Università della Calabria
Dipartimento di Matematica e Informatica

Dottorato di Ricerca in Matematica e Informatica
XXXI ciclo

Tesi di Dottorato

Enhancing and Applying Answer Set Programming:
Lazy Constraints, Partial Compilation and Question Answering

Settore Scientifico Disciplinare INF/01 – INFORMATICA

Coordinatore: Ch.mo Prof. Nicola Leone

Supervisore: Prof. Francesco Ricca

Dottorando: Dott. Bernardo Cuteri

Contents

Sommario

Questo lavoro è incentrato sull'Answer Set Programming (ASP), il quale è un formalismo espressivo per la rappresentazione della conoscenza e il ragionamento automatico. Nel corso del tempo, ASP è stato sempre più dedicato alla risoluzione di problemi reali grazie alla disponibilità di sistemi efficienti. Questa tesi presenta due contributi in questo ambito: *(i)* nuove strategie per migliorare la valutazione dei programmi ASP, e *(ii)* un'applicazione di ASP per il Question Answering in Linguaggio Naturale.

Nel primo contributo studiamo alcuni casi in cui la strategia di valutazione classica di ASP fallisce a causa del cosiddetto *grounding bottleneck*. Dapprima ci concentriamo su casi in cui ci sono dei vincoli la cui istanziazione è problematica. Affrontiamo il problema usando propagatori ad-hoc e istanziatori lazy, e dimostriamo empiricamente sotto quali condizioni queste soluzioni sono efficaci. Due svantaggi delle tecniche basate sui propagatori/istanziatori sono la necessità di avere una conoscenza approfondita dei sistemi ASP e il fatto che l'implementazione sia a carico dell'utente e scritta in un lionguaggio non dichiarativo. Per superare questi svantaggi, abbiamo ideato una tecnica di compilazione parziale dei programmi ASP. Questa nuova tecnica consente di generare automaticamente i propagatori per alcune delle regole logiche di un programma. Un'analisi empirica mostra i benefici, in termini di prestazioni, che possono essere ottenuti introducendo la compilazione parziale nella valutazione dei programmi ASP. Per quanto ne sappiamo, questo è il primo lavoro su tecniche di compilazione per programmi ASP.

Per quanto riguarda la seconda parte della tesi, presentiamo un sistema di risposta alle domande in linguaggio naturale la cui implementazione è basata su ASP. Il sistema da noi realizzato trasforma le domande di input in query SPARQL che vengono eseguite su una base di conoscenza ontologica. Nel sistema sono integrati diversi modelli e strumenti allo stato dell'arte per il processamento del linguaggio naturale con particolare riferimento alla lingua italiana e al dominio dei beni culturali.

Il sistema che è stato da noi realizzato rappresenta il modulo software principale nell'ambito del progetto PIUCULTURA che è un progetto finanziato dal Ministero dello Sviluppo Economico italiano e il cui obiettivo è quello di promuovere e migliorare la fruizione dei beni culturali.

Abstract

This work is focused on Answer Set Programming (ASP), that is an expressive formalism for Knowledge Representation and Reasoning. Over time, ASP has been more and more devoted to solving real-world problems thanks to the availability of efficient systems. This thesis brings two main contributions in this context: *(i)* novel strategies for improving ASP programs evaluation, and *(ii)* a real-world application of ASP to Question Answering in Natural Language.

Concerning the first contribution, we study some cases in which classical evaluation fails because of the so-called *grounding bottleneck*. In particular, we first focus on cases in which the standard evaluation strategy is ineffective due to the grounding of problematic constraints. We approach the problem using custom propagators and lazy instantiators, proving empirically when this solution is effective, which is an aspect that was never made clear in the existing literature. Despite the development of propagators can be effective, it has two main disadvantages: it requires deep knowledge of the ASP systems, and the resulting solution is not declarative. We propose a technique for overcoming these issues which we call program compilation. In our approach, the propagators for some of the logic rules (not only for the constraints) of a program are generated automatically by a compiler. We provide some sufficient conditions for identifying the rules that can be compiled in an approach that fits a propagator-based system architecture. An empirical analysis shows the performance benefits obtained by introducing (partial) compilation into ASP programs evaluation. To the best of our knowledge, this is the first work on compilation-based techniques for ASP.

Concerning the second part of the thesis, we present the development of a Natural Language Question Answering System whose core is based on ASP. The proposed system gradually transforms input questions into SPARQL queries that are executed on an ontological knowledge base. The system integrates several state-of-the NLP models and tools with a special focus on the Italian language and the Cultural Heritage domain. ASP is used to classify questions from a syntactical point of view. The resulting system is the core module of the PIUCULTURA project, funded by the Italian Ministry of Economic Development, that has the aim to devise a system for promoting and improving the fruition of Cultural Heritage.

Introduction

Context and motivation. Answer Set Programming (ASP) is a powerful formalism that has roots in Knowledge Representation and Reasoning and is based on the stable model semantics [43, 16]. ASP is a viable solution for representing and solving many classes of problems thanks to its high expressive power and the availability of efficient systems [41]. Indeed, ASP has been successfully applied to several academic and industrial applications such as product configuration [50], decision support systems for space shuttle flight controllers [67], explanation of biomedical queries [31], construction of phylogenetic supertrees [51], data-integration [62], reconfiguration systems [6], and more. A key features of ASP consists in the capability to model hard combinatorial problems in a declarative and compact way.

But, how far can the spectrum of applicability of ASP be extended in real-world scenarios? Even though ASP is supported by efficient systems, is there still room for improvements?

On one hand, there is a general need for assessing ASP on a wider and wider spectrum of real-world applications. This is particularly useful to provide a more general understanding of the usability of ASP in practice. On the other hand, the evaluation of ASP programs is a complex issue, and there are still cases where ASP systems can be improved, especially in presence of the so-called *grounding-bottleneck* [18]. Thus there is the need to overtake their weak spots and possibly gain performance improvements to keep up with the challenges of emerging applications.

This thesis has two main goals in this context: *(i)* devise novel strategies for improving ASP programs evaluation, and *(ii)* applying ASP to the challenging problem of Question Answering in Natural Language. These two goals and the related contributions are introduced in the following and treated in detail in the two parts of this thesis. The two parts are related in the sense that they tackle two aspects that are both important to the de-

velopment of the ASP ecosystem. Indeed, theoretical/methodological results make real-worlds ASP applications possible, and on the other hand, real-world applications push the innovation of existing systems, methods and theories.

Part 1: Enhancing ASP Systems on Grounding-intensive Programs State-of-the-art ASP systems evaluate ASP programs in a two-step computation by first removing variables in the so-called *grounding phase* and then performing a stable models search on the resulting variable-free program in the so-called *solving phase*. This approach is typically referred to as the "ground+solve" approach. One of the main criticism to the "ground+solve" approach is that the two phases are completely separated and the grounding of an ASP program might require, both in theory and in practice, a memory space and an amount of time that exceed usability limits. Not only the grounding phase can be too costly to perform by itself, but also it can generate a propositional program that is too big for solvers to tackle. This problem is often referred to as the *grounding bottleneck* of ASP [18]. The grounding bottleneck has been the subject of several studies in recent years, and various alternative approaches to overcome it have been proposed such as ASPeRiX [55], GASP [20], and OMiGA [22]. Albeit alternative approaches obtained promising preliminary results, they cannot yet reach the performance of state of the art systems in many benchmarks [18, 55].

The first part of this thesis presents some alternative strategies for solving classes of ASP programs where the classical "ground+solve" evaluation fails or is inefficient. First, we focus on problematic ASP constraints and present three different evaluation strategies that are implemented as programmatic extensions of WASP [4], that is a state-of-the-art ASP solver. We provide an empirical evaluation conducted on real and synthetic benchmarks that confirms that the usage of custom propagators or lazy instantiators can be effective in presence of problematic ASP constraints. Moreover, we investigate the applicability of a portfolio approach to automatically select the best strategy when more than one solution is available. The result is positive in the sense that, empirically, a naive portfolio approach is faster than the best approach. Then, we present partial compilation of ASP programs. By using partial compilation we can generate lazy instantiators of constraints automatically. The applicability of partial compilation is not limited to constraints, but allows is extended to rules under some provided conditions. An

empirical analysis shows the performance benefits that can be obtained by introducing partial compilation into ASP programs evaluation.

Part 2: An Application of ASP to Closed-Domain Question Answering. Question Answering (QA) attempts to find direct answers to questions posed in natural language. There are two families of QA: in open domain QA there is no restriction to the domain of the questions, while in closed domain QA questions are bound to a specific domain.

In open domain QA, most systems are based on a combination of Information Retrieval and NLP techniques [47]. Such techniques are applied to a large corpus of documents: first attempting to retrieve the best documents to look into for the answer, then selecting the paragraphs which are more likely to bear the desired answer and finally processing the extracted paragraphs by means of NLP. Many closed domain question answering systems also adopt IR approaches, but in this context, we might benefit from existing structured knowledge. Some of the very early question answering systems were designed for closed domains and they were essentially conceived as natural language interfaces to databases [45][80].

Closed Domain QA differs from Open Domain QA mainly because the set of possible questions is more limited and there is a higher chance of being able to rely on a suitable model for representing the data that is specific to the domain at hand.

In the second part of the thesis, we present a system developed in the context of Closed Domain Question Answering. The idea comes from the PICULTURA project, which is a project funded by the Italian Ministry for Economic Development and for which the University of Calabria is a research partner. The PIUCULTURA project has the intent to promote and improve the fruition of Cultural Heritage and a core module of the project is a Question Answering System that had to be able to answer natural language questions posed by the users on Cultural Heritage. The project was centered on the Italian language and the Cultural Heritage domain, but the approach that we developed is general and can work similarly for other languages and, to some extents, to other domains.

Cultural Heritage can benefit from structured data sources: in this context, data has already started to be saved and shared with common standards. One of the most successful standards is the CIDOC Conceptual Reference Model, that has been identified as the ontological model of reference on

cultural heritage for our Question Answering system. It provides a common semantic framework for the mapping of cultural heritage information and can be adopted by museums, libraries, and archives. We developed a system that gradually transforms input natural language questions into formal queries that are executed on an ontological Knowledge Base.

The presented system integrates state-of-the-art models and tools for Natural Language Processing (NLP).

ASP plays a crucial role in the syntactic categorization of questions in the form of what we call *question templates* that are expressed in terms of ASP rules. This is indeed a new application scenario for ASP and the proposed solution fits the requirements and the expectations of the companies involved in the PIUCULTURA project that will transform our prototypical system in a full-fledged implementation.

Additional Notes An update version of the thesis can be downloaded here [1].

Contributions. Summarizing the main contributions of the thesis are:

- We identify empirically the cases in which a propagator-based solution can be used effectively to circumvent the grounding-bottleneck in presence of problematic constraints, reporting on an exhaustive experiment that systematically compares the standard "ground+solve" approach with several alternatives that are based on propagators.
- A technique for partial compilation of ASP programs which is embeddable in state-of-the-art ASP systems. To the best of our knowledge, this is the first work on compilation-based techniques for ASP programs evaluation.
- A system for Closed-Domain Question Answering that is based on ASP.

Publications. Some of the contributions of the thesis have been subject of the following scientific publications.

- Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *TPLP*, 17(5-6):780–799, 2017.

[1] `http://bit.ly/2VQjJEE`

- Bernardo Cuteri and Francesco Ricca. A compiler for stratified datalog programs: preliminary results. In *SEBD*, volume 2037 of *CEUR Workshop Proceedings*, page 158. CEUR-WS.org, 2017.

- Bernardo Cuteri, Alessandro Francesco De Rosis, and Francesco Ricca. lp2cpp: A tool for compiling stratified logic programs. In *AI*IA*, volume 10640 of *Lecture Notes in Computer Science*, pages 200–212. Springer, 2017.

- Bernardo Cuteri. Closed domain question answering for cultural heritage. In *DC@AI*IA*, volume 1769 of *CEUR Workshop Proceedings*, pages 17–22. CEUR-WS.org, 2016.

Part I

Enhancing ASP Systems on Grounding-intensive Programs

Answer set programming (ASP) is a declarative formalism for knowledge representation and reasoning based on stable model semantics [43, 16], for which robust and efficient implementations are available [35]. State-of-the-art ASP systems are usually based on the "ground+solve" approach [49], in which a *grounder* module transforms the input program (containing variables) in an equivalent variable-free one, whose stable models are subsequently computed by the *solver* module. ASP implementations adopting this traditional approach are known to be effective for solving complex problems arising from academic and industrial applications, including: product configuration [50], decision support systems for space shuttle flight controllers [67], explanation of biomedical queries [31], construction of phylogenetic supertrees [51], data-integration [62], reconfiguration systems [6], and more. Nonetheless, there are some classes of programs whose evaluation is not feasible with the "ground+solve" approach. One notable case is due to a combinatorial blow-up of the grounding phase (cf. [18]). An issue that is usually referred to as the *grounding bottleneck* of ASP. In this first part of this thesis, we present some alternative approaches for solving ASP programs, that attempt to succeed where the standard "ground+solve" approach fails or can be improved, especially in presence of the *grounding bottleneck*.

This part is organized as follows:

- in chapter 1 we provide some preliminaries on ASP;
- in chapter 2 we empirically compare the "ground+solve" approach with lazy instantiation and custom propagators for solving ASP programs with problematic constraints;
- in chapter 3 we present partial compilation of ASP programs, that is the application of compilation-based technique to the evaluation of ASP programs (or parts of them).

Chapter 1

Preliminaries and Notation

An ASP program Π is a finite set of rules of the form:

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_j, {\sim}b_{j+1}, \ldots, {\sim}b_m \tag{1.1}$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms and $n \geq 0$, $m \geq j \geq 0$. In particular, an *atom* is an expression of the form $p(t_1, \ldots, t_k)$, where p is a predicate symbol and $t_1, \ldots, t_k$ are *terms*. Terms are alphanumeric strings, and are distinguished in variables and constants. According to the Prolog's convention, only variables start with an uppercase letter. A *literal* is an atom a_i (positive) or its negation ${\sim}a_i$ (negative), where $\sim$ denotes the *negation as failure*. Given a rule r of the form (1.1), the disjunction $a_1 \vee \ldots \vee a_n$ is the *head* of r, while $b_1, \ldots, b_j, {\sim}b_{j+1}, \ldots, {\sim}b_m$ is the *body* of r, of which $b_1, \ldots, b_j$ is the *positive body*, and ${\sim}b_{j+1}, \ldots, {\sim}b_m$ is the *negative body* of r. A rule r of the form (1.1) is called a *fact* if $m = 0$ and a *constraint* if $n = 0$. An object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables. Rules and programs are *positive* if they contain no negative literals, and *general* otherwise. Given a program Π, let the *Herbrand Universe* U_Π be the set of all constants appearing in Π and the *Herbrand Base* B_Π be the set of all possible ground atoms which can be constructed from the predicate symbols appearing in Π with the constants of U_Π. Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of U_Π. Similarly, given a program Π, the *ground instantiation* $Ground(\Pi)$ of Π is the set $\bigcup_{r \in \Pi} Ground(r)$.

For every program Π, its stable models are defined using its ground instantiation $Ground(\Pi)$ in two steps: First stable models of positive programs

are defined, then a reduction of general programs to positive ones is given, which is used to define stable models of general programs.

A set L of ground literals is said to be *consistent* if, for every literal $\ell \in L$, its negated literal $\sim\ell$ is not contained in L. Given a set of ground literals L, $L_{|+} \subseteq L$ denotes the set of positive literals in L. An interpretation I for Π is a consistent set of ground literals over atoms in B_Π. A ground literal ℓ is *true* w.r.t. I if $\ell \in I$; ℓ is *false* w.r.t. I if its negated literal is in I; ℓ is *undefined* w.r.t. I if it is neither true nor false w.r.t. I. A constraint c is said to be *violated* by an interpretation I if all literals in the body of c are true. An interpretation I is *total* if, for each atom a in B_Π, either a or $\sim a$ is in I (i.e., no atom in B_Π is undefined w.r.t. I). Otherwise, it is *partial*. A total interpretation M is a *model* for Π if, for every $r \in Ground(\Pi)$, at least one literal in the head of r is true w.r.t. M whenever all literals in the body of r are true w.r.t. M. A model X is a *stable model* for a positive program Π if any other model Y of Π is such that $X_{|+} \subseteq Y_{|+}$.

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program Π w.r.t. an interpretation X is the positive ground program Π^X, obtained from Π by (i) deleting all rules $r \in \Pi$ whose negative body is false w.r.t. X and (ii) deleting the negative body from the remaining rules. A stable model of Π is a model X of Π such that X is a stable model of $Ground(\Pi)^X$. We denote by $SM(\Pi)$ the set of all stable models of Π, and call Π *coherent* if $SM(\Pi) \neq \emptyset$, *incoherent* otherwise.

Example 1. *Consider the following program* Π_1*:*

$$\begin{array}{lll} r_1 : a(1) \leftarrow \sim b(1) & r_2 : b(1) \leftarrow \sim a(1) & r_3 :\leftarrow a(X),\ b(X) \\ r_4 : c(1) \leftarrow \sim d(1) & r_5 : d(1) \leftarrow \sim c(1) & r_6 :\leftarrow a(X),\ \sim b(X) \end{array}$$

The ground instantiation $Ground(\Pi_1)$ *of the program* Π_1 *is the following program:*

$$\begin{array}{lll} g_1 : a(1) \leftarrow \sim b(1) & g_2 : b(1) \leftarrow \sim a(1) & g_3 :\leftarrow a(1),\ b(1) \\ g_4 : c(1) \leftarrow \sim d(1) & g_5 : d(1) \leftarrow \sim c(1) & g_6 :\leftarrow a(1),\ \sim b(1) \end{array}$$

Note that $M = \{\sim a(1),\ b(1),\ c(1),\ \sim d(1)\}$ *is a model of* $Ground(\Pi_1)$*. Since* $Ground(\Pi_1)^M$ *comprises only the facts* $b(1)$ *and* $c(1)$*, and constraint* g_3*,* M *is a stable model of* Π*.*

Support. Given a model M for a ground program Π, we say that a ground atom $a \in M$ is *supported* with respect to M if there exists a *supporting* rule

$r \in \Pi$ such that a is the only true atom w.r.t. M in the head of r, and all literals in the body of r are true w.r.t. M. If M is a stable model of a program Π, then all atoms in M are supported.

Chapter 2

Custom Propagators for Constraints

The grounding bottleneck has been subject of several studies in recent years, and various alternative approaches to overcome it have been proposed. Some of these are based on syntactic extensions that enable the combination of ASP solvers with solvers for external theories [68, 11, 10, 7, 23, 75, 30]; whereas, the most prominent approach working on plain ASP is *lazy grounding*, which was implemented by ASPERIX [55], GASP [21], and OMIGA [22]. Roughly, the idea of lazy grounding is to instantiate rules only when it is required during the search for a stable model [60]. In this way, it is possible to prevent the grounding of rules that are unnecessary for the computation. Albeit lazy grounding techniques obtained promising preliminary results, they cannot yet reach the performance of state of the art systems in many benchmarks [18, 55]. One of the reasons is probably that fully-fledged lazy grounding techniques could not be easily integrated within solvers based on the very efficient Conflict-Driven Clause Learning (CDCL) algorithm [73, 49, 79]. Nonetheless, in many applications, the grounding bottleneck is merely caused by rules of a specific kind, namely constraints. For example, the following constraint has been identified as the bottleneck in programs solving a problem of natural language understanding:

$$\leftarrow eq(X, Y),\ eq(Y, Z),\ {\sim}eq(X, Z)$$

Its grounding, which features a cubic number of instances with respect to the extension of predicate eq in the worst case, is often not feasible for real world instances [72].

In this work, we first focus on the above practically-relevant case of problematic constraints. In particular, we systematically compare alternative strategies that avoid the instantiation of some constraints by extending a CDCL-based ASP solver. In a nutshell, the input program is simplified by omitting problematic constraints and it is grounded; then, the resulting ground program is provided as input to a solver that is extended to emulate the presence of missing constraints. Among the strategies for extending the solver, we considered *lazy instantiation of constraints* and *custom propagators*. In the first strategy, the solver searches for a stable model S of the simplified program. Then, S is returned as a solution if it satisfies also the omitted constraints, otherwise the violated instances of these constraints are lazily instantiated, and the search continues (Sec. 2.1.2). In the second strategy, the solver is extended (in possibly alternative ways) by custom *propagators*, which emulate the presence of missing constraints during the search (Sec. 2.1.3). The above-mentioned strategies can be implemented by using the API of existing CDCL-based ASP solvers [35, 25].

An empirical evaluation conducted on real and synthetic benchmarks (Sec. 2.2) confirms that the usage of lazy instantiation and custom propagators is effective when the grounding bottleneck is due to some constraint. The analysis of the results highlights strengths and weaknesses of the different strategies. Moreover, it shows there is not always a clear winner for a given problem, and the choice depends also on the characteristics of the instances to solve. This observation suggested to investigate the applicability of algorithm selection techniques. The results are positive, in the sense that already a basic portfolio is faster than any of the considered approaches taken individually.

2.1 Solving Strategies

2.1.1 Classical Evaluation

The standard solving approach for ASP is instantiation followed by a procedure similar to CDCL for SAT with extensions specific to ASP [49]. The basic algorithm *ComputeStableModel*(Π) for finding a stable model of program Π is shown in Algorithm 1. The Function 2 combines unit propagation (as in SAT) with some additional ASP-specific propagations, which ensure the model is stable (cf. [49]).

Given a partial interpretation I consisting of literals, and a set of rules Π, *unit propagation* infers a literal ℓ to be true if there is a rule $r \in \Pi$ such that r can be satisfied only by $I \cup \{\ell\}$. Given the nogood representation $C(r) = \{\sim a_1, \ldots, \sim a_n, b_1, \ldots, b_j, \sim b_{j+1}, \ldots, \sim b_m\}$ of a rule r, then the negation of a literal $\ell \in C(r)$ is unit propagated w.r.t. I and rule r iff $C(r) \setminus \{\ell\} \subseteq I$. To ensure that models are supported, unit propagation is performed on the Clark completion of Π or alternatively a *support propagator* is used [3].

Example 2. *Consider the ground program Π_1 from Example 1. ComputeStableModel(Π_1) starts with $I = \emptyset$ and does not propagate anything at line 2. I is partial and consistent, so the algorithm continues at line 15. Assume no restart and no deletion is performed, and assume ChooseLiteral returns $\{a(1)\}$, i.e., $I = \{a(1)\}$. Next, Propagate(I) is called, which yields $I = \{a(1),\ b(1),\ \sim b(1)\}$: $\sim b(1)$ comes from unit propagation on g_3 and $b(1)$ from unit propagation on g_6. Thus, I is inconsistent and I is analyzed to compute a reason explaining the conflict, i.e., CreateConstraint(I) = $\{g_7\}$ with $g_7 : \leftarrow a(1)$. Intuitively, the truth of $a(1)$ leads to an inconsistent interpretation, thus $a(1)$ must be false. Then, the consistency of I is restored (line 5), i.e., $I = \emptyset$, and g_7 is added to Π_1. The algorithm again restarts at line 2 with $I = \emptyset$ and propagates $I = \{\sim a(1),\ b(1)\}$, where $\sim a(1)$ comes from unit propagation on g_7, and b from unit propagation on g_2. I is partial and consistent, therefore lines 15 and 16 are executed. Assume again that no restart and no constraint deletion happens, and that ChooseLiteral(I) = $\{c(1)\}$. Therefore, the algorithm continues at line 2 with $I = \{\sim a(1),\ b(1),\ c(1)\}$. Propagation yields $I = \{\sim a(1),\ b(1),\ c(1),\ \sim d(1)\}$ because $\sim d(1)$ is support-propagated w.r.t. g_4 and I (or unit-propagated w.r.t. the completion of g_4 and I). I is total and consistent, therefore the algorithm returns I as the first stable model.*

For the performance of this search procedure, several details are crucial: learning effective constraints from inconsistencies as well as heuristics for restarting, constraint deletion, and for choosing literals.

2.1.2 Lazy Constraints

The algorithm presented in this section is reported as Algorithm 3. The algorithm takes as input a program Π and a set of constraints $C \subseteq \Pi$. Then, the constraints in C are removed from Π, obtaining the program $\mathcal{P}$. A stable model of *Ground*($\mathcal{P}$) is searched (line 2). Two cases are possible: (i) $\mathcal{P}$ is

Algorithm 1 ComputeStableModel

Input: A ground program $\mathcal{P}$
Output: A stable model for $\mathcal{P}$ or $\bot$

1: $I := \emptyset$
2: $I :=$ *Propagate(I)*
3: **if** I is inconsistent **then**
4: $\quad r :=$ *CreateConstraint(I)*
5: $\quad I :=$ *RestoreConsistency*(I)
6: $\quad$ **if** I is consistent **then**
7: $\quad\quad \mathcal{P} := \mathcal{P} \cup \{r\}$
8: $\quad$ **else**
9: $\quad\quad$ **return** $\bot$
10: $\quad$ **end if**
11: **else if** I total **then**
12: $\quad$ **return** I
13: **else**
14: $\quad I :=$ *RestartIfNeeded(I)*
15: $\quad \mathcal{P} :=$ *DeleteConstraintsIfNeeded*$(\mathcal{P})$
16: $\quad I := I \cup$ *ChooseLiteral(I)*
17: **end if**
18: **goto** 2

Algorithm 2 Propagate(I)

1: $\mathcal{I} = I$
2: **for all** $\ell \in \mathcal{I}$ **do**
3: $\quad \mathcal{I} := \mathcal{I} \cup$ *Propagation*$(\mathcal{I}, \ \ell)$
4: **end for**
5: **return** $\mathcal{I}$

Algorithm 3 LazyConstraintInstantiation

Input: A nonground program Π, a set of nonground constraints $C \subseteq \Pi$
Output: A stable model for Π or $\bot$

1: $\mathcal{P} := Ground(\Pi \setminus C)$
2: $I := ComputeStableModel(\mathcal{P})$
3: **if** $I == \bot$ **then**
4: **return** $\bot$
5: **end if**
6: $\mathcal{C} = \{c \mid c \in Ground(C),\ c \text{ is violated}\}$
7: **if** $\mathcal{C} == \emptyset$ **then**
8: **return** I
9: **end if**
10: $\mathcal{P} := \mathcal{P} \cup \mathcal{C}$
11: **goto** 2

incoherent (line 4). Thus, the original program Π is also incoherent and the algorithm terminates returning $\bot$. (*ii*) $\mathcal{P}$ is coherent. Thus, a stable model, say I, is computed. In this case, a set of constraints $\mathcal{C} \in Ground(C)$ that are violated under the stable model I are extracted (line 6) and added to $\mathcal{P}$ (line 10). The process is repeated until either a stable model of $\mathcal{P}$ violating no constraints in $Ground(C)$ is found or $\mathcal{P}$ is incoherent. Importantly, $Ground(C)$ is never represented explicitly in the implementation of line 5.

Example 3. *Again consider program* Π_1 *from Example 1 and the set of constraints* $C = \{r_3,\ r_6\}$. *The algorithm computes a stable model, say* $I_1 = \{a(1),\ {\sim}b(1),\ c(1),\ {\sim}d(1)\}$, *of* $\mathcal{P}_1 = Ground(\Pi_1 \setminus C)$. *Thus, the ground instantiation* g_6 *of* r_6 *is violated under* I_1 *and therefore* g_6 *is added to* $\mathcal{P}$. *Then, a stable model of* $\mathcal{P}$ *is computed, say* $I_2 = \{{\sim}a(1),\ b(1),\ c(1),\ {\sim}d(1)\}$. *At this point,* I_2 *violates no constraint in* $Ground(C)$. *Thus, the algorithm terminates returning* I_2. *Note that all instantiations of constraint* r_3 *will be never violated since rules* r_1 *and* r_2 *enforce that exactly one of* $a(1)$ *and* $b(1)$ *can be true in a stable model. Thus,* r_3 *will never be instantiated by the algorithm.*

An important feature of Algorithm 3 is that it requires no modifications to the search procedure implemented by the underlying ASP solver.

2.1.3 Constraints via Propagators

In this section, constraints are replaced using the concept of *propagator*, which can set truth values of atoms during the solving process, based on truth values of other atoms. An example of a propagator is the unit propagation, detailed in Section 2.1.1. In contrast to the lazy instantiation of constraints that aims at adding violated constraints when a stable model candidate is found, propagators usually are used to evaluate the constraints during the computation of the stable model. Given a program Π, traditional solvers usually apply propagators on the whole set of rules and constraints in $Ground(\Pi)$. An alternative strategy is to consider a variant of the program, say $\mathcal{P} = \Pi \setminus C$, where C is a set of constraints. The solver is then executed on $Ground(\mathcal{P})$ and a propagator is used to guarantee the coherence of partial interpretations with the constraints in $Ground(C)$. Constraints in C are not instantiated in practice but their inferences are simulated by an ad-hoc procedure implemented for that purpose. This approach requires a modification of the Propagation function in algorithm 2, such that Propagation considers the additional set C of constraints, verifies which constraints would result in a propagation on the partial interpretation, and propagate truth values due to inferences on C in addition to unit propagation.

Example 4. *Again consider program Π_1 from Example 1 and the set of constraints $C = \{r_3,\ r_6\}$. The idea is to execute Algorithm 1 on $Ground(\mathcal{P}_1)$, where $\mathcal{P}_1 = \Pi_1 \setminus C$. $ComputeStableModel(\mathcal{P}_1)$ starts with $I = \emptyset$ and does not propagate anything at line 2. I is partial and consistent, so the algorithm continues at line 15. Assume no restart and no deletion is performed, and assume ChooseLiteral returns $\{a(1)\}$, i.e., $I = \{a(1)\}$. Next, $Propagate(I,\ C)$ is called. In this case, the propagation yields $I = \{a(1),\ b(1),\ {\sim}b(1)\}$, where ${\sim}b(1)$ comes from unit propagation on g_1, while $b(1)$ comes from unit propagation on the ground instantiation g_6 of the rule r_6. Thus, I is inconsistent and I is analyzed to compute a reason that explains the conflict, i.e., $CreateConstraint(I) = \{g_7\}$ with $g_7 : \leftarrow a(1)$. Then, the algorithm continues as shown in Example 2. Note that, from this point of the computation, the ground instantiations of constraints r_3 and r_6 will never be violated again, since g_7 assure that $a(1)$ will be false in all partial interpretations under consideration.*

We classify constraint propagators according to the priority given to them. In particular, they are considered *eager* if propagation on non-ground con-

straints is executed as soon as possible, i.e., during unit propagation of already grounded constraints; moreover, they are called *postponed* (or *post*) if propagation on constraints is executed after all other (unit, support, etc.) propagations.

2.2 Implementation and Experimental Analysis

2.2.1 Implementation

The lazy instantiation of constraints and the propagators have been implemented on top of the ASP solvers WASP [5] and CLINGO [35]. The Python interface of WASP [25] follows a synchronous message passing protocol implemented by means of method calls. Basically, a Python program implements a predetermined set of methods that are later on called by WASP whenever specific points of the computation are reached. The methods may return some values that are then interpreted by WASP. For instance, when a literal is true the method *onLiteralTrue* of the propagator is called, whose output is a list of literals to infer as true as a consequence (see [25] for further details). CLINGO 5 [35] provides a Python interface where a propagator class with an interface similar to WASP can be registered.

Two important differences exist between WASP and CLINGO. Firstly CLINGO provides only a post-propagator interface and no possibility for realizing an eager propagator (that runs before unit propagation is finished). Secondly, WASP first collects nogoods added in Python and then internally applies them and handles conflicts, while CLINGO requires an explicit propagation call after each added nogood. If propagation returns a conflict then no further nogoods can be added in CLINGO, even if further nogoods were detected. After consulting the CLINGO authors, we implemented a queue for nogoods and add them in subsequent propagations if there is a conflict. This yields higher performance than abandoning these nogoods.

2.2.2 Description of Benchmarks

In order to empirically compare the various strategies for avoiding the instantiation of constraints, we investigated several benchmarks of different nature, namely Stable Marriage, Packing, and Natural Language Understanding. All

benchmarks contain one or few constraints whose grounding can be problematic.

Stable Marriage. The *Stable Marriage* problem can be described as follows: given n men and m women, where each person has a preference order over the opposite sex, marry them so that the marriage is stable. In this case, the marriage is said to be stable if there is no couple (M, W) for which both partners would rather be married with each other than their current partner. We considered the encoding used for the fourth ASP Competition. For the lazy instantiation and for the ad-hoc propagators the following constraint has been removed from the encoding:

```
:- match(M,W1), manAssignsScore(M,W,Smw), W1 ≠ W,
manAssignsScore(M,W1,Smw1), Smw > Smw1,
match(M1,W), womanAssignsScore(W,M,Swm),
womanAssignsScore(W,M1,Swm1), Swm ≥ Swm1.
```

Intuitively, this constraint guarantees that the stability condition is not violated.

Packing. The *Packing Problem* is related to a class of problems in which one has to pack objects together in a given container. We consider the variant of the problem submitted to the ASP Competition 2011. In that case, the problem was the packing of squares of possibly different sizes in a rectangular space and without the possibility of performing rotations. The encoding follows the typical guess-and-check structure, where positions of squares are guessed and some constraints check whether the guessed solution is a stable model. We identified 2 expensive sets of constraints. The first set comprises the following two constraints:

```
:- pos(I,X,Y), pos(I,X1,Y1), X1 ≠ X
:- pos(I,X,Y), pos(I,X1,Y1), Y1 ≠ Y
```

which enforce that a square is not assigned to different positions. The second set comprises constraints forbidding the overlap of squares. One of these constraints is reported in the following:

```
:- pos(I1,X1,Y1), square(I1,D1), pos(I2,X2,Y2),
square(I2,D2), I1 ≠ I2, W1 = X1+D1,
H1 = Y1+D1, X2 ≥ X1, X2 < W1, Y2 ≥ Y1, Y2 < H1.
```

Other constraints are similar thus they are not reported.

Natural Language Understanding (NLU). The *NLU* benchmark is an application of ASP in the area of Natural Language Understanding, in particular the computation of optimal solutions for First Order Horn Abduction problems under the following cost functions: cardinality minimality, cohesion, and weighted abduction. This problem and these objective functions have been described by Schüller [72]. In this problem, we aim to find a set of explanatory atoms that makes a set of goal atoms true with respect to a First Order Horn background theory. We here consider the acyclic version of the problem where backward reasoning over axioms is guaranteed to introduce a finite set of new terms. A specific challenge in this problem is that input terms and terms invented via backward chaining can be equivalent to other terms, i.e., the unique names assumption is partially not true. Equivalence of terms must be handled explicitly in ASP, which is done by guessing an equivalence relation. This makes the instantiation of most instances infeasible, as the number of invented terms becomes large, due to the grounding blow-up caused by the following constraint:

```
:- eq(A,B), eq(B,C), not eq(A,C).
```

2.2.3 Hardware and Software Settings

The experiments were run on a Intel Xeon CPU X3430 2.4 GHz. Time and memory were limited to 600 seconds and 4 GB, respectively. In the following, WASP-LAZY refers to WASP implementing lazy instantiation of constraints, while WASP-EAGER and WASP-POST refer to WASP implementing eager and postponed propagators, respectively. All versions of WASP use GRINGO version 5.1.0 as grounder, whose grounding time is included in the execution time of WASP. Moreover, CLINGO LAZY and CLINGO POST refer to CLINGO implementing lazy and postponed propagators, respectively. For the NLU benchmark, we always use unsat-core optimization.

2.2.4 Discussion of Results

Stable Marriage. Concerning Stable Marriage, we executed the 30 instances selected for the Fourth ASP Competition. CLINGO and WASP executed on the full encoding are able to solve 29 out of the 30 instances

with an average running time of 50 and 29 seconds, respectively. On the same instances, ad-hoc propagators cannot reach the same performance. Indeed, WASP-LAZY and WASP-POST perform the worst solving 0 and 5 instances, respectively, whereas WASP-EAGER is much better with 17 solved instances. The same performance is obtained by CLINGO-LAZY and CLINGO-POST which can solve 0 and 17 instances in the allotted time, respectively. The poor performance of the lazy instantiation can be explained by looking at the specific nature of the instances. Indeed, each instance contains a randomly generated set of preferences of men for women (resp. women for men). By looking at the instances we observed that each man (resp. woman) has a clear, often total, preference order over each woman (resp. man). This specific case represents a limitation for employing the lazy instantiation. Indeed, WASP and CLINGO executed on the encoding without the stability constraint perform naive choices until a stable model candidate is found. Then, each candidate contains several violations of the stability condition and many constraints are added. However, those constraints are not helpful since they only invalidate the current stable model candidate. In general, for instances where the program without the stability condition

Table 2.1: Stable Marriage: Number of solved instances and average running time (in seconds).

Pref. (%)	**wasp**		**wasp-lazy**		**wasp-eager**		**wasp-post**		**clingo**		**clingo-lazy**		**clingo-post**	
	sol.	**avg t**	**sol.**	**avg t**	**sol.**	**avg t**	**sol.**	**avg t**	**sol.**	**avg t**	**sol.**	**avg t**	**sol.**	**avg t**
0	10	4.1	10	4.7	10	4.6	10	4.7	10	10.6	10	4.2	10	4.2
5	9	16.2	10	4.7	10	4.3	10	4.9	10	23.0	10	4.6	10	4.4
10	10	19.2	10	4.7	10	4.3	10	4.6	10	34.6	10	6.4	10	8.2
15	9	24.3	10	4.7	10	4.4	10	4.8	10	42.9	10	9.6	10	17.5
20	8	35.2	10	4.8	10	4.6	10	5.2	10	48.9	10	16.5	10	24.7
25	10	34.8	10	4.8	10	5.4	10	6.0	10	53.9	10	22.2	10	42.8
30	6	97.0	10	5.0	10	7.7	10	7.6	10	59.5	10	32.2	10	92.1
35	10	42.1	10	5.0	10	8.2	10	10.0	10	65.8	10	62.4	10	115.9
40	9	51.3	10	5.2	10	7.6	10	9.2	10	68.4	10	81.8	10	117.5
45	10	113.4	10	5.4	10	10.8	10	12.0	10	71.0	10	97.7	10	140.8
50	6	74.6	10	5.1	10	22.4	10	20.3	10	72.0	10	153.6	10	143.4
55	9	44.5	8	5.9	10	39.4	10	23.6	10	72.9	10	193.8	10	166.5
60	9	70.9	10	7.7	10	23.8	10	25.0	10	74.6	10	241.1	10	181.6
65	7	99.3	10	11.4	10	64.7	10	54.2	10	74.7	10	295.6	10	209.8
70	9	89.3	5	25.5	10	121.8	10	101.8	10	75.0	10	361.1	10	235.3
75	8	77.0	0	-	10	184.0	10	146.7	10	75.1	6	472.1	10	311.0
80	7	85.5	0	-	10	248.6	8	274.7	10	76.3	0	-	10	434.3
85	4	259.5	0	-	10	232.3	1	337.2	10	82.3	0	-	7	569.7
90	9	79.2	0	-	5	449.4	0	-	10	251.1	0	-	1	577.7
95	10	46.3	0	-	0	-	0	-	6	273.6	0	-	3	580.8
100	8	67.6	1	81.2	10	133.3	10	153.6	10	74.1	6	493.3	10	323.9

is under-constrained many stable model candidates need to be invalidated before an actual solution is found (intuitively, given a program Π and a set of constraints $C \subseteq \Pi$, $|SM(\Pi \setminus C)| \gg |SM(\Pi)|$).

In order to further analyze this behavior empirically, we have conducted an additional experiment on the same problem. In particular, we randomly generated instances where each man (resp. woman) gives the same preference to each woman (resp. man), so basically the stability condition is never violated. Then, we consider a percentage k of preferences, i.e., each man (resp. woman) gives the same preference to all the women (resp. men) but to $k\%$ of them a lower preference is given. In this way, instances with small values of k should be easily solved by lazy instantiation, whereas instances with high values of k should be hard. For each considered percentage k, we executed 10 randomly generated instances. Results are reported in Table 2.1, where the number of solved instances and the average running time are shown for each tested approach. Concerning WASP, as observed before, for instances where the value of k is small (up to 50%) the lazy approach can solve all the instances with an average running time of about 5 seconds. On the other hand, for high values of k the advantages of the lazy approach disappear, as observed for the competition instances. Interestingly, the eager propagator obtained the best performance overall. For the tested instances, it seems to benefit of a smaller program and generation of the inferences does not slow down the performance as observed for competition instances. Concerning CLINGO, the lazy approach is the best performing one for instances where the value of k is up to 35%. As shown for WASP, the performance of the lazy approach are worse for high values of k.

Packing. Concerning Packing problem, we considered all 50 instances submitted to the Third ASP Competition. Interestingly, when all constraints are considered none of the instances can be instantiated within the time limit. Thus, CLINGO and WASP do not even start the computation of a stable model. The grounding time substantially decreases when the two sets of expensive constraints described in Section 2.2.2 are removed from the encoding. Indeed, in this case, the grounding time on the tested instances is 5 seconds on average, with a peak of 16 seconds. Results of the lazy constraint instantiation and of constraint propagators on the resulting program are reported in the cactus plot of Figure 2.1. The graph highlights that WASP-EAGER, WASP-POST, and CLINGO-POST basically obtained the same

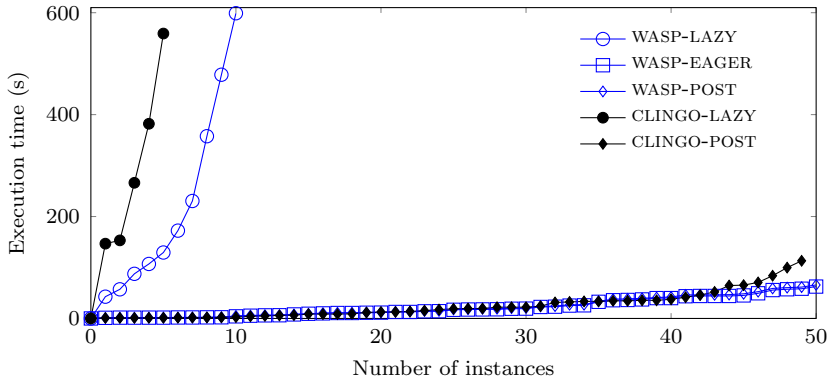

Figure 2.1: Packing: Comparison of LAZY and PROPAGATORS approaches on 50 instances.

performance. Indeed, the first two solve all the tested instances with an average running time of 22 and 23 seconds, respectively, while CLINGO-POST solves 49 out of 50 instances with an average running time of 25 seconds. Both WASP-POST and CLINGO-POST outperform their lazy counterparts. Indeed, WASP-LAZY solves 10 instances, with an average running time of 226 seconds, while CLINGO-LAZY solves 5 instances, with an average running time of 301 seconds. As already observed on the Stable Marriage instances, lazy instantiation cannot compete with constraint propagators. In this experiment, we observed that WASP and CLINGO perform naive choices on the encoding without the expensive constraints, thus each candidate stable model contains several violations of constraints, leading to inefficient search in harder instances.

Natural Language Understanding (NLU). Concerning NLU, we considered all 50 instances and all three objective functions used in [72]. Results

Table 2.2: NLU Benchmark: Number of solved instances and average running time (in seconds).

Obj. Func.	wasp		wasp-lazy		wasp-eager		wasp-post		clingo		clingo-lazy		clingo-post	
	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t
Card.	43	39.7	50	2.3	50	4.3	50	3.3	41	30.7	50	4.5	50	1.5
Coh.	43	40.1	50	18.5	50	8.8	50	6.3	41	30.7	49	24.6	49	15.8
W. Abd.	43	49.3	50	26.6	49	66.1	50	62.6	41	33.9	48	31.9	50	24.0

are reported in Table 2.2. As a general observation, all the tested instances are solved by WASP-LAZY and WASP-POST, no matter the objective function. Moreover, WASP-LAZY is on average faster than all other alternatives for both the objective functions cardinality and weighted abduction. The good performance of lazy instantiation is related to the small number of failing stable model checks performed. Indeed, only 2, 16, and 64 invalidations are on average required for cardinality, coherence, and weighted abduction, respectively. The number of propagation calls is much higher for WASP-EAGER than for WASP-POST (approximately WASP-EAGER performs 3 times more propagation calls than WASP-POST). However, the number of propagated literals that are not immediately rolled back because of a conflict is very similar, hence it is clear that WASP-EAGER performs a lot of unnecessary propagations in this benchmark and WASP-POST should be preferred. Note that this is not generally the case for other benchmarks. Concerning CLINGO, 45, 248, and 321, stable model candidates are invalidated with CLINGO-LAZY, respectively, and a similar amount (26, 589, and 700, respectively) with CLINGO-POST. This shows that CLINGO tends to produces more stable models that violate lazy constraints. These violations are detected earlier with CLINGO-POST, therefore it outperforms CLINGO-LAZY in all objectives. None of the CLINGO propagators is able to solve all instances with all objectives, whereas WASP-POST solves all of them within 600 s. In particular for objective functions cardinality and coherence, WASP is always slightly faster and uses slightly more memory than CLINGO. For weighted abduction, CLINGO-POST is most efficient with WASP-LAZY in second place. Nevertheless, using CLINGO or WASP with a LAZY or POST propagator will always be an advantage over using the pure ASP encoding where the constraints are instantiated prior to solving. Hence the choice of the method for instantiating constraints is more important than the choice of the solver.

Discussion. We empirically investigated whether lazy instantiation or propagators can be a valid option for enhancing the traditional "ground+solve" approach. When the full grounding is infeasible, then both lazy instantiation and propagators can overcome this limitation, even though they exhibit different behaviors depending on the features of the problem and of the instances. This is particularly evident in Packing, where no instance can be grounded within the time limit. Since propagators are activated during the search, while lazy instantiation intervenes only when a total interpretation

is computed, propagators are preferable when the problematic constraint is important to lead the search toward a solution (as overlap constraints in Packing). On the other hand, a high number of unnecessary propagations can make propagators inefficient and even slower than the lazy approach. In these cases, we observed that post propagators are better than eager propagators as remarked by the results on the objective function 'weighted abduction' in the NLU benchmark. The experiment on Stable Marriage highlights that lazy instantiation is effective when few constraints are instantiated during the search. This is the case when: (i) it is very likely that a stable model of the simplified (i.e., without problematic constraints) input program also satisfies the lazy constraints; or (ii) the solver heuristics is such that one of the first candidate total interpretations also satisfies the lazy constraints. This is also confirmed in the NLU benchmark where the instances often have the above characteristics, and the propagator is better only when the constraints generated by the lazy approach do not fit the working memory. Moreover, from case (ii), we conjecture that the lazy approach can be effective in combination with domain-specific heuristics [40, 24].

Finally, we conducted an additional experiment, where we do not oppose our approaches with the ground+solve one as in the previous cases, but it only aims at comparing the lazy propagation versus propagators in a controlled setting. In particular, we considered a synthetic benchmark based on the well-known 3-SAT problem that is interesting for our study since it allows us to control both the hardness of the instances and the probability that an interpretation satisfies the constraint. Indeed, we generated the instances uniformly at random in a range centered on the phase transition [1]. We used a straightforward ASP encoding where we guess an interpretation and we check by a single (non-ground) constraint whether this satisfies all clauses. The results are summarized in Figure 2.4 where we present two representative runs on formulas with 220 and 280 Boolean variables, respectively. Since eager and post propagators behave very similarly we only show comparisons between eager propagator and a lazy instantiation.

Expectedly, execution times follow the easy-hard-easy pattern [1], centered on the phase transition, while varying the ratio R of clauses over variables. Initially, the problem is very easy and both approaches are equally fast. Then there is an interval in which the lazy approach is preferable, and finally the eager approach becomes definitely better than the lazy. Note that, on formulas with 220 variables (see Figure 2.2) the lazy approach is preferable also on the hardest instances, instead with 280 variables (see Figure 2.3) the

eager approach becomes more convenient before the phase transition. To explain this phenomenon we observe that the lazy approach can be exemplified by assuming that the solver freely guesses a model and then the lazy instantiator checks it, until every clause is satisfied by an assignment or no model can be found. The probability that a random model satisfies all clauses is $(\frac{7}{8})^k$ where k is the number of clauses, thus fewer tries are needed on average to converge to a solution if the formula has fewer clauses. This intuitively explains why, as the number of variables increases, the eager approach becomes more convenient at smaller and smaller values of R. It is worth pointing out that this simplified model does not fully capture the behavior of lazy instantiation that is more efficient in practice, since the implementation learns from previous failures (by instantiating violated constraints).

2.2.5 On the applicability of techniques for automatic algorithm selection

The analysis conducted up to now shows that there is not always a clear winner among the strategies for realizing constraints, since the best solving method depends on characteristics of the encoding and the instance at hand. In similar scenarios, portfolio approaches which automatically choose one out of a set of possible methods have proven to be very effective in increasing system performance, since they combine the strengths of the available methods. Therefore, we investigated whether algorithm selection techniques can improve performance in our context.

We apply basic algorithm selection based on classification with machine learning: we extract some natural features from each instance, and train a C4.5 [69] classifier to predict the best solving method (i.e., the one that required least amount of time) among all the available ones (including the plain solver). We limit our analysis to Stable Marriage and NLU, because in these domains none of the available methods is clearly superior. As features for stable marriage we used the number of persons and the percentage or

Table 2.3: Applicability of a portfolio approach: experimental results.

		WASP-based				CLINGO-based			
Problem	**#instances**	**Prec**	**Recall**	**F-Meas**	**Perf. gain**	**Prec**	**Recall**	**F-Meas**	**Perf. gain**
Stable Marriage	500	0.66	0.67	0.63	26.7%	0.74	0.75	0.74	13.6%
NLU	150	0.88	0.92	0.90	38.3%	0.84	0.84	0.84	10.0%

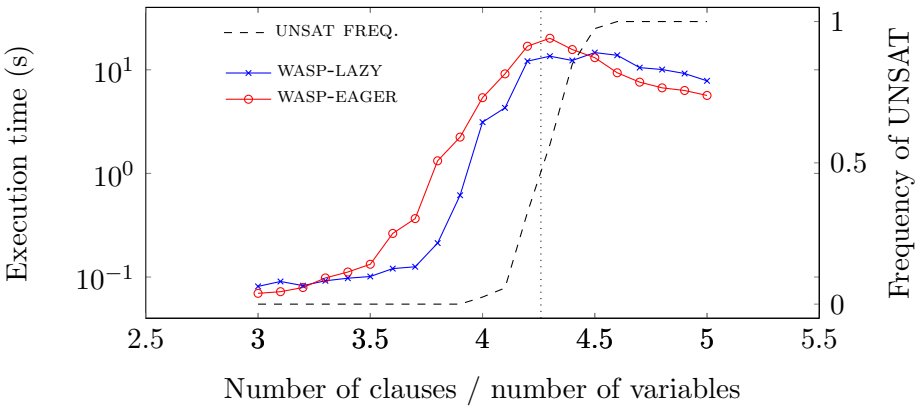

Figure 2.2: Results with 220 variables

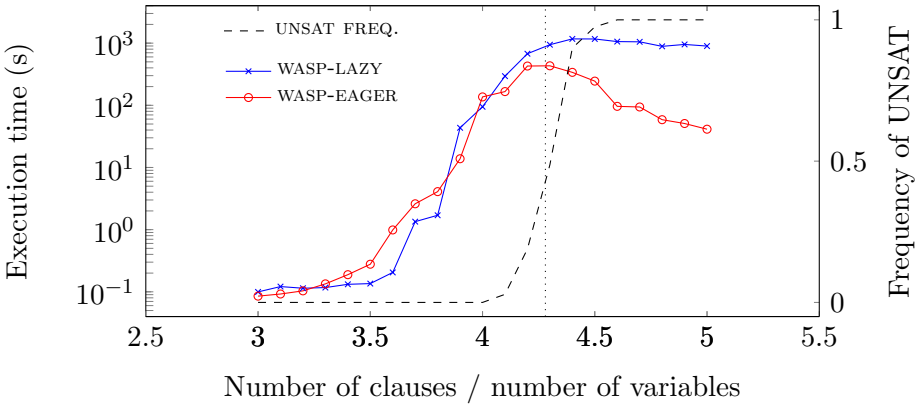

Figure 2.3: Results with 280 variables

Figure 2.4: *3-SAT experiments. Red and blue lines correspond to eager propagators and lazy instantiation respectively. The dashed black line represents the percentage of UNSAT instances, while the vertical dotted line evidences the phase transition point (frequency is about* 0.5 *at* $R = 4.26$*).*

preferences, for NLU we used the number of facts and the number of distinct constants and (instance-specific) predicates. We create portfolios for both WASP-based and CLINGO-based implementations.

Table 2.3 shows the results of our evaluation using 10-fold cross-validation (i.e., we split the set of instances into 10 partitions and use each partition as test set while training on the remaining partitions). For each problem we report (weighted average) precision, recall, and f-measure of the prediction, as well as the average performance gain of the portfolio (i.e., by gain we mean the difference in percentage between the sum of the execution times measured for the portfolio and for its best method). We observe that the classifier is able to choose the best algorithm in many cases, and the choice is almost ideal in NLU (f-measure of 0.9 for WASP and 0.84 for CLASP). The portfolios are always faster (in terms of execution times) than the corresponding best method for the respective problem. The performance gain peaks to 38% for the WASP-based, and is less pronounced for the CLINGO-based (peak at 13.6%). This is expected since CLINGO features a basic solver that is more competitive with propagator-based solutions in these domains.

Summarizing, these results confirm that already the application of basic portfolio techniques is a viable option for improving the performance when propagators are available.

2.3 Related Work

The grounding bottleneck in ASP has been subject of various studies. The most prominent grounding-less approach that works on plain ASP is *lazy grounding*, which was implemented by ASPERIX [55], GASP [21], and OMIGA [22]. Differently from our approach that is focused on constraints, these solvers perform lazy instantiation for *all* the rules of a program, and do not perform (conflict) clause learning. Weinzierl [79] recently investigated learning of non-ground clauses.

Lazy instantiation of constraints was topic of several works on *integrating ASP with other formalisms*. These include CASP [13, 68, 11], ASPMT [75], BFASP [7], and HEX [29]. Differently from our approach, these approaches are based on syntactic extensions that enable the combination of ASP solvers with solvers for external theories. HEX facilitates the integration of generic computation oracles as literals in ASP rule bodies, and allows these computations not only to return true or false, but also to inject constraints into

the search. This gave rise to the 'on-demand constraint' usage pattern of external atoms [30] which roughly corresponds with the LAZY propagators in this work. HEX also permits a declarative specification of properties of external computations [70], e.g., antimonotonicity with respect to some part of the model. Such specifications automatically generate additional lazy constraints. Integration of ASP with continuous motion planning in robotics, based on HEX, was investigated in [32]: adding motion constraints in a POST propagator was found to be significantly faster than checking only complete stable model candidates (LAZY). For integrating *CModels with BProlog* [10] it was shown that using BProlog similar as a POST propagator (clearbox) performs better than using it as a LAZY propagator (black-box).

De Cat et al. [23] provide a theory and implementation for *lazy model expansion* within the FO(ID) formalism which is based on *justifications* that prevent instantiation of certain constraints under assumptions. These assumptions are relative to a model candidate and can be revised from encountered conflicts, leading to a partially lazy instantiation of these constraints.

We finally observe that lazy constraints can be seen as a simplified form of lazy clause generation that was originally introduced in Constraint Programming [34].

2.4 Discussion

In this chapter, we compared several solutions for addressing the problem of the grounding bottleneck focusing on the practically-relevant case of problematic constraints without resorting to any language extension. The considered approach can be seen as a natural extension of the "ground+solve" paradigm, adopted by state of the art ASP systems, where some constraints are replaced either by lazy instantiators or propagators. The solutions fit CDCL-based solving strategies, and can be implemented using APIs provided by state of the art solvers.

Experiments conducted on both real-world and synthetic benchmarks clearly outline that all the approaches can solve instances that are out of reach of state of the art solvers because of the grounding blowup. Lazy instantiation is the easiest to implement, and it is the best choice when the problematic constraints have a high probability to be satisfied. Otherwise, eager and post propagators perform better, with the latter being slightly more efficient when the constraint is activated more often during propaga-

tion. Our empirical analysis shows that there is not always a clear winner for a given problem, thus we investigated the applicability of algorithm selection techniques. We observed that a basic portfolio can improve on the best strategy also on these cases. In the next section we are presenting a continuation of this work, that lead to the automatic generation of lazy propagators and moves some first steps towards compilation-based evaluation of ASP.

Chapter 3

Partial Compilation

As seen in the previous chapter, the standard "ground+solve" approach can fail at solving hard ASP instances, especially when the grounding phase is costly. We empirically showed how lazy or eager propagators of constraints can be better suited than the "ground+solve" approach in several use-cases. One of the main drawbacks of using custom propagators is that the user has to write a custom procedure that simulates the constraint (either eagerly or lazily). We moved forward with the intent to produce such procedures automatically and did it for lazy propagators. Moreover, we extended the previous work to more general cases of application, covering not only constraints, but more general classes of sub-programs.

In an attempt to make the whole process as efficient as possible, we decided to adopt a compilation-based approach, where an ASP sub-program can be compiled into a specialized procedure that is in principle (and in practice, as we will show) faster than a general-purpose procedure thanks to its specificity. This attempt led to what is (to the best of our knowledge) the first work of partial compilation of ASP programs. Indeed, automatic lazy grounding of constraints became a by-product of partial compilation, and the partial compilation approach can be adopted also in more general cases.

Our partial compilation is embeddable in existing ASP solvers thanks to programmatic APIs and we implemented it within the WASP solver.

In the next sections, we first provide the conditions under which our compilation-based approach can be adopted and then we discuss the approach itself and evaluate and analyze it in different settings. The system is available

for download. [1]

3.1 Conditions for Splitting and Compiling

In this section, we describe the conditions for our partial compilation of ASP programs. See chapter 1 for a reference to ASP syntax and semantics. We call our compilation approach partial because it cannot be used to compile any ASP program in general, but it is suitable to compile certain classes of subsets of rules of an input program and, as we will see later, it is suitable to be nested into other evaluation strategies for ASP.

A sub-program of π is a set of rules $\lambda \subseteq \pi$. Note that any sub-program is also an ASP program. In what follows, we denote with H_r the set of predicate names appearing in the head of a rule r and with B_r^+ (resp. B_r^-) the set of predicate names appearing in the positive (resp. negative) body of r. We denote with $\mathcal{Q}_\pi$ the set of predicate names appearing in an ASP program π, and we denote with $\mathcal{H}_\pi$ the set of predicate names appearing in the head of any rule of an ASP program π.

We provide the conditions for a sub-program to be compilable under our compilation-based approach by using the concept of labeled dependency graph of an ASP program.

Definition 1. *Given an ASP program π, the dependency graph of π denoted with DG_π is a labeled graph (V, E) where V is the set of predicate names appearing in some head of π, and E is the smallest subset of $V \times V \times \{+, -\}$ containing*

- $(V_1, V_2, +) \in E$ *if* $\exists r \mid V_1 \in B_r^+ \wedge V_2 \in H_r$;
- $(V_1, V_2, -) \in E$ *if* $\exists r \mid V_1 \in B_r^- \wedge V_2 \in H_r$; *and*
- $(V_1, V_2, -) \in E$ *if* $\exists r \mid V_1, V_2 \subseteq H_r$.

Intuitively, the dependency graph contains positive (resp., negative) arcs from positive (resp., negative) body literals to head atoms, and negative arcs between atoms in a disjunctive head.

Definition 2. *Given an ASP program π, π is said to be stratified iff its labeled dependency graph has no loop containing a negative edge.*

[1] `https://goo.gl/Varv4y`

Note that we here consider constraints to be stratified, as we consider them to have an empty head.

Definitions provided above are classical definitions for ASP programs.

Now we can define when an ASP sub-program is compilable:

Definition 3. *Given an ASP program π, an ASP sub-program $\lambda \subseteq \pi$ is* compilable with respect to π *iff:*

1. *λ is a stratified ASP program and*
2. *$p \in \mathcal{H}_\lambda \implies p \notin \mathcal{Q}_{\pi \setminus \lambda}$ (i.e. no predicate appearing in an head of λ can appear in $\pi \setminus \lambda$)*

Intuitively, stratified sub-programs are compilable if they do not produce atoms that appear elsewhere in the input program.

By defining compilation on stratified sub-programs we allow for the compilation of constraints, which satisfies the original goal of replacing hand-written constraints instantiators with automatically generated ones. Moreover, rules allow the compilation of more generic fragments.

Secondly, we want to comply with the propagators' input/output interface provided by the programmatic APIs of ASP solvers in which no new variables are introduced in the solver. This is the reason why we enforce that atoms defined in the compiled program do not appear elsewhere in the original input program.

By considering Definition 3, we can observe that an ASP constraint is always compilable because:

- constraints are stratified because they produce an empty dependency graph which is stratified by definition
- constraints have empty heads, thus condition 2 is trivially satisfied

Consider the following example:

```
(1) in(X) | out(X) :- v(X).
(2) r(X,Y) :- e(X,Y).
(3) r(X,Y) :- e(X,Z), r(Z,Y).
(4) :- a(X), a(Y), X ≠ Y, not r(X,Y).
```

The example contains two compilable sub-programs, one given by constraint (4) and one given by constraint (4) together with rules (2) and (3).

Compilable subprograms are related to Rule Splitting Sets of HEX programs [28], however, we here define them on the basis of predicates, not partially ground atoms. ASP Modules [48] are more permissive than compilable subprograms because they permit mutually cyclic (negative) dependencies among modules, which is not possible in compilable subprograms.

3.1.1 Largest Compilable Sub-program

Selecting compilable sub-programs might not be an easy task. In this section, we provide a procedure that, given an ASP program, determines the largest compilable sub-program. This can be useful in selecting compilable subprograms because we can start from a maximal subprogram and we can then discard unwanted rules and constraints. Moreover, we provide a sketched proof that the union of two compilable sub-programs is still a compilable sub-program and thus every ASP program has exactly one (possibly empty) largest compilable sub-program.

Theorem 1. *Given two compilable sub-programs λ_1 and λ_2, $\lambda_1 \cup \lambda_2$ is also a compilable sub-program.*

Proof sketch. Condition 2 of definition 3 holds because both sub-programs cannot define atoms that appear elsewhere, so the predicates in the union of their defined predicates do not appear elsewhere. Condition 1 holds too, because of the following properties:

- an SCC is either completely included in a compilable sub-program or completely excluded because of condition 1
- there is no loop between two SCCs

□

Corollary 1.1. *Given an ASP program π, π has exactly one largest compilable sub-program λ (possibly empty), that is the union of all compilable sub-programs of π.*

In the following, we provide a procedure that computes the largest compilable sub-program by working on the strongly connected components of the labeled dependency graph.

Definition 4. *A strongly connected component SCC of DG_π is a subgraph of DG_π that is strongly connected (i.e. every vertex in SCC is reachable from every other vertex in SCC) and is maximal (i.e. no other vertex can be added to SCC without breaking the strongly connected property).*

By considering a *SCC* as a single node we can build a contraction of DG_π as a graph DAG_π where there is an edge between two *SCC* iff any two nodes of them form an edge in DG_π. DAG_π is a directed acyclic graph and we can compute the largest compilable sub-program of π by first building a topological sort and then iterating over the SCCs once in reverse order w.r.t. the topological sort, collecting those SCCs that are stratified and whose children are also compilable as illustrated in algorithm 4. Note that we also add to the largest compilable sub-program all constraints.

Algorithm 4 Find largest compilable sub-program

Input: ASP program π
Output: largest compilable sub-program of π

1: $\lambda = \emptyset$
2: *//add all constraints in* π
3: **for all** $r \in \pi \mid H_r = \emptyset$ **do**
4: $\quad \lambda = \lambda \cup \{r\}$
5: **end for**
6: $DG = \text{dependecy\_graph}(\pi)$
7: $DAG = \text{components\_acyclic\_graph}(DG)$
8: $(SCC_1, \ldots, SCC_n) = \text{topological\_order}(DAG)$
9: *//compilable components*
10: $CC = \emptyset$
11: **for all** $i \in (n, \ldots, 1)$ **do**
12: $\quad$ **if** $is\_stratified(SCC_i)$ **then**
13: $\quad\quad to\_insert = \top$
14: $\quad\quad$ **for all** $j \in (n, \ldots, i+1)$ **do**
15: $\quad\quad\quad$ **if** $(SCC_i, SCC_j) \in \text{edges}(DAG) \wedge SCC_j \notin CC$ **then**
16: $\quad\quad\quad\quad to\_insert = \bot$
17: $\quad\quad\quad$ **end if**
18: $\quad\quad$ **end for**
19: $\quad\quad$ **if** $to\_insert = \top$ **then**
20: $\quad\quad\quad$ **for all** $p \in \text{vertices}(SCC_i)$ **do**
21: $\quad\quad\quad\quad$ **for all** $r \in \pi \mid H_r = p$ **do**
22: $\quad\quad\quad\quad\quad \lambda = \lambda \cup \{r\}$
23: $\quad\quad\quad\quad$ **end for**
24: $\quad\quad\quad$ **end for**
25: $\quad\quad\quad CC = CC \cup SCC_i$
26: $\quad\quad$ **end if**
27: $\quad$ **end if**
28: **end for**
29: **return** λ

3.2 Architecture for Partial Compilation

Partial compilation is designed to be embeddable in existing ASP systems. This is possible due to recent works on ASP solvers to enable solvers exten-

sion by means of carefully designed APIs [36][26]. This makes ASP compilation more appealing due to the consolidated performances of state-of-the-art solvers. Figure 3.1 shows a high-level architecture of our partial compilation embedded in an ASP solver. ASP programs can be split into two (or more) parts, one being evaluated with the standard "ground+solve" approach and the other being evaluated by compilation. The director of the solving process is the ASP solver which interacts with the compiled program with a well-defined input/output interface. In the presented architecture the role of *compiler* is to take as input an ASP program that meets the conditions presented in the previous section and build a compiled component that materializes (i.e. generates source code) an evaluation procedure for the input program.

Algorithm 5 presents a procedure that integrates an ASP solving component with a compiled sub-program. Given an ASP program π which is split into two sub-programs π' and λ (which is a compilable program compiled into λ_c^{eval}), the solver searches for an answer set M of π'; if the solver fails to find an answer set it returns $\perp$ which means that π' and π are unsatisfiable, otherwise it provides M to the compiled program which returns a set of nogoods N and a model M_{λ_R}. In this thesis, nogoods identify ground constraints over the universe of ground atoms of the solving component. This notion has basically the same behaviour as the one given in [39].

$N = \emptyset$ means that $\lambda \cup M$ is satisfiable, then M_{λ_R} is an answer set of $\pi' \cup \lambda$. Otherwise (i.e. when $N \neq \emptyset$) the solver learns the set of nogoods N and searches for another answer set until λ_c^{eval} returns an empty set of no-goods or no more answer sets of π' are found.

An important fact is that, by using our compilation-based approach, the compiled rules are not involved in the grounding process and we have to account for the fact that the solver does not know about predicates appearing in the head of the compiled program (see Definition 3). This fact leads to the need of introducing some algorithms for explaining why certain literals are true in the compiled program, and such explanations (or reasons) are intended to be expressed in terms of literals that are known to the ASP solver. We will talk more about literal reasons in the next sections.

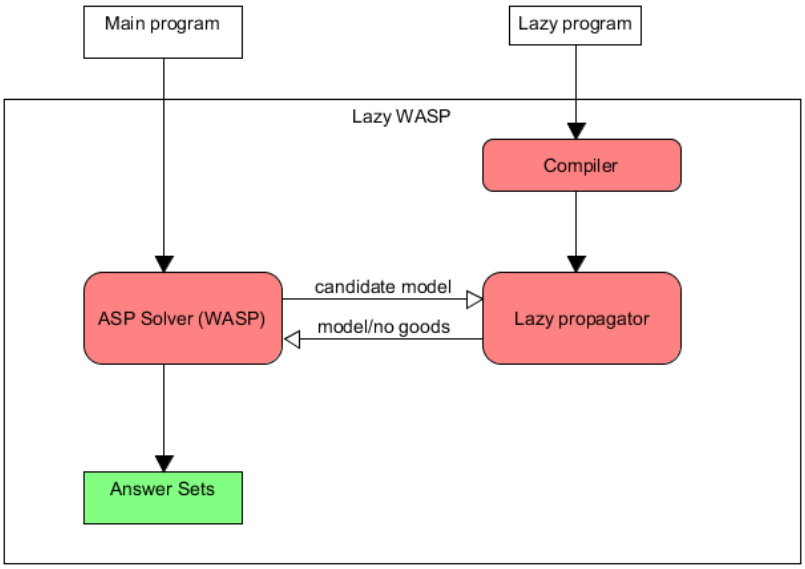

Figure 3.1: Partial compilation architecture

Algorithm 5 Solving with a compiled program

Input: ASP program π', ASP compilable program λ

1: λ_c^{eval} =compile(λ)
2: M = next_answer_set(π')
3: **while** $M \neq \bot$ **do**
4: $(N, M_{\lambda_R}) = \lambda_c^{eval}(M)$
5: **if** $N \neq \emptyset$ **then**
6: learn(N)
7: **else**
8: **return** M_{λ_R}
9: **end if**
10: M = next_answer_set(π')
11: **end while**
12: **return** $\bot$

3.3 Compiled Program Evaluation

In the following sections π is the input ASP program, π' is the solver program and λ is the compiled program, with the conditions that $\pi = \pi' \cup \lambda$ and λ is a compilable ASP program. λ can be seen as the union of a set of ASP constraints λ_C and a set of ASP rules (with a non-empty head) λ_R.

$$\lambda = \lambda_R \cup \lambda_C$$

We have to clarify that by compilation we mean the instantiation of a general-purpose evaluation strategy with respect to a fixed ASP (sub-)program. For the sake of simplifying presentation, we first present a general-purpose evaluation strategy that is a generic version (i.e., valid for any compilable input program) of what is compiled, and then we describe how this strategy can be instantiated during compilation depending on the input program. Basically, the procedure is compiled in a version that is specific for the rules at hand simplifying it during the compilation process. This will be made more clear in the following.

Our general-purpose evaluation strategy is presented in the next two sections. It is essentially composed of two steps:

1. a bottom-up evaluation of λ w.r.t. an ASP solver model of π' M, followed by

2. a top-down nogoods building for failed constraints checks in λ_C.

3.3.1 Bottom-up Evaluation

The first step of our partial evaluation consists of a bottom-up evaluation performed on λ where the set of input facts is given by an answer set M of π. Historically, bottom-up strategies are the standard way to approach stratified programs evaluation [76]. Algorithm 6 presents a pseudo-code of the bottom-up evaluation that we use in our implementation. It closely follows the computation of ASP grounders, but the presented version is organized in a way that is easy to compile. For example, loops are written in such a way that compile-time information is pushed to outer loops so that they can be unrolled, while instance specific information is only present in the innermost loops which cannot be handled at compilation-time. In the presented algorithm R is the model of λ, $SCCs$ is the ordered set (w.r.t. a topological

sort) of the strongly connected components of the dependency graph DG of λ. The evaluation starts with the computation of the dependency graph. Then it performs the evaluation of one strongly connected component at a time by following a topological sort of the dependency graph. The for loops at line 5 and 14 iterates over all predicate names in the current SCC.

For each component, firstly so-called exit rules are evaluated, then recursive rules in a nested loop. A rule r in a component C is an exit rule iff all predicates in B_r belong to a component that precedes C in the topological sort. This means that the body predicates extensions are already fixed and they will not change anymore during the computation. Otherwise, r is said to be recursive (i.e. there is some body predicate in the body of r that belongs to C).

R_P denotes the extension of a predicate P in the current program model R. W sets are instead used as working sets in recursive rules and are used to accumulate recursive predicates in the evaluation of a recursive SCC. W is split into individual working sets W_P for each predicate in SCC.

The function *evaluate* stands for a nested-join loop of a rule r starting from a ground atom s (i.e. it finds all rule instantiation with given s). In exit rules, a body positive predicate is selected to start the nested loop, while for recursive components, the evaluation continues as long as there are new ground atoms for the predicates in the recursive component using accumulation sets in W.

The computation of constraints violation is done at the end of the bottom-up evaluation. Here again, any positive predicate can be used to start a nested join evaluation. For constraints we use the term *ground* instead of *evaluate* to denote the fact that the set C will contain all ground constraints violations of λ w.r.t. the input candidate model M.

Algorithm 6 bottom_up_evaluation

Input: ASP stratified program $\lambda = \lambda_R \cup \lambda_C$, answer set of π' M
Output: answer set of $\lambda \cup M$

```
1: DG = dependecy_graph(λ)
2: R = load_facts(M)
3: SCCs = topological_sort(DG)
4: for all SCC ∈ SCCs do
5:    for all predicates P ∈ SCC do
6:       for all exit rules r ∈ λ_R with P = H_r do
7:          S = starter_predicate(r)
8:          for all s ∈ R_S do
9:             R_P = R_P ∪ evaluate(r, s, R)
10:         end for
11:      end for
12:   end for
13:   W = ∅
14:   for all P ∈ SCC do
15:      W_P = R_P
16:      W = W ∪ W_P
17:   end for
18:   while ∃W_P ∈ W | W_P ≠ ∅ do
19:      for all W_P ∈ W do
20:         while W_P ≠ ∅ do
21:            for all r ∈ λ_R | H_r ∈ SCC ∧ P ∈ B_r^+ do
22:               for all s ∈ W_P do
23:                  E = evaluate(r, s, R)
24:                  W_{H_r} = W_{H_r} ∪ (E \ R_{H_r})
25:                  R_{H_r} = R_{H_r} ∪ E
26:                  W_P = W_P \ {s}
27:               end for
28:            end for
29:         end while
30:      end for
31:   end while
32: end for
33: C = ∅
34: for all r ∈ λ_C do
35:    S = starter_predicate(r)
36:    for all s ∈ R_S do
37:       C = C ∪ ground(r, s, R)
38:    end for
39: end for
40: return R
```

3.3.2 Nogoods for Failed Constraints Checks

In this section, we describe how we build the reasons (in terms of nogoods) of failure of constraints checks. Before directly discussing the algorithms we first provide a theoretical framework that arises from the use-case of partial compilation. The theoretical framework defines the scope of the implementation and motivates the solution adopted. After the theoretical part, we propose a mathematical operator that follows the theory of the theoretical framework and of which the algorithms proposed are a *smarter* implementation.

Theoretical framework

Definition 5. *Given an ASP program π, a nogood w.r.t. π is a set of ground literals N s.t. $\nexists A \in ANS(\pi) \mid N \subseteq A$. In other words $N \cup \pi$ is incoherent.*

As introduced in section 3.2, in our setting, π is an ASP program that is split into two sub-programs, i.e. $\pi = \pi' \cup \lambda$ where π' is the external program and λ is the propagator program. Moreover we have $\lambda = \lambda_R \cup \lambda_C$ where λ_R is a set of stratified rules and λ_C is a set of constraints.

Definition 6. *Given two sets of ground literals L^1, L^2 we say that L^1* leads to *L^2 w.r.t. π', λ and we denote it with $L^1 \Rightarrow L^2$ iff $\forall M \in ANS(\pi) \mid L^1 \subseteq M_{\lambda_R}$ we have that $L^2 \subseteq M_{\lambda_R}$, where M_{λ_R} denotes the unique model of $M \cup \lambda_R$.*

Definition 7. *Given a set of (possibly non-ground) literals $\mathcal{L}$ and a model M of π', we denote with $\overline{\mathcal{L}}$ the set $\{l \in M_{\lambda_R} \mid \exists L \in \mathcal{L}$ s.t. $l \doteq L\}$*

Theorem 2. *If we have that $L^1 \Rightarrow L^2$ and $L^2 \Rightarrow L^3$, then $L^1 \Rightarrow L^3$.*

Proof. By def 6, $L^1 \Rightarrow L^2$ iff $\{M \mid L^1 \subseteq M_{\lambda_R}\} \subseteq \{M \mid L^2 \subseteq M_{\lambda_R}\}$ and $L^2 \Rightarrow L^3$ iff $\{M \mid L^2 \subseteq M_{\lambda_R}\} \subseteq \{M \mid L^3 \subseteq M_{\lambda_R}\}$. Then we have $\{M \mid L^1 \subseteq M_{\lambda_R}\} \subseteq \{M \mid L^3 \subseteq M_{\lambda_R}\}$, which by def 6 implies $L^1 \Rightarrow L^3$. □

Definition 8. *A reason of a set of literals $\mathcal{L}$ is a set of literals $R(\mathcal{L}) \mid \overline{R(\mathcal{L})} \Rightarrow \overline{\mathcal{L}}$.*

We denote with $\mathcal{R}(\mathcal{L})$ the set of all possible reasons of $\mathcal{L}$.

Theorem 3. *Given a set of literals $\mathcal{L}$ it holds that any $R(R(\mathcal{L})) \in \mathcal{R}(\mathcal{L})$, i.e. any $R(R(\mathcal{L}))$ is a reason for $\mathcal{L}$.*

Proof. By def 7, $\overline{R(R(\mathcal{L}))} \Rightarrow \overline{R(\mathcal{L})}$ and $\overline{R(\mathcal{L})} \Rightarrow \overline{\mathcal{L}}$. By theorem 1, $\overline{R(R(\mathcal{L}))} \Rightarrow \overline{\mathcal{L}}$. □

Definition 9. *Given M a model of π' and $R(\mathcal{L})$ a reason for a set of literals $\mathcal{L}$, we say that $R(\mathcal{L})$ is a complete reason for $\mathcal{L}$ iff $R(\mathcal{L}) \subseteq M$. In such case, we denote it with $R^C(\mathcal{L})$*

Theorem 4. *Given $M \in ANS(\pi')$ and λ_C^G the set of ground constraints of $M \cup \lambda$, then $\{R^C(C) \mid C \in \lambda_C^G\}$ is a nogoods set for π and is expressed in terms of $U_{\pi'}$.*

Proof. A complete reason is a subset of M, thus being $\{R^C(C) \mid C \in \lambda_C^G\}$ composed of complete reasons this fact implies that it is expressed in terms of $U_{\pi'}$. Now, given $c \in \lambda_C^G$, by def 8 we have that $\overline{R^C(C)} \Rightarrow \overline{C}$ and since $R^C(C) \subseteq M \subseteq M_{\lambda_R}$ and $C \subseteq M_{\lambda_R}$, we can conclude that $\overline{R^C(C)} = R^C(C)$ and $\overline{C} = C$. So, $R^C(C) \Rightarrow C$. By def 6, we have that $\forall M \in ANS(\pi) \mid R^C(C) \subseteq M_{\lambda_R}, C \subseteq M_{\lambda_R}$, thus $R^C(C) \cup \pi$ is incoherent, thus $R^C(C)$ is a nogood for π (def 5) and $\{R^C(C) \mid C \in \lambda_C^G\}$ is a nogoods set for π. □

Theorem 5. *Given a set of literals L, $(\bigcup_{l \in L} R(l)) \in \mathcal{R}(\mathcal{L})$ (i.e. the union of the reasons of each $l \in L$ is a reason for L).*

Proof. $R(l)$ a reason for l means that, by def 6 and 8, for all l we have that $\{M|\overline{R(l)} \in M_{\lambda_R}\} \subseteq \{M|\bar{l} \in M_{\lambda_R}\}$. Thus, by the fact that $A \subseteq B \wedge C \subseteq D \implies A \cup C \subseteq B \cup D$, we have $\{M|\overline{\bigcup_{l \in L} R(l)} \in M_{\lambda_R}\} \subseteq \{M|\overline{\bigcup_{l \in L} l} \in M_{\lambda_R}\}$. By def 6 this holds iff $\overline{\bigcup_{l \in L} R(l)} \Rightarrow \overline{L}$, which by 8 implies that $(\bigcup_{l \in L} R(l)) \in \mathcal{R}(\mathcal{L})$. □

Seminaive reasons operator

Here we introduce a fixpoint operator R that, applied recursively, is able to compute a nogood starting from a failed constraint check. In particular, R is a function R that maps a couple of the type $\langle L, S\rangle$ into a couple in the same domain; L and S are sets of (possibly non-ground) literals. The operator is defined as follows:

$$\mathsf{R}(\langle S, L\rangle) = \langle S', (\bigcup_{l \in L} r(l)) \setminus S')\rangle$$

Where $S' = (L \cup S) \setminus M$ and the expression $r(l)$ is then defined as follows:

$$r(l) = \begin{cases} \{l' \in M \mid l' \doteq l\} & \text{if } p(l) \in p(\pi') \\ \{\sigma(B_r) | H_r \overset{\sigma}{=} l' \wedge l' \in M_{\lambda_R} \wedge l' \doteq l\} & \text{if } p(l) \in p(\lambda_R) \wedge l \text{ positive} \\ \{\sigma(B_r^{negated}) | {\sim}H_r \overset{\sigma}{=} l\} & \text{if } p(l) \in p(\lambda_R) \wedge l \text{ negative} \end{cases} \tag{3.1}$$

In the above definition, $B_r^{negated}$ is the set obtained by negating all body literals (e.g. if r is $a(X) : -b(X,Y), {\sim}c(Y)$, then $B_r^{negated} = \{{\sim}b(X,Y), C(Y)\}$), while σ denotes a substitution with the convention that $\sigma(expression)$ applies σ to the expression, while $expr1 \overset{\sigma}{=} expr2$ denotes a substitution from variables in $expr1$ to terms in $expr2$ s.t. $\sigma(expr1) = expr2$. The operator R defines a recursive sequence where $\langle S^i, L^i \rangle = \mathsf{R}(\langle S^{i-1}, L^{i-1} \rangle)$. Finally, by fixing $L^0 = C$ where C is the set of literals of a ground violated constraint and $S^0 = \emptyset$, we have that:

1. the operator reaches a fixed point at step k with k finite
2. at the step k the operator computes a nogood for C expressed as a subset of M, i.e. it computes a complete reason for C (see def. 9)

We omit a complete formal proof of termination and correctness of the operator, but we informally argument its termination and correctness in the remaining of the paragraph. We leave the formalization of the proof as a future work. The fact that the operator reaches a fixed point is ensured by the following property: at each step i, it either leave literals in L as they are, (in case they are already in M) or it replaces them by literals that are new (i.e. that are not in S), thus, since there is a finite number of possible literals (in the universe), it will eventually leave L as it is and reach the fixed point. Termination also ensures that all literals at the fixed point are in M because otherwise they would be removed and replaced by new literals. For what concerns condition 2, at each step the following property holds: L^n is a reason for L^{n-1}, thus at each step L^n is a reason for C which is L^0. First, we note that $r(l)$ computes a reason for l: indeed, for positive literals it adds all body literals of all rules whose head unifies with l, hence, every literal of every possible body that would produce l is in the reason; for negative literals it adds the negation of the literals of all rules whose head unifies with l, hence, every literal that would falsify a rule that could produce l is in the

reason. Finally, we note that by theorem 5 the union of the reasons of the literals in L is a reason for L.

Algorithm 7 build_reason_for_violated_constraint

Input: Violated ground constraint C, Candidate Model M, Extendend Model M_{λ_R}, Open Set O

1: $R = \emptyset$
2: **for all** $l \in C$ **do**
3: $\quad R = R \cup \text{explain\_literal}(l, M, M_{\lambda_R}, O)$
4: **end for**
5: **return** R

Implementation

Algorithms 7, 8 and 9 present in pseudo-code a procedure for building nogoods from ground constraints failure of the compiled program. Such algorithms are essentially a smarter version of the operator presented above.

In the same spirit of the bottom-up evaluation, the presented procedures are general-purpose and are like a template of what code runs in the compiled program because the compiler generates a custom instantiation of such procedures by exploiting the fact that rules are known at compilation time.

In all algorithms, M_{λ_R} is an answer set of $M \cup \lambda_R$ where M is an answer set of π'. Since λ_R is stratified there is only one such answer set for each M.

Algorithm 7 builds nogoods from a ground violated constraint C. A ground violated constraint is a ground instantiation of a constraint c in λ that is obtained by applying all possible substitution of ground atoms to c. See grounding of ASP programs [59].

Our compiler performs body reordering as preprocessing to ensure that negative literals are pushed to the end of the rule body (or at so far that all variables appearing in that negative literal were already substituted with a constant value in a preceding positive literal).

Algorithm 8 builds nogoods for a given input literal l. Literal l is either a positive or a negative literal. If it is a negative literal it can also be non-ground, while if it is positive it is ground, since in no point we are interested in the reason of a positive non-ground literal. First the literal is transformed in a canonical form. The canonical form of a literal $l(t_1, \ldots, t_n)$, is a literal $l(t'_1, \ldots, t'_n)$ where $t'_1, \ldots, t'_n$ are constants or variables in a fixed variables set

Algorithm 8 explain_literal

Input: Literal l, Candidate Model M, Extended Model M_{λ_R}, Open Set O
Output: reason for literal l being true

1: *//l can only be non-ground if it is negative, thus it is a positive ground literal or a negative literal*
2: *//variables are meaningless in this context, e.g. we do not distinguish between p(X) and p(Z)*
3: $l \leftarrow$ canonical_form(l)
4: *//if literal is known to the ASP solver return it*
5: **if** is_ground(l) $\wedge$ $l \in M$ **then**
6: **return** {l}
7: **end if**
8: *//stop, if we are in a loop*
9: **if** $l \in O$ **then**
10: **return** $\emptyset$
11: **end if**
12: $O = O \cup \{l\}$
13: $R = \emptyset$
14: **if** is_positive(l) **then**
15: *// we have partial reasons for l, so we build the reason recursively using its partial reasons*
16: **for all** $r \in$ evaluation_reasons(l) **do**
17: $R = R \cup$ explain_literal(r, M, M_{λ_R}, O)
18: **end for**
19: $O = O \setminus \{l\}$
20: **return** R
21: **end if**
22: *//negative case, here the literal can be either ground or non-ground*
23: *//we put the literals in the model_generator that unifies the atom of l*
24: *//for example, if $M = \{\sim a(1), \sim a(2)\}$ the reason of $\sim a(X)$ is $\{\sim a(1), \sim a(2)\}$*
25: **for all** $l' \in M \mid l'$ is negative $\wedge$ $l' \doteq l$ **do**
26: $R = R \cup l'$
27: **end for**
28: *//we build the reason for a negative literal going from head to body in the rules whose head unifies with l*
29: *//it is false because all rules that could have fired did not fire*
30: **for all** $r \in \lambda\_R \mid l \doteq \sim H_r$ **do**
31: explain_negative_literal_from_rule$(R, r, 0,$ compute_substitution$(l, H_r), M, M_{\lambda_R}, O)$
32: **end for**
33: $O = O \setminus \{l\}$
34: **return** R

Algorithm 9 explain_negative_literal_from_rule

Input: OutputParameter R, Rule r, index i, Candidate Model M, Extended Model M_{λ_R}, Substitutions σ, Open Set O

```
1: //We select the i-th literal of r and apply the substitution to it
2: l = apply_substitution(body_literal(r, i), σ)
3:
4: if is_positive(l) then
5:    //add all reasons of possibly positive literals unifying lit (join stops)
6:    //i.e. reasons for substitutions where the join stops at i-th body literal

7:    R = R ∪ explain_literal(∼l, M, I, O)
8:    for all b ∈ M_λR | b ≐ l do
9:       //recursive call to the next body literal for succeeding joins
10:      //i.e. we continue the join and it will stop after l
11:      explain_negative_literal_from_rule(R, r, i + 1, M, M_λR,
         σ ∪ compute_substitution(l, b), O)
12:   end for
13:
14: else
15:   //in the else we have a negative ground literal because rules are safe
      and we push negation to the end
16:   if l ∉ I then
17:      //join stops here
18:      R = R ∪ explain_literal(∼l, M, M_λR, O)
19:   else
20:      //recursive call to the next literal for succeeding joins
21:      explain_negative_literal_from_rule(R, r, i + 1, σ, M, M_λR, O)
22:   end if
23: end if
```

$\{X_1, X_2, X_n\}$. Canonical transformation proceeds left to right. A constant t_i is mapped to itself, while a variable t_i is mapped to the next unused variable in $\{X_1, X_2, X_n\}$ if there is no $t_j = t_i$ with $j < i$, otherwise it applies the mapping $t'_i = t'_j$. If the literal is known to the ASP solver we return the literal itself. The open set O is used to prevent loops (e.g. while building the nogoods for a literal l we ask for the nogoods of l). At line 14, if literal is positive (thus ground because of preconditions of algorithm 8, it means that $l \in M_{\lambda_R} \setminus M$. Thus, it was produced in the compiled program by a rule evaluation and we have partial reasons for it (that is a set of literals B such that $\exists r \in \lambda \mid B = B_r \wedge l = H_r$). In such case we iterate over such B that we call *evaluation reasons* of l and we call *explain_literal* recursively.

In case l is negative, we explain why the atom of l is false. In case the predicate of l is known to the model generator we return all negative ground literals in M that unifies l. Symbol $\doteq$ represent unification check between two literals.

Otherwise (i.e. the predicate of l only appears in the compiled program), we look for rules whose head could have produced the atom of l in the evaluation and did not. Thus, we iterate over all rules whose head unifies the atom of l and we call the procedure presented in algorithm 9 for getting an explanation of why a rule did not produce the atom of l. Here we work top-down, attempting to build rules by starting from a head literal that could have been produced by such rules. We add to the reasons all literals that stop the top-down search. It always stops at some point before reaching the end of a rule because otherwise the literal would have been true (and we would not have looked for a reason of its falsity).

Thus, in algorithm 9 we want to explain why a given rule did not fire. Such procedure receives an index parameter which iterates over rules bodies through recursive calls and it is initialized to 0 in the first call that starts from algorithm 8, where 0 means the first body literal of the rule.

The first time we call algorithm 9 we prepare a substitution function that maps variables in rules head to constants in the literal we are explaining (function compute_substitution). For example, if we are explaining $\sim a(1, X)$ and we start algorithm 9 on a rule whose head is $a(Y, Z)$ we map Y to 1 in the substitution function. The first step of 9 is to apply the substitution to the i-th body literal. Then, if l is positive we add the reasons for $\sim l$ (i.e. where the join stops because of l is false). For the cases where the join stops later we extend the substitution sigma and recursively call 9 incrementing the index i.

If l is negative, we know that l is also ground because rules are safe and we push negation to the end of rules. Thus, if l is false it means that $a \in M_{\lambda_R}$, thus the join stops here and we add the reasons for a. Finally, if l is true, the join stops later and we call algorithm 9 recursively for the next body literal.

The presented algorithms follow the theory presented at the beginning of this section. The reasons are built by producing complete reasons of violated ground constraints. Indeed, the algorithm is a smarter version of the operator R where the function r(l) is replaced by a smarter implementation that uses evaluation reasons as the reason of positive literals and uses a smaller reason for negative literals thanks to the fact that we partially ground such rules instead of using the whole negated body. So, the algorithms compute reasons recursively, by exploiting the fact that, as stated above, the reason for a reason for a literal l is still a reason for l. For positive literals, we use evaluation reasons that is a set of positive true literals: the condition of definition 8 holds trivially since, given a positive literal l the evaluation reasons consists of a positive body of a rule where l is the head, thus whenever all such literals are true the literal in the head is also true due to the samantics of ASP programs. For a negative literal l, we essentially produce a set of literals s.t. if they are all true, no rule that could produce $\sim l$ (i.e. all rules whose head unifies with $\sim l$) can have a true body. To guarantee termination we the open set O, in the same spirit of the S set of the seminaive operator.

3.4 Compilation Algorithm

We do not go into the details of the compiler algorithm. The implementation is very technical and a complete description would be hard to follow and very long. In this section, we provide an intuition of how the compilation algorithm works and some compilation techniques adopted in our compiler.

Our compilation algorithm generates the compiled component by instantiating algorithms 6, 7, 8, 9 w.r.t. a given ASP compilable program λ.

We use compilation techniques to eliminate or simplify code of the corresponding general purpose strategy by exploiting information known at compilation time. For example, the program dependency graph DG and the strongly connected components $SCCs$ are computed at compilation time, while facts loading cannot be resolved at compilation time since facts (i.e. the input interpretation) are not known.

All loops presented in algorithm 6 with the exception of loops at lines 8,

Algorithm 10 compiled_bottom-up_evaluation_example

Input: Interpretation M
Output: answer set of $\lambda_{ex} \cup M$

1: $R_a, R_b, R_e, R_r = load\_facts(M)$
2: **for all** $a \in R_a$ **do**
3: $X = a[0]$
4: $Y = a[1]$
5: **if** $(Y) \notin R_b$ **then**
6: $R_e = R_e \cup (X, Y)$
7: **end if**
8: **end for**
9: **for all** $e \in R_e$ **do**
10: $X = e[0]$
11: $Y = e[1]$
12: $R_r = R_r \cup (X, Y)$
13: **end for**
14: $W_r = R_r$
15: **while** $W_r \neq \emptyset$ **do**
16: **for all** $r \in W_r$ **do**
17: $Z = r[0]$
18: $Y = r[1]$
19: **for all** $X \mid e(X, Z) \in R_e$ **do**
20: $W_r = W_r \cup (\{(X, Y)\} \setminus R_r)$
21: $R_r = R_r \cup \{(X, Y)\}$
22: $W_r = W_r \setminus \{(Z, Y)\}$
23: **end for**
24: **end for**
25: **end while**
26: **return** $R_a \cup R_b \cup R_e \cup R_r$

22 and 36 are unrolled. Loops unrolling is a technique for rewriting loops into repeating instructions, eliminating or reducing loop control instructions. Loops unrolling potentially improves programs efficiency at the cost of increasing program size. Such unrollings can be done because loops conditions are known at compilation time and depend on the structure of the input program. Loops at lines 8, 22 and 36 cannot be unrolled since the extensions of predicates are not known at compilation time. Unrolling proceeds from outer loops to inner loops and works by duplicating as many times as the number of loop iterations the code inside a loop. Moreover, variables that would be assigned in the loop execution are replaced by constants at compilation time. Functions *evaluate* and *ground* are also partially unrolled because they both have an outer loop that iterates over body literals of r and such loop can be unrolled since rule bodies are known at compilation time.

Consider the input program λ_{ex} below:

```
e(X,Y) :- a(X,Y), not b(Y).
r(X,Y) :- e(X,Y).
r(X,Y) :- e(X,Z), r(Z,Y).
```

Algorithm 10 presents a pseudo-code of compilation applied to λ_{ex}. The only evaluation difference in structure w.r.t. the general-purpose version is that here we expand and unroll *evaluate* function of algorithm 6 lines 9 and 23, and we perform the operations of inserting evaluate results to result sets or working sets in the inner-most loop. In the algorithm, expressions like $p[i]$ stand for the term at index i of a ground atom p, with the first term having index 0.

We also want to clarify, that ad-hoc data-structures are declared in the compiled program in order to allow efficient set operations (see for example lines 12, 20, 21, 22) and sets data access by key (line 19).

For what concerns algorithms 7, 8, 9, we also apply unrolling as much as possible. In particular, while applying unrolling we avoid producing dead code inside conditional statements that are known to be false at compilation time like the one at line 14 of algorithm 8 or unreachable code below a return like the one at line 20 of algorithm 8.

Algorithm 9 is unrolled into separate procedures, one for each negative literal on which it can be invoked, while its inner recursive calls (line 11 and 22) are unrolled with a series of nested for loops.

Positive literals explanations cannot be unrolled instead, because there can be several rules that can potentially produce the same positive literal

(with different bodies), thus the specific procedure calls are determined at evaluation time.

We will not provide an example of compiled procedures for no-goods because they are long and hard to understand, so we prefer to leave the reader with the intuitions provided above which are the very core of the compilation strategy.

3.5 Experiments

We experimented with partial compilation in four different settings:

1. Compilation of stratified programs
2. Partial compilation of constraints
3. Partial compilation of rules and constraints
4. Partial compilation of rules

Time and memory for each run are limited to 10 minutes CPU-time and 6GB, respectively.

Compilation of stratified programs Here we used some benchmarks from OpenRuleBench, which is an open community benchmark designed to test rule engines. The benchmark includes queries evaluations, which is not the target of this work, thus we removed queries and run perfect model computation as done for example in [17]. Figure 3.2 shows a cactus plot of execution times. The baseline consists of three well-known ASP systems, namely Clingo, DLV and I-DLV.

Partial compilation of constraints For what concerns constraints, we used the same benchmarks that are used in our work on lazy grounding of constraints presented in chapter 2 i.e. stable marriage, packing and NLU.

Partial compilation of rules and constraints In setting number 3, i.e. compilation of rules and constraints, we devised two experiments:

- *non-partition removal coloring*: a benchmark taken from [15], where the authors uses a benchmark inspired by a real-world application in [42].

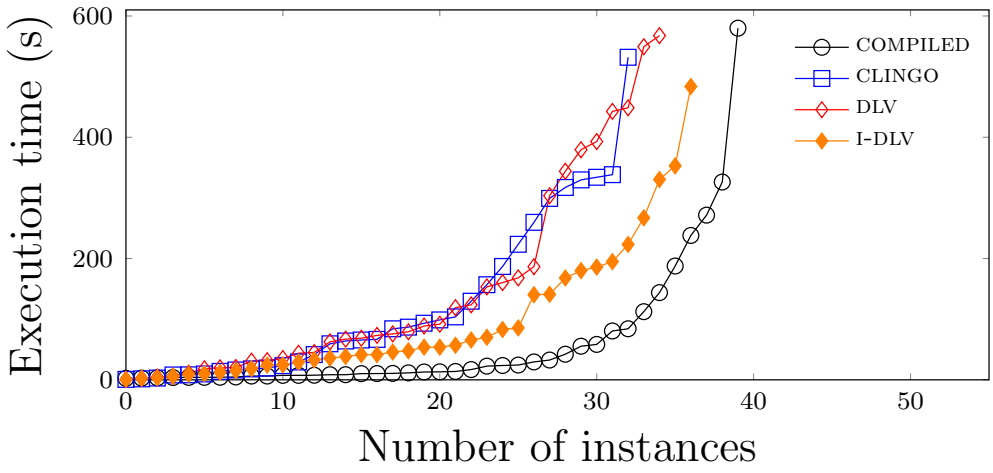

Figure 3.2: OpenRuleBench benchmark

- *connected k-cut*: a graph problem where we want to find a cut of size at least k where the two formed partitions are connected

In *non-partition removal coloring* the problem is the following: given a directed graph, remove one vertex in such a way that the transitive closure of the original and the resulting graph are equal on the remaining nodes and that the resulting graph is 3-colorable.

For what concerns *connected k-cut*, we generated instances at random experimenting on graphs with different numbers of nodes (from 200 to 800), different densities (from 0.001 to 0.25) and different cut sizes (from 50 to 800). The compiled sub-program computes the transitivity closure of the two partitions and contains a constraint that enforces connectivity.

The baseline here is composed of Clingo and WASP without compilation, and for the first benchmark, we also include the Alpha system [79], which is an ASP solver that exploits lazy solving techniques. Alpha authors are also the authors of this benchmark.

Partial compilation of rules Finally, for setting number 4 we present an experiment where we first solve a minimum cost cut problem and then we compute the transitive closure of the two partitions. The transitive closure in this benchmark introduces a hard (yet polynomial) post-processing to a hard (NP-hard) problem. Results are reported in figure 3.5.

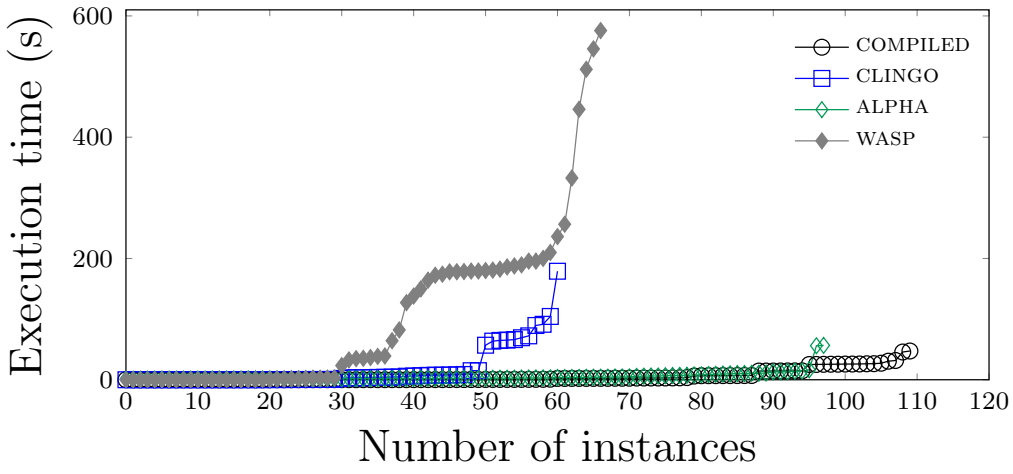

Figure 3.3: Non-partition removal coloring benchmark

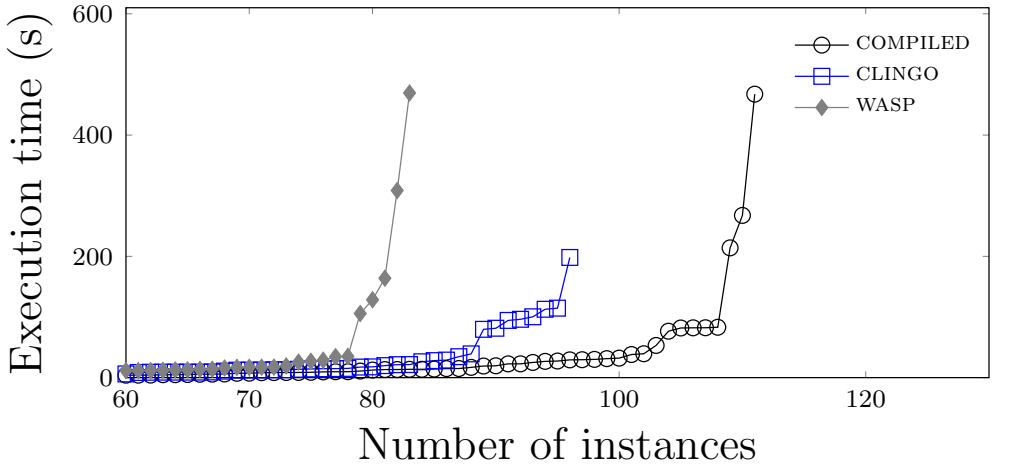

Figure 3.4: Connected k-cut benchmark

Table 3.1: Stable Marriage: Number of solved instances and average running time (in seconds).

Pref. (%)	**wasp**		**wasp-lazy**		**compiled**	
	sol.	**avg t**	**sol.**	**avg t**	**sol.**	**avg t**
0	10	6.2	10	5.8	10	5.6
5	10	25.3	10	5.7	10	5.8
10	8	48.2	10	5.4	10	5.6
15	9	38.3	10	6.8	10	5.6
20	9	50	10	5.9	10	5.4
25	7	52.6	10	5.9	10	5.9
30	10	60.1	10	6	10	5.7
35	5	111.4	10	6.3	10	8.3
40	7	63.3	10	9.4	10	20
45	8	83.8	10	6.3	10	11.3
50	9	67.9	10	6.4	10	8.3
55	7	124.4	9	7.2	9	9.4
60	8	63.3	10	11.5	9	10.7
65	6	66.7	6	18.2	9	17.1
70	6	71	3	21.8	5	132.3
75	8	89.9	0	-	1	13.8
80	7	148.9	0	-	0	-
85	6	107.2	0	-	0	-
90	9	152.2	0	-	0	-
95	10	70.3	0	-	0	-
100	8	61.9	1	7.3	0	-

Table 3.2: NLU Benchmark: Number of solved instances and average running time (in seconds).

Obj.Func.	**wasp**		**wasp-lazy**		**compiled**	
	sol.	**avg t**	**sol.**	**avg t**	**sol.**	**avg t**
card	48	83.0	50	2.8	50	2.3
coh	48	83.0	50	26.8	49	18.3
wa	48	103.2	49	23.6	49	38.5

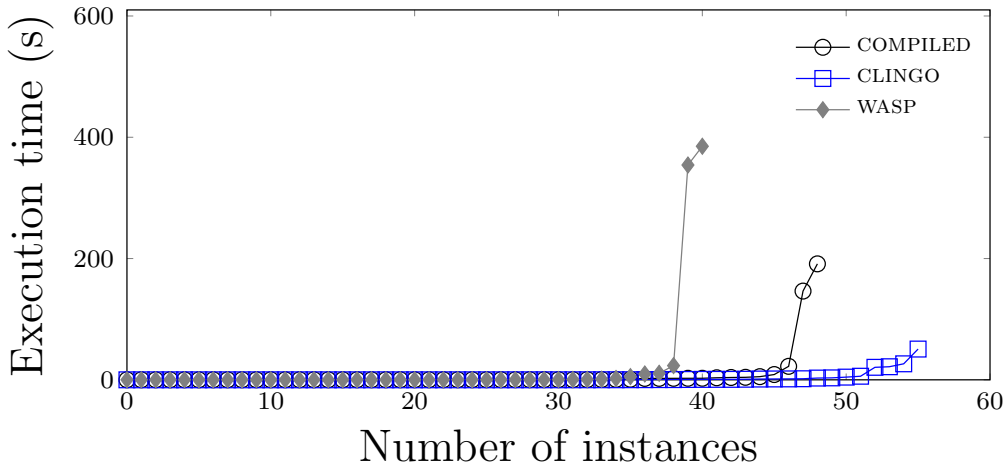

Figure 3.5: Mincut with transitive closure

3.5.1 Discussion of Experiments

Experimental results are promising. Stratified programs evaluation on the OpenRuleBench benchmark clearly shows the performance benefits that can be obtained by a compilation-based approach. In this benchmark, we solve more instances than classical approaches and we have lower execution times in general.

For what concerns experimental setting number 2, we are expecting the same behaviour of lazy propagators developed in our previous work (see chapter 2). The reason is that our evaluation follows the same execution pattern of lazy constraints evaluation, i.e. check the constraint on answer set candidates of the original input program without the lazy constraint. In our previous work we had to hand-write custom Python scripts for emulating the constraint, but here there is no such need because the compiler generates the propagator automatically. Indeed, here we are experiencing similar performances.

Setting number 3 includes a benchmark taken from [15] where the authors of the Alpha system present a benchmark in which lazy grounding pays-off w.r.t. "ground+solve" approaches. In this benchmark, we perform better than both standard solvers and Alpha (see 3.3). Another benchmark of the same setting is the *connected k-cut* benchmark and also, in this case, partial compilation pays-off, mainly because the connectivity sub-program is hard

to ground.

Finally, in setting number 4, we boost the performance of WASP, but we do not surpass Clingo because WASP seems to be much slower than Clingo at solving this problem.

3.6 Related Work

Traditional evaluation strategy of ASP systems is based on two steps, namely *grounding* and *solving*; for both phases, several efficient systems have been proposed during the years.

Concerning the grounding, state-of-the-art grounders are DLV [33], GRINGO [38] and IDLV [17]; all systems are based on seminaïve database evaluation techniques [76] for avoiding duplicate work during grounding.

Concerning ASP solvers, the first generation, i.e. SMODELS [74] and DLV [56], was based on a DPLL-like algorithm extended with inference rules specific to ASP. Modern ASP solvers, including CLASP [37] and WASP [5], include mechanisms for conflict-driven clause learning and for non-chronological backtracking. Both solvers also offer an external interface to simplify the integration of custom solving strategies in the main search algorithm. In particular, we used the interface of WASP to implement the techniques described in the thesis.

Alternative approaches are based on lazy grounding of the whole program, e.g., GASP [20], ASPeRiX [54], or Alpha [79], where all rules are instantiated lazily; this makes the search less informed but might have a better memory footprint. These 'fully lazy' approaches have in common, that they instantiate even the non-stratified part of the program only when rule bodies of the respective rules are satisfied in the current assignment of the search process, as opposed to our approach where all guesses are instantiated upfront and only stratified parts depending on guesses (including constraints) are computed lazily.

3.7 Discussion

Compilations-based approaches are meant to speed-up computation by exploiting information known at compilation time to create custom procedures that are specific to the problem at hand. In this chapter, we presented what

is, to the best of our knowledge, the first work on partial compilation of ASP programs. In our approach, we allow compilation of ASP sub-programs and we define what a compilable sub-program is, i.e. we specify what are the conditions under which our approach can be adopted. The presented approach has been developed as a solver extension of WASP which is a state-of-the-art ASP solver. The evaluation strategy presented includes a bottom-up evaluation for computing the unique stable model of the compilable sub-program and a top-down evaluation for computing failed constraints reasons in terms of literals that are known to the ASP solver. An experimental analysis shows the benefits that can be obtained in different use-cases by a compilation-based approach. The approach is particularly suited for solving Datalog programs, and for compiling ground-intensive sub-programs, as for example the ones presented in chapter 2. In the future, we are planning to extend the presented approach to the eager/post propagators case, i.e. where the evaluation is performed also on partial interpretations every time a new literal is chosen (eager) or when unit propagation ends (post). Another possible extension would be the development of an ASP grounder that uses compilation-based techniques. Indeed the problem of instantiating ASP programs can be seen as an extension of Datalog programs evaluation.

Part II

An Application of ASP to Closed-Domain Question Answering

Introduction

The information need of a user often resolves in a simple question where it would be useful to have brief answers instead of whole documents to look into. IR techniques have proven to be successful at locating relevant documents to the user query into large collections [8], but the effort of looking for a specific desired information into such documents is then left to the user. Question answering attempts to find direct answers to user questions. Intuitively, answering to any kind of question, with no linguistic and no domain restriction is a very hard task. When no restriction is made on the domain of the questions we are talking about open domain question answering while, when questions are bound to a specific domain, we are in closed (or restricted) domain question answering (CDQA) [2]. In open domain QA, most systems are based on a combination of Information Retrieval and NLP techniques [47]. Such techniques are applied to a large corpus of documents: first attempting to retrieve the best documents to look into for the answer, then selecting the paragraphs which are more likely to bear the desired answer and finally processing the extracted paragraphs by means of NLP. This type of approach is also adopted in several closed domain question answering systems, but in this context, we might benefit from existing structured knowledge. Some of the very early question answering systems were designed for closed domains and they were essentially conceived as natural language interfaces to databases [45][80].

In this work, we present a closed domain question answering system for the cultural heritage domain that comes from the PIUCULTURA project, which is a project of which the University of Calabria is a research partner.

The project was centered on the Italian language, but the presented approach is general and can work similarly for other languages.

Cultural Heritage can benefit from structured data sources: in this context, data has already started to be saved and shared with common stan-

dards. One of the most successful standards is the CIDOC Conceptual Reference Model [27], that has been identified as the ontological model of reference on cultural heritage for our Question Answering prototype. It provides a common semantic framework for the mapping of cultural heritage information and can be adopted by museums, libraries, and archives.

We designed and implemented a system capable of interpreting natural language questions regarding cultural heritage objects and facts, map the input questions into formal queries compliant to the CIDOC-crm model and execute such queries to retrieve the desired information.

We believe that the choice of using CIDOC-crm is valid because of the following key factors:

- Museums and institutions typically have structured sources in which they store information about their artifacts
- The availability of documentary sources is limited. If we take into consideration freely available documentary sources such as Wikipedia, we realize that the percentage coverage of works and authors can only be very low. For example, a museum like the British Museum has about 8 million artifacts while on Wikipedia there are in total around 500 thousand articles about works of art (from all over the world)
- The CIDOC-crm model has been specifically designed as a common language for the exchange and sharing of data on cultural heritage without loss of meaning, supporting the implementation of local data transformation algorithms towards this model
- The CIDOC-crm is a standardized maintained model and is periodically released in RDFs format

In summary, the Question Answering system has, therefore, the task of finding the information required by the user questions on an RDF knowledgebase that follows the CIDOC-crm model. The query language of RDF is SPARQL. So, in first approximation, we can say that the Question Answering system has to transform natural language questions into SPARQL queries. Our approach follows a waterfall model in which the user question is first processed from a syntactic point of view and then from a semantic point of view. Our syntactic processing model focuses on a concept of *template*, where a template represents a category of syntactically homogeneous questions. In

our system, templates are encoded in terms of Answer Set Programming rules. By using ASP we can work in a declarative fashion and avoid implementing the template matching procedure from scratch. The semantic processing is instead focused on a concept of *intent*. By intent we mean the purpose (i.e. the intent) of the question: two questions can belong to two disjoint syntactical categories but have the same intent and vice versa. To give an example: *who created Guernica?* and *who is the author of Guernica?* have a quite different syntactic structure, but have the same intent, i.e. *know who made the work Guernica*. On the other hand, if we consider *who created Guernica?* and *who restored Guernica?* we can say that they are syntactically similar (or homogeneous), but semantically different: the purpose of the two questions is different. Intents are mapped into SPARQL queries and the query result set is then converted in a natural language form by using a template metalanguage. The system favors components reuse and threats intents as functions that can be composed together to create larger intents (and queries).

Chapter 4

Preliminaries on Question Answering

Question Answering is a long-standing computer science discipline that is usually referred to in the field of Information Retrieval and Natural Language Processing. The first prototypes of Question Answering systems date back to the 60s: Lunar [80] and Baseball [45]. These two systems implemented the two main paradigms of question answering: answer to questions based on IR and answer to questions on structured knowledge bases. Countless Question Answering systems have been developed over the years. One of the most famous is IBM Watson that in 2011 beat human competitors at the American television game Jeopardy and although the goal of the game was the entertainment, the technology used to answer the questions of the television quiz led to an effective advancement of the techniques of QA. Most QA systems focus on factual questions, which are questions whose answer can be expressed in terms of a short text that represents a fact. The following questions fall into this category and can be answered in simple facts that denote respectively a person, an artwork and a place:

- Who painted Guernica?
- Which Van Gogh artwork represents sunflowers?
- Which museum houses the Rosetta Stone?

Below we will briefly talk about the two main paradigms of QA focusing on their application in answering factual questions and we will focus on the type

that is better suited to out context. The first paradigm is called Information Retrieval Question Answering or also with the simple name of text-based Question Answering. Systems of his type typically rely on a vast amount of information found in the form of text on the Web or in specific document collections. Given a user question, the QA system uses techniques similar to those of IR to extract passages directly from the documents, guided by the text of the input question. The second type of paradigm is known as Knowledge-Based Question Answering and is based on a semantic representation of the question in the form of a formal query on a knowledge base.

4.1 IR-based Question Answering

The models of QA that are based on Information Retrieval answer to user questions by searching for answers in text segments taken from the Web or other document collections. Typically, the flow of execution of an IR-based QA system consists of the following steps:

1. Question processing
2. Question classification
3. Query formulation
4. Passage extraction
5. Answer extraction

In the question processing phase, the goal is to extract useful information from the input question. The answers types specify the types of entities involved in the answer (e.g., people, places, etc.). The query specifies keywords that can be used by techniques similar to those of IR for the retrieval of relevant documents. Some systems perform an initial classification of the type of question at this stage (for example, if it is a question that asks for a definition, a mathematical question or a question from which there is a list of terms as an answer). In the question classification, the type of expected response is determined. For example, the question *who has painted Guernica?* has a PERSON-like answer, while the question *where was Picasso born?* is expected to have a PLACE-like answer. In this way, once the entity type of the expected answer is known, we can concentrate only on the segments of

text that concern such entity type. In this phase, classification systems can be used, also in terms of entity categorization, which is often organized hierarchically following a taxonomy that can be constructed semi-automatically or manually. In this phase, the use of lexical resources such as WordNet [63] is often introduced to facilitate the identification of the entities and identify the related words. In the query formulation phase, the task is to create a list of keywords to be used in a search process using IR techniques. This phase is influenced by the type of sources available. In the case of Web search, web search engines can sometimes be used directly. The passage extraction phase starts from the documents retrieved by the execution of the query built in the previous phase and further narrows down the search space by analyzing the most promising documents and extracting the text passages that potentially contain the answer. Typically this phase follows a supervised approach where the features are relevant features of the individual steps through a multi-level match with the input question. Finally, the answer extraction phase typically involves an extractive process which consists in extracting a specific portion of the text that has been selected in the previous steps. Two common answer extraction techniques are answer-type pattern extraction and N-gram tiling.

4.2 Knowledge-Based Question Answering

Although a vast amount of data is coded in terms of text on the web, information also exists in more structured forms. The term Knowledge-Based Question Answering refers to the problem of answering questions in natural language by querying a structured knowledge base. As is the case with the text-based QA paradigm, the origins of this approach date back to the 1960s with the Baseball system, which provided answers by using a structured database of baseball games and statistics. Knowledge-base QA systems typically map questions in the form of predicates computation or query languages such as SQL or SPARQL. Often the knowledge base consists of a relational database or a less structured database such as triple-store databases for RDF. Among the most frequently used methods for question classification, we find the approaches based on rules that typically lead to good accuracy, but require a consistent contribution of grammatical rules provided by experts. Another family of widespread approaches is based on supervised methods, where the question classification process is more automated, but large training sets are often required to function properly.

Chapter 5

ASP-based Question Answering for Cultural Heritage

In this chapter, we will illustrate and motivate the design choices of a Question Answering system, which is consistent with the project objectives and can be implemented in a software system in the project times and constraints. The idea is to realize a Knowledge-Based Question Answering System following a rules-based approach to obtain high precision in the management of the questions, with the support of valid tools to help the construction of the rules in order to speed up the management process of new questions within the system. The entire QA process is exemplified in Figure 5.1, which shows the interaction among the various modules of the system. In particular, the question answering process is split into the following phases:

1. the input question is transformed into a three-level syntactic representation: **Question NLP Processing**

2. the representation is categorized by a template system that can be implemented by means of logical rules with Answer Set Programming: **Template Matching**

3. the next phase, managed through imperative code (Java), allows passing from a template to an intent, where the intent identifies precisely the intent (or purpose) of the application: **Intent Classification**

4. the intent generates a formal query which is performed to find the answer to the question based on knowledge: **Query Generation**

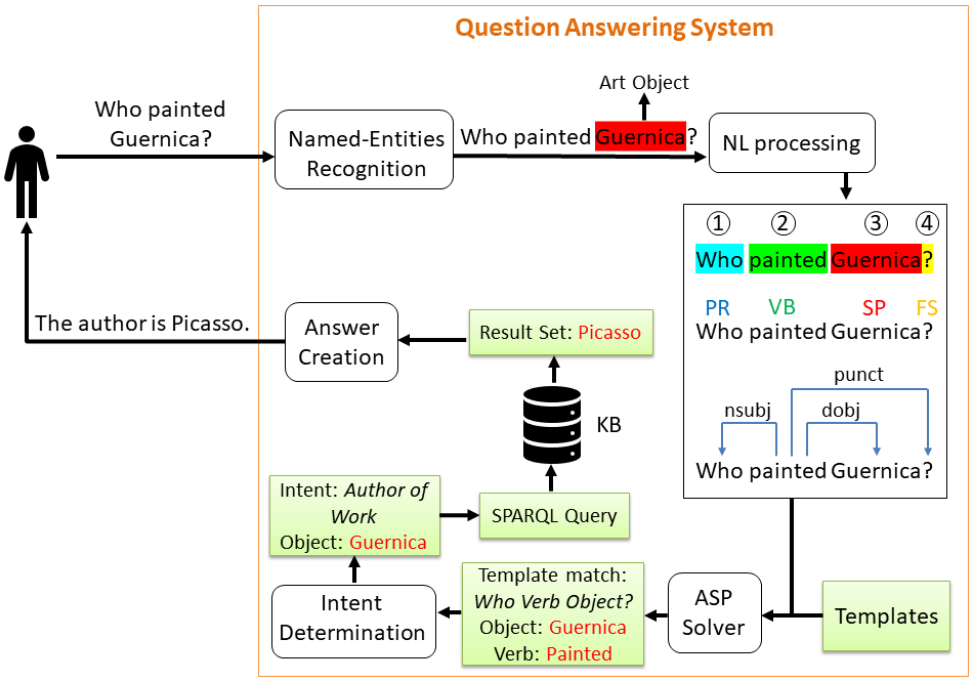

Figure 5.1: Scenario of interaction with the Question Answering System

5. the Query is physically executed on the knowledge-base: **Query Execution**

6. the result produced by the interrogation is transformed into a natural language answer: **Answer Generation**

Splitting the QA process into distinct phases allowed us to implement a system by connecting loosely-coupled modules dedicated to each phase. In this way we also achieved better maintainability, and extensibility. In the following sections, we analyze in detail the 6 phases just listed.

5.1 Question NL Processing

The NL processing phase deals with building a formal and therefore tractable representation of the input question. The question is decomposed and analyzed highlighting the atomic components that compose it, the morphological properties of the components and the relationships that bind them. Fortu-

nately, at this stage, it is possible to use pre-existing computational tools and models in the field of natural language processing, also for the Italian language. This phase is in turn divided into sub-processes:

- Recognition of named entities
- Tokenization
- Part-of-speech tagging
- Dependency parsing

5.1.1 Named Entities Recognition

The named entities of a text are portions of text that identify the names of people, organizations, places or other elements that are referenced by a proper name. For example, in the phrase *Michelangelo has painted the Last Judgment* we can recognize two entities, that are Michelangelo that could belong to a *Person* category and the Last Judgment that could belong to an *Artwork* category. The recognition of named entities is an NLP task that deals with the identification and categorization of the entities that appear in a text. In a QA system, a NER phase makes it possible to identify the topics of the input question, classify them and eventually simplify the subsequent phases of NLP. To date, there are several existing implementations of NER. Mostly, the NER algorithms are based on grammars of natural languages or on statistical approaches such as Machine Learning. Grammar-based approaches typically require a consistent supply of grammar rules provided by grammar experts, offering high accuracy, but low recall and consistent rule-making work that can take months. Instead, statistical approaches require large amounts of training data that are manually annotated. More recently, semi-supervised approaches have been developed to allow faster creation of the training set. When the entities of the text have been recognized, they can be replaced with placeholders that are easier to manage in the subsequent stages of processing of natural language. For example, it is possible to replace long and decomposed names with atomic names (that is, composed of a single word) that are more easily handled during tokenization, Parts-of-speech tagging, and Dependency Parsing. In our implementation we use CRF++ [52] that implements a supervised model based on conditional random fields that are probabilistic models for segmenting and labeling sequence data [53].

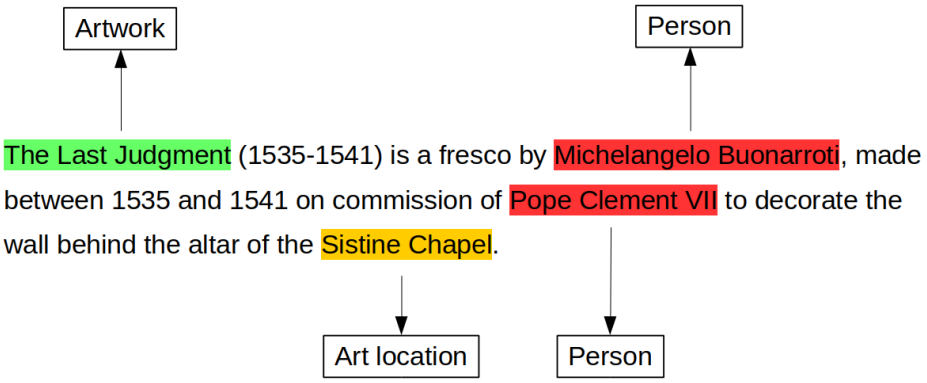

Figure 5.2: Example of Named-Entities Recognition

To train the CRF model, we generated a training set built from some question patterns specified by using a metalanguage and expansible into a set of training questions. An example of question patterns is the following:

```
who {painted,created} [the work,the artwork] <W>.
```

The pattern is expanded by using, for each expansion, exactly one word from curly brackets set, at most one word from square brackets set, and one available values from a predefined entities set identified with the id in angle brackets. So, if $W = \{Guenica, theRosettaStone, theMonalisa\}$, the resulting patterns expansion are all the followings:

```
who painted Guenica.
who created Guenica.
who painted the work Guenica.
who created the work Guenica.
who painted the artwork Guenica.
who created the artwork Guenica.
who painted the Rosetta Stone.
...
```

The resulting questions can then be down-sampled. We randomly sample expanded patterns by using a fixed maximum size for each question pattern.

Figure 5.3: Example of tokenization

5.1.2 Tokenization

Tokenization consists of splitting text into words (called tokens). A token is an indivisible unit of text. Tokens are separated by spaces or punctuation marks. In Italian, as in other western languages, the tokenization phase turns out to be rather simple, as these languages place quite clear word demarcations. In fact, the approaches used for natural language tokenization are based on simple regular expressions. Tokenization is the first phase of lexical analysis and creates the input for the next Part-of-Speech Tagging phase. Figure 5.3 shows an example of tokenization.

5.1.3 Parts-Of-Speech Tagging

The part-of-speech tagging phase consists in assigning to each word the corresponding part of the speech. Common examples of parts-of-speech are adjectives, nouns, pronouns, verbs, or articles. The part-of-speech assignment is typically implemented with supervised statistical methods. There are, for several languages, large manually annotated corpora that can be used as training sets to train a statistical system. Among the best performing approaches are those based on Maximum Entropy [6]. The set of possible parts of the speech (called tag-set) is not fixed, and above all, it can present substantial differences depending on the language taken into consideration. For Italian, a reference tag-set is the Tanl tag-set: [1]. This tag-set distinguishes between coarse-grained tags and fine-grained tags and allows specifying tags that include morphological information such as gender and number. Figure 5.4 shows an example of POS tagging. In the example, the symbols above the words indicate the tags corresponding to the words below: *NP* indicates a proper name, *VB* indicates a verb, and *F* indicates a punctuation mark.

For tokenization and POS-tagging we used the Apache OpenNLP library[2]

[1] `http://medialab.di.unipi.it/wiki/Tanl_POS_Tagset`

[2] `https://opennlp.apache.org`

Figure 5.4: Example of POS-tagging

with pretrained models[3].

5.1.4 Dependency Parsing

Dependency Parsing is the identification of lexical dependencies of the words of a text according to a grammar of dependencies. The dependency grammar (DG) is a class of syntactic theories that are all based on the dependency relationship (as opposed to the circumscription relation). Dependency is the notion that linguistic units, e.g. words, are connected to one another by directed connections (or dependencies). A dependency is determined by the relationship between a word (a head) and its dependencies. The methods for extracting grammar dependencies are typically supervised and use a reference tag-set and a standard input representation format known as the CoNLL standard, developed and updated within the CoNLL scientific conference (Conference on Computational Natural Language Learning). An updated version of the CoNLL format is the CoNLL-U[4] version. In CoNLL-U, lexical information is encoded as plain text UTF-8. The information is organized by line and there are 3 types of line:

- Word lines: contain annotations related to a word (token) in 10 fields separated by single tabs.
- Blank lines: used to separate sentences.
- Comment lines: start with a hash symbol (#) and are not interpreted by the interpreters of the CoNLL format, but are used to add comments.

A sentence consists of one or more word lines. A word line contains the following attributes:

[3] `https://github.com/aciapetti/opennlp-italian-models`
[4] `http://universaldependencies.org/format.html`

- ID: Index of the word, is an integer starting from 1 for each new sentence; it can cover a multi-word token
- FORM: Shape of the word (that is the word itself), possibly a punctuation symbol
- LEMMA: Lemma or stem or canonical form of the word. E.g. run is the lemma of running, apple is the lemma of apples
- UPOSTAG: universal part-of-speech tag
- XPOSTAG: language-specific part-of-speech tag
- FEATS: (Features) List of morphological characteristics associated with the word (eg gender, number, time)
- HEAD: Head of the current word. It allows representing a direct grammatical dependency; it is the ID of the related word (head)
- DEPREL: tag that identifies the type of grammatical dependency
- DEPS: used for the analysis of the dependencies whose structure is not a tree, but rather a graph and there may be more arcs coming out of a node
- MISC: any other annotation

In our implementation we used MaltParser[5] that is a system for data-driven dependency parsing [66].

5.2 Template Matching

Once the NLP phases are completed we perform one of the core phases of the system, that is the template matching phase. Template matching is in charge of classifying question from the syntactical point of view and extract the question terms that are needed to instantiate the query for retrieving the answer. Basically, a template represents a category of syntactically homogeneous questions. In our system, templates are encoded in terms of ASP rules.

[5] `http://www.maltparser.org/`

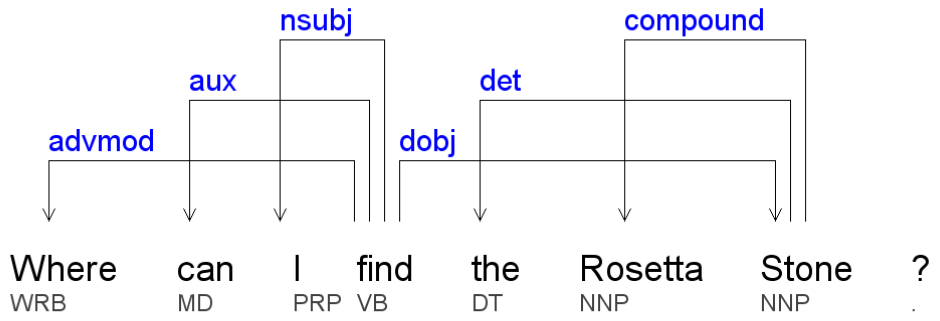

Figure 5.5: Example of Dependency Parsing

By using ASP we can work in a declarative fashion and avoid implementing the template matching procedure from scratch.

To this end, the output of the NLP phase is modeled by using ASP facts that constitute the input of the template matching module. In particular, words are indexed by their position in the sentence and they are associated with their morphological feature by using facts of the following forms:

```
word(pst,wrd).      pos(pst,pos_tag).
        gr(pst1,pst2,rel_tag).
```

the first kind of fact associates position of words (`pst`) in a sentence to the word itself (`wrd`); the second associates words (`pst`) with their POS tags (`pos_tag`), and the latter models grammatical relations (a.k.a. typed dependencies) specifying the type of grammatical relation (`rel_tag`) holding among pair of words (`pst1,pst2`). The possible tags and relations terms are constants representing the labels produced by the NLP tools mentioned in the previous subsection.

Consider, for example, the question *who painted Guernica?*, the output of the NLP phase would result in the following facts (cfr., Figure 5.1 for a graphical representation of this NLP output).

```
word(1, "who"), word(2, "painted").
word(3, "Guernica").  word(4, "?").
```

```
pos(1, pr), pos(2, vb).  pos(3, np).  pos(4, f).
gr(2, 1, nsubj), gr(2, 3, dobj).  gr(2, 4, punct).
```

We denote with F_Q the set of facts produced by the application of mentioned NLP phases and transformations to an input question Q.

In the template matching phase, questions are matched against question templates. Templates identify categories of questions that are uniform from the syntactic point of view and we express them in the form of ASP rules.

Definition 10. *A template rule R is a rule having in the head an ASP atom of the form*

$$\text{template}(ID, terms\_K(V_1, \ldots, V_K), W)$$

where template is a constant predicate name (that is the same for all templates), ID is an ASP constant that identifies the template, K is an integer that we call template arity, $V_1, \ldots, V_K$ are ASP variables and W is an integer that defines the template rule weight.

Basically, each template rule models a condition under which we identified a possible syntactic question pattern for a template. The function *terms_K* conveniently groups the terms that are extracted from the match. Finally the weight is numerical value that expresses the importance of a pattern. By using weights one can freely express preferences among patterns; for instance in our implementation we set this number to the size of the body to favor more specific templates rules over more generic ones. An example of template rule that matches questions of the form *who action object?* is the following:

```
template(who_action_object, terms_2(V, O), 8) :-
word (1, "who"),
word(2, V), word(3, O), word(4, "?"),
pos(1, pr), pos(2, vb), pos(3, np), pos(4, f),
gr(2, 1, nsubj), gr(2, 3, dobj), gr(2, 4, punct).
```

In the example, *who_action_object* is a constant that identifies the template, while $terms(V, W)$ is a function symbol that allows extracting the terms of the input question, respectively the verb V and the object O. The weight of the template rule is 8, which corresponds to the body size as described above.

Definition 11. *A template T is a set of template rules having the same ID and arity.*

Basically, a template collects a number of possible syntactic patterns (one per template rule), roughly corresponding to different ways of formulating a kind of question.

Definition 12. *A template matching program P is an ASP program that contains at least one template, and the following rule, defining the best matches (i.e., the ones with highest weight):*

```
bestMatch(T,R) :- template(T,R,M), #max{W: template(_,_,W)} =
                                                    M.
```

Definition 13. *Given a template matching program P and the set of facts F_Q coming from the NLP phase w.r.t. a question Q, we say that (T, R) is a best match for Q iff $bestMatch(T, R) \in A$ where A is the answer set of $F_Q \cup P$. In such case, T identifies a best matching template and R defines the terms extracted from the match.*

Note that, it was by design that one can retrieve more than one best match, to give more freedom to the design of the interaction with the user. Pragmatically, in the first prototypical implementation, we simply select the first best-match assuming that all best-matches represent the question equally good.

Question templates are intended to be defined by the application designer, which is a reasonable choice in applications like the one we considered, where the number of templates to produce is limited. Nonetheless, to assist the definition of templates we developed a graphical user interface. Such interface helps the user at building template rules by working and generalizing examples, and does not require any previous knowledge of ASP or specific knowledge of NLP tools. The template editing interface is not described in this paper for space reasons.

In our prototype, we used DLV [57] as the ASP solver that computes the answer sets (thus the best matches) of the template matching phase, and the DLV Wrapper library [71] to programmatically manage DLV invocations from Java code.

The intent determination process is based on the lexical expansion of question terms extracted in the template matching phase.

5.3 Intent Determination

The identification of a question by templates is typically not sufficient to identify its intent or purpose. For example, *who painted Guernica?* and *who killed Caesar?* have a very similar syntactic structure and may fall into the same template, but they have two different purposes. The intent determination process is based on the lexical expansion of question terms extracted in the template matching phase and has the role of identifying what the question asks (i.e., its intent), starting from the result of the template matching phase. In other words, it disambiguates the intent of questions that fall into the same syntactic category (and that therefore have a match on the same template). In the previous example, painting is *hyponym* (i.e., a specific instance) of creating and this fact allows us to understand that the intent is to determine the creator of a work, while killing does not have such relationships and we should, therefore, instantiate a different intent. In the same way, *who painted Guernica?*, *who made Guernica?* or *who created Guernica?* are all questions that can be correctly mapped with a single template and can be correctly recognized by the same intent thanks to the fact that all three verbs are hyponyms or synonyms of the verb *create*. Words semantic relations can be obtained by using dedicated dictionaries, like wordnet [63] or BabelNet [65]. In our system we used BabelNet and we implemented the intent determination module in Java and used the BabelNet API library for accessing word relations. In particular, intent determination is implemented as a series of Java conditional checks (possibly nested) on word relations. Such conditional checks are expressed as a question term Q, a word relation R and a target word T. The BabelNet service receives such triple and returns true/false depending on whether Q is in relation R w.r.t. T, R is either synonymy, hyponymy or hyperonymy. Algorithm 11 presents a pseudo-code of the intent determination process for the *who_action_object* template with one verb and one object as terms.

The implementation of intent determination is done by the designer as template definition. Our system implements a set of intents that were identified during the analysis by a partner of the project.

Note that intent determination could also be easily encoded by means of ASP rules, which would have allowed having a single ASP program for handling both template matching and intent determination. However, the access to external dictionaries was not efficient in practice, and we decided to go for a straight implementation with imperative code.

Algorithm 11 Determine intent for who_action_object template

Input: matched terms: verb, object
Output: intent of question

```
1: if inDictionary(verb, "synonym", "created") then
2:     return AUTHOR_OF_WORK
3: end if
4: if inDictionary(verb, "synonym", "found") then
5:     return FOUNDER_OF_WORK
6: end if
7: if inDictionary(verb, "synonym", "married") then
8:     return SPOUSE_OF_PERSON
9: end if
10: if inDictionary(verb, "hyponym", "created") then
11:     return AUTHOR_OF_WORK
12: end if
13: ...
14: return FAIL
```

5.4 Query Execution

The intents identified in the previous phase are mapped one to one with template queries, called prepared statements in programming jargon. In the Query Execution phase, the query template corresponding to the identified event is filled with the slots with terms extracted from the template matching phase and executed over the knowledge base. The CIDOC-crm specification is, by definition, an RDF knowledge base [19], thus we implemented the queries corresponding to intents in the SPARQL language [46]. The problem of programmatically running a query on an RDF knowledge base is a problem for which there are already several solutions. Among the many, we mention Apache Jena for Java and Rasqal for C / C++. In our prototype, we store our data and run our queries using Apache Jena, as programmatic query API, and Virtuoso Open-Source Edition as knowledge base service.

5.5 Answer Generation

Finally, the latest phase of a typical user interaction with the QA system is the so-called Answer Generation. In this phase, the results produced by the execution of the query on the knowledge base are transformed into a natural language answer that is finally returned to the user. To implement this phase we have designed answer templates that are in some ways similar to the one seen for generating the test set for the NER phase. In this case, the idea is to have natural language patterns with parameterized slots that are filled according to the question intent and the terms extracted from the database. These answer templates can be expressed in a compact way through a metalanguage that allows expressing sentences with variations according to the subjects of question or answer. The example below presents a possible answer template for questions concerning the materials of a work.

```
The material{s:s}[R] of <Q> {s:is,p:are}[R] <R>.
```

The curly brackets denote a sequence of variants and the square brackets denote the term (or terms) with respect to which the block preceding it refers: R stands for answer (or response) and Q stands for question. A variant consists of a property and a value separated by a colon symbol. The block delimited by the braces is replaced by the value of the variant appropriate to the term enclosed between square brackets that follows the block. The determination of the appropriate variant can be implemented within the system using, for example, a dictionary of terms. In the example, the s variant is for singular forms and the p variant is for plural forms. The variants may possibly be extended into more complex types (possibly organized in hierarchies) and take into consideration other characteristics of the terms of answer and question extracted from appropriate dictionaries of terms or explicitly represented in the knowledge base. Finally, the $\langle Q \rangle$ tag is replaced by the question terms and the $\langle R \rangle$ tag from the answer terms. So, suppose we want to apply the answer template from the previous example to the fact that the Rosetta Stone is made of granodiorite we would get the answer: *The material of the Rosetta Stone is granodiorite.*

5.6 System Performance on Real-world Data

In this section, we report on the results of an experimental analysis conducted to assess the performance of the system, and in particular, we have checked

whether it scales well w.r.t the number of template rules present in a use case developed in the PIUCULTURA project.

Table 5.1: Template matching time (average times on a sample of 167 questions)

Number of templates	20	30	40	50	60
Average matching time (milliseconds)	30	30	31	33	34

We devised 60 template rules, which are able to handle basic question forms and types for the cultural heritage domain distributed in 20 different intents (e.g., authors, materials, dimensions, techniques of artworks, dates/locations of birth and death of artists, and so on). The queries have been executed on a dump of the *British Museum*[6] Knowledge-Base that consists of more than 200 million RDF triples. The hardware used is an Intel i7-7700HQ CPU with 16GB of ram running a Linux operating system. The knowledge base was handled by Virtuoso ver. 7.2.4, connected to our system with JENA ver. 3.6.0. The ASP system we have used is DLV build BEN/Dec 17 2012.

The average execution times of the template matching phase measured on a sample of 167 questions and on an increasing number of template rules are reported in table 5.1. Execution times are in the order of some milliseconds and seems to scale well w.r.t the number of templates. The DLV system performs well on the template programs we have implemented, which by design fall in the stratified [12] syntactic subclass of ASP, which is comptationally cheap and efficiently handled by the ASP system we employ [57, 58]. For what concerns the other phases of the QA system, we report that, on the same sample of 167 questions, the NL phase average execution time is of 30 milliseconds and is at most 50 milliseconds, the intent determination phase average execution time is of 50 milliseconds and is at most 580 milliseconds and the average query execution time is of 8 milliseconds and is at most 32 milliseconds. Overall the system presents good execution times, which are acceptable for a real-time QA system.

[6] `http://www.britishmuseum.org/`

5.7 Related Work

This work is mostly related to other approaches and forms of QA. Most QA systems in the literature are concerned to retrieving answers from collections of documents, or on the Web, also thanks to the work developed in the context of the Text Retrieval Conference (TREC) that popularized and promoted this form of QA [78]. Systems that fall in this category are mainly based on information retrieval techniques. The most prominent differences to our approach are that we are collecting data from a structured knowledge base, instead of text collections.

For what concerns closed-domain QA, early examples are Baseball [45] and Lunar [80], they were essentially natural language interfaces to relational data-bases. Lunar allowed asking geologist questions about rocks, while Baseball answered questions about data collected from a baseball season. AquaLog [61] is an ontology-portable Question Answering system that tries to map input questions into linguistic triples and then into ontology triples by mainly using similarity services. In our approach, there is no triple representation of questions and we implemented an intent layer that separates the NL and the ontology world. The intent layer allows implementing intents in actions/queries that are not SPARQL queries. Our approach is less general to be ported to other ontologies, but we provide more control to the developer to create precise NL-to-ontology mappings.

WEBCOOP [14] is a QA system for the tourism domain and implements a cooperative QA on unstructured data. WEBCOOP works on text collections instead of a structured knowledge base. In [77] input questions are transformed into SPARQL templates that mirror the internal structure of the question and are then filled with question terms. As for AcquaLog, the system is domain-independent and in contrast to AcquaLog, they capture questions that are not fully represented by triple clauses, but also those that need filtering and aggregations functions to be handled. Again, our approach is less general and less automatic, but also more controllable and less dependent on the data representation formalism that can vary independently from the question templates.

There are some systems that approach questions by transforming them into logic forms so as to be able to perform reasoning tasks [64, 44, 9], both in open and closed domains. This is particularly useful for difficult questions that need advanced reasoning capabilities to be answered. Our approach also uses logic, but for the different task of expressing question patterns matching

input questions.

5.8 Discussion

In this last part of the thesis, we tackled the problem of Question Answering in Closed Domains, with a specific focus on the Cultural Heritage domain and the Italian language. We proposed a solution that is based on Answer Set Programming and is materialized as a software prototype. The presented solution gradually transforms input questions into SPARQL queries that are executed on an ontological Knowledge-Base. It integrates state-of-the-art NLP tools and models in a modular architecture. The core of the question answering process is organized in a two-steps classification process:

1. a rule-based syntactic matching implemented in terms of Answer Set Programming rules and called *template matching*

2. a semantic classification based on lexical expansions of question terms that we called *intent classification*

The system is designed for allowing the fast integration of new question forms (by adding templates) and new question intents (by adding intents and templates) and behaves well in closed domains that are characterized by a limited number of intents and question forms.

As for future works, it would be interesting to compare the presented approach against other forms of question answering, as the ones based on machine learning.

Chapter 6

Conclusion

In this thesis we presented our work in two parts, reporting on two research efforts belonging to the field of Answer Set Programming. In the first part of the thesis, we approached the problem of improving ASP systems where state-of-the-art systems suffer from the grounding bottleneck, providing several contributions in this context. In the second part of the thesis we described a new real-world application of ASP to a problem of question answering in natural language. In the following we draw the conclusion of our studies; two separate paragraphs summarize the contributions and discuss future works for the two parts of the thesis.

Lazy Constraints and Partial Compilation. We investigated some issues that are related to the evaluation of ASP programs in cases where state-of-the-art systems are not effective. At first, the focus is on problematic ASP constraints for which the grounding phase represents a bottleneck of the computation. Such conditions have been observed in synthetic and real-world problems and prevent the standard ASP solving approaches from being effective. We study how and when the replacement of constraints with ad-hoc solvers extensions is beneficial with a distinction between three flavors of replacements: lazy instantiators, eager propagators and post propagators. Indeed, an experimental analysis shows that such approaches are effective when constraints are hard to ground, with lazy instantiators being better on easier constraints (in terms of satisfiability) and eager/post on harder constraints, while the standard "ground+solve" approach is generally better for constraints that are anyway easy to ground. We concluded the first work with a demonstrative portfolio approach that allows a smart combination

of all the approaches when they are all available. A disadvantage of custom propagators was that they had to be written manually. In an attempt to generate them automatically we decided to adopt a compilation-based approach, where an ASP sub-program can be compiled into a specialized procedure that is in principle (and in practice, as we have empirically shown) faster than a general-purpose procedure thanks to its specificity. Our partial compilation is embeddable in existing ASP solvers thanks to programmatic APIs and we implemented it within the WASP solver. Experiments on benchmarks from the literature witness the efficacy of our technique.

The partial compilation technique proposed in this thesis is (to the best of our knowledge) the first approach of this kind to the evaluation of ASP programs.

For what concerns future works, automatic generation of eager and post propagators would be an interesting advancement. Another possible extension of ASP compilation would be the development of an ASP grounder that uses compilation-based techniques. Indeed the problem of instantiating ASP programs can be seen as an extension of Datalog programs evaluation.

ASP for Question Answering on Cultural-Heritage. We presented an application of ASP to Closed Domain Question Answering with a focus on the Italian language and the Cultural Heritage domain. We first collect and define some necessary building blocks and a preliminary architecture that is able to tackle the problem. We selected a standard ontological reference model for representing the cultural heritage data and collected some models and tools that are able to perform some standard Natural Language Processing tasks. We defined a two-steps question classification process, the first one classifying questions by means ASP rules and the second one being a more semantical classification based on words semantic relations. The syntactic classification has been named *template matching* where a template is a set of ASP rules.

The result of the classification process is given by a question intent and a set of matched terms that are the subjects of the questions and are extracted in the template matching process. Finally, by using the intent and the matched terms the systems generates a SPARQL query that is executed on an ontological knowledge base to retrieve the answer to the user question.

The core part of the system is the template matching process that is implemented in ASP. By using ASP we could rely on efficient existing imple-

mentations without the need to implement a matching system from scratch. Moreover, new question forms are easily integrated in the system by means of new templates, while new question types can be handled by introducing new question intents (and related templates and queries).

The presented QA system has been integrated in the system that is under development for the PIUCULTURA project, which is a project funded by the Italian Ministry of Economic Development and whose aim is to promote and improve the fruition of Cultural Heritage. The commercial partners of the project are the IT companies Neatec[1] and Softlab[2], which are planning to use the software for commercial purposes in the next years, when the project is completed.

As for future works, it would be interesting to compare the presented approach against other forms of question answering, like the ones based on machine learning.

[1] `http://www.neatec.it`
[2] `http://www.soft.it`

Bibliography

[1] Dimitris Achlioptas. Random satisfiability. In *Handbook of Satisfiability*, volume 185 of *FAIA*, pages 245–270. IOS Press, 2009.

[2] Ali Mohamed Nabil Allam and Mohamed Hassan Haggag. The question answering systems: A survey. *International Journal of Research and Reviews in Information Sciences (IJRRIS)*, 2(3), 2012.

[3] Mario Alviano and Carmine Dodaro. Completion of disjunctive logic programs. In *IJCAI*, pages 886–892. IJCAI/AAAI Press, 2016.

[4] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native ASP solver based on constraint learning. In *LPNMR*, volume 8148 of *LNCS*, pages 54–66. Springer, 2013.

[5] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In *LPNMR*, volume 9345 of *LNCS*, pages 40–54. Springer, 2015.

[6] Markus Aschinger, Conrad Drescher, Gerhard Friedrich, Georg Gottlob, Peter Jeavons, Anna Ryabokon, and Evgenij Thorstensen. Optimization methods for the partner units problem. In *CPAIOR*, pages 4–19, 2011.

[7] Rehan Abdul Aziz, Geoffrey Chu, and Peter J. Stuckey. Stable model semantics for founded bounds. *TPLP*, 13(4-5):517–532, 2013.

[8] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[9] Marcello Balduccini, Chitta Baral, and Yuliya Lierler. Knowledge representation and question answering. *Foundations of Artificial Intelligence*, 3:779–819, 2008.

[10] Marcello Balduccini and Yuliya Lierler. Integration schemas for constraint answer set programming: a case study. *TPLP*, 13(4-5-Online-Supplement), 2013.

[11] Marcello Balduccini and Yuliya Lierler. Constraint answer set solver EZCSP and why integration schemas matter. *CoRR*, abs/1702.04047, 2017.

[12] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2010.

[13] Sabrina Baselice, Piero A. Bonatti, and Michael Gelfond. A preliminary report on integrating of answer set and constraint solving. In *Answer Set Programming*, volume 142 of *CEUR Workshop Proceedings*, 2005.

[14] Farah Benamara. Cooperative question answering in restricted domains: the webcoop experiment. In *Proceedings of the Conference on Question Answering in Restricted Domains*, 2004.

[15] Bart Bogaerts and Antonius Weinzierl. Exploiting justifications for lazy grounding of answer set programs. In *IJCAI*, pages 1737–1745. ijcai.org, 2018.

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

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

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

[19] World Wide Web Consortium et al. Rdf 1.1 concepts and abstract syntax. 2014.

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

[21] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. GASP: answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322, 2009.

[22] Minh Dao-Tran, Thomas Eiter, Michael Fink, Gerald Weidinger, and Antonius Weinzierl. Omiga : An open minded grounding on-the-fly answer set solver. In *JELIA*, volume 7519 of *LNCS*, pages 480–483. Springer, 2012.

[23] Broes de Cat, Marc Denecker, Maurice Bruynooghe, and Peter J. Stuckey. Lazy model expansion: Interleaving grounding with search. *J. Artif. Intell. Res. (JAIR)*, 52:235–286, 2015.

[24] Carmine Dodaro, Philip Gasteiger, Nicola Leone, Benjamin Musitsch, Francesco Ricca, and Konstantin Schekotihin. Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). *TPLP*, 16(5-6):653–669, 2016.

[25] Carmine Dodaro, Francesco Ricca, and Peter Schüller. External propagators in WASP: preliminary report. In *RCRA*, volume 1745 of *CEUR Workshop Proceedings*, pages 1–9. CEUR-WS.org, 2016.

[26] Carmine Dodaro, Francesco Ricca, and Peter Schüller. External propagators in wasp: Preliminary report. In *RCRA@ AI* IA*, pages 1–9, 2016.

[27] Martin Doerr. The cidoc conceptual reference module: an ontological approach to semantic interoperability of metadata. *AI magazine*, 24(3):75, 2003.

[28] T. Eiter, M. Fink, G. Ianni, T. Krennwallner, C. Redl, and P. Schüller. A model building framework for answer set programming with external computations. *Theory and Practice of Logic Programming*, 16(4), 2016.

[29] 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.

[30] Thomas Eiter, Christoph Redl, and Peter Schüller. Problem solving using the HEX family. In *Computational Models of Rationality, Essays*

dedicated to Gabriele Kern-Isberner on the occasion of her 60th birthday, pages 150–174. College Publications, 2016.

[31] Esra Erdem and Umut Öztok. Generating explanations for biomedical queries. *TPLP*, 15(1):35–78, 2015.

[32] Esra Erdem, Volkan Patoglu, and Peter Schüller. A systematic analysis of levels of integration between high-level task planning and low-level feasibility checks. *AI Commun.*, 29(2):319–349, 2016.

[33] Wolfgang Faber, Nicola Leone, and Simona Perri. The intelligent grounder of DLV. In *Correct Reasoning*, volume 7265 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2012.

[34] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In *CP*, volume 5732 of *LNCS*, pages 352–366. Springer, 2009.

[35] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *ICLP TCs*, volume 52 of *OASICS*, pages 2:1–2:15, 2016.

[36] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *OASIcs-OpenAccess Series in Informatics*, volume 52. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[37] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Javier Romero, and Torsten Schaub. Progress in clasp series 3. In *LPNMR*, volume 9345 of *Lecture Notes in Computer Science*, pages 368–383. Springer, 2015.

[38] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in *gringo* series 3. In *LPNMR*, volume 6645 of *LNCS*, pages 345–351. Springer, 2011.

[39] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *IJCAI*, volume 7, pages 386–392, 2007.

[40] Martin Gebser, Benjamin Kaufmann, Javier Romero, Ramón Otero, Torsten Schaub, and Philipp Wanko. Domain-specific heuristics in answer set programming. In *AAAI*. AAAI Press, 2013.

[41] Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca, and Torsten Schaub. Evaluation techniques and systems for answer set programming: a survey. In *IJCAI*, pages 5450–5456. ijcai.org, 2018.

[42] Martin Gebser, Anna Ryabokon, and Gottfried Schenner. Combining heuristics for configuration problems using answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 384–397. Springer, 2015.

[43] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[44] Cordell Green. Theorem proving by resolution as a basis for question-answering systems. *Machine intelligence*, 4:183–205, 1969.

[45] Bert F Green Jr, Alice K Wolf, Carol Chomsky, and Kenneth Laughery. Baseball: an automatic question-answerer. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 219–224. ACM, 1961.

[46] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. Sparql 1.1 query language. *W3C recommendation*, 21(10), 2013.

[47] Lynette Hirschman and Robert Gaizauskas. Natural language question answering: the view from here. *natural language engineering*, 7(4):275–300, 2001.

[48] Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran. Modularity Aspects of Disjunctive Stable Models. *Journal Of Artificial Intelligence Research*, 35:813–857, 2009.

[49] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI Magazine*, 37(3):25–32, 2016.

[50] Tero Kojo, Tomi Männistö, and Timo Soininen. Towards intelligent support for managing evolution of configurable software product families. In *SCM*, volume 2649 of *LNCS*, pages 86–101. Springer, 2003.

[51] Laura Koponen, Emilia Oikarinen, Tomi Janhunen, and Laura Säilä. Optimizing phylogenetic supertrees using answer set programming. *TPLP*, 15(4-5):604–619, 2015.

[52] Taku Kudo. Crf++. *http://crfpp. sourceforge. net/*, 2013.

[53] John Lafferty, Andrew McCallum, and Fernando CN Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[54] Claire Lefevre, Christopher Béatrix, Igor Stéphan, and Laurent Garcia. Asperix, a first-order forward chaining approach for answer set computing. *Theory and Practice of Logic Programming*, 17(3):266–310, 2017.

[55] Claire Lefèvre and Pascal Nicolas. The first version of a new ASP solver: Asperix. In *LPNMR*, volume 5753 of *LNCS*, pages 522–527. Springer, 2009.

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

[57] 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.

[58] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. Openrulebench: an analysis of the performance of rule engines. In *WWW*, pages 601–610. ACM, 2009.

[59] Vladimir Lifschitz. What is answer set programming?. In *AAAI*, volume 8, pages 1594–1597, 2008.

[60] Lengning Liu, Enrico Pontelli, Tran Cao Son, and Miroslaw Truszczynski. Logic programs with abstract constraint atoms: The role of computations. *Artif. Intell.*, 174(3-4):295–315, 2010.

[61] Vanessa Lopez, Michele Pasin, and Enrico Motta. Aqualog: An ontology-portable question answering system for the semantic web. In *European Semantic Web Conference*, pages 546–562. Springer, 2005.

[62] Marco Manna, Francesco Ricca, and Giorgio Terracina. Taming primary key violations to query large inconsistent data via ASP. *TPLP*, 15(4-5):696–710, 2015.

[63] George Miller. *WordNet: An electronic lexical database*. MIT press, 1998.

[64] Dan Moldovan, Christine Clark, Sanda Harabagiu, and Steve Maiorano. Cogex: A logic prover for question answering. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 87–93. Association for Computational Linguistics, 2003.

[65] Roberto Navigli and Simone Paolo Ponzetto. Babelnet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artificial Intelligence*, 193:217–250, 2012.

[66] Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülşen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135, 2007.

[67] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A prolog decision support system for the space shuttle. In *Answer Set Programming*, 2001.

[68] Max Ostrowski and Torsten Schaub. ASP modulo CSP: the clingcon system. *TPLP*, 12(4-5):485–503, 2012.

[69] J.R. Quinlan. *C4.5: Programs for Empirical Learning*. Morgan Kaufmann, 1993.

[70] Christoph Redl. The dlvhex system for knowledge representation: recent advances (system description). *TPLP*, 16(5-6):866–883, 2016.

[71] Francesco Ricca. The dlv java wrapper. In *APPIA-GULP-PRODE*, pages 263–274. Citeseer, 2003.

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

[73] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[74] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.

[75] Benjamin Susman and Yuliya Lierler. SMT-based constraint answer set solver EZSMT (system description). In *ICLP TCs*, volume 52 of *OASICS*, pages 1:1–1:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[76] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*, volume 14 of *Principles of computer science series*. Computer Science Press, 1988.

[77] Christina Unger, Lorenz Bühmann, Jens Lehmann, Axel-Cyrille Ngonga Ngomo, Daniel Gerber, and Philipp Cimiano. Template-based question answering over rdf data. In *Proceedings of the 21st international conference on World Wide Web*, pages 639–648. ACM, 2012.

[78] Ellen M Voorhees and Hoa Trang Dang. Overview of the trec 2003 question answering track. In *TREC*, volume 2003, pages 54–68, 2003.

[79] Antonius Weinzierl. Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In *LPNMR*, volume 10377 of *LNCS*, pages 191–204, 2017.

[80] William A Woods. Semantics and quantification in natural language question answering. In *Advances in computers*, volume 17, pages 1–87. Elsevier, 1978.