described procedure for the execution of *DLV* on Android grants the porting of any C++ code on Android platform and therefore using NDK and JNI is possible also the porting of *clingo* and *DLV2* systems.

`DLVDesktopService`, `ClingoDesktopService` and `DLV2DesktopService` are specific versions tailored for the *DLV*, *clingo* and *DLV2* reasoners, respectively, on the desktop platform; they extend the `DesktopService` with functions needed to invoke the embedded solver(s).

[5]http://docs.oracle.com/javase/8/docs/technotes/guides/jni

Chapter 7

Embedding ASP Programs

In this chapter we show how to employ the Java libraries generated via EMBASP for making use of ASP within an actual Java application. Then, in the following, we shall describe the development of an Android application based on ASP, for solving Sudoku puzzles. Notably, although the following example application makes use of one formalism via one solver, EMBASP also allows the user to deploy applications that rely on multiple logic formalisms and multiple solvers at once.

Suppose that a user designed (or has been given) a proper logic ASP program *P* to solve a Sudoku puzzle, and also that she has been provided with an initial schema to be solved. For instance, *P* can correspond to the logic program presented in Section 4.1, so that, coupled with a set of facts *F* representing the given initial schema, this allows the user to obtain the only possible solution (i.e., a single answer set). By means of the annotation-guided mapping, the initial schema can be expressed in forms of Java objects. To this extent, we define the class `Cell`, aimed at representing a single cell of the Sudoku schema, as follows:

```
1 @Id("cell")
2 public class Cell {
3
4   @Param(0)
5   private int row;
6
7   @Param(1)
8   private int column;
9
10
```

```
11   @Param(2)
12   private int value;
13
14      public Cell() {
15          row = column = value = 0;
16     }
17
18     [...]
19
20
21 }
```

It is worth noting how the class has been annotated by two custom annotations, as introduced above. Thanks to these annotations the `ASPMapper` will be able to map `Cell` objects into strings recognizable from the ASP solver as logic facts of the form $cell(Row, Column, Value)$. Also, the mapped class must be a JavaBean class and in particular follow this convention:

- The class must have a public default constructor. This allows instantiation of the objects.
- The class properties must be accessible using *getter*, *setter* methods . This allows easy automated inspection and updating the field of the objects.

At this point, we can create an `Android Activity Component` [1], and start deploying our Sudoku application:

```
1 public class MainActivity extends AppCompatActivity {
2
3   [...]
4   private Handler handler;
5
6   @Override
7   protected void onCreate(Bundle bundle) {
8     handler = new AndroidHandler(getApplicationContext(),
9         DLVAndroidService.class);
10    [...]
11  }
12
13  public void onClick(final View view){
14    [...]
15    startReasoning();
16  }
17
18  public void startReasoning() {
```

[1] https://developer.android.com/reference/android/app/Activity.html

```
19      InputProgram inputProgram =
20        new ASPInputProgram();
21      for ( int i = 0; i < 9; i++){
22        for ( int j = 0; j < 9; j++)
23          try {
24            if(sudokuMatrix[ i ] [ j ]!=0) {
25              inputProgram.addObjectInput(
26                new Cell(i, j, sudokuMatrix[i][j]));
27            }
28          } catch (Exception e) {
29            // Handle Exception
30          }
31      }
32      handler.addProgram(inputProgram);
33
34      String sudokuEncoding =
35        getEncodingFromResources();
36      handler.addProgram(new
37        ASPInputProgram(sudokuEncoding));
38
39      Callback callback = new MyCallback();
40      handler.startAsync(callback);
41    }
42 }
```

The previous class contains a `Handler` instance, that is initialized when the `Activity` is created as an `AndroidHandler`. Also, required parameters include the `Android Context` (an Android utility, needed to start an Android Service Component) and the type of `AndroidService` to use in our case, a `DLVAndroidService`. Besides, the initial Sudoku schema is represented by a matrix of integer 9×9 where position (i, j) contains the value of cell (i, j) in the initial schema; cells initially empty are represented by positions containing zero. The method `startReasoning` is in charge of actually managing the reasoning: in our case, it is invoked in response to a "click" event that is generated when the user asks for the solution. Lines 19–32 create an `InputProgram` object that first is filled with `Cell` objects, representing the initial values of the schema, and then served to the handler. Using the utility function $getEncodingFromResources()$ the ASP program is loaded from the *Android Resources folder* and lines 34–37 provide it to the handler. At this point, the reasoning process can start, and a callback object is in charge of fetching the output when the ASP system has done (Lines 39–40) (since for Android we provide only the asynchronous execution mode). Once the computation is over, the output can be retrieved directly in the form of Java objects and eventually a possible solution can be displayed.

For instance, in our case an inner class `MyCallback` implements the interface `Callback`:

```
1 private class MyCallback implements Callback {
2
3   @Override
4   public void callback(Output o) {
5     if(!(o instanceof AnswerSets))
6       return;
7     AnswerSets answerSets=(AnswerSets)o;
8         if(answerSets.getAnswersets().isEmpty())
9           return;
10        AnswerSet as = answerSets.getAnswersets().get(0);
11    try {
12      for(Object obj:as.getAtoms()) {
13        Cell cell = (Cell) obj;
14        sudokuMatrix[cell.getRow()]
15          [cell.getColumn()] = cell.getValue();
16      }
17    } catch (Exception e) {
18      // Handle Exception
19    }
20    displaySolution();
21  }
22}
```

The complete code of the previous example is in Appendix B

Notably, the architecture and the design of EMBASP do not affect the performance of the ASP systems. Indeed, we test the performance of the presented program against the simple running of *DLV* via command line, i.e. without using EMBASP, with different initial Sudoku schema. The results show comparable time proving that the two-way translation of Java objects and the call of the ASP system implemented in EMBASP not influence the performance of the ASP-based applications.

7.1 Generalizing the Framework

The implementation illustrated above relies on Java: a very popular, solid and reliable programming language. Besides, the choice of Java was also justified motivated by the intention to foster the use of ASP in new scenarios, in particular in the mobile one and Android is by far the most widespread mobile platform, and its development and deployment models heavily rely on Java. However, the abstract architecture of EMBASP can be made concrete using other object-oriented

programming languages. Most of components in the Java implementation presented herein have been accomplished thanks to features that are typical of any object-oriented language, such as *inheritance* and *polymorphism*. The only exception is represented by the *ASPMapper* component, implemented by means of Java peculiar features, such as *annotations* and *reflection*. In case of other languages that feature similar constructs, such as `C#` [2], the approach can resemble the herein presented Java implementation.

With different languages that lack such features, the mapping mechanism can still be implemented with a simulation via inheritance and polymorphism and applying typical Software Engineering patterns [49]. As an example, one possible implementation can be accomplished with the *Prototype design pattern*, that is well-suited to our purposes, as it allows the user to "specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype" [49]. Indeed, a Python version of the framework is available in [43].

[2]Microsoft Developer Network, MSDN: C# Attributes (https://msdn.microsoft.com/en-us/library/mt653979), C# Reflection (https://msdn.microsoft.com/en-us/library/mt656691)

Chapter 8

EMBASP on the Field: some actual ASP-Based Applications

In this chapter, we describe some ASP-based applications developed in the context of a university course that covers ASP topics, in particular, developed by some of the course attendants, i.e., undergraduate students. The most important aspect is the engagement of university (under)graduate students in ASP capabilities, in order to make them able to take advantage of it when solving problem and designing solutions, in the broadest sense.

In the following, we briefly introduce four Android applications that make use EMBASP framework along with the project links that contains the full code of the apps, where available.

GuessAndCheckers

GuessAndCheckers[1] is a mobile application that works as a helper for users that play "live" games of the (Italian) checkers (i.e., using physical board and pieces). The application runs on Android and helps a player at any time by means of the camera that takes a picture of the board, and infers the information about the current status of the game thanks to OpenCV[2], an open source computer vision and machine learning software. Then, an ASP-based artificial intelligence

[1]https://github.com/vincenzoarieta93/GuessAndCheckers
[2]http://opencv.org

module suggests the next move.

Thanks to EMBASP and the use of ASP, *GuessAndCheckers* features a fully-declarative approach that made it easy to develop and improve several different strategies, also experimenting with many combinations thereof.

DLVEdu

DLVEdu is an educational Android App for children, that integrates well-established mobile technologies, such as voice or drawn text recognition, with the modeling capabilities of ASP. In particular, it can guide the child throughout the learning tasks, by proposing a series of educational games, and developing a personalized educational path. The games are divided into four macro-areas: Logic, Numeric-Mathematical, Memory, and Verbal Language. The usage of ASP allows the application to adapt to the games already played by the user, her learning gap, and the obtained improvements.

The application continuously profiles the user by recording mistakes and successes, and dynamically builds and updates a customized educational path along the different games.

The application features a “Parent Area”, that allows parents to monitor child’s achievements and to express preferences, such as directions regarding access to certain games or educational areas.

Connect4

The popular turn-based *Connect Four* game is played on a vertical 7×6 rectangular board, where two opponents drop their disks with the aim of creating a line of four, either horizontally, vertically, or diagonally. The *Connect4* application allows a user to play the game (also known as *Four-in-a-Row*) against an ASP-based artificial player. Notably, the declarative nature of ASP, its expressive power, and the possibility to compose programs by selecting rules to design and implement different AIs, ranging from the most powerful one, that implements advanced techniques for the perfect play, to the simplest one, that relies on traditional heuristic strategies. Furthermore, by using EMBASP, two different versions of the same app have been built: one for Android, making use of *DLV*, and one for desktop platforms, making use of *clingo*.

DLVfit

The *DLVFit*[3] Android App was the first application making use of the framework; it was conceived as a proof of concept, in order to show the framework features and capabilities. To our knowledge, it is also the first mobile app natively running an ASP solver.

DLVfit is a health app that offers suggestions to the owner of a mobile device the "best" way to achieve fitness goals. The app lets the user express her own goals and preferences in a customizable way along with many composable dimensions, such as calories to burn, time to spend, differentiation over several physical activities and time constraints. Then, it monitors her actual activities throughout the day by means of the Google Activity Recognition APIs [4], a defacto standard on Android, thus relying on these for the accuracy of detection. At any time, the user might ask for a suggestion about a workout plan for the rest of the day, and the reasoning module hence prepares a (set of) workout plans in line with the personal goals and preferences previously expressed.

[3] https://github.com/brainatwork/DLVfit

[4] https://developers.google.com/android/reference/com/google/android/gms/location/ActivityRecognition

Part III

Easing Interoperability of ASP Systems

Chapter 9

Improving Interoperability: State of the Art and Motivations

While the Answer Set Programming has been increasingly employed in many different domains, and also used for the development of industrial-level and enterprise applications, current trends expressed the need for access to external sources.

The availability of external sources of computation has been already recognized as a desirable feature of ASP systems in the literature; indeed, all major ASP systems are endowed with such capabilities to different extents: we mention here the *HEX-programs* supported by *dlvhex* [38] and the support for Python computation by *clingo* [51]. Notably, some forms of external computation and value invention were also supported by the ASP system *DLV* [15, 16].

The embedding of external sources of computation in a logic program is useful for many tasks, as it allows to provide a general interface between high-level reasoning modules and different sources of both computation and data/knowledge. One of the first attempt to embed external computations in ASP was in [27] that introduce a framework aimed at enabling ASP to deal with external sources of computation (*DLV-EX*). The framework includes the explicit possibility of invention of new values from external sources in order to deal in a satisfactory way with infinite domains such as strings or natural numbers, or with such data types need of ad hoc manipulation constructs, which are typically difficult to be encoded and cannot be efficiently evaluated in logic programming.

HEX-atoms [42] have been introduced with the capabilities for higher-order reasoning and interfacing with external sources of computation for interoperability with the Ontology Layer of the Semantic Web. The HEX-semantics was implemented in the *dlvhex* [38] system, and more precisely an *external atoms* have the form:

$$\&p[Y_1,\ldots,Y_n](X_1,\ldots,X_m)$$

where $Y_1,\ldots,Y_n$ are input parameters (which can be either a constant or variable term, or a predicate), $X_1,\ldots,X_m$ are output terms and p is an external predicate name. External atoms are realized as plugins of the reasoner using a programming interface. To this end, a user of an external source basically implements its oracle function.

In an early version of *dlvhex* external atoms were seen as black boxes and it was assumed that the system did not have any information. However, in many applications, a provider has additional knowledge about the behavior of the source, for instance, that the source is monotonic, functional, has a limited domain. Knowing such properties, in *dlvhex* a user can specify a set of *properties* that external sources might have, and allow the system to use more specialised algorithms which are tailored to the particular external sources.

Utilizing scripting languages also the ASP *clingo* [51] enriches the input language with external computations. It allows functions that are evaluated during grounding and the external function syntax look like function terms but are preceded by "@".

However, unlike HEX-atoms, the communication between the ASP system and external scripts is only possible in grounding phases and is not tightly coupled to the model generation. Consequently, external atoms in *dlvhex* offer complete supports for external source of knowledge but are inherently more difficult to evaluate. The difficulty comes especially in nonmonotonic behavior because HEX-atoms can take as input predicate extensions and then a coupling with a model generator is needed for the evaluation of the external function.

The above described systems support external sources of computation using external atoms or functions seen as oracle function by the ASP systems and implemented using an imperative programming language, like Python. External atoms improve interoperability of ASP with external systems, however, even though this approach relying on external atoms perfectly reaches the original

goal, there are some reasons in favour of a "native" support for such features. For instance, as we show in Section 4.2, in scenarios in which a management of high volume of data is needed the use of external atoms is not crucial while accessing standard knowledge sources is vital. Therefore, in the next chapters, we present the advancements in *I-DLV* aiming at easing the integration with external systems: the handling of external computations with explicit calls to Python scripts via external atoms, and interoperability mechanisms for the connection with relational and graph databases via explicit directives.

Chapter 10

External Sources of Computations

I-DLV supports the call to external sources of computations within ASP programs by means of *external atoms* [22, 24],whose extension is not defined by the semantics within the logic program, but rather is specified by means of external Python programs. They are inspired by the ones supported in *dlvhex* [38], although there are some differences; for instance, in *dlvhex* external atoms can have as input parameters also predicates, while relational inputs are not permitted in *I-DLV*; this implies that *I-DLV* external atoms can be completely evaluated at the grounding stage, while in *dlvhex* an external atom, in general, might need to wait for the solving phase in order to be evaluated, depending on the interpretation in question.

In the next sections we describe syntax and semantics of the constructs, then illustrating their use by a few examples.

10.1 External Atoms

An *external atom* have the form:

$$\&p(t_0,\ldots,t_n;u_0,\ldots,u_m)$$

where:

- p is an *external predicate*;

- $t_0, \ldots, t_n$ are *input* terms;
- $u_0, \ldots, u_m$ are *output* terms;
- $n + m > 0$.

The name of an external predicate starts with "&" symbol and input terms are separated from the output terms by ";". Note that an input or output term can be either a *constant* or a *variable*.

Example 10.1. *Examples of external atoms:*

$$\&e1(X;Y),\ \&e2(c1,X;Z),\ \&e3(;Z),\ \&e4(\text{"}word\text{"};Z)$$

An external literal is either `not` e or e, where e is an external atom, and "`not` " represents default negation. An external literal is safe if all input terms are safe, according to the safety definition in section 2.1. External atoms can appear only in rules bodies, so external atoms in rules head are not allowed, nor two external atoms with the same external predicate but a different number of input or output terms. Intuitively, the computation of output terms is carried out according to the semantics which is externally defined via Python scripts. In particular, the user must define a Python function for each external predicate $\&p$ with n/m input/output terms. Moreover, the Python function receives n parameters and returns m output values and has to be compliant with Python[1] version 3. Note that each instance of an external predicate must appear with the same number of input and output terms throughout the program.

Example 10.2. *The following rule makes use of an external predicate with one input and output terms:*

$$compute(X,Z) :\!-\, number(X), \&square_root(X;Z).$$

A program containing this rule must come with a Python script that contains the definition of the external atom &square_root within a Python function, as, for instance, the one reported next.

[1] https://docs.python.org/3

```
1   import math
2
3   def square_root(X):
4       return math.sqrt(X)
```

The previous program calculates just the square root of a number given as input. Therefore, given to I-DLV *a program containing the previous rule with the facts* $\{number(9).\ number(16).\ number(25).\}$ *and a Python script containing the previous function, the systems calculate:* $\{compute(9,3), compute(16,4), compute(25,5)\}$. *Notably,* I-DLV *can manage multiple Python scripts and is also possible to import extern system or user-defined modules.*

External atoms can be both functional or relational, i.e., they can return a single tuple or a sequence of tuples, as output. In the Example 10.2 the external atom *square_root* is *functional* because the Python function for each input value returns a single value. In general, a functional external atom with $m > 0$ output terms return a *sequence*[2] containing m values; if $m = 0$, the associated Python function return a Boolean value. A relational external atom with $m > 0$ is defined by a Python function that returns a sequence of m-sequences, where each inner sequence is composed of m values (the output values).

Example 10.3. *Consider the following rule:*

$$prime_factor(X,Z) \text{ :- } number(X),\ \&compute_prime_factor(X;Z).$$

Intuitively, the rule, given a number, is intended to compute prime factors of it. This task is demanded to a relational external atom, that receives as input the number X, and returns as output its factors. The semantics of $\&compute_prime_factor$ *is provided by a Python function:*

```
1   def compute_prime_factors(n):
2       i = 2
3       factors = []
4       while i * i <= n:
```

[2]https://docs.Python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

```
5            if n % i:
6                i += 1
7            else:
8                n //= i
9                factors.append(i)
10       if n > 1:
11           factors.append(n)
12       return factors
```

Therefore, the previous Python function returns a sequence of 1-sequence due to $m = 1$ that can be simplified in a simple sequence. For this reason, the returned variable $factors$ is a simple list.

Given an external atom $\&p(In;Out)$, where In and Out represent input and output terms respectively, and a substitution σ, a ground instance of such external atom is obtained by applying σ to variables appearing in In and Out, obtaining:

$$\sigma(\&p(In;Out)) = \&p(In_g;Out_g)$$

The truth value of a ground external atom is given by the value $f_{\&p}(In_g;Out_g)$ of a decidable $n+m-ary$ two-valued Boolean oracle function, where n and m are the lengths of In_g and Out_g, respectively. A negative ground external literal `not` e is $true/false$ if e is $false/true$. Intuitively, output terms are computed on the basis of the input ones, according to a semantics which is provided externally (i.e., from the outside of the logic program) by means of the definition of oracle functions and according to the given definition, external atoms are completely evaluated by *I-DLV* as *true* or *false*, i.e. external predicates do not appear in the produced instantiation.

10.2 External Atoms Mapping Policy

After the evaluation of a Python function for each returned value, a proper conversion is necessary from its Python type to an ASP-Core-2 term type. On the Python side, the following types are permitted:

- *numeric*;

- *boolean*;
- *string*[3].

These types are mapped to terms accordingly to the following default policy:

1. an integer returned value is mapped to a corresponding *numeric constant*;
2. while all other values:
   (a) if the form is compatible to the *constant* of ASP-Core-2 syntax then is associated to a *symbolic constant*;
   (b) otherwise, values are associated to a *string constant*.

Nevertheless, *I-DLV* allows the user to customize the mapping policy of a particular external predicate by means of a directive of the form:

`#external_predicate_conversion(&p,type:`T_1`,...,`T_N`).`

where:

- $\&p$ is an external predicate;
- $T_1,\ldots,T_N$ are output terms.

Intuitively, the directive specifies the sequence of conversion types for an external predicate $\&p$ featuring n output terms. A conversion type can be:

1. `@U_INT` (the value is converted to an unsigned integer);
2. `@UT_INT` (the value is truncated to an unsigned integer);
3. `@T_INT` (the value is truncated to an integer);
4. `@UR_INT` (the value is rounded to an unsigned integer);
5. `@R_INT` (the value is rounded to an integer);
6. `@CONST` (the value is converted to a string without quotes);

[3]https://docs.python.org/3/library/stdtypes.html

7. @Q_CONST (the value is converted to a string with quotes).

In cases $(1-5)$, the value is mapped to a *numeric term*, in case (6) to a *symbolic constant*, while in case (7) to a *string constant*.

Directives can be specified at any point within an ASP program, and have a global effect: once a conversion directive for an external predicate has been included, say $\&p$, it is applied each time the predicate is found.

Example 10.4. *Consider the following directive with the program in Example 10.2:*

```
#external_predicate_conversion(&square_root,type:Q_CONST).
```

Then, for each external call, the output variable Z is bounded to the value returned by the Python function interpreted as a quoted string and the output of I-DLV *would be:* $\{compute(9,"3"), compute(16,"4"), compute(25,"5")\}$.

10.3 Taking Advantage from External Computations for KRR

Interfacing ASP with external tools represents an important additional power for modeling KRR tasks. Let us consider, as a running example, the problem of automatically assigning a score to students after an assessment test: given a list of students, a list of topics, and a set of questions regarding the given topics along with corresponding student answers, our aim is to determine the score of each student. In particular, let us suppose that each student is represented by a fact of the form $student(id)$, where *id* is a unique identifier code, and topics are expressed by facts of type $topic(to)$, where *to* is a string representing the topic name, such as "Computer_Science". Each question can be expressed by a fact of the form $question(id,to,tx,ca)$ where *id* is a unique identifier number, *to* is the topic covered, *tx* is the text containing also the possible answers, and *ca* is the only option which is the correct answer. Each student's answer is modeled by a fact of the form $answer(sid,qid,ans)$ where *ans* represents the answer given by student *sid* to the question *qid*.

In addition, let us suppose that the score is computed as the sum of the single scores obtained by answering the questions on each topic, and that some topics

can have more weight than others; for instance, the score obtained in *Mathematics* might be more important than the one in *English*. Interestingly, the score could be computed differently each time the assessment test takes place; for instance, *English* might be the most important topic, in a different session. Moreover, the number of wrong answers may also be considered while computing the score, for instance in case of negative marking.

Given these requirements, the following program encodes our problem:

$$s_1: \ correctAnswers(St,To,N) \ {:\!-} \ topic(To), student(St), N = \#count\{QID : question(QID,To,Tx,Ca), answer(St,QID,Ca)\}.$$

$$s_2: \ wrongAnswers(St,To,N) : -topic(To), student(St), N = \#count\{QID : question(QID,To,Tx,Ca), answer(St,QID,Ans), Ans! = Ca\}.$$

$$s_3: \ topicScore(St,To,Sc) \ {:\!-} \ correctAnswers(St,To,Cn), wrongAnswers(St,To,Wn), \&assignScore(To,Cn,Wn;Sc).$$

$$s_4: \ \#external_predicate_conversion(predicate = \&assignScore, type = R_INT).$$

$$s_5: \ testScore(St,Sc) \ {:\!-} \ student(St), Sc = \#sum\{Sc : topicScore(St,To,Sc)\}.$$

Intuitively, rule s_1 and rule s_2 count the number of correct and wrong answers for each student on each topic, respectively. Rule s_3 determines the score on each topic for each student; to this end, an external atom is in charge of assigning the score. Interestingly, each time the relative weight of the topics needs to change, it is enough to change the Python function defining its semantics. For instance, supposing that among the concerned topics *Mathematics* and *Computer Science* have the highest importance, and that for each wrong answer the value 0.5 is subtracted from the score, a possible implementation for the Python function defining the score computation could be:

Problem	# inst.	DLVHEX		GRINGO		I-DLV	
		#solved	time	#solved	time	#solved	time
Attachment	10	0	TO	10	149,55	10	45,50
Growth	10	0	TO/MO	9	164,21	10	67,20
Move	10	0	TO/MO	9	163,89	10	68,08
Contact	10	6	93,05	10	11,21	10	4,94
Disconnect	10	8	127,72	10	10,85	10	4,86
Discrete	10	8	127,44	10	10,85	10	4,96
Equal	10	8	101,07	10	10,95	10	4,93
Externally Connect	10	8	100,43	10	10,79	10	4,68
Nontangential Proper Part	10	7	107,14	10	10,89	10	4,88
Overlap	10	6	92,80	10	11,11	10	4,94
Part Of	10	8	126,56	10	11,00	10	4,93
Partially Overlap	10	8	126,37	10	11,00	10	4,83
Proper Part	10	6	93,34	10	11,21	10	4,99
Tangential Proper Part	10	7	106,54	10	10,90	10	4,72
String Concatenation	5	0	TO/MO	5	64,73	5	52,09
Prime Numbers	10	1	93,34	10	21,94	10	13,26
Reachability	10	0	TO/MO	10	36,71	10	37,18
Solved Instances		**81/165**		**163/165**		**165/165**	
Total Running Time		**59341**		**8362**		**3109**	

Table 10.1: External atoms: experiment results (TO/MO stands for Time/Memory Out).

```
1 def assignScore(topic, numCorrectAns, numWrongAns):
2      if(topic==''ComputerScience''
3                or topic==''Mathematics''):
4        return numCorrectAns*2-numWrongAns*0.5
5      return numCorrectAns-numWrongAns*0.5
```

Statement s_4 is a directive stating that values returned by the Python function have be rounded to ASP-Core-2 integers, while rule s_5 computes the final score by summing up the scores on each topic.

10.4 Assessing External Computation Machinery

We compared *I-DLV* with other currently available systems supporting external computation via Python with the aim of assessing their efficiency at integrating external computations.

Experiments have been performed on a NUMA machine equipped with two 2.8GHz AMD Opteron 6320 processors and 128 GiB of main memory, running

Linux Ubuntu 14.04.5; binaries were generated with the *GNU* `C++` compiler v. 4.9. As for memory and time limits, we allotted 15 GiB and 600 seconds for each system, per each run.

Then, we compare *I-DLV* against both the ASP grounder *gringo* [54] and the *dlvhex* [38] system. In particular, we considered the latest available releases at the time of writing: *clingo* 5.2.0 executed with the `--mode=gringo` and *dlvhex* 2.5.0 executed with the default provided ASP solver that combines *gringo* 4.5.4 and *clasp* 3.1.4, respectively.

Hence, we first adapted a set of already-proposed problems, and then enriched them with further domains testing different aspects.

The first set of benchmarks is focused on the spatial representation and reasoning domain; these problems originally appeared in [98]. In this setting, two scenarios have been taken into account:

- The first scenario requires the determination of relations among randomly-generated circular objects in a 2-*D* space. For each pair of circles the aim is to find out which of the following relations hold: having some contacts, being disconnected, being externally connected, overlapping or partially overlapping, one being a part of the other, one being proper part of the other, one being a tangential proper part of the other, one being non-tangential proper part of the other. For each possible relation, an ASP encoding that makes use of an external Python script checks if it holds. The encodings have been paired with instances of increasing sizes containing random generated circles, from 10 to 190.

- In a second scenario we adapted the encodings of *Growth*, *Move* and *Attachment* problems introduced in [98], that solves some geometrical problems over triples of circular objects in a 2D space. Again, instances of increasing size have been given to the tested systems: in this case, we generated triples of circles, from 7 to 70.

Notably, ASP-Core-2 only supports integer numbers; hence, the encodings have been adapted in order to become independent from the way data are expressed. Each object is associated with an identifier, and information about coordinates and dimensions are stored in a csv file; thus, from the ASP side, objects are managed via their ids, and computations involving real numbers are handled

externally via the same Python scripts that, in turn, are invoked by the external mechanisms typical of each tested system. Hence, the encodings reported in [98] have been carefully translated into ASP-Core-2 encodings. In this respect, it is worth noting that in [98] two versions for Attachment problem are reported: in our setting, due to the described translation, they coincide.

Since the benchmarks introduced above involve non-disjunctive stratified encodings, relational external atoms and only numeric (integer) constants as ground terms, we considered three further domains: the reachability problem, where edges are retrieved via Python scripts; concatenation of two randomly-generated strings with arbitrary lengths varying from 1000 to 3000 chars; generation of first k prime numbers, with k ranging from 0 to 100000.

Table 10.1 reports the result of the benchmark evaluation: after the domain name and the corresponding number of instances, the next three pairs of columns show the number of solved instances and the average running time. The last line reports the total running times for each system. Figures 10.2 and 10.2 summarizes the results for the two different sets of benchmark.

Results show satisfactory performance for *I-DLV*, both in comparison with *gringo*, which solves approximately the same number of instances but spending larger times and with *dlvhex*, which shows uncompetitive performance in part due to its architecture, that makes use of an ASP solver as a black box, and especially because it offers a complete support for external source of knowledge and a more expressive language, which in turns requires more computational cost.

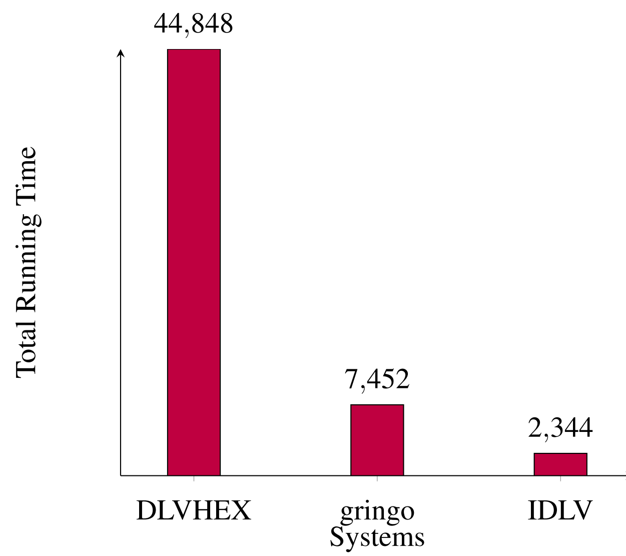

Figure 10.1: Set of Benchmarks 1 on the spatial representation and reasoning domain

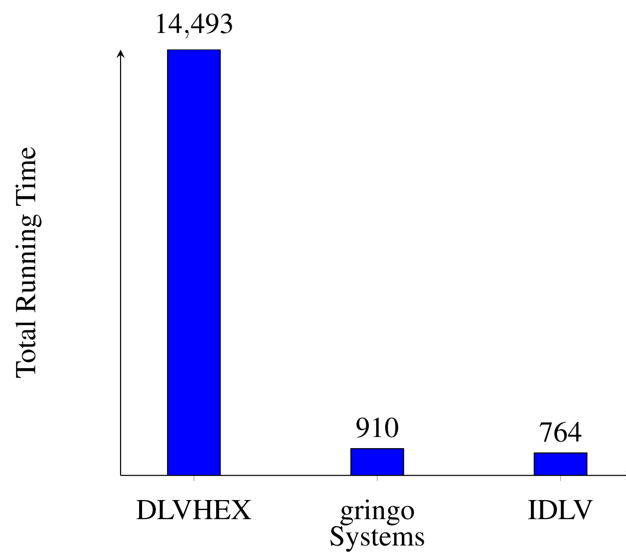

Figure 10.2: Set of Benchmarks 2 on ad-hoc domains

Chapter 11

Exchanging Data with External Sources of Knowledge

To further ease the use of *I-DLV* in real-world applications, it has been extended with the aim of easing the interoperability and integration with external systems. We define and implement a framework for creating ad-hoc directives for interoperability and make use of it for providing some ready-made ones for the connection with relational and graph databases.

11.1 Connecting with Relational and Graph Databases

I-DLV inherits from *DLV* directives for *importing/exporting* data from/to relational DBs.

An import directive is of the form:

```
#import_sql(db_name, "user", "pwd", "query",
                              pred_name [, type_conv]).
```

where:

- the first three parameters are the database name, the username and password used for the connection to the database;

- `query` is an *SQL* statement that defines the imported table (must be double quoted);
- `pred_name` indicates the predicate whose extension will be enriched with the result of the query;
- `type_conv` specifying how DBMS data types have to be mapped to ASP-Core-2 terms and it is optional.

The `type_conv` parameter is a string with the same syntax described in Section 10.2.

An export directive is of the form:

```
#export_sql(db_name, "user", "pwd", p_name,
                                    p_arity, table_name).
```

where:

- first three params are the same of connection parameters of *#import_sql* directive;
- `p_name/p_arity` indicate *name/arity* of the predicate whose extension will be exported into the external `table_name` table.

The previous directives mimic the old *DLV* with slight differences due to the fact that *I-DLV* complies with the ASP-Core-2 syntax. In particular, ASP-Core-2 features negative numbers, thus requiring more conversion types and allows a predicate name to be associated with predicates with different arities, thus requiring that in an `export` directive the predicate has to be uniquely specified by indicating both the predicate name and arity.

Interoperability with graph databases is a novel feature of *I-DLV* in which data can be imported via SPARQL queries, both from *local* RDF/XML files and *remote* SPARQL *EndPoints*.

The import directives is of the form:

```
#import_local_sparql("rdf_file","query",
                                predname,predarity,[,typeConv]).
```

and

```
#import_remote_sparql("endpnt_url","query",
                              predname,predarity,[,typeConv]).
```

where:

- `query` is a SPARQL statement defining data to be imported;
- `typeConv` is optional and specifies the conversion for mapping RDF data types to ASP-Core-2 terms (similar to *#import_sql* directive).

For the local import the `rdf_file` in the directive *#import_local_sparql* can be either a local or remote URL pointing to an RDF/XML file. In the second case, the file is downloaded and treated as a local RDF/XML file and in any case, the graph is built in memory. On the other hand, for the remote import, the `endpnt_url` in the directive *#import_remote_sparql* , it refers to a remote endpoint and building the graph is up to the remote server. This second option might be more convenient in the case of large datasets.

Graph and relational databases connection directives can be simulated using external atoms; in addition, it shows how external atoms give the user a powerful means for extending the grounder capabilities significantly. Nevertheless, a "native" support for interoperability with graph and relational databases has some benefits. First of all, it is easy to imagine that native support should enjoy much better performance, as we discuss in later on in Section 10.1. Furthermore, in many scenarios (as it is often the case in the deductive database settings) the use of external atoms is not crucial, while accessing standard knowledge sources is vital: in such cases, taking care of writing efficient Python code and the burden of the external Python runtime machinery does not seem worthwhile. Hence, the idea is to incorporate into the system the directives that are most likely to be used "per se", and let external atoms address cases that need extended functionalities.

Example 11.1. *Suppose that general data of the example in Section 10.3, such as questions and topics, are permanently stored in a relational database, while test-related data, such as the student answers, are gathered into an RDF/XML file which is specifically referred to the test itself.*

The following directive allows retrieval of questions from the relational DB:
```
#import_sql(relDB, "user", "pwd", "SELECT * FROM question",
```

```
          question, type:U_INT,Q_CONST,Q_CONST,Q_CONST ).
```

where:

- `relDB` *is the name of the database;*
- `"user"` *and* `"pwd"` *are the data for the user authentication;*
- `"SELECT * FROM question"` *is an SQL query that constructs the table to import;*
- `question` *is the predicate name used for constructing the new facts;*
- *type : U_INT, Q_CONST, Q_CONST, Q_CONST are the mapping policy where the first field (the question identifier) is imported as an integer, while the remained values are converted to quoted strings.*

Topics can be retrieved similarly:

```
#import_sql(relDB, "user", "pwd",,
             "SELECT * FROM topic", topic, type:Q_CONST ).
```

Moreover, student answers can be retrieved with the following directive:

```
 #import_local_sparql("answers.rdf",
"PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX my:  <http://sample/rdf#>
    SELECT ?St, ?Qe, ?Ans
    WHERE {?X rdf:type my:test.  ?X my:student ?St.
    ?X my:question ?Qe.  ?X my:answer ?Ans.}",
answer, 3, type:U_INT,U_INT,Q_CONST).
```

where:

- `"answers.rdf"` *is the XML file containing the answers;*
- *the long quoted string is the SPARQL query;*
- `answer/3` *is the predicate used for constructing the new facts;*
- *type : U_INT, U_INT, Q_CONST are the mapping policy where the first two fields (the question's and student's identifiers) are imported as integers and the latter is converted to a quoted string.*

11.2 Experimentally Assessing the Interoperability Mechanisms

In this section we analyze the effective gain on performance obtainable with a native support of SQL/SPARQL local import directives against the same directives implemented via Python scripts. In detail, we compare two different importing approaches:

- a version exploiting explicit directives natively implemented in `C++`;
- a version where the import mechanism is performed externally.

Experiments have been performed on a machine equipped with an Intel Core i7-4770 processor and 16 GiB of main memory, running Linux Ubuntu 14.04.5; binaries were generated with the *GNU* `C++` compiler v. 4.9. As for memory and time limits, we allotted 15 GiB and 600 seconds for each system, per each single run.

The benchmarks are divided into two categories:

- Importing data from a Relational Database, by means of SQL statements. To this end, a DB containing a randomly generated table has been created: this table contains 1000000 tuples and features three columns, one of integer type and two of alphanumeric type. Several encodings have then been tested, each one importing a different number of tuples from the aforementioned table, ranging from 100000 to 1000000. In both cases, each SQL column is mapped, respectively, to a numeric term, a symbolic constant and a string constant (we refer the reader to ASP-Core-2 term types [17]).
- SPARQL imports from a local RDF/XML file. In particular, we generated some OWL ontologies via the Data Generator(UBA) by means of the Lehigh benchmark LUBM [65]. Such ontologies are referred to a University context: each university has a number of departments ranging from 15 to 22. The considered encodings import graduate and undergraduate students from a different number of universities, ranging from 1 to 5.

Notably, in both SQL and SPARQL benchmarks the complexity of the query is fixed. We choose this benchmarks because in this experiments the goal is to analyze the effective gain on performance with a native support implemented in C++

against the same directives implemented via Python scripts that have the burden of the external Python runtime machinery. Therefore, the study of the query used regarding expressivity is not relevant because rely on the SQL/SPARQL engine used.

Results, reported in Figure 11.1 and FIgure 11.2, show that the native approach clearly outperforms the other by 66% when dealing with SQL directives, and by 43% when dealing with SPARQL local import directives.

Intuitively, an internal management of import/export mechanism can be performed directly interfacing `C++` and SQL/SPARQL, while with external atoms Python acts as a mediator causing an overhead which is not always negligible as our tests evidenced. Hence, a "native" support for interoperability enjoy much better performance in all the cases, as the previous test shows, and in many scenarios scalability is indispensable especially for accessing standard knowledge sources. However, external atoms easy the integration of custom functionalities with external functions while defining new directives requires changes in the source code of the systems followed by the rebuilt of it.

Therefore, the idea is to incorporate into the system the directives that are most likely to be used "per se", and let external atoms address cases that need extended functionalities.

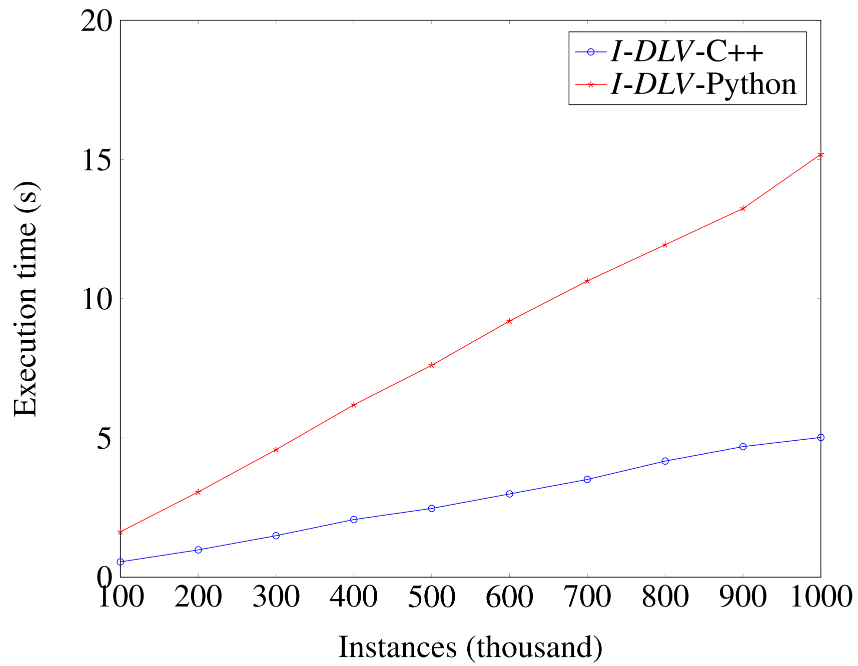

Figure 11.1: Experimental results importing data from a Relational Database.

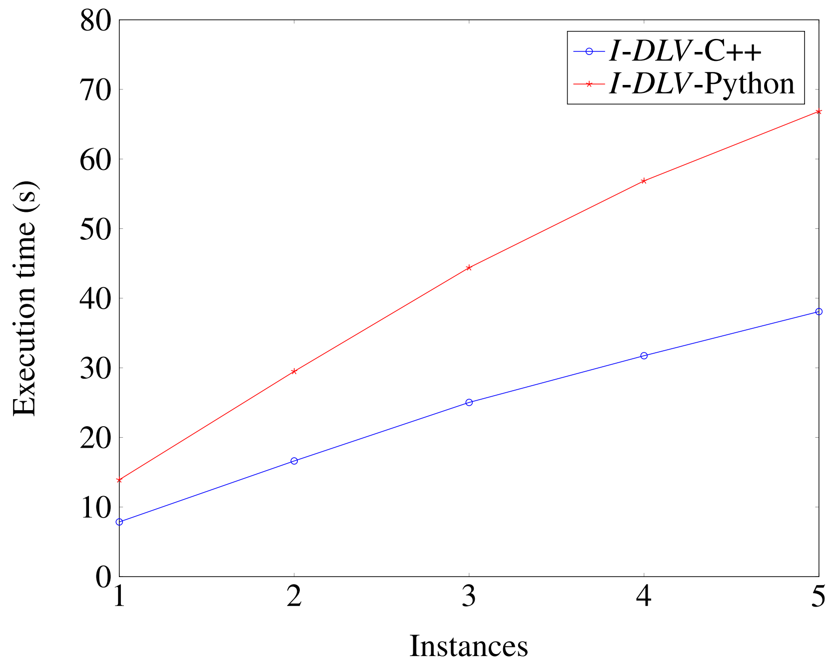

Figure 11.2: Experimental results importing data from a local RDF.

Part IV

Boosting Performance

Chapter 12

Improving Performance: the Need for Speed

As already discussed above, the high expressivity of the language made ASP a powerful tool for developing advanced applications in different areas. These applications prove the viability of the use of ASP; nevertheless, they also point out the need for efficient implementations, as we reported in Section 4.2. Nonetheless, the interest in developing more efficient and faster systems is still a crucial and challenging research topic, as witnessed by the results of the ASP Competition series [56, 29].

The competition evaluates ASP systems in order to compare them on the basis of fixed input language and fixed conditions. For each benchmark problem, a fixed ASP-Core-2 program encodes it, and each system is run on the same platform with limited time and memory available. The selected benchmark domains are classified according to the computational complexity of the related problem, in *Polynomial*, *NP*, and *Beyond-NP*. The benchmark suite included different domains, like temporal and spatial scheduling problems, combinatory puzzles, graph problems. Moreover, several benchmarks domains are based on particular applications. For instance, *Incremental Scheduling* [7] deals with assigning jobs to devices such that the makespan of a schedule stays within a given budget, *Partner Units* [5] has applications in the configuration of surveillance, electrical engineering, computer network, and railway safety systems, *Video Streaming* [95] aims at an adaptive regulation of resolutions and bit rates in a

content delivery network. Comparing the results with the last available competition and the previous one highlights significant advances in the state of the art. Indeed, during the different competition, several domains have been classified in *very easy*, i.e. the systems taking the first three places of the previous competition can solve all the instances in less than 20 seconds. These achievements have been obtained thanks to the new solving technique introduced such as the *Conflict-Driven Clause Learning* proposed in *clasp* [35] that combines the high-level modeling capacities of ASP with state-of-the-art techniques from the area of Boolean constraint solving, the unsatisfiable core shrinking implemented in *wasp* [2] that can boost the computation of optimum stable models for logic programs with weak constraints employing unsatisfiable core analysis. However, there are still several benchmark instances classified as untractable, such as the *non-groundable*, i.e. systems cannot finish the grounding within the allotted time, and *too hard*, i.e. systems cannot finish the solving during the allotted time. To overcome these challenges the portfolio approach is an active research area because it can take advantage of existing technology conversely to develop a competitive ASP system from scratch by no means an easy task.

The non-groundable problems highlighted the weaknesses of the traditional grounding and solving approaches used by the state-of-the-art systems. In several real-world applications, as we show in Section 4.2, the exponentially grow off the ground program size is also called *grounding bottleneck*. Lazy-grounding tries to overcome this problem by interleaving grounding and solving [32, 70, 99]. However, this technique cannot benefit from the solver optimization techniques due to the fact that the ground program is computed during the solving phase. Therefore, an active research aim is to couple the grounding and the solving system in order to improve the performance of ASP systems with a tight integration to avoid grounding bottleneck.

In the next chapters, we present two systems with that aim at reaching a more efficient ASP evaluation: *DLV2* and *I-DLV+MS*. The first is a recently released system that combines the *I-DLV* grounder with the solver *wasp*, extending the core modules by application-oriented features that customise heuristics of the system and extend its solving capabilities. The second is an ASP system relying on *I-DLV* as a grounder, and makes use of an automatic solver selector that inductively choose the best solver, depending on some inherent features of the instantiation.

Chapter 13

A New Efficient ASP System

The ASP computation, as briefly discussed in Section 3.2, is typically split into grounding and solving, and many ASP tools actually focus on one of the two processes, due to the complexity of implementing a *monolithic* ASP system. Therefore, a user has to pipe the output of the grounder to the solver system that automatically outputs the answer sets. However, a monolithic system offers more control over the grounding and solving process. Also, a tight integration of grounder and solver can improve the performance of the entire system, because, unfortunately, simply improving the grounding times does not necessarily imply improvements on the solving side, since the ASP solvers heavily depend on the form of the produced instantiation, therefore a specific tailoring of the grounding output can definitely benefit the solving side.

The first monolithic ASP system was called *DLV* [72], and more recently, also the grounder *gringo* and the solver *clasp* have been released together in the *clingo* system [51].

In this light, we worked on the design and development of a new ASP system, namely *DLV2*, a brand new and improved version of *DLV*, that updates its predecessor with modern evaluation techniques and development tools. For the solving part it relies on the well-assessed solver *wasp* [3], while the instantiation task is in charge of *I-DLV* [23], a new intelligent grounder that, as already mentioned, we actively contribute to design and implement.

In the next sections, we first present an overview of the *I-DLV* system and then we describe major features of *DLV2*; among them, annotations and direc-

tives for customizing heuristics of the solver, and also an empirical analysis conducted on benchmark from ASP competitions that show good performance of *DLV2* when compared against the old system and *clingo*.

13.1 The *I-DLV* Grounder

The new intelligent instantiator *I-DLV*, starting from the solid theoretical foundations of *DLV*, has been redesigned and re-engineered to build a renovated ASP grounder with the purpose of improving the performance and native support the ASP-Core-2 standard language [17]. The core strategies of the system rely on a bottom up evaluation based on a semi-naïve approach [50], that has been extended in *I-DLV* with a number of optimization techniques that have been explicitly designed by contextualizing it in the setting of an ASP grounder.

Optimized Grounding Process

Optimization implemented into *I-DLV* include: *rule body back-jumping*, *magic-sets*, and, in a significantly enhanced version, *body-reordering* and *indexing* strategies. The first two techniques have been adopted from the *DLV* grounder, and adapted to the new data structures and architecture; the latter two have been significantly enhanced and/or extended.

In particular, the body reordering technique in *I-DLV* compute new statistics for variables involved in comparison predicates, covering cases that previously were not properly addressed [23]; indexing strategies with an on-demand policy manage single- and multiple-argument indices, along with heuristics for selecting the best strategy [23].

Furthermore, a number of additional techniques and features have been designed and introduced in *I-DLV*, not included in *DLV*, such as the *anticipated evaluation of strong constraints*, the *aligning substitutions* mechanism, the *two-fold management* of isolated variables. More in detail, the anticipated evaluation of strong constraints process the strong constraints as soon as the extensions of its body predicates are available and due to the simplification mechanism, literals appearing in the body which are already known to be true can be removed, possibly leading to an empty-body constraint. By definition, such constraints

are always violated; thus, the input program is inconsistent, and the grounding process can be safely aborted [23].

The aligning substitutions search for an "agreement" between body literals on variable replacements. Basically, before processing a rule r, for each variable X, we compute the intersection of all sets of possible substitutions for all the occurrences of X in r. Intuitively, this reduces, in general, the number of possible values for X, by skipping those that would not match among distinct variable occurrences. Such technique performs best when the sets of substitutions differ significantly [23].

Isolated variables are variables that not affect any failed match for other atoms, nor the instances obtained for the head atoms. Thus, we can safely eliminate the isolated variables by projecting them in auxiliary predicates or if the predicate depends only on EDB and solved predicates, a special filter mechanism when looking for next matches can be applied, suggesting only the relevant instances [23].

Flexibility and Customizability

I-DLV allows the user to enable, disable, and customize every single strategy, thus resulting in a flexible tool for experimenting with ASP and its applications. To this end, it has been designed in order to allow a fine-grained control over the whole computational process, both via command-line options and a new special feature for facilitating system customization and tuning: *annotations* of ASP code. *I-DLV* annotations allow to give explicit directions on the internal grounding process, at a more fine-grained level with respect to the command-line options: they "annotate" the ASP code in a Java-like fashion, while embedded in comments: hence, the resulting programs can still be given as input to other ASP systems that do not support them, without any modification. In particular, our annotations can have two different scopes: at the global level, meaning that they are applied to the whole program, or at the rule level, and hence annotations acts just on the rule they precede.

Syntactically, all annotations start with the prefix "%@" and end with a dot ("."). The current *I-DLV* release supports annotations for customizing two of the major aspects of the grounding process: body ordering and indexing.

A specific body ordering strategy can be explicitly requested for any rule,

simply preceding it with the line:

```
%@rule_ordering(@value=Ordering_Type).
```

where `Ordering_Type` is a number representing an ordering strategy. In addition, it is possible to specify a particular partial order among atoms, no matter the employed ordering strategy, by means of `before` and `after` directives. For instance, in the next example, *I-DLV* is forced to always put literals $a(X,Y)$ and $X = \#count\{Z : c(Z)\}$ before literal $f(X,Y)$, whatever the order chosen:

```
%@rule_partial_order(
    @before={a(X,Y), X=#count{Z:c(Z)}},
    @after={f(X,Y)}).
```

As for indexing, directives on a per-atom basis can be given; the next annotation, for instance, requests that, in the subsequent rule, atom $a(X,Y,Z)$ is indexed, if possible, with a double-index on the first and third arguments:

```
%@rule_atom_indexed(@atom=a(X,Y,Z),
    @arguments={0,2}).
```

Multiple preferences can be expressed via different annotations; in case of conflicts, priority is given to the first appearing in the program. In addition, preferences can also be specified at a global scope, by replacing the `rule` directive with the `global` one. Such kind of annotations are applied on the rules, if possible. While a `rule` annotation must precede the intended rule, `global` annotations can appear at any line in the input program. Both `global` and `rule` annotations can be expressed in the same program; in case of overlap on a particular rule/setting, priority is given to the more specific `rule` ones.

Intuitively, the way annotations change the grounding mechanisms can noticeably affect performance on the program at hand. It is worth noting that the heuristic-based nature of the default strategies does not guarantee optimality, and thus it is possible that the introduction of further policies, directly indicated by the user or inferred from other criteria, lead to improve the performance of *I-DLV*.

13.2 The ASP System *DLV2*

13.2.1 *DLV2* Overview

DLV2 consist of the combination of the grounder *I-DLV* [23] and the solver *wasp* [3] that compose the core module of the system. Currently, *DLV2* implements a one-directional communication between the grounder to the solver. Nevertheless, this first integration allow *DLV2* to improve the usability of the main application-oriented features of *I-DLV* and *wasp* systems.

wasp

The solver module implements a modern CDCL backtracking search algorithm, extended with custom propagation functions to handle the specific properties of ASP programs. The computation of optimum answer sets can be carried out by using either *model-guided* or *core-guided* algorithms [2]. Core-guided algorithms can also be combined with strategies for shrinking the size of the cores that can boost the computation of optimum stable models for logic programs with weak constraints.

RDBMS Data Access

DLV2 can import/export data from/to an RDBMS by means of directives, carried out by the *I-DLV* core. Therefore, the syntax of the directives are the same as those used in *I-DLV*, Chapter 11. Hence, with `#import_sql` is possible to import table from an RDBMS, otherwise with `#export_sql` are used to populate specific tables with the extension of a predicate.

Query Answering

DLV2 supports cautious reasoning over non-ground queries. Then, if the ASP program contains a query the system applies the magic-sets technique [4] to optimize the evaluation of queries, performed by *I-DLV* core. Afterwards, the computation of cautious consequences is done according to *anytime* algorithms [4], performed by *wasp* core, so that answers are produced during the computation even in computationally complex problems.

Python Interface

Similarly to *I-DLV* (Section 10.1), using *DLV2* input program can be enriched by external atoms of the form $\&\mathtt{p}(\mathtt{i_1},\ldots,\mathtt{i_n};\mathtt{o_1},\ldots,\mathtt{o_m})$, where p is the name of a Python function, $\mathtt{i_1},\ldots,\mathtt{i_n}$ and $\mathtt{o_1},\ldots,\mathtt{o_m}$ $(n,m \geq 0)$ are input and output terms, respectively.

On the solving side, the input program can be enriched by external propagators. The computation of answer sets in *wasp* solver, as mentioned above, is carried out by employing an extended version of the Conflict-Driven Clause Learning (CDCL) algorithm, introduced for SAT solving [36]. An important feature of this algorithm is *propagation*, which extends the interpretation with the literals that can be deterministically inferred. *wasp* system supports *external propagators*, which allows a user to embed new external propagators (using Python language) in the solver. A user can write an external Python module that complies with a particular interface, whose methods are associated with events occurring during the search for an answer set. Whenever a specific point of the computation is reached, the corresponding event is triggered, and a method of the module is called. However, *wasp* takes as input a ground program, in particular, a numeric format produced by a grounder system, and external propagator works with predicates belonging to the non-ground program. In some case, the numeric format does not contain all the information to reconstruct the original non-ground program and then is hard for a user to write an external propagator. *DLV2* simplifies this process using specific directive for external propagator. An example is reported below, and also used in experiments.

Example 13.1. *Consider the following program, for a given positive n:*

$$
\begin{array}{l}
edb(1..n).\\
\{value(X) : edb(X)\}.\\
\{in(X) : edb(X)\}.\\
:\!-value(X), X! = \#count\{Y : in(Y)\}.
\end{array}
$$

The instantiation of the last constraint above results into n ground rules, each one comprising n instances of $in(Y)$*. It turns out that the above program cannot be instantiated for large values of n due to memory consumption. Notably, in this example, we use choice rules, a syntactic shortcut defined in the new ASP Core*

2 standard [17].

Within DLV2*, the expensive constraint can be replaced by the following propagator:*

```
#propagator(@file="propagator.py",
            @elements={X,value:value(X); X,in:in(X)}).
```

where `propagator.py` *is a Python script reacting on events involving instances of value(X) and in(X).*

The directive *#propagator* takes as input the name of the Python script that implements the external propagator and also a set of atoms used during the propagation. In particular, after the grounding, a dictionary of all the instances of the atoms appearing in the directives is built. Each element of the dictionary has a key that represents uniquely the literal, and a tuple that stores its values. Then, the script checks whether candidate answer sets satisfy the propagator using the dictionary to recover the values of the literals and also, to access the status of the current interpretation.

Java-like Annotations

In order to ease the system customization and tuning, *DLV2* enrich the ASP programs by global and local annotations, where each local annotation only affects the immediate subsequent rule. The system takes advantage of annotations to customize some of its heuristics and annotations are processed by the *I-DLV* core [23]. Customizations include *body ordering* and *indexing*, two of the crucial aspects of the grounding. For example, the order of evaluation of body literals in a rule, say $\prec$, is constrained to satisfy `p(X)` $\prec$ `q(Y)` by adding the following local annotation: `%@rule_partial_order(@before={p(X)}, @after={q(Y)}).`

Note that annotations do not change the semantics of input programs. For this reason, their notation starts with %, which is used for comments in ASP-Core-2, so that other systems can simply ignore them.

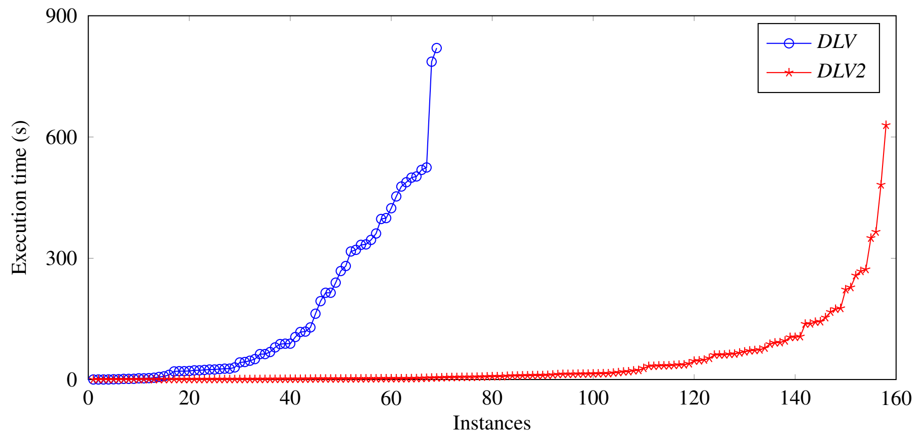

Figure 13.1: *DLV2* vs *DLV*: cactus plot on benchmarks from ASP Competition 2011.

13.3 Experiments

In this section we compare *DLV2* with *clingo* 5.1.0 [51] and *DLV* [72]. All systems were tested with their default configuration. *clingo* and *DLV2* were tested on benchmarks taken from the latest ASP competition [56]. However, since *DLV* does not fully support the ASP-Core-2 standard language, the comparison between *DLV* and *DLV2* is performed on benchmarks taken from the third competition [28]. An additional benchmark is obtained from the programs in Example 13.1. Experiments were performed on a NUMA machine equipped with two `2.8GHz AMD Opteron 6320` processors. The time and memory were limited to 900 seconds and 15 GB, respectively.

Results reported in the cactus plot in Figure 13.1 shows the comparison between *DLV* and *DLV2*. A sensible improvement is obtained by the new version of the system. Indeed, the percentage gain of the solved instances is 128%, and it is even higher, namely 273% if the running time is bounded to 60 seconds.

Results reported in Table 13.1 shows the comparison with *clingo*. Overall, *DLV2* solved 306 instances, 35 more than *clingo*. Such an advantage can be explained by the fact that *DLV2* handles query answering, while *clingo* does not. If benchmarks with queries are ignored, the difference between the two systems is five instances in favour of *clingo*.

As already observed in Example 13.1, the instantiation of the constraint is expensive in terms of memory consumption and results are reported in the plots

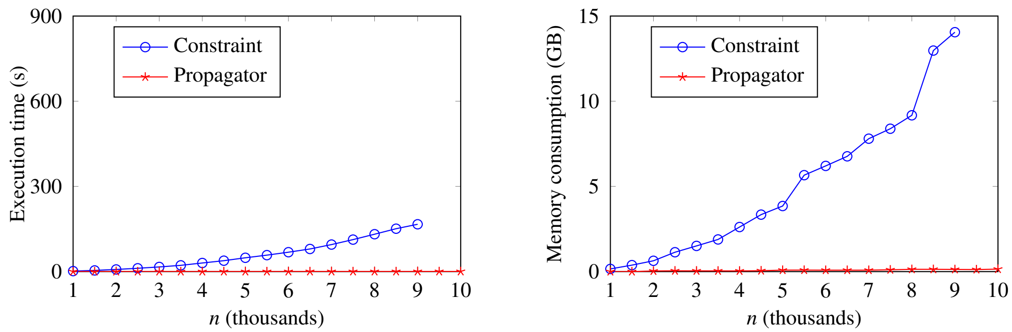

Figure 13.2: Evaluation of *DLV2* on programs from Example 13.1.

in Fig. 13.2. The average memory consumption is around 5 GB on the 17 solved instances. Such inefficiency also impacts on the execution time (left plot), which

Table 13.1: *DLV2* vs *clingo*: solved instances and average running time (in seconds) on benchmarks from ASP Competition 2015 (20 instances per benchmark).

Benchmark	*clingo*		*DLV2*	
	time	solved	time	solved
Abstract Dialectical Frameworks WF Model (optimization)	8.89	20	137.31	15
Combined Configuration	286.73	10	150.62	1
Complex Optimization of Answer Sets	174.61	18	120.08	6
Connected Maximim-density Still Life (optimization)	193.63	6	73.40	9
Crossing Minimization (optimization)	65.63	6	2.90	19
Graceful Graphs	191.42	11	59.17	5
Graph Colouring	215.98	17	204.83	9
Incremental Scheduling	131.21	13	166.21	8
Knight Tour With Holes	15.00	10	41.83	10
Labyrinth	105.12	13	181.75	12
Maximal Clique Problem (optimization)	—	0	168.84	15
MaxSAT (optimization)	44.33	7	90.83	20
Minimal Diagnosis	7.74	20	38.39	20
Nomistery	163.46	8	118.42	9
Partner Units	35.40	14	375.99	9
Permutation Pattern Matching	180.30	12	153.64	20
Qualitative Spatial Reasoning	174.81	20	326.47	18
Ricochet Robots	130.42	8	267.87	9
Sokoban	86.52	10	174.69	10
Stable Marriage	430.49	5	459.31	9
System Synthesis (optimization)	—	0	—	0
Steiner Tree (optimization)	242.45	3	—	0
Valves Location (optimization)	42.51	16	68.40	16
Video Streaming (optimization)	56.96	13	0.10	9
Visit-all	248.40	11	68.94	8
Consistent Query Answering (query)	—	—	252.77	13
Reachability (query)	—	—	131.48	20
Strategic Companies (query)	—	—	30.07	7
TOTAL		271		306

is on average 61 seconds on the solved instances. On the other hand, *DLV2* takes a sensible advantage from the ad-hoc propagator, solving each tested instance in less than 1 second and 80 MB of memory.

Chapter 14

Automating Solver Selection

The performance of current ASP systems can be defined as good enough for different real-world applications. However, they feature several different optimization techniques, which cause systems to outperform each other depending on the domain at hand. The capability to enjoy good performance over various problems domains has already been studied by other communities, by means of proper strategies of *algorithm selection* [90]; for instance, what has been done for solving propositional satisfiability (SAT) [101] and Quantified SAT (QSAT) [84]. This approach consists of building machine learning techniques to inductively choose the "best" solver on the basis of some input program characteristics, or *features*. As far as ASP is concerned, some interesting works in this respect have already been carried out in [78].

In this chapter, we present *I-DLV+MS* [25], a new ASP system that integrates an efficient grounder, namely *I-DLV* [23], with an automatic solver selector: machine-learning techniques are applied to inductively choose the best solver, depending on inherent features of the instantiation produced by *I-DLV*.

We define a particular set of features, and then carry out an experimental analysis for computing them over the ground versions of benchmarks submitted to the 6th ASP Competition [56]. Also, we build our classification method for selecting the solver that is supposed to be the "best" for each input among the two state-of-the-art solvers *clasp* [52] and *wasp* [3]. Furthermore, we test *I-DLV+MS* performance both against the state-of-the-art ASP systems and the best established multi-engine ASP system ME-ASP [78], that is the winner of the

6th ASP Competition.

Notably, *I-DLV+MS* participated in the latest (7*th*) ASP competition [58], winning in the regular track, category *SP* (i.e., one processor allowed).

In this section first, we introduce *I-DLV+MS* with an overview of the system, and we then describe the proposed classification method along with the selected features. Afterwards, we discuss a thorough experimental activity against the state-of-the-art ASP multi-engine system.

14.1 *I-DLV+MS* Overview

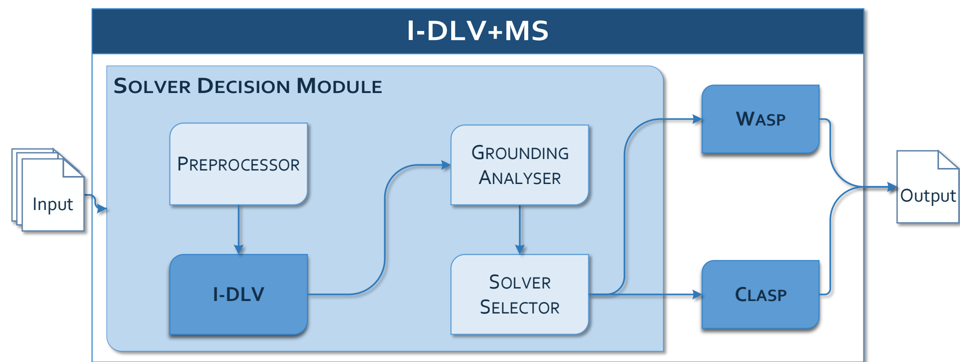

Figure 14.1: *I-DLV+MS* Architecture.

The architecture of *I-DLV+MS* is reported in Figure 14.1. Given an ASP program *P*, the PREPROCESSOR module analyzes it, and interacts with the *I-DLV* system to determine if the input program is non-disjunctive and stratified, because these kinds of programs are completely evaluated by *I-DLV* without the need for a solver. If this is not the case, the GROUNDING ANALYSER module extracts the signified features from the ground program produced by *I-DLV* and passes them to the classification module. Then, the SOLVER SELECTOR, based on proper classification algorithms, tries to predict, among the available solvers, which one would perform better and selects it. This module is based on *Decision Trees*, a non-parametric supervised learning method used for classification [85]; this classifier aims at creating a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. We use an optimized version of tree algorithm implemented in *scikit-learn* library [82],

namely *CART* (Classification and Regression Trees) [12]. The algorithm is a variant of C4.5 [86], but differs from it in that it supports numerical target variables and does not compute rule sets. *I-DLV+MS* currently supports the two state-of-the-art ASP solvers *clasp* and *wasp*. Nonetheless, the modular architecture of *I-DLV+MS* easily allows one to update the solvers or even add additional ones. Clearly, such changes would require the prediction model to be retrained with appropriate statistics on the new solvers.

Features

As already introduced, the machine-learning technique selects the best solver according to specific features of the input program. In this work, we selected several features with the aim of catching two fundamental aspects of ASP programs:

- *Atoms ratios.* We considered five different ratios that represent the type of atoms and a raw measure of their distribution in the input ground program:

  $$(a) : \frac{F}{R} \quad (b) : \frac{PA}{R} \quad (c) : \frac{NA}{R} \quad (d) : \frac{PA}{BA} \quad (e) : \frac{NA}{BA}$$

  where F is the total number of facts and always true atoms, R the total number of ground rules, PA/NA the total number of positive/negative atoms and BA the total number of atoms appearing in rule bodies.

- *Rules ratios.* We considered five different ratios that represent the type of rules and a raw measure of their distribution in the input ground program, also taking into account advanced constructs of the ASP-Core-2 standard language [18], such as choices, aggregates, and weak constraints:

  $$(f) : \frac{C}{R} \quad (g) : \frac{W}{R} \quad (h) : \frac{SR}{R} \quad (i) : \frac{CR}{R} \quad (j) : \frac{WR}{R}$$

  where C is the total number of strong constraints, W the total number of weak constraints, SR the total number of standard rules, CR the total number of choice rules and WR the total number of weight rules. Please note that with standard rules we denote rules without aggregate or choice atoms; weight and choice rules, instead, handle aggregate literals and choice atoms generated by the grounder.

Table 14.1: *I-DLV+MS*, *I-DLV +clasp* and *I-DLV +wasp*: number of solved instances and average running times (in seconds) on benchmarks from the 6th ASP Competition (20 instances per problem). In bold is outlined the top-performing system among *I-DLV +clasp* and *I-DLV +wasp*.

Benchmark	*I-DLV+MS*		*I-DLV +clasp*		*I-DLV +wasp*	
	solved	time	solved	time	solved	time
AbstractDialecticalFrameworks	20	10,91	**20**	**7,17**	14	80,38
CombinedConfiguration	13	103,83	**12**	**108,78**	6	52,93
ComplexOptimizationOfAnswerSets	19	127,68	**19**	**105,12**	5	127,58
ConnectedMaximim-densityStillLife	10	86,02	5	283,53	**8**	**120,60**
CrossingMinimization	19	2,46	6	91,88	**19**	**0,65**
GracefulGraphs	9	122,36	**10**	**59,77**	5	22,12
GraphColouring	16	104,05	**16**	**117,28**	6	137,67
IncrementalScheduling	11	42,09	**11**	**35,74**	8	113,75
KnightTourWithHoles	14	29,95	**14**	**25,87**	10	34,76
Labyrinth	10	134,82	**11**	**78,49**	11	134,06
MaximalCliqueProblem	14	49,06	0	-	**15**	**143,18**
MaxSAT	19	17,55	8	47,70	**20**	**50,91**
MinimalDiagnosis	20	27,55	**20**	**14,50**	20	24,86
Nomistery	8	26,21	**8**	**25,26**	8	37,09
PartnerUnits	14	23,29	**14**	**45,03**	8	210,47
PermutationPatternMatching	20	117,40	**20**	**16,86**	20	129,84
QualitativeSpatialReasoning	20	106,06	**20**	**113,75**	12	124,32
RicochetRobots	10	95,43	**12**	**130,96**	7	166,09
Sokoban	8	343,63	9	23,71	**10**	**153,95**
StableMarriage	6	111,30	**9**	**268,98**	7	397,86
SteinerTree	8	14,60	**8**	**84,82**	4	0,15
ValvesLocationProblem	16	6,99	**16**	**12,10**	16	37,04
VideoStreaming	15	18,89	**15**	**5,63**	9	1,94
Visit-all	8	162,88	**9**	**57,18**	8	69,74

14.2 Experimental Evaluation

To evaluate the *I-DLV+MS* system we report the results of an experimental activity, and in particular, we performed two distinct sets of experiments, which are discussed in the following.

Experiments have been performed on a NUMA machine equipped with two `2.8GHz AMD Opteron 6320` and 128 GiB of main memory, running `Linux Ubuntu 14.04.4`. Binaries were generated with the `GNU C++ compiler 4.9.0`. As for memory and time limits, we allotted 15 GiB and 600 seconds for each system per each single run.

14.2.1 Impact of Solver Selection

With the first set of experiments, we check the quality of the choice performed by the machine-learning-based model for solver selection. To this end, we took the benchmarks from the latest available ASP Competition [56] and ran *I-DLV+MS* along with two distinct combinations of the *I-DLV* grounder with *clasp* and *wasp* solvers.

This benchmark allows us to compare the performance of the solver chosen by *I-DLV+MS* against the best one. Ideally, a system equipped with a perfect selector would match the performance of the best solver for each benchmark, net of possible overheads.

Results are reported in Table 14.1: first column shows the name of the benchmark, while the next pairs report the solved instances and average times per each tested system (each benchmark featured 20 instances). The best performing combination among *I-DLV +clasp* and *I-DLV +wasp* is highlighted, on a benchmark basis, by means of bold values.

Results show that *I-DLV+MS* performance is very close to the best solver, in almost all domains, thus implying that the defined measures along with the chosen model lead to the right choices. Furthermore, the total number of instances solved by *I-DLV+MS* is 327, while for *I-DLV +clasp* and *I-DLV +wasp* is 292 and 256, respectively.

It is worth noting that, even when the choice was right, *I-DLV+MS* pays some overhead. This is due to the fact that the ground program produced by *I-DLV* is not directly processed to the solver; rather, it must be analyzed in order to extract the features needed for making the choice: hence, huge ground programs might cause a loss of performance during the features extraction.

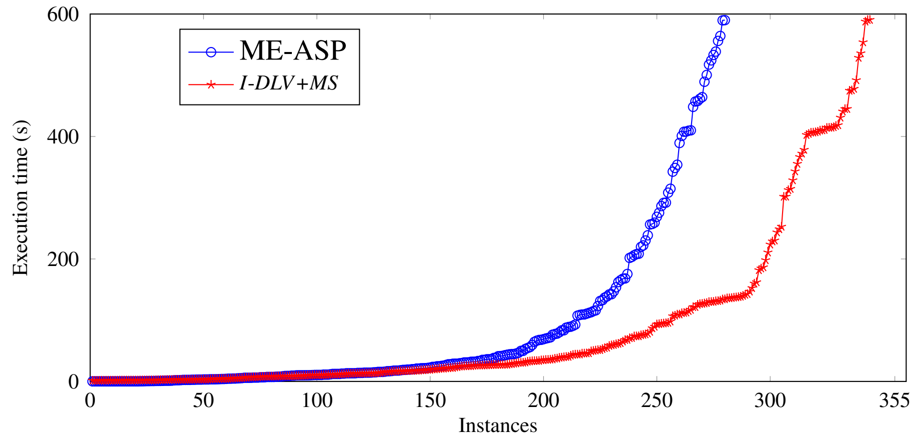

Figure 14.2: *I-DLV+MS* and ME-ASP comparison on benchmarks from the 6th ASP Competition.

14.2.2 Comparison to the State of the Art

In the second set of experiments we compared *I-DLV+MS* against the latest available version of ME-ASP[1]), the state-of-the-art among multi-engine ASP solvers, the winner of the 6th ASP Competition.

Figure 14.2 reports the cactus plot comparing *I-DLV+MS* and ME-ASP. Interestingly, even though *I-DLV+MS* is a prototypical system, it performed well overall: indeed, it solved 327 instances, 49 more than ME-ASP.

The two systems are similar, yet there are some key differences; in particular, the main differences are due to the nature and the number of systems used. First of all, ME-ASP computes features of the input program at hand over a ground program produced by *gringo* system, while *I-DLV+MS* makes use of *I-DLV*. Furthermore, the main interesting difference is due to the fact that ME-ASP manages five solvers, far more than the mere two taken into account by *I-DLV+MS*. The strategy of using a large pool of solver engines, as in the case of ME-ASP, allows the system to solve a significant number of instances uniquely, i.e., instances solved by only one solver, as the different engines use evaluation strategies that can be substantially different. Nevertheless, such differences imply that a high price is paid in case of a wrong choice. On the other hand, when the space for choices is narrowed, the probability of picking the wrong solver decreases, and this might lead to a more consistent behavior, as in the case of *I-DLV+MS*.

[1]http://aspcomp2015.dibris.unige.it/participants

Chapter 15

Conclusion

The main advantage of Answer Set Programming is the high expressive power of its language, allowing to represent complex problems in a straightforward, concise and elegant way. Due to the steady work of the scientific community, that led to significant improvements of systems supporting the formalism and multiple language extensions, ASP has been used in many different domains, thus shifting ASP from a strict theoretical scope to more practical scenarios. However, such use of ASP made several challenges arise; for instance, it became evident the need for proper tools and interoperability mechanisms that ease the development of ASP-based applications, helping the integration with external systems and different source of knowledge; also, the effective use of the potential of ASP in new real contexts requires significant improvements of performance.

The work presented in this thesis aims at addressing the aforementioned challenges, and introduces new tools and techniques for easing the application of ASP. In particular, the contributions of this thesis can be summarized as follows:

- We present EMBASP: a framework for the integration of ASP in external systems for general applications to different platforms and ASP reasoners. The framework features explicit mechanisms for two-way translations between strings recognisable by ASP solvers and objects in the programming language of choice. In order to illustrate the use of EMBASP, we present an actual Java implementation and several specialized libraries for the state-of-the-art ASP systems *clingo*, *DLV* and *DLV2*, on desktop platforms and another specialization for *DLV* for the Android mobile platform, showing four applications developed using EMBASP that prove the effectiveness of the framework.

- We extend the language of ASP in order to make it capable of handling external computations with explicit calls to Python scripts via external atoms; furthermore, we define interoperability mechanisms for the connection with relational and graph databases via explicit directives for importing/exporting data. We implement this features into the *I-DLV* system, the new instantiator of *DLV*, with the main purpose of obtaining a flexible tool for experimenting with ASP and its applications.

- We work on techniques and tools for improving the performance of ASP computation; in particular, we implement our proposals into two new ASP systems: *DLV2* and *I-DLV+MS*. *DLV2* updates *DLV* with modern evaluation techniques, combining *I-DLV* with the solver *wasp*. Also, *DLV2* extends the core modules by application-oriented features. *I-DLV+MS* is a new ASP system that integrates *I-DLV* with an automatic solver selector for inductively choose the best solver, depending on some inherent features of the instantiation produced by *I-DLV*. To this end, we define a specific set of features, and then carry out an experimental analysis for computing them over the ground versions of benchmarks submitted to the 6th ASP competition.

As a future work we would like to test the EMBASP framework over different platforms and solvers and properly evaluate performances. Although the framework has been mainly conceived for fostering the usage of ASP, its abstract core makes it also adaptable to other declarative knowledge representation formalisms; indeed, we introduce a proper extension supporting the *PDDL* [64, 63] planning language in [43], far beyond logic formalisms similar to ASP.

Moreover, we plan to increase the functionalities of *I-DLV* system related to the language extensions for easing the interoperability with the external system. More in detail, more native directives for interoperating with external data will be added with a tighter integration with the ASP solver *wasp* in the new ASP system *DLV2*.

Notwithstanding the good performance of *I-DLV+MS*, the system is still in a prototype phase. As future work, we aim to test additional supervised learning method and also several frameworks for automatic algorithm configuration, like Autofolio [77] or Auto-WEKA [69]. We also plan to significantly extend experiments over additional domains and analyze the possible overfitting of the model and try different splits of the dataset for the train and test set among the available problems. Moreover, we aim to both include additional ASP solvers with different parameterizations, and explore more features for improving the classification capabilities and achieve better overall performance.

In addition, we are studying the possibility of taking advantage from machine-learning techniques for improving performance of ASP grounding engines; in particular, we plan to develop a built-in automatic algorithm selector within the *I-DLV* system (which *I-DLV+MS* is based on), thus opening up the possibility to dynamically adapt all the optimization strategies to the problem at hand.

Appendix A

Embed ASP in Android Application

This appendix reports the full code of the example proposed in Chapter 7.

The Android app is divided into two classes:

- *Cell* class represent a single cell of the Sudoku schema;
- *MainActivity* class is the activity called when the application is launched.

In the following sections, we show the full code of the two classes that implement an Android application for solving Sudoku puzzles.

A.1 The Cell Class

By means of the annotation-guided mapping of EMBASP, the initial schema can be expressed in forms of Java objects. Then, a Cell object represents a cell in a Sudoku schema. Therefore a Cell class has three fields representing the row, the column and the value, i.e. a number between 1 and 9. Also, the mapped object follows the JavaBean convention, hence the below class implements the *getter* and *setter methods*.

```
1 @Id("cell")
2 public class Cell {
3
4   @Param(0)
5   private int row;
6
7   @Param(1)
8   private int column;
9
10  @Param(2)
```

```
11   private int value;
12
13    public Cell() {
14       row = column = value = 0;
15   }
16
17   public Cell(final int row, final int column, final int value) {
18       this.row = row;
19       this.column = column;
20       this.value = value;
21   }
22
23   public int getColumn() {
24       return column;
25   }
26
27   public int getRow() {
28       return row;
29   }
30
31   public int getValue() {
32       return value;
33   }
34
35   public void setColumn(final int column) {
36       this.column = column;
37   }
38
39   public void setRow(final int row) {
40       this.row = row;
41   }
42
43   public void setValue(final int value) {
44       this.value = value;
45   }
46
47
48 }
```

A.2 The Main Activity

The MainActivity Class contains the main functions of the app. The first function called when the application is run is `onCreate` and performs the basic application startup logic. The function `displaySolution` display on the screen the value of the Sudoku schema stored in the `sudokuMatrix`, that is initialized with an initial schema with cells initially empty are represented by positions containing zero. Furthermore,the utility function `getEncodingFromResources` loads the ASP program from the *Android Resources*

folder,while the method `startReasoning` is in charge of managing the reasoning and is invoked in response to a “click” event called by the `onClick` function. After the reasoning process a callback object `MyCallback` is in charge of fetching the output that is retrieved directly in the form of Java objects.

```
1 package embasp.mat.unical.it.embasp_android_sudoku;
2
3 import android.os.Bundle;
4 import android.support.v7.app.AppCompatActivity;
5 import android.view.View;
6 import android.widget.Button;
7 import android.widget.TextView;
8
9 import java.io.BufferedReader;
10 import java.io.IOException;
11 import java.io.InputStream;
12 import java.io.InputStreamReader;
13
14 import it.unical.mat.embasp.base.Callback;
15 import it.unical.mat.embasp.base.Handler;
16 import it.unical.mat.embasp.base.InputProgram;
17 import it.unical.mat.embasp.base.Output;
18 import it.unical.mat.embasp.languages.asp.ASPInputProgram;
19 import it.unical.mat.embasp.languages.asp.AnswerSet;
20 import it.unical.mat.embasp.languages.asp.AnswerSets;
21 import it.unical.mat.embasp.platforms.android.AndroidHandler;
22 import it.unical.mat.embasp.specializations.dlv.android.
      DLVAndroidService;
23
24 public class MainActivity extends AppCompatActivity {
25
26     public final int N=9;
27     private String encodingResource="sudoku";
28     private int[][] sudokuMatrix =     {{1,0,0,0,0,7,0,9,0},
29                                         {0,3,0,0,2,0,0,0,8},
30                                         {0,0,9,6,0,0,5,0,0},
31                                         {0,0,5,3,0,0,9,0,0},
32                                         {0,1,0,0,8,0,0,0,2},
33                                         {6,0,0,0,0,4,0,0,0},
34                                         {3,0,0,0,0,0,0,1,0},
35                                         {0,4,1,0,0,0,0,0,7},
36                                         {0,0,7,0,0,0,3,0,0}};
37     private Handler handler;
38
39     private class MyCallback implements Callback {
40         @Override
41         public void callback(Output o) {
42              if(!(o instanceof AnswerSets))return;
43             AnswerSets answerSets=(AnswerSets)o;
44             if(answerSets.getAnswersets().size()==0)return;
45             AnswerSet as = answerSets.getAnswersets().get(0);
```

```
46
47             try {
48                 for(Object obj:as.getAtoms()) {
49                     Cell cell = (Cell) obj;
50                     sudokuMatrix[cell.getRow()][cell.getColumn()]
51                                        =cell.getValue();
52                 }
53             } catch (Exception e) {
54                 e.printStackTrace();
55             }
56
57             displaySolution();
58         }
59     }
60
61     @Override
62     protected void onCreate(Bundle savedInstanceState) {
63         super.onCreate(savedInstanceState);
64         setContentView(R.layout.activity_main);
65         handler = new AndroidHandler(getApplicationContext(),
66                                  DLVAndroidService.class);
67         displaySolution();
68     }
69
70     private String getEncodingFromResources(){
71         InputStream ins = getResources().openRawResource(
72                 getResources().getIdentifier(encodingResource,
73                         "raw", getPackageName()));
74         BufferedReader reader=
75                        new BufferedReader(new InputStreamReader(ins));
76         String line="";
77         StringBuilder builder=new StringBuilder();
78         try {
79             while ((line = reader.readLine()) != null) {
80                 builder.append(line);
81                 builder.append("\n");
82             }
83         } catch (IOException e) {
84             e.printStackTrace();
85         }
86         return builder.toString();
87     }
88
89     public void startReasoning(){
90         InputProgram inputProgram=new ASPInputProgram();
91         for ( int i = 0; i < N; i++)
92             for ( int j = 0; j < N; j++)
93                 try {
94                     if(sudokuMatrix[ i ] [ j ]!=0) {
95                         inputProgram.addObjectInput(
96                                 new Cell(i, j, sudokuMatrix[i][j]));
97                     }
98                 } catch (Exception e) {
```

```
 99                    e.printStackTrace();
100                }
101        handler.addProgram(inputProgram);
102
103        String sudokuEncoding = getEncodingFromResources();
104        handler.addProgram(new InputProgram(sudokuEncoding));
105
106        Callback callback = new MyCallback();
107        handler.startAsync(callback);
108     }
109
110    public void onClick(final View view){
111        Button button=(Button) findViewById(R.id.button);
112        button.setEnabled(false);
113        startReasoning();
114    }
115
116    private void displaySolution() {
117        String out="";
118        for ( int i = 0; i < N; i++) {
119            for (int j = 0; j < N; j++) {
120                out += sudokuMatrix[i][j]+"␣";
121            }
122            out+="\n";
123        }
124
125        final String finalOut=out;
126        runOnUiThread(new Runnable() {
127            @Override
128            public void run() {
129                TextView text = (TextView) findViewById(R.id.result);
130                text.setText(finalOut);
131            }
132        });
133    }
134
135}
```

Appendix B

External Computation and Sources of Knowledge Encodings

In this appendix, we give details to some of those benchmark encodings which are not described in Sections 10.4 and 11.2.

B.1 External Computations Benchmarks

In Section 10.4 we compare *I-DLV* with other currently available systems supporting external computation via Python to assess their efficiency at integrating external computations.

Then, we compare *I-DLV* against: the ASP grounder *gringo* [54] and the *dlvhex* [38]. In order to compare the different systems, we use a distinct ASP encoding for each of them due to the different syntax of the external atoms implemented in each system.

We compare the systems on two set of benchmarks:

- the first set of benchmarks is focused on the spatial representation and reasoning domain originally appeared in [98];
- three further ad-hoc domains:
  - concatenation of two randomly-generated strings with arbitrary lengths varying from 1000 to 3000 chars;
  - generation of first k prime numbers with k ranging from 0 to 100000;
  - the recursive reachability problem, with edges retrieved via Python scripts.

In the next sections we propose the ASP encodings used for the three ad-hoc problems which are not described in the existing literature.

String Concatenation

In the paragraphs below, we present for each system the related ASP encoding that solve the concatenation of two randomly-generated strings using the *concat* external atom. It receives as input two strings and compute the concatenation. Notably, the input strings are given by the *EDB* predicate *b*.

- ***I-DLV***

$$a(X,Y,Z) :- \ b(X),\ b(Y),\ \&concat(X,Y;Z).$$

- ***dlvhex***

$$a(X,Y,Z) :- b(X),\ b(Y),\ \&concat[X,Y](Z).$$

- ***gringo***

$$a(X,Y,@concat(X,Y)) :- b(X),\ b(Y).$$

First k-prime

The below ASP encodings solve the problem of generating the first k-prime number. The fact $num(K)$ is the only input of the program and in particular: the external atom *range* returns a range of number between 2 to K and the external atom *prime* takes as input a number and checks if it is prime. Moreover, the second rule, calculates how many prime numbers are in the range 2 to k.

- ***I-DLV***

$$prime(Y) :- num(K),\ \&range(K;Y),\ \&prime(Y;).$$
$$primeN(Z) :- \#count\{X : prime(X)\} = Z.$$

- ***dlvhex***

$$prime(Y) :- num(K),\ \&range[K](Y),\ \&prime[Y](Z),\ Z = 1.$$
$$primeN(Z) :- \#count\{X : prime(X)\} = Z.$$

- ***gringo***

$$prime(Y) \mathrel{:-} num(K),\ \ @range(K) = Y,\ \ @prime(Y) = 1.$$
$$primeN(Z) \mathrel{:-} \#count\{X : prime(X)\} = Z.$$

Reachability

The Deductive Database problem *Reachability*, already presented in Section 4.1, computes all pairs of reachable nodes in a given graph G. However, in next paragraph, we encode the reachability problem with edges retrieved via external atoms.

- ***I-DLV***

$$edge(X,Y) \mathrel{:-} \&getEdges(;X,Y).$$
$$reachable(X,Y) \mathrel{:-} edge(X,Y).$$
$$reachable(X,Y) \mathrel{:-} reachable(Z,Y),\ \ edge(X,Z).$$

- ***dlvhex***

$$edge(X,Y) \mathrel{:-} \&getEdges[](X,Y).$$
$$reachable(X,Y) \mathrel{:-} edge(X,Y).$$
$$reachable(X,Y) \mathrel{:-} reachable(Z,Y),\ \ edge(X,Z).$$

- ***gringo***

$$edge(X,Y) \mathrel{:-} (X,Y) = @getEdges().$$
$$reachable(X,Y) \mathrel{:-} edge(X,Y).$$
$$reachable(X,Y) \mathrel{:-} reachable(Z,Y),\ \ edge(X,Z).$$

For the sake of completeness in the following we report the Python code of the relative external atoms for *I-DLV*, *dlvhex* and *gringo* systems. Notably, *edges* variables are list of tuples where a tuple represents an edge that changes according to the input instance.

- ***I-DLV***

```
1  def concat(X,Y):
2    Z=str(X)+str(Y)
3    return Z
```

```
1  def range(X):
2    return range(2,X)
3
4  def prime(X):
5    if X==2:
6      return True
7    for i in range(2,X):
8      if X % i == 0:
9        return False
10   return True
```

```
1  def getEdges():
2    return edges
3  edges=[...]
```

- ***dlvhex***

```
1  import dlvhex
2
3  def concat(tup):
4    ret = ""
5    for x in tup:
6      ret = ret + x.value()
7    dlvhex.output((ret, ))
8
9  def register():
10   dlvhex.addAtom("concat",(dlvhex.TUPLE, ),1
       )
```

```
1  import dlvhex
2
3  def r(X):
```

```
4       y = int(X.value())
5       for i in range(2,y):
6         dlvhex.output((i,))
7
8     def prime(X):
9       y = int(X.value())
10      if y==2:
11        dlvhex.output((int(1),))
12        return
13      for i in range(2,y):
14        if y % i == 0:
15          dlvhex.output((int(0),))
16          return
17      dlvhex.output((int(1),))
18
19    def register():
20      dlvhex.addAtom("r",(dlvhex.CONSTANT,),1)
21      dlvhex.addAtom("prime",(dlvhex.CONSTANT,)
          ,1)
```

```
1     import dlvhex
2     def getEdges():
3       for e in edges:
4         dlvhex.output((e[0],e[1]))
5
6     def register():
7       dlvhex.addAtom("getEdges",(),2)
8     edges=[...]
```

- ***gringo***

```
1     #script (python)
2     def concat(X,Y):
3       Z=str(X)+str(Y)
4       return Z
5     #end.
```

```
1   #script (python)
2   def range(X):
3     return range(2,X)
4
5   def prime(X):
6     if X==2:
7       return True
8     for i in range(2,X):
9       if X % i == 0:
10        return False
11    return True
12  #end.
```

```
1   #script (python)
2   def getEdges():
3     return edges
4   edges=[...]
5   #end.
```

B.2 Interoperability Benchmarks

In Section 11.2 we analyze the effective gain in performance obtainable with a native support of SQL/SPARQL local import directives against the same directives implemented via Python scripts.

The first benchmark we import data from a Relational Database using SQL statements. The ASP encoding that natively import data from a database is a single line directive

```
#import_sql(DLVDB0,"root","root",
      "SELECT * FROM T0 LIMIT K;",
         p1,type: U_INT, CONST, Q_CONST).
```

where K ranging from 100000 to 1000000 according to the instance, $p1$ indicates the predicate whose extension will be enriched with the result of the query and $type : U_INT, CONST, Q_CONST$ are the mapping policy where the first field of the table $T0$

is imported as an integer, the second field as symbolic constant and the remaining values are converted to quoted strings.

Meanwhile, the ASP program that imports data from relational database using an external atom is defined below.

$$p1(X,Y,Z) :- \&importDB(\text{"}DLVDB0\text{"}, \text{"}SELECT * FROM T0 LIMIT K;\text{"}; X,Y,Z).$$

Moreover, the relative Python code that defines the external atom *importDB* is:

```
1   import pymysql
2   from pymysql import connect, err, sys, cursors
3
4   def importDB(db,q):
5
6     conn = connect( host = 'localhost',
7                     port = 3306,
8                     user = 'root',
9                     passwd = 'root',
10                    db = db );
11
12    cursor = conn.cursor( cursors.DictCursor );
13
14    cursor.execute( q )
15    data = cursor.fetchall()
16    l = []
17    for row in data:
18      p1 = row["F0"]
19      p2 = row["F1"]
20      p3 = row["F2"]
21      l.append((p1,p2,p3))
22    return l
```

In the second benchmark, we import data from a local RDF/XML file and in particular, OWL University ontologies.

The ASP encoding that natively import data using SPARQL query is composed of n directives, where n is the number of universities to import:

```
#import_local_sparql("file:University1.owl",
"PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub:  <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?X WHERE {{?X rdf:type ub:GraduateStudent.}}
UNION {?X rdf:type ub:UndergraduateStudent.} }",
student,1, type:Q_CONST).
...
#import_local_sparql("file:UniversityN.owl",
"PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub:  <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?X WHERE {{?X rdf:type ub:GraduateStudent.}}
UNION {?X rdf:type ub:UndergraduateStudent.} }",
student,1, type:Q_CONST).
```

where the SPARQL queries import graduate and undergraduate students from a different number of universities in predicate *student* with the first field as constant

Meanwhile, the ASP program that imports using an external atom is defined below.

$$
\begin{aligned}
&student(X0) :- \; university(X), \; \&sparql(X, \\
&\quad "PREFIX\ rdf :< http : //www.w3.org/1999/02/22-rdf-syntax-ns\# > \\
&\quad PREFIX\ ub :< http : //swat.cse.lehigh.edu/onto/univ-bench.owl\# > \\
&\quad SELECT\ ?X\ WHERE\ \{\{?X\ rdf : type\ ub : GraduateStudent.\} \\
&\quad UNION\ \{?X\ rdf : type\ ub : UndergraduateStudent.\}\}";X0).
\end{aligned}
$$

The relative Python code:

```
1     import rdflib
2
3     def sparql(dataset, query):
4         g = rdflib.Graph()
5         g.parse(dataset)
6         qres = g.query(query)
7         return qres
```

Appendix C

External Propagators in *DLV2*

DLV2 offers the possibility to customize heuristics of the system and extend its solving capabilities by means of directives. For instance, the input program can be enriched by external propagators that allow a user to embed new external propagators (using Python language) in the solver, as we show in Section 13.2.

For a better understanding of the external propagators, in the following, we report the full ASP and Python code of the example 13.1 proposed in Chapter 13.

$\#propagator(@file = "prop.py", @elements = X, value : value(X); X, in : in(X)).$

$edb(1..2).$
$\{value(X) : edb(X)\} = 1.$
$\{in(X) : edb(X)\}.$

The directive #*propagator* calls the python file *prop.py* reported below.

```
1    import wasp
2
3    # dictionary 'elements' is created by DLV2
4
5    valueAtoms = [key for key in elements.keys()
6                  if elements[key][1] == "value"]
7    inAtoms = [key for key in elements.keys()
8               if elements[key][1] == "in"]
```

```
 9
10    answer = []
11    valueAtom = None
12
13    def checkAnswerSet(*answer_set):
14        global answer
15        global valueAtom
16
17        count = sum([1 for i in inAtoms
18                 if answer_set[i] > 0])
19        value = None
20        for i in valueAtoms:
21            if answer_set[i] > 0:
22                value = elements[i][0]
23                valueAtom = i
24                break
25
26        # it's an answer set!
27        if count != value: return 1
28
29        # not an answer set!
30        answer = answer_set
31        return 0
32
33    def getReasonsForCheckFailure():
34        global answer
35        global valueAtom
36
37        reason = [-valueAtom]
38        reason.extend([(-i if answer[i] > 0
39                    else i) for i in inAtoms])
40        return wasp.
41            createReasonsForCheckFailure([reason])
```

The above propagator simulates the following constraint:

$$:\!-\, value(X),\ \#count Y : in(Y) = X.$$

where lines $5-8$ find in the dictionary `elements` all the ground atoms with predicate *value* and *in* storing them in two separate lists. Then the function `checkAnswerSet` implements the external propagator and receives an interpretation `answer_set` and returns true if it is an answer set, false otherwise. The interpretation `answer_set` is a dictionary of literal id and assignment, that can be 1 if the literal is true, false if it 0. Indeed, in lines $17-18$ count the true *in* atoms respect the interpretation `answer_set` (simulating the #count aggregate) and lines $20-24$ find the value of the *value* atom with respect to the interpretation `answer_set`. Notably, we can break the iteration to the first value found because according to the ASP encoding only one *value* can be true. Therefore, if the count of *in* atoms is equal to *value* atom `answe_set` is a possible answer set and the function returns true, otherwise, it is not, and returns false. The global variables defined in lines $10-11$ are used in function `getReasonsForCheckFailure` invoked after a failure of `checkAnswerSet` function and returns a list of clauses modeling the reasons for the failure and consequently in lines $37-41$ return the value of the true literal *value* and the *in* literals, previously stored in the global variables, lines 23 and 30.

Bibliography

[1] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP system DLV2. In Balduccini and Janhunen [8], pages 215–221.

[2] Mario Alviano and Carmine Dodaro. Anytime answer set optimization via unsatisfiable core shrinking. *TPLP*, 16(5-6):533–551, 2016.

[3] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In Calimeri et al. [30], pages 40–54.

[4] Mario Alviano, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets for disjunctive datalog programs. *Artificial Intelligence*, 187:156–192, 2012.

[5] Markus Aschinger, Conrad Drescher, Gerhard Friedrich, Georg Gottlob, Peter Jeavons, Anna Ryabokon, and Evgenij Thorstensen. Optimization methods for the partner units problem. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 4–19, 2011.

[6] Yuliya Babovich and Marco Maratea. Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. Available on http://www.cs.utexas.edu/users/tag/cmodels.html, 2003.

[7] Marcello Balduccini. Industrial-size scheduling with asp+ cp. *Logic Programming and Nonmonotonic Reasoning*, pages 284–296, 2011.

[8] Marcello Balduccini and Tomi Janhunen, editors. *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, volume 10377 of *Lecture Notes in Computer Science*. Springer, 2017.

[9] Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[10] Rachel Ben-Eliyahu and Luigi Palopoli. Reasoning with Minimal Models: Efficient Algorithms and Applications. In *Proceedings Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, pages 39–50, 1994.

[11] Piero Bonatti, Francesco Calimeri, Nicola Leone, and Francesco Ricca. Answer set programming. In *A 25-year perspective on logic programming*, pages 159–182. Springer-Verlag, 2010.

[12] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[13] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[14] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and Weak Constraints in Disjunctive Datalog. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in AI (LNAI)*, pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.

[15] Francesco Calimeri, Susanna Cozza, and Giovambattista Ianni. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence*, 50(3–4):333–361, 2007.

[16] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. An ASP System with Functions, Lists, and Sets. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning — 10th International Conference (LPNMR 2009)*, volume 5753 of *Lecture Notes in Computer Science*, pages 483–489. Springer Verlag, September 2009.

[17] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. Asp-core-2: Input language format, 2012. https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf.

[18] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2: 4th ASP Competition Official Input Language Format, 2013. https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.01c.pdf.

[19] Francesco Calimeri, Michael Fink, Stefano Germano, Andreas Humenberger, Giovambattista Ianni, Christoph Redl, Daria Stepanova, Andrea Tucci, and Anton Wimmer. Angry-hex: An artificial player for angry birds based on declarative knowledge bases. *IEEE Trans. Comput. Intellig. and AI in Games*, 8(2):128–139, 2016.

[20] Francesco Calimeri, Davide Fuscà, Stefano Germano, Simona Perri, and Jessica Zangari. Embedding ASP in mobile systems: discussion and preliminary implementations. In *Proceedings of the Eighth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2015), workshop of the 31st International Conference on Logic Programming (ICLP 2015)*, August 2015.

[21] Francesco Calimeri, Davide Fuscà, Stefano Germano, Simona Perri, and Jessica Zangari. Boosting the development of asp-based applications in mobile and general scenarios. In *AI*IA 2016: Advances in Artificial Intelligence - XVth International Conference of the Italian Association for Artificial Intelligence, Genova, Italy, November 29 - December 1, 2016, Proceedings*, pages 223–236, 2016.

[22] Francesco Calimeri, Davide Fuscà, Giovambattista Ianni, Giovanni Melissari, and Jessica Zangari. The new DLV Grounder: External Computations, Interoperability and Customizability. In *Proceedings of the 4th International Workshop on Grounding and Transformations for Theories with Variables (GTTV2017), workshop of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR17)*, 2017.

[23] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale*, 11(1):5–20, 2017.

[24] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. External computations and interoperability in the new dlv grounder. In *AI* IA 2017 Advances in Artificial Intelligence*. To appear.

[25] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. I-DLV+MS: preliminary report on an automatic ASP solver selector. In *Proceedings of the 24th RCRA International Workshop on "Experimental Evaluation of*

*Algorithms for Solving Problems with Combinatorial Explosion" (RCRA2017), workshop of the 16th Conference of the Italian Association for Artificial Intelligence (AI*IA 2017)*, to appear.

[26] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artificial Intelligence*, 231:151–181, 2016.

[27] Francesco Calimeri and Giovambattista Ianni. External sources of computation for Answer Set Solvers. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 105–118. Springer Verlag, September 2005.

[28] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third open answer set programming competition. *TPLP*, 14(1):117–135, 2014.

[29] Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. The third answer set programming competition: Preliminary report of the system competition track. In *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2011.

[30] Francesco Calimeri, Giovambattista Ianni, and Miroslaw Truszczynski, editors. *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, volume 9345 of *LNCS*. Springer, 2015.

[31] Francesco Calimeri and Francesco Ricca. On the application of the answer set programming system dlv in industry: a report from the field. *Book Reviews*, 2013(03), 2013.

[32] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. GASP: Answer set programming with lazy grounding. *Fundamenta Informaticae*, 96(3):297–322, 2009.

[33] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[34] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *Journal of Experimental & Theoretical Artificial Intelligence*, 21(2):79–121, 2009.

[35] Christian Drescher, Martin Gebser, Torsten Grote, Benjamin Kaufmann, Arne König, Max Ostrowski, and Torsten Schaub. Conflict-Driven Disjunctive Answer Set Solving. In Gerhard Brewka and Jérôme Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, pages 422–432, Sydney, Australia, 2008. AAAI Press.

[36] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003.*, pages 502–518. LNCS Springer, 2003.

[37] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving using the DLV system. In *Logic-based artificial intelligence*, pages 79–103. Springer, 2000.

[38] Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, Christoph Redl, and Peter Schüller. A model building framework for answer set programming with external computations. *TPLP*, 16(4):418–464, 2016.

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

[40] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer Set Programming: A Primer. In *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School - Tutorial Lectures*, pages 40–110, Brixen-Bressanone, Italy, August-September 2009.

[41] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *IJCAI*, volume 5, pages 90–96, 2005.

[42] Thomas Eiter, Giovambattista Ianni, Hans Tompits, and Roman Schindlauer. Effective Integration of Declarative Rules with External Evaluations for Semantic

Web Reasoning. In *Proceedings of the 3rd European Semantic Web Conference (ESWC 2006)*, pages 273–287, June 2006.

[43] EMBASP. https://www.mat.unical.it/calimeri/projects/embasp/.

[44] Esra Erdem, Erdi Aker, and Volkan Patoglu. Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics*, 5(4):275–291, Oct 2012.

[45] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. 37:53, 10 2016.

[46] Esra Erdem, Volkan Patoglu, Zeynep G Saribatur, Peter Schüller, and Tansel Uras. Finding optimal plans for multiple teams of robots through a mediator: A logic-based approach. *Theory and Practice of Logic Programming*, 13(4-5):831–846, 2013.

[47] Davide Fuscà, Stefano Germano, Jessica Zangari, Marco Anastasio, Francesco Calimeri, and Simona Perri. A framework for easing the development of applications embedding answer set programming. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom*, pages 5–7, 2016.

[48] Davide Fuscà, Stefano Germano, Jessica Zangari, Francesco Calimeri, and Simona Perri. Answer set programming and declarative problem solving in game ais. In *PAI@ AI*IA*, pages 81–88, 2013.

[49] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: elements of, 1994.

[50] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, 2000.

[51] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *ICLP TCs*, pages 2:1–2:15, 2016.

[52] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Javier Romero, and Torsten Schaub. Progress in clasp series 3. In Calimeri et al. [30], pages 368–383.

[53] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo= asp+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*, 2014.

[54] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in *gringo* series 3. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*, pages 345–351. Springer, 2011.

[55] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 386–392. Morgan Kaufmann Publishers, January 2007.

[56] Martin Gebser, Marco Maratea, and Francesco Ricca. The design of the sixth answer set programming competition – report. In *LPNMR*, volume 9345 of *LNCS*, pages 531–544, 2015.

[57] Martin Gebser, Marco Maratea, and Francesco Ricca. What's hot in the answer set programming competition. In Dale Schuurmans and Michael P. Wellman, editors, *Proc. of the 13th AAAI Conference on Artificial Intelligence, Feb 12-17, 2016, Phoenix, Arizona, USA.*, pages 4327–4329. AAAI Press, 2016.

[58] Martin Gebser, Marco Maratea, and Francesco Ricca. The design of the seventh answer set programming competition. In Balduccini and Janhunen [8], pages 3–9.

[59] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271, Tempe, Arizona, May 2007. Springer Verlag.

[60] Martin Gebser, Torsten Schaub, Sven Thiele, and Philippe Veber. Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming*, 11(2-3):323–360, 2011.

[61] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, WA, Aug 15-19, 1988 (2 Volumes)*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[62] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4):365–385, 1991.

[63] Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artif. Intell.*, 173(5-6):619–668, 2009.

[64] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL — The Planning Domain Definition language. Technical report, Yale Center for Computational Vision and Control, October 1998. Available at http://www.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz.

[65] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[66] Salvatore Maria Ielpa, Salvatore Iiritano, Nicola Leone, and Francesco Ricca. An ASP-Based System for e-Tourism. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2009*), volume 5753 of *Lecture Notes in Computer Science*, pages 368–381. Springer, 2009.

[67] Tomi Janhunen. Some (in) translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1-2):35–86, 2006.

[68] Tomi Janhunen and Ilkka Niemelä. Gnt - a solver for disjunctive logic programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, volume 2923 of *Lecture Notes in AI (LNAI)*, pages 331–335, Fort Lauderdale, Florida, USA, January 2004. Springer.

[69] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *Journal of Machine Learning Research*, 17:1–5, 2016.

[70] Claire Lefèvre and Pascal Nicolas. The first version of a new asp solver : Asperix. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning — 10th International Conference (LPNMR 2009)*, volume 5753 of *Lecture Notes in Computer Science*, pages 522–527. Springer Verlag, September 2009.

[71] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Georg Gottlob, Giovambattista Ianni, Giuseppe Ielpa, Christoph Koch, Simona Perri, and Axel Polleres. The DLV System. In Sergio Flesca, Sergio Greco, Giovambattista Ianni, and Nicola Leone, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA)*, volume 2424 of *Lecture Notes in Computer Science*, pages 537–540, Cosenza, Italy, September 2002. (System Description).

[72] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, 2006.

[73] Nicola Leone and Francesco Ricca. Answer set programming: a tour from the basics to advanced development tools and industrial applications. In *Reasoning Web. Web Logic Rules*, pages 308–326. Springer, 2015.

[74] Yuliya Lierler. Disjunctive Answer Set Programming via Satisfiability. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer Verlag, September 2005.

[75] Vladimir Lifschitz. Answer Set Planning. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

[76] Fangzhen Lin and Yuting Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, Edmonton, Alberta, Canada, 2002. AAAI Press / MIT Press.

[77] Marius Lindauer, Holger H Hoos, Frank Hutter, and Torsten Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.

[78] Marco Maratea, Luca Pulina, and Francesco Ricca. A multi-engine approach to answer-set programming. *TPLP*, 14(6):841–868, 2014.

[79] Victor W. Marek and Mirosław Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In Krzysztof R. Apt, V. Wiktor Marek, Mirosław

Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[80] Ilkka Niemelä. Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[81] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A System for Answer Set Programming. In Chitta Baral and Mirosław Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, April 2000. Online at http://xxx.lanl.gov/abs/cs/0003033v1.

[82] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[83] Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[84] Luca Pulina and Armando Tacchella. A multi-engine solver for quantified boolean formulas. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 574–589. Springer, 2007.

[85] J Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[86] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[87] Jakob Rath and Christoph Redl. Integrating answer set programming with object-oriented languages. In *International Symposium on Practical Aspects of Declarative Languages*, pages 50–67. Springer, 2017.

[88] Francesco Ricca. The DLV Java Wrapper. In Marina de Vos and Alessandro Provetti, editors, *Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation*, pages 305–316, Messina, Italy, September 2003. Online at http://CEUR-WS.org/Vol-78/.

[89] Francesco Ricca, Giovanni Grasso, Mario Alviano, Marco Manna, Vincenzino Lio, Salvatore Iiritano, and Nicola Leone. Team-building with answer set programming in the gioia-tauro seaport. *Theory and Practice of Logic Programming.* Cambridge University Press, 12(3):361–381, 2012.

[90] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[91] Peter Schüller. Modeling variations of first-order horn abduction in answer set programming. *Fundamenta Informaticae*, 149(1-2):159–207, 2016.

[92] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[93] Tommi Syrjänen. Omega-restricted logic programs. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, volume 2173 of *Lecture Notes in Computer Science*, pages 267–279. Springer, 2001.

[94] Matthias Thimm. Tweety: A comprehensive collection of java libraries for logical aspects of artificial intelligence and knowledge representation. In *KR*, 2014.

[95] Laura Toni, Ramon Aparicio-Pardo, Gwendal Simon, Alberto Blanc, and Pascal Frossard. Optimal set of video representations in adaptive streaming. In *Proceedings of the 5th ACM Multimedia Systems Conference*, pages 271–282. ACM, 2014.

[96] Nam Tran and Chitta Baral. Hypothesizing about signaling networks. *Journal of Applied Logic*, 7(3):253–274, 2009.

[97] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[98] Przemyslaw Andrzej Walega, Carl P. L. Schultz, and Mehul Bhatt. Non-Monotonic Spatial Reasoning with Answer Set Programming Modulo Theories. *CoRR*, abs/1606.07860, 2016.

[99] Antonius Weinzierl. Blending lazy-grounding and cdnl search for answer-set solving. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 191–204. Springer, 2017.

[100] Johan Wittocx and Marc Denecker. The idp system: a model expansion system for an extension of classical logic. In *Logic and Search, Computation of Structures from Declarative Descriptions (LaSh)*, pages 153–165, 2008.

[101] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *CoRR*, abs/1111.2249, 2011.

[102] Shiqi Zhang, Mohan Sridharan, and Jeremy L Wyatt. Mixed logical inference and probabilistic planning for robots in unreliable worlds. *IEEE Transactions on Robotics*, 31(3):699–713, 2015.