FAKULTÄT FÜR !NFORMATIK

MASTER THESIS
Evaluation of Answer Set Programs with Bounded Predicate Arities

carried out at

Faculty of Informatics
Department of Knowledge Based Systems

Vienna University of Technology

under the supervison of

O.Univ.Prof. Dipl.-Ing. Dr. techn. Thomas Eiter

by

MUSHTHOFA Mushthofa

Hollergasse. 49/11, A1150 Wien

September 11, 2009

Evaluation of Answer Set Programs with Bounded Predicate Arities*

Mushthofa Mushthofa

September 11, 2009

Abstract

We consider the class of logic programs under the restriction of bounded predicate arities. Previous results showed that the complexity answer set semantics for such class of logic programs is lower than unrestricted programs. In particular, evaluation under answer set semantics is possible within polynomial space. However, current ASP solvers and grounders do not seem to respect this complexity bound, and may produce exponential size ground programs, even for programs with bounded predicate arities. We present three methods for evaluation of logic programs with bounded predicate arities which stays within polynomial space. We developed an evaluation framework built on top of current ASP solvers based on the methods, and also provided a prototype implementation of the framework. An experiment was conducted to measure the feasibility and performance of the methods, and to compare it with current ASP solvers, DLV and claspD. The test results showed that the proposed methods and framework are able to evaluate many test instances more efficiently than DLV and claspD. Evaluations by the prototype system stay within polynomial space, and hence, avoid the bottleneck associated with exponential size grounding.

Acknowledgement

Contents

List of Algorithms

List of Figures

List of Tables

1

Introduction

Answer Set Programming (ASP) has been, in the last decade, evolving as one of the powerful and appealing methods for problem solving and declarative knowledge representation. It allows for a purely declarative form of problem representation and solution, and has many desirable properties, such as: termination guarantee and a clear, easily-understood semantics. Moreover, it is powerful and expressive enough to represent problems of complexity Σ_2^P and beyond.

Following the development and study of answer set semantics, research works leading towards building systems capable of performing evaluation under the semantics has also been pursued. These systems are usually called *ASP solvers*. Early ASP solvers were mostly prototypes lacking the efficiency to solve real-world problems. With the advancements in techniques, heuristics and algorithms used in these solvers, their performances have since been significantly improved. Nowadays, systems such as: clasp,[1] DLV[2] and SMODELS[3] are among the best performing and most popularly used ASP solver systems, capable of solving many real-life hard problems.

In this thesis, we consider the class of logic programs where the arities of the predicates in the program are bounded by a fixed constant. It has been shown that the class of logic programs with bounded predicate arities has a complexity class which is lower than that of the full language of logic programs with unrestricted predicate arities, but is still higher than the one of respective ground programs or propositional logic programs [Eiter et al., 2007]. One of the implications of such result concerns the implementations of ASP solvers, in that it should be possible to perform evaluation of a logic program with bounded predicate arities within polynomial space.

However, current ASP solvers do not seem to respect this complexity bound. Evaluation of logic programs using current ASP solvers involve a grounding step, which in general, may produce an exponential size ground program that is logically equivalent to the input program, even when the predicate arities of the program is bounded. This problem manifests as what is known to be the "grounding bottleneck".

To overcome this problem, we propose three evaluation methods to perform evaluation of logic programs with bounded predicate arities which stays within polynomial space. Each of the methods can be seen as a preprocessing step, reducing the task of evaluating a logic program into a series of smaller tasks, each of which can be performed using one of the

---

[1]http://potassco.sourceforge.net/
[2]http://www.dbai.tuwien.ac.at/proj/dlv/
[3]http://www.tcs.hut.fi/Software/smodels/

available ASP solvers without causing it to produce exponential space grounding. We also present a framework architecture defined on top of an ASP solver and a Prolog system, which implements the methods and provides a clearer view on the details of each method.

As a concrete actualization of the proposed methods and framework, we have developed a prototype system which utilizes DLV as the underlying ASP solver and XSB as the Prolog engine to perform variable substitutions-related tasks. We then performed some experiments on the prototype system and compared its performance to current ASP solvers, with the intention to show the feasibility and performance of the proposed methods.

To give a general overview of the work being done in this thesis, this chapter will provide a summary to several foundational concepts of logic programming and answer set programming in particular, as well as a brief discussion on the topic of computational complexity in relation with logic programming. We will discuss the class of logic programs with bounded predicates, its complexity properties and the issues related to evaluation of logic programs with bounded predicate arities using current ASP solvers. Finally, we will provide a brief summary to the main contributions presented in this thesis.

1.1 Logic Programming

In the field of computer science, one of the main aspect discussed is about algorithms and computer programming. As one of the main areas of computer science, computer programming contains many different paradigms and and methods to achieve its results. Unfortunately, when one talks about computer programming, it has become almost synonymous with only one of these different paradigms of computer programming, which is called *imperative/procedural programming*. In *imperative/procedural programming*, one writes a program to *tell* a computer how a computation is performed; what operation should be carried out, what values should be stored, what values to read and so on following a certain *recipe* which is usually called an *algorithm*. Many *imperative/procedural* programming languages have been developed and have been used successfully in many areas and this contributes to the popularity of this programming paradigms. Programming languages such as: Fortran, Pascal, and C are well-known examples of programming languages which use the imperative programming paradigm. Further developments in the field of computer programming languages have enriched this programming paradigm with other ones, of which Object Oriented Programming (OOP) paradigm has been the most popular. This has resulted in new computer languages such as: C++, Objective-C, Java and C#.

However, there are also other paradigms and approaches used in computer programming: notably, the *logic programming* paradigm and *functional programming* paradigm. Even though they are being studied and developed more recently, in comparison with the *imperative programming* languages, both of these approaches have been gaining many progress in the recent years. These might be attributed to the development of other areas of information technology that turn out to have relationships with these two approaches. Functional programming has become quite relevant in areas such as scripting in web development, as well as development of highly parallel/concurrent computations. Meanwhile, interest in logic programming has sparked recently due to the development in the areas of Artificial Intelligence, Knowledge Representation and in applications such as Semantic Web.

Informally, logic programming is a paradigm in computer programming where problems are encoded in the forms of rules and facts in a language based on a logical syntax with a certain semantics attached, where the solution of the problems will be computed as logical consequences of the representation of the problems with respect to the semantics. A fact is represented syntactically as a predicate relation with zero or more constant arguments,

denoting the individuals/objects in the relation. A rule is usually written as

$$H \leftarrow B_1, \ldots, B_n$$

which intuitively means "conclude H whenever B_1, and $\ldots, B_n$ are true". To solve a certain problem, a declarative logical representation of the problem, or *encoding* is then written, and fed to a system capable of performing evaluation using a certain logical semantics. The solutions to the problem are then derived as consequences of the problem representation.

Example 1.1. Consider the following simple problem: given a directed graph and a certain starting node in the graph, find all nodes in the graph reachable from the node using a (non-zero) even number of steps. For example, given the graph in Figure 1.1 and node a as the starting node, the solution should be $\{c, d, f\}$.

Figure 1.1: Input graph for Example 1.1

Using logic programming, we can find the solution of this problem by writing the necessary logical rules and facts to represent the problem. First, we will need the facts to represent the input graph and the chosen starting node. These can done by denoting the edges as a predicate relation *edge* with two arguments,[4] start and end node of the edge, while the starting node can be represented by just one predicate *start_node* with an argument denoting the selected starting node. For instance, the graph in the above figure would be represented as the following facts:

$$edge(a, b). \quad edge(b, c). \quad edge(b, d).$$
$$edge(c, e). \quad edge(d, e).$$
$$edge(e, f). \quad start\_node(a).$$

We then need the rules to represent the property of being "two steps away" and "reachable in an even steps away" from the starting node. The following rules will perform that:

$$two\_steps(X, Y) \leftarrow edge(X, Z), edge(Z, Y)$$
$$reachable(X) \leftarrow start\_node(S), two\_steps(S, X)$$
$$reachable(X) \leftarrow two\_steps(Y, X), reachable(Y)$$

With the right semantic attached to the above program, the logical consequences of the representation encoded by the program will represent the solution of the problem: $\{reachable(c), reachable(d), reachable(f)\}$. In fact, because the rules represent a general description of the problem, they will produce the correct solutions for all possible input graph and starting node.

---

[4]Usually written as $edge/2$ to signify that it has 2 arguments

One of the earliest paper in the field of logic programming is by McCarthy [McCarthy, 1959], which pointed out the need for common-sense reasoning in computer programming. Studies in the field have resulted different kinds of syntactic and semantic flavors of logic programs, as well as several implementation of logic programming engines/solvers. Prolog, with a semantics usually referred to as SLDNF resolution, is arguably one of the most well-known logic programming language.

1.2 Non-monotonic Reasoning

In describing a certain problem, it might be necessary to be able to express about the falseness of a certain fact, instead of its truthness. Consider the following example.

Example 1.2. The following program is a possible logic program representing how an agent makes a decision about crossing a road.

$$cross\_road \leftarrow safe$$
$$safe \leftarrow \neg cars\_passing$$

where the symbol $\neg cars\_passing$ represent the fact there are no cars passing the road at the moment. The symbol "$\neg$" denotes the concept of "classical negation", and is related to the notion of negation in classical logic. The rules intuitively means that the agent should cross the road only when it is known to be safe to do so. It is safe to do so, if it is known that there are no cars passing. In the absence of the information about whether there are cars crossing or not, the agent should not deduce $cross\_road$, since it naturally, such an agent might not want to risk taking the wrong decision.

However, such notion of negation might not be appropriate for representing another type of "common-sense" reasoning. In some cases, we might want to be able to express negation as a condition where the information about the truth (or provability) of a certain fact is not available. This is usually the case when we want to express *normative* statements, i.e., statements of the form "A's usually true, unless B is known to be true". In logic programming, the absence of information about a certain facts is expressed using *default negation/negation-as-failure*(NAF), and is usually written using the symbol "*not*". In contrast to classical negation, negation-as-failure does not require the availability of the explicit information of whether a certain fact is false. The absence of any information about the truth of a (or $\neg a$) is enough to deduce the *not* a (or *not* $\neg a$) is true. Let us consider the following well-known example.

Example 1.3. The following program represents a simple reasoning about the flying ability of birds.

$$flies(X) \leftarrow bird(X), not\,abnormal(X)$$
$$bird(X) \leftarrow penguin(X)$$
$$abnormal(X) \leftarrow penguin(X)$$
$$bird(tux).$$

The first rule represents the normative statement about birds' flying ability. A bird normally flies, unless if it is an exceptional type of bird which is "abnormal" with respect to the flying ability. The second rule specifies that a penguin is one kind of bird, while the third

rule specifies that penguins are abnormal type of birds (w.r.t flying ability). Given the fact $bird(tux)$, the first rule will imply the normative statement $flies(tux)$, since there is no reason to believe that tux is abnormal. In this case, $not\,abnormal(tux)$ is *assumed* to be true, since there is no indication otherwise. However, if later we are given that $penguin(tux)$ is true, then the program derives $abnormal(tux)$ and hence, the previous assumption about $not\,abnormal(tux)$ no longer holds. At this point, we cannot deduce $flies(tux)$ anymore.

One of the properties satisfied by classical logic is the property of **monotonicity**: if a formula ϕ is derivable from a theory T, then the formula is still derivable if we add more theories to T, i.e., $T \models \phi$ implies $T' \models \phi$ for any theory $T' \supseteq T$. Consider now the program in Example 1.3. Initially, the program concludes $flies(tux)$. By adding the fact $penguin(tux)$, $flies(tux)$ no longer holds. This examples illustrates that the property of monotonicity no longer holds in a program with negation-as-failure. Reasoning in a logic system which does not satisfy the monotonicity property is called a **non-monotonic** reasoning. Note that when a logic program such as the one presented in Example 1.3 does not contain negation-as-failure, then monotonicity will still hold.

The early studies on non-monotonic reasoning include formalisms such as: Reiter's *closed world assumption* (CWA) [Reiter, 1980], Clark's program completion [Clark, 1978], McCarthy's circumscription [McCarthy, 1977] as well as Reiter's default reasoning [Reiter, 1980]. Today, answer set semantics is becoming one of the most-widely used and studied semantics for non-monotonic reasoning.

1.3 Stable Model Semantics and Answer Set Programming

The field of Answer Set Programming started with the introduction of what is known as the *stable model semantics* in a seminal paper by Gelfond and Liftschitz [Gelfond and Lifschitz, 1988]. It opened up a new paradigm of on logic programming, which improved upon the semantics of traditional Prolog-based programming. It has since been refined with additional features such as classical negation [Gelfond and Lifschitz, 1990, Przymusinski, 1991] and disjunction in the head [Minker, 1982, Gelfond and Lifschitz, 1991, Przymusinski, 1991] into what is now known as Answer Set Programming. The complexity properties of classes of logic programs under answer set semantics have also been studied and are well-understood. See e.g.,[Dantsin et al., 2001, Eiter and Gottlob, 1993, Eiter et al., 1997, Ben-Eliyahu and Dechter, 1994, Eiter et al., 1998] for discussions on the complexity results.

One of the many improvements which Answer Set Programming presents over Prolog-based programming is a clear semantics guaranteeing termination of the program. Despite being considered "declarative", Prolog programs are not purely declarative, in the sense that, the semantics provided by Prolog still depends on the procedural aspects of the program, such as: the ordering of the body literals in a rule, or the ordering of the clauses in the program. The presence of Prolog cut operator ("!") is also an evidence of Prolog's procedural properties.

This procedural nature can lead to problems such as non-termination of the evaluation of some programs, depending on how the particular Prolog engine is implemented. For example, Prolog solvers usually evaluate the body literals of a rule in the order from left to right. Each body literal is a goal to be executed according to the clauses defining the goal. If a recursive call appears first in the body of a rule, then termination is not guaranteed, and some logical conclusions of the program might not be derivable. Consider again the

program in Example 1.1. If we modify the last two rules, such that the ordering becomes:

$$reachable(X) \leftarrow reachable(Y), two\_steps(Y, X)$$
$$reachable(X) \leftarrow start\_node(Y), two\_steps(Y, X)$$

then traditional Prolog solvers will not be able to derive all the $reachable/1$ extensions, since the first call to the goal $reachable(X)$ will not terminate. Of course, one might argue that this problem can be avoided if we order the body literals in such a way that, indefinite recursion will not occur. However, there are some cases where the declarative logical representation of the problem has to contain such recursion. For example, one might want to represent the transitivity property of the subset relations among sets, i.e., $A \subseteq B$ and $B \subseteq C$ imply $A \subseteq C$. The natural logical representation of this property is

$$subset(A, C) \leftarrow subset(A, B), subset(B, C)$$

In such cases, there is no way to reorder the literals in the body of the rule such that indefinite recursion is avoided. Answer set semantics provides a pure declarative semantics which does not depend on the ordering of the body literals or the ordering of the rules. All the previously shown programs will be evaluated correctly in definite time by an ASP solver.

Another improvement which Answer Set Programming offers is a clear semantics for unstratified negation.[5] Prolog does not provide any clear semantics for any unstratified negation occurring in the program.

Example 1.4. One might want to express the fact that each person is either a female or a male. We exploit the ability of default negation in expressing the following normative statements: "If it can be safely assumed that a person is *not* a male, then the person must be a female", and the corresponding converse statement. The following rules then express these statements:

$$female(X) \leftarrow person(X), not\, male(X)$$
$$male(X) \leftarrow person(X), not\, female(X)$$

For any facts of the form $person(a)$, the answer set semantics computes one answer set containing $female(a)$ and one answer set containing $male(a)$.

Another important feature of Answer Set Programming is the ability to use disjunctive expressions in the head of the rules. Each head atoms represents one possible consequence or outcome of a certain condition. This allows for a very declarative and intuitive encoding of certain problems. For example, the program in Example 1.4 above can be more succinctly written using only one rule as follows:

$$female(X) \lor male(X) \leftarrow person(X)$$

The use of disjunctions in the head does not merely constitute a syntactic addition to the language of logic programs, but also increases the expressive power (and hence, also complexity) of ASP. It is known [Ben-Eliyahu and Dechter, 1994, Dix et al., 1996] that, for a subset of disjunctive logic programs called the *head-cycle-free*(HCF) programs, disjunctions in the head can be rewritten as a set of non-disjunctive rules containing unstratified negations. However, for the non-HCF programs, this no longer applies. See Section 2.2.4 for more discussion on HCF and non-HCF programs.

---

[5]See Section 2.2.3 for an introduction to program stratifications.

A common problem-solving approach used in ASP is the so-called "guess-and-check" approach. It divides the process of problem-solving into two main parts: i) guess a possible solution to the problem and 2) eliminate any candidate solution which does not satisfy the requirements from the problem description. Guessing a possible solution usually involves the use of disjunctions or unstratified negations. These candidates solutions are then checked using a set of rules called *integrity constraints*(or sometimes, hard constraints), which is a set of rules of the following form:

$$false \leftarrow \ B(r), not\ false$$

This rule will eliminate any answer sets which satisfies $B(r)$. Rules of this form are usually written in a shorter form:

$$\leftarrow \ B(r)$$

We illustrate the use of the "guess-and-check" paradigm, by showing the following more realistic example.

Example 1.5. One of the well-know NP-complete problems is the **Graph 3-Colorability** problem [Garey and Johnson, 1979]: given an (undirected) graph $G = \langle V, E \rangle$, decide whether it is possible to color each node in the graph with one of three colors, such that no two adjacent nodes are assigned the same color. Formally, is there any function $f : V \mapsto \{0, 1, 2\}$ such whenever $(v, w) \in E$, then $f(v) \neq f(w)$?

We assume that the graph has been encoded by facts of the form $edge(v, w)$ for every edge between nodes v and w in G. We assume also that a node n being colored by color c is represented by facts of the form $color(n, c)$. The following program solves the problem:

$$node(X) \leftarrow \ edge(X, Y)$$
$$node(X) \leftarrow \ edge(Y, X)$$
$$color(X, 0) \vee color(X, 1) \vee color(X, 2) \leftarrow \ node(X)$$
$$\leftarrow \ color(X1, C), color(X2, C), edge(X1, X2)$$

The first two rules only derive the predicate for the nodes of the graph from the *edge* relations in the facts. The third rule performs the "guessing" part, generating any possible combination of coloring for the graph. The last rule then performs the "checking" part, eliminating any candidate solution which does not satisfy the requirement of the problem. Given the graph in Figure 1.2 as input, there will be six answer sets for the program. One of them is[6]

$$A = \{color(a, 0), color(b, 1), color(c, 1), color(d, 2), \ldots\}$$

which corresponds to a valid coloring scheme.

1.4 Programs with Bounded Predicate Arities

This thesis works is inspired by the work done in [Eiter et al., 2007], which presented the theoretical foundations in the complexity results of programs with bounded predicate arities. It pointed out the problems with current ASP solvers with regards to the space complexity of their computation, and laid out a sketch of a solution to the problem using

---

[6]We show only atoms from the *color* predicate

Figure 1.2: Input graph for Example 1.5

the *meta-interpreter* approach. We briefly summarize the results of the paper, and discuss some findings on the meta-interpreter approach.

Complexity results in [Eiter et al., 2007] show that logic programs with bounded predicate arities have significantly lower complexities than unrestricted programs. Compared to the class of propositional programs, they are one level higher in the polynomial hierarchy. In particular, it is also concluded that polynomial reductions to propositional ASP (with disjunction) from HCF programs with bounded predicate arities are possible. This has at least one important implications towards the implementation of ASP solvers: that reasoning tasks such as answer set existence, brave and cautious reasoning are possible in polynomial space.

One of the steps of logic program evaluation under the answer set semantics is called *grounding*. It transforms logic programs with variables into equivalent variable-free programs, by instantiating each occurrence of a variable with a constant. Grounding is necessary since the answer set semantics is defined on programs without variables. ASP solvers such as clasp and SMODELS have traditionally used an external grounder systems, e.g., Lparse and GrinGo. DLV on the other hand, is an integrated system capable of performing grounding and evaluation by itself.

Observations on current ASP systems show that the grounding step performed by these systems can be the source of the "bottleneck" in the computation. In particular, grounding using current ASP systems may results in an exponentially many ground rules, even for programs with bounded predicate arities. First, we note that, if the program is stratified, current ASP solvers can evaluate the program without performing full grounding. However, if the program contains unstratified negation and/or disjunctions, then current grounders cannot evaluate the rules which depend on the unstratified negations and disjunctions without generating the the full ground instantiations of the rules, which in general may be exponential in size with respect to the input size. The previously stated result has shown that this is not necessary, and it should be possible to avoid producing exponential size ground program at once.

This thesis builds on these premises, first by considering the meta-interpreter approach given in [Eiter et al., 2007]. Modifications on the meta-interpreter schema allowing negation in the body of the rules has been considered. Furthermore, several attempts at improving the efficiency of the approach have also been pursued. Finally, a prototype implementation of the approach was also built to show its feasibility. The system works as a sort of translator, transforming a program with bounded predicate arities into the meta-interpreted form which, in theory, should not cause exponential size grounding when evaluated using current

ASP solvers. Hence, it should provide a better performance than directly evaluating the program using those solvers.

However, some experimental results indicated that the approach might not be so practical, and carries with it a heavy overhead. We thus pursued another direction, the results of which are presented in this thesis. We provide the summaries to the main results and contributions of the thesis in the following sections.

1.4.1 Proposed Evaluation Methods

This thesis proposes three evaluation methods under the answer set semantics for logic programs with bounded predicate arities. The first two methods, which we call Method 1 and Method 2, deal with normal/HCF programs. We start with the observation that a normal/HCF programs has the property that the answer sets of the program are computable by considering only polynomially bounded subsets of the ground program. Both methods first generate the so-called *local answer sets*: answer sets computed from the subsets of the ground program which satisfies certain conditions. Since these subsets are ground and polynomially bounded in size, current ASP solvers can evaluate them without the need perform any grounding, thus avoiding exponential space computation. The local answer sets are then "checked" to ensure that they satisfy all the rules in the program. This checking step is formulated into an evaluation of a program which also does not cause exponential space grounding to be performed by the solvers.

We illustrate the main idea of the methods using the following example.

Example 1.6. Consider the following program, P:

$$
\begin{aligned}
a &\vee b \\
p(X) &\leftarrow a, d(X) \\
q &\leftarrow d(X_1), p(X_1), \ldots, d(X_k), p(X_k) \\
ok &\leftarrow p(X) \\
&\leftarrow not\ ok \\
d(0). \quad &d(1).
\end{aligned}
$$

Lparse, GrinGo and DLV's grounder produce virtually the same ground program for P, with $2^k + 6$ ground rules (excluding the facts), given below:

$$
\begin{aligned}
a &\vee b \\
p(0) &\leftarrow a, d(0) \\
p(1) &\leftarrow a, d(1) \\
q &\leftarrow d(0), p(0), \ldots, d(0), p(0) \\
&\vdots \\
q &\leftarrow d(1), p(1), \ldots, d(1), p(1) \\
ok &\leftarrow p(0) \\
ok &\leftarrow p(1) \\
&\leftarrow not\ ok \\
d(0). \quad &d(1).
\end{aligned}
$$

This program has only one answer set, $A = \{a, d(0), d(1), p(0), p(1), q, ok\}$. Intuitively we feel that this answer set should be derivable using only a small part of this exponentially

many ground rules. Unfortunately, current ASP solvers must perform the full instantiations producing the $2^k + 6$ ground rules given above, before computing any answer set.

On the other hand, our methods for evaluation of normal/HCF programs recognize that an answer set of program P can be derived by only considering a subset of the ground rules which contains only one ground instance for each head atom. The methods then select such subset and compute the answer set(s). Indeed, the answer set A given above is immediately found by considering the following subset of the ground rules:

$$
\begin{aligned}
a &\vee b \\
p(0) &\leftarrow a, d(0) \\
p(1) &\leftarrow a, d(1) \\
q &\leftarrow d(0), p(0), \ldots, d(0), p(0) \\
ok &\leftarrow p(0) \\
&\leftarrow not\ ok \\
d(0). \quad &d(1).
\end{aligned}
$$

thus avoiding the bottleneck associated with the grounding step.

For non-HCF programs, we recognize that the approach given above may not work. In general, it may not be possible to determine polynomially bounded subsets of the ground rules which may contribute answer sets in a non-HCF program. We hence propose another method for dealing with this type of programs. The approach for non-HCF programs primarily consists of two steps: 1) model generation, in which each model of the program which may potentially be an answer set is computed, and 2) minimality checking, where the models produced by the previous step are checked for minimality. For complete discussions on all the three methods, we refer the reader to Chapter 4.

1.4.2 Framework Architecture

To provide a complete view on how one might implement the three methods proposed, we also present a framework architecture to perform evaluation on programs with bounded predicate arities. It provides a decomposition of the evaluation process using the methods proposed into smaller subtasks which can be understood more clearly. A framework component is then defined for each subtask, and a detailed algorithmic description for each of them is then given. Finally, an overall evaluation strategy combining the framework components is presented.

The framework architecture uses an external ASP solver to perform some of its subtasks, including evaluating ground program subsets and model checking, but it ensures that no program gets submitted into the external ASP solver that will cause the solver to perform exponential space grounding. In a sense, the architecture "shields" the ASP solver from having to evaluate programs that will cause the unwanted grounding bottleneck. At the same time, by using the currently available ASP solvers, we can take advantage of many optimization techniques that have been implemented in the solvers.

The framework architecture also makes use of a Prolog engine, mainly to help during the process of generating subsets of the ground program. The framework component responsible for this task performs some rewritings to the original (non-ground) rules to obtain a Prolog program used to compute subsets of the ground rules. It then queries the Prolog engine to obtain a list of rules to be instantiated along with a set of variable substitutions for each rule. The information obtained from this query is used to construct

the corresponding ground rules. Prolog was chosen for this task since Prolog is considered to be sufficiently efficient for this purpose. This choice also helps to simplify the task of generating the subsets of the ground rules, relieving us from having to write a special purpose engine to perform the task. However, in principle, it should be possible to use any procedure for this purpose, as long as it can perform the task as specified in the framework.

Finally, an overall evaluation strategy is presented for the framework architecture. The evaluation strategy employs dependency analysis and strongly-connected components decomposition to split the program into smaller program components. This allows the three methods to be applied independently at each program components, thereby increasing the overall efficiency of the evaluation. In line with the goal of keeping the computation to stay within polynomial space, the evaluations of the program components are arranged in such a way that exponential space requirement is avoided. This consists of arranging the flow of the answer sets between program components in a streaming fashion, so that at most only one answer set per program component is stored at any given time. The evaluation along the program components is then performed in a backtracking way to allow all possible answer sets to be considered. We refer to Chapter 5 for the complete description of the framework architecture.

1.4.3 Implementation and Experiments

The final contribution presented in this thesis is the development of a prototype system implementing the three evaluation methods and the framework architecture described above. It uses DLV as the external ASP solver and XSB as the selected Prolog engine. The system is then used to conduct an experiment measuring the performance of the system and comparing it against DLV and claspD. We selected six problems for the experiment, and we generated several instances for each problem with increasing input size and problem parameters. The goal of the experiment is to compare both time and space consumptions of the three systems.

The results obtained from the experiment confirm our expectations. Evaluation of programs with bounded predicate arities using the implemented system clearly stays within polynomial space, avoiding the problems associated with exponential space grounding. Efficiency of the evaluation with respect to time consumption seems to show improvement as well, for some problem instances. In summary, the pattern observed from the results of the experiment is that, program instances which are "easy", in the sense that it has many (easily found) answer sets, are evaluated faster using the proposed methods. For such problem instances, relatively few guesses need to be made by the three proposed methods before an answer set is found, hence the smaller time consumption. ASP solvers such as DLV and claspD do not enjoy the same benefit since they need to wait until all the (exponentially many) ground rules are instantiated and stored.

However, not all test cases show such good results. Some problem instances, notably the inconsistent ones, have shown that the proposed evaluation methods might have a higher time consumption. Intuitively, this is due to the fact that those methods do not have any sophisticated way to detect inconsistencies in a program early in the evaluation, and have to resort to generating many guesses before concluding that the input program is inconsistent. For a more detailed description on the implemented system, as well as the results and analysis of the experiment, we refer to Chapter 6.

1.5 Thesis Organization

This thesis is organized in the following structure:

1. Chapter 2 reviews some preliminaries and previous results related to Answer Set Programming and the complexity of the semantics of Answer Set Programming.

2. Chapter 3 introduces the notion of programs with bounded predicate arities, summarizes the complexity results of programs with bounded predicate arities as presented in [Eiter et al., 2007], as well as discussing the Meta-Interpreter approach in overcoming the problem of exponential space grounding.

3. Chapter 4 comprises the main contributions of the thesis work. It presents three evaluation methods under the answer set semantics which stay within polynomial space for logic programs with bounded predicate arities.

4. Chapter 5 describes a framework architecture designed to perform evaluation based on the three proposed methods and provides a general evaluation strategy and algorithms which implement those methods.

5. In Chapter 6, we provide a description of the system called BPA, which we have developed to perform the evaluation methods using the the framework architecture described in Chapter 5. This chapter also details the experiment we conducted to measure the performance of the proposed methods and its implementation.

6. Finally, Chapter 7 provides a summary and closing remarks for the thesis work, as well as a brief discussion on possible future works based on this thesis.

2

Preliminaries

In this chapter, we provide a review of the definitions and results in logic programming and answer set semantics. We begin with a discussion on the general topic of declarative logic programming, and continue with a branch of declarative logic programming relevant to this work: answer set programming. Several important concepts in answer set programming will be summarized. This chapter will conclude with a discussion on the topic of computational complexity and a review of complexity results in logic programming.

2.1 Logic Programs

Declarative logic programming is a paradigm in computer programming, whereby one solves an instance of a problem by specifying the conditions of the problems and the solutions one wants using a certain logical syntax and obtains the solutions as sets of answers derived from the semantics of the logical syntax. Using this paradigm, the programmer is usually only concerned with how to declare the problem and the solutions, but not with how the answers are computed using the semantics. A declarative logic program can be written in a more compact and concise way to solve a certain problem than it would otherwise be required when one uses an imperative programming language.

Several approaches to declarative logic programming have been studied and implemented, with varying degree in how declarative these approaches are. For example, Prolog is a programming language developed as one of the declarative programming languages, however it is not purely declarative since the order of the rules in the program and the order of atoms in the body of a rule affects the results. Moreover, the semantics provided by Prolog does not guarantee termination for all syntactically correct Prolog programs.

Another approach for declarative logic programing which has gained popularity in the recent years is the approach using the *stable model semantics* and *answer set semantics* [Gelfond and Lifschitz, 1988, 1990]. Stable model semantics builds up from Prolog by allowing unstratified negations and defining a formal semantics which guarantees termination. Answer set semantics extended stable model semantics by allowing classical negation as well as disjunction in the head. For the rest of this chapter, we will focus on logic programming using these semantics. The next section reviews the syntax and semantics of logic programming under the answer set semantics.

2.2 Logic Programs under the Answer Set Semantics

The answer set semantics is one of the most popular and generally accepted semantics in logic programming. It has its roots on the so called *stable model* semantics defined in [Gelfond and Lifschitz, 1988], which was further extended to allow classical negation [Gelfond and Lifschitz, 1990, Przymusinski, 1991]. The use of disjunction in the head has been considered since the early work in [Minker, 1982], and [Gelfond and Lifschitz, 1991, Przymusinski, 1991] extended the semantics to allow disjunction in the rule head.

2.2.1 Syntax

Let σ^P, σ^C and σ^V be disjoint sets of predicate symbols, constant symbols and variables, respectively, from a first order vocabulary Φ, where σ^V is infinite while σ^P and σ^C are finite. In accordance with the convention used in most answer set solvers, we assume that each predicate symbol and constant symbol is either a string constant which begins with a lowercase letter or is double quoted, while the variable symbols are string contants which begin with an uppercase letter. A constant symbol may also include numeric values (non-negative integers). A *term* is either a constant or a variable. Given a predicate symbol $p \in \sigma^P$, an *atom* is defined as $p(t_1, \ldots, t_k)$, where each t_i, $1 \leq i \leq k$ is a term, while k is called the *arity* of p. Atoms with arity 0 are called propositional atoms.

A *classical literal* (or simply *literal*) l is an atom a or a classically negated atom $\neg a$. The *complementary literal* for a literal l is $\neg l$, and $\neg\neg l$ is defined to be simply l. A *negation as failure* (*NAF*) literal is a literal l or a default negated literal $not\, l$. Negation as failure is an extension to the concept of classical negation, whereby a default negated literal $not\, l$ is true if all attempts to prove it fail. Thus, $not\, l$ evaluates to true if l is provably false, or when no proof for the truthness of l can be found.

A rule r is an expression of the form:

$$a_1 \vee \ldots \vee a_k \leftarrow b_1 \ldots, b_m, not\, c_{m+1}, \ldots, not\, c_n \tag{2.1}$$

with $k \geq 0, m \geq 0, n \geq 0$ and $a_1, \ldots a_k, b_1, \ldots, b_m, c_{m+1}, \ldots c_n$ are classical literals. We say that the set $\{a_1, \ldots, a_k\}$ is the *head* of the rule r, denoted by $H(r)$. Similarly, the set $\{b_1, \ldots, b_m, not\, c_{m+1}, \ldots, not\, c_n\}$ is called the *body* of r and is denoted by $B(r)$. We distinguish between the *positive body literals* and the *negative body literals* of r, denoted as $B^+(r)$ and $B^-(r)$, respectively, where $B^+(r) = \{b_1, \ldots, b_m\}$ and $B^-(r) = \{not\, c_{m+1}, \ldots, not\, c_n\}$ and $B(r) = B^+(r) \cup B^-(r)$. To denote the set of all literals appearing in r, we use the notation $Lit(r)$.

A rule r without head literals (i.e., $k = 0$) is called an *integrity constraint* or *hard constraint*. A rule with exactly one head literal (i.e., $k = 1$) is called a *normal rule*. A rule with $k > 1$ is called a *(proper) disjunctive* rule. If the body is empty (i.e., $k = m = 0$), then the rule is called a *fact*, and for simplicity, we omit the symbol "$\leftarrow$". Traditionally, we also write a fact with literal a as $a.$ (with a dot). A rule is called *positive* if $n = m$. A positive normal rule is called a *Horn* rule.

An *extended disjunctive logic program* (*EDLP*) or simply, a *program* $\mathcal{P}$ is a finite set of rules of the form (2.1). The set of (proper) rules (rules which are not facts) is called the *intensional database* (*IDB*) of $\mathcal{P}$ and is denoted by $IDB(\mathcal{P})$, while the set of all facts in $\mathcal{P}$ is called the *extensional database* (*EDB*) of $\mathcal{P}$ and is denoted by $EDB(\mathcal{P})$. We denote by $Lit(\mathcal{P})$, the set of all literals appearing in a program $\mathcal{P}$ (i.e $Lit(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} Lit(r)$). Throughout this thesis, we assume that a logic program satisfies *safety condition* defined in Definition 2.1.

Definition 2.1. A rule r is called *safe* iff it holds that any variable X appearing in $H(r) \cup B^-(r)$ also appears in at least one literal in $B^+(r)$. A logic program $\mathcal{P}$ is *safe* iff all the rules of $\mathcal{P}$ is safe. Specifically, any fact of a program must be ground.

A program $\mathcal{P}$ is called *normal, positive* or *Horn*, respectively, if every rule $r \in \mathcal{P}$ is normal, positive or Horn. Additional classes are summarized in the Table 2.1

Name	restriction
definite Horn	$k = 1, n = m$
Horn	$k \le 1, n = m$
normal	$k \le 1$
definite	$k \ge 1, n = m$
positive	$n = m$
disjunctive	no restriction

Table 2.1: Program classes

2.2.2 Answer Set Semantics

The semantics of an EDLP is defined in terms of a variable-free programs. Thus, we defined first the *ground instantiation* of a program that eliminates all variables in the program.

The *Herbrand universe* of the program $\mathcal{P}$, denoted by $HU_{\mathcal{P}}$ is the set of all constant symbols $c \in \sigma^C$ appearing in $\mathcal{P}$. If no constant symbol appears in $\mathcal{P}$, then $HU_{\mathcal{P}} = \{a\}$, where a is an arbitrary constant symbol taken from Φ. A term, literal, rule or program is called *ground* iff they do not contain any variables. The *Herbrand base* $HB_{\mathcal{P}}$ of a program $\mathcal{P}$ is the set of all ground (classical) literals that can be constructed from the predicate symbols appearing in $\mathcal{P}$ and the constant symbols in $HU_{\mathcal{P}}$. A *ground instance* of a rule r, denoted by $Ground(r)$ is obtained by replacing each variable that occurs in r by a constant symbol in $HU_{\mathcal{P}}$. We denote by $Ground(\mathcal{P})$, the set of all ground instances of the rules in $\mathcal{P}$.

We define the semantics of EDLP by first considering the positive ground programs. A set of literals $L \subseteq HB_{\mathcal{P}}$ is called *consistent* iff every atom $a \in HB_{\mathcal{P}}$ satisfies $\{a, \neg a\} \not\subseteq L$. An *interpretation* I with respect to a program $\mathcal{P}$ is a consistent subset of $HB_{\mathcal{P}}$. We say that a set of literals S *satisfies* a rule r iff $H(r) \cap S \ne \emptyset$ whenever $B^+(r) \subseteq S$ and $B^-(r) \cap S = \emptyset$ hold. In such case, we write $S \models r$. A set of literal S satisfies a program $\mathcal{P}$ iff it satisfies all the rules in $\mathcal{P}$, and similarly, we also write $S \models \mathcal{P}$. A *model* of a program $\mathcal{P}$ is an interpretation $I \subseteq HB_{\mathcal{P}}$ such that I satisfies P. An answer set of a positive ground program $\mathcal{P}$ is the minimal model of $\mathcal{P}$ w.r.t set inclusion.

To extend the definition of the answer set semantics to programs with default negation, we define the notion of *Gelfond-Lifschitz reduct*[1] or simply *GL-reduct*, as follows:

Definition 2.2. The *GL-reduct* of a program $\mathcal{P}$ w.r.t an interpretation I, denoted by $\mathcal{P}^I$, is the ground positive program obtained from $Ground(\mathcal{P})$ by:

(i) deleting every rule $r \in Ground(\mathcal{P})$ such that $B^-(r) \cap I \ne \emptyset$ and

(ii) deleting the negative body literals from the remaining rules.

An answer set of a program $\mathcal{P}$ is an interpretation $I \subseteq HB_{\mathcal{P}}$ such that I is an answer set of $\mathcal{P}^I$. The set of all answer sets of $\mathcal{P}$ is denoted by $ANS(\mathcal{P})$. The program $\mathcal{P}$ is said to be

---

[1]In the literature, it is also called the *Gelfond-Lifschitz transform*

consistent if it has at least one answer set (i.e., $ANS(\mathcal{P}) \neq \emptyset$), and *inconsistent* otherwise. A ground classical literal a is said to be *bravely true* w.r.t a program $\mathcal{P}$ under the answer set semantics iff there exist an answer set $A \in ANS(\mathcal{P})$ such that $a \in A$, otherwise a is said to be *bravely false* w.r.t $\mathcal{P}$. A ground classical literal a is said to be *cautiously true* w.r.t a program $\mathcal{P}$ iff for every answer set $A \in ANS(\mathcal{P})$, it holds that $a \in A$. Note that if $\mathcal{P}$ is inconsistent, every literal $l \in HB_{\mathcal{P}}$ is cautiously true, for trivial reason.

Consider the special case where the program $\mathcal{P}$ is definite Horn. In this case, it is known that $\mathcal{P}$ has exactly one answer set, and that it is characterized using the fixpoint of the so-called *immediate consequence* operator of $\mathcal{P}$, usually denote by $\mathcal{T}_{\mathcal{P}}$.

Definition 2.3. Let I be an interpretation of a definite Horn program $\mathcal{P}$. The *immediate consequence* operator $\mathcal{T}_{\mathcal{P}}$ is defined as $\mathcal{T}_{\mathcal{P}}(I) = \{H(r) \mid B(r) \subseteq I, r \in \mathcal{P}\}$. Furthermore, let the sequence I_i, $i \geq 0$ be defined as follows: $I_0 = \emptyset$ and $I_i = \mathcal{T}_{\mathcal{P}}(I_{i-1})$. Then I_i is monotone and has a least fixpoint, denoted by $Least(\mathcal{P})$.

It is well known that for a definite Horn program $\mathcal{P}$, $Least(\mathcal{P})$ is the only answer set of $\mathcal{P}$. Consider now that $\mathcal{P}$ is normal. By definition, any answer set $A \in \mathcal{ANS}(\mathcal{P})$ is the answer set of $\mathcal{P}^A$. Since $\mathcal{P}^A$ is definite, it must be the case that $A = Least(\mathcal{P}^A)$.

Under the answer set semantics, the following reasoning tasks are defined:

(i) **Answer set existence** Given a program $\mathcal{P}$, decide whether $\mathcal{P}$ has at least one answer set (i.e., whether the program is consistent).

(ii) **Brave reasoning** Given a program $\mathcal{P}$ and an atom a (called the *brave query* atom), decide whether a is bravely true w.r.t $\mathcal{P}$.

(iii) **Cautious reasoning** Given a program $\mathcal{P}$ and an atom a (called the *cautious query* atom), decide whether is cautiously true w.r.t $\mathcal{P}$.

(iv) **Generate all answer sets** Given a program $\mathcal{P}$, compute all answer sets $A \in ANS(\mathcal{P})$.

Example 2.1. Suppose that P_1 is the following program:

$$
\begin{aligned}
a(X) \vee b(X) &\leftarrow d(X) \\
c(X) &\leftarrow a(X) \\
c(X) &\leftarrow b(X) \\
d(0).d(1).
\end{aligned}
$$

We have that $HB_{\mathcal{P}} = \{a(0), b(0), c(0), d(0), a(1), b(1), c(1), d(1)\}$. The ground program $Ground(P_1)$ is:

$$
\begin{aligned}
a(0) \vee b(0) &\leftarrow d(0) \\
c(0) &\leftarrow a(0) \\
c(0) &\leftarrow b(0) \\
a(1) \vee b(1) &\leftarrow d(1) \\
c(1) &\leftarrow a(1) \\
c(1) &\leftarrow b(1) \\
d(0).d(1).
\end{aligned}
$$

There are exactly four answer sets of P_1:

- $A_1 = \{d(0), d(1), a(0), a(1), c(0), c(1)\}$

- $A_2 = \{d(0), d(1), a(0), b(1), c(0), c(1)\}$

- $A_3 = \{d(0), d(1), a(1), b(0), c(0), c(1)\}$ and

- $A_4 = \{d(0), d(1), b(0), b(1), c(0), c(1)\}$.

Notice that the GL-reduct of P_1 w.r.t each A_i, $P_1^{A_i}$, $1 \leq i \leq 4$ is the same as $Ground(P_1)$. The atoms $d(0)$ and $d(1)$ has to be in each model of $P_1^{A_i}$ since they are given as facts. Moreover, from the rules $a(0)vb(0) \leftarrow d(0)$ and $a(1)vb(1) \leftarrow d(1)$ we infer that at least one of $a(0)$ or $b(0)$ and $a(1)$ or $b(1)$ must be in each of the models as well. The subset minimal sets satisfying these properties are exactly the answer sets given above.

It is clear that each of $c(0), c(1), d(0)$ and $d(1)$ is bravely and cautiously true w.r.t P_1. On the other hand, each of $a(0), a(1), b(0)$ and $b(1)$ is also bravely true, but cautiously false, w.r.t P_1.

We will now review some restrictions and analysis to logic programs which will allow for an efficient evaluation of the programs.

2.2.3 Program Stratification

The notion of program stratification was first introduced independently in [Apt et al., 1988] and [Gelder, 1989]. Przymusinski generalized the notion to constraint-free disjunctive logic programs [Przymusinski, 1989, 1991].

Definition 2.4. We say that a ground program $\mathcal{P}$ is *stratified* iff there exists a function $\lambda : Lit(\mathcal{P}) \mapsto \mathbb{N}^+$ such that for every rule $r \in \mathcal{P}$ of the form 2.1, there exists a $c \in \mathbb{N}^+$ such that:

1. $\lambda(h) = c$ for all $h \in H(r)$

2. $\lambda(b) \leq c$ for all $b \in B^+(r)$ and

3. $\lambda(b) < c$ for all $b \in B^-(r)$

It is well known that such stratification function λ can be efficiently found, if it exists. In particular, if a program is positive, then it is clearly stratified. Stratified programs allow for a more efficient evaluation than unrestricted programs. If the program is free of integrity constraints, then it is guaranteed to have at least one answer set.

The main idea in the evaluation of a stratified program is to perform a layered computation of the answer set of the rules in the program, where each layer is identified by the values of the stratification function λ on the negative body literals of the rules.

Example 2.2. Suppose P_2 is the following program:

$$a \leftarrow b, not\, c$$
$$d \leftarrow b, not\, a$$
$$p \vee q \leftarrow a, not\, d$$
$$b.$$

Program P_2 is stratified, since the function λ' defined as:

- $\lambda'(c) = \lambda'(b) = 0$

- $\lambda'(a) = 1$

- $\lambda'(d) = 2$ and

- $\lambda'(p) = \lambda'(q) = 3$

satisfies all the conditions for the stratification function λ described in Definition 2.4. We can compute the answer sets of P_2 in the following manner:

1. We start off with the fact b, which must be true and belong to any answer set of P_2.

2. Since no rule concludes c, it must not belong to any answer set of P_2, hence $not\ c$ is true. By the rule $a \leftarrow b, not\ c$, it is concluded that a is true as well

3. Since a is provably true, the rule $d \leftarrow b, not\ a$ cannot be applied to derive d.

4. Because a is true and d is not provably true, the last rule $p \lor q \leftarrow a, not\ d$ concludes p or q must be true in one of the answer sets. In conclusion, we derive two answer sets of P_2: $\{a, b, p\}$ and $\{a, b, q\}$.

Example 2.3. Consider the program P_3 as follows:

$$
\begin{aligned}
a &\leftarrow not\ b \\
b &\leftarrow not\ p \\
p &\leftarrow not\ q \\
q &\leftarrow not\ a
\end{aligned}
$$

Program P_3 is not stratified, hence a layered evaluation as performed in Example 2.2 is not possible. To compute the answer sets, one must resort to *guessing* model candidates and then *checking* whether the candidates satisfy the condition of being an answer set.

In the case where $\mathcal{P}$ is non-ground, stratification is defined on the ground program $Ground(\mathcal{P})$. Consider the following example:

Example 2.4. Take program P_4 as follows:

$$
\begin{aligned}
p(X) &\leftarrow q(X), not\ r(X) \\
r(X) &\leftarrow s(X) \\
s(0).q(0).q(1).&
\end{aligned}
$$

The ground program $Ground(P_4)$ is:

$$
\begin{aligned}
p(0) &\leftarrow q(0), not\ r(0) \\
r(0) &\leftarrow s(0) \\
p(1) &\leftarrow q(1), not\ r(1) \\
r(1) &\leftarrow s(1) \\
s(0).q(0).q(1).&
\end{aligned}
$$

Stratification on $Ground(P_4)$ can be obtained by considering the function λ' defined as:

- $\lambda'(s(0)) = \lambda'(s(1)) = \lambda'(q(0)) = \lambda'(q(1)) = 0.$

- $\lambda'(r(0)) = \lambda'(r(1)) = 1.$

- $\lambda'(p(0)) = \lambda'(p(1)) = 2.$

2.2.4 Head-cycle-freeness

Another possible restriction to an EDLP is called *head-cycle-freeness*. To describe the definition of *head-cycle free*, we first introduce the notion of *positive dependency graph* of a program.

The *positive dependency graph* of a program $\mathcal{P}$ is a directed graph where each predicate p occurring in $\mathcal{P}$ is a node and there is an edge from p_1 to p_2 if there is a rule $r \in \mathcal{P}$ such that $p_1 \in H(r)$ and $p_2 \in B^+(r)$. A program $\mathcal{P}$ is called *head-cycle-free* (HCF) iff its positive dependency graph does not contain cycles that go through two literals occurring in the head of a rule in $\mathcal{P}$. For such programs, [Ben-Eliyahu and Dechter, 1994] showed the result described in Theorem 2.5:

Theorem 2.5. (cf. [Ben-Eliyahu and Dechter, 1994]) Given a HCF EDLP $\mathcal{P}$, a consistent interpretation S is an answer set iff

1. S satisfies each rule in $\mathcal{P}$, and

2. there is a function $\phi : Lit(\mathcal{P}) \mapsto \mathbb{N}^+$ such that for each literal l in S there is a rule r in $\mathcal{P}$ satisfying the conditions:

   a) $B^+(r) \subseteq S$
   b) $B^-(r) \cap S = \emptyset$
   c) $l \in H(r)$
   d) $S \cap (H(r) \setminus \{l\}) = \emptyset$
   e) $\phi(l') < \phi(l)$ for each $l' \in B^+(r)$

The essence of this theorem is that HCF programs (in contrast with unrestricted disjunctive programs) allow for a leveled evaluation of the logic program if we know the function ϕ satisfying the above properties. In fact, head-cycle-free disjunction does not increase the expressivity of the program over the normal logic programs, since a head cycle free disjunctive rule can be replaced by a set of semantically equivalent normal rules using the so-called *shifting process* [Dix et al., 1996].

Definition 2.5. If r is a disjunctive rule of the form 2.1 in a HCF program $\mathcal{P}$, we define $shift(r)$ as the following set of k normal rules:

$$a_1 \leftarrow B(r), not\, a_2, \ldots, not\, a_k$$

$$\vdots$$

$$a_i \leftarrow B(r), not\, a_1, \ldots, not\, a_{i-1}, not\, a_{i+1}, \ldots, not\, a_k$$

$$\vdots$$

$$a_k \leftarrow B(r), not\, a_1, \ldots, not\, a_{k-1}$$

For a HCF program $\mathcal{P}$, we define $shift(\mathcal{P})$ as the normal program obtained by shifting every disjunctive rule in $\mathcal{P}$.

It has been shown [Ben-Eliyahu and Dechter, 1994, Dix et al., 1996] that for a HCF program $\mathcal{P}$, $shift(\mathcal{P})$ is a semantically equivalent program to $\mathcal{P}$. However, shifting a non-HCF program does not yield an equivalent program.

Example 2.6. The following HCF program P_3:

$$a \vee b \leftarrow c$$

$$c.$$

is semantically equivalent to the program $shift(P_3)$ below:

$$a \leftarrow c, not\ b$$
$$b \leftarrow c, not\ a$$
$$c.$$

Example 2.7. It is easy to verify that the following non-HCF program, P_4:

$$a \vee b \leftarrow$$
$$a \leftarrow b$$
$$b \leftarrow a$$

has exactly one answer set $\{a, b\}$, whereas $shift(P_4)$:

$$a \leftarrow b$$
$$b \leftarrow a$$
$$a \leftarrow not\ b$$
$$b \leftarrow not\ a$$

has no answer set.

2.2.5 Program Modularity and Dependency Analysis

In devising an operational semantics for logic programs, analyzing dependency information between head and body of a rule is a common tool that can be used to efficiently evaluate the program. The common approaches that have been studied in this topic include the notion of stratification and *local stratification* [Przymusinski, 1989], *modular stratification* [Ross, 1990] and *splitting sets* [Lifschitz and Turner, 1994]. In [Eiter et al., 1997, Oikarinen and Janhunen, 2008], strongly connected components (SCCs) of the program dependency graph is used to find a modular decomposition of disjunctive logic programs, while [Schindlauer, 2006] described an architecture where dependency analysis and splitting sets are used in conjunction with SCC analysis to perform modular evaluation of a more expressive class of logic programs called HEX-programs. We will review the results obtained in these works, focusing on the relevant points with respect to this thesis.

Following [Lifschitz and Turner, 1994], a *splitting set* for a ground program $\mathcal{P}$ is any set U of literals such that for every rule $r \in \mathcal{P}$, if $H(r) \cap U \neq \emptyset$ then $Lit(r) \subset U$. If U is a splitting set for $\mathcal{P}$, we say that U *splits* $\mathcal{P}$. The set of rules $r \in P$ such that $Lit(r) \subseteq U$ is called the *bottom* of $\mathcal{P}$ relative to the splitting set U and denoted by $b_U(\mathcal{P})$. The program $\mathcal{P} \setminus b_U(\mathcal{P})$ is called the *top* of $\mathcal{P}$ relative to U. It is clear that the head literals of every rule in $\mathcal{P} \setminus b_U(\mathcal{P})$ belong to $Lit(\mathcal{P}) \setminus U$.

Let U be a splitting set for a program $\mathcal{P}$. Consider a set of literals X. For each rule $r \in Ground(\mathcal{P})$ such that $B^+(r) \cap Ground(U) \subseteq X$ and $(B^-(r) \cap Ground(U)) \cap X = \emptyset$, create a new rule r', where $H(r') = H(r)$, $B^+(r') = B^+(r) \setminus Ground(U)$ and $B^-(r') = B^-(r) \setminus Ground(U)$. We define $e_U(\mathcal{P}, X) = \{r' \mid r \in \mathcal{P}\}$.

A *solution* to $\mathcal{P}$ w.r.t U is a pair $\langle X, Y \rangle$ of sets of literals such that:

- X is an answer set for $b_U(\mathcal{P})$,

- Y is an answer set for $e_U(\mathcal{P} \setminus b_U(\mathcal{P}), X)$,

- $X \cup Y$ is consistent.

Theorem 2.8. Splitting Set Theorem (*cf.* [Lifschitz and Turner, 1994]). Let U be a splitting set for $\mathcal{P}$. A set of literals A is a consistent answer set of $\mathcal{P}$ iff $A = X \cup Y$ for some solution $\langle X, Y \rangle$ to $\mathcal{P}$ w.r.t U.

Example 2.9. Consider the following program P_5:

$$
\begin{aligned}
a &\leftarrow c, not\, b \\
b &\leftarrow c, not\, a \\
p \vee q &\leftarrow a, not\, b \\
c.
\end{aligned}
$$

The set $U = \{a, b, c\}$ is a splitting set for P_5. The bottom of P_5 w.r.t U, $b_U(P_5)$ is

$$
\begin{aligned}
a &\leftarrow c, not\, b \\
b &\leftarrow c, not\, a \\
c.
\end{aligned}
$$

A solution to P_5 is $\langle \{a, c\}, \{p\} \rangle$ since $\{a, b\}$ is an answer set of $b_U(P_5)$ and $\{p\}$ is an answer set of $e_U(P_5 \setminus b_U(P_5), \{a, c\}) = p \vee q \leftarrow .$ Hence, $\{a, c, p\}$ is an answer set of P_5.

Splitting sets allow for a modular evaluation of logic programs in the sense that programs are divided into modules which can be evaluated separately. To further take advantage of such modular property of logic programs, we analyze the dependency information in the program and apply the splitting sets theorem on different parts of the program. In order to proceed in this direction, first we define the program *dependency graph*.

Definition 2.6. For a program $\mathcal{P}$, let $a, b \in Lit(\mathcal{P})$ be classical literals appearing in $\mathcal{P}$. We say that a *depends positively* on b ($a \rightarrow_p b$), if one of the following conditions holds:

1. There is some rule $r \in \mathcal{P}$ such that $a \in H(r)$ and $b \in B^+(r)$

2. There are some rules $r_1, r_2 \in \mathcal{P}$ such that $a \in B(r_1)$ and $b \in H(r_2)$, and a and b are unifiable.

We say that a *depends negatively* on b ($a \rightarrow_n b$), iff there is some rule $r \in \mathcal{P}$ such that $a \in H(r)$ and $b \in B^-(r)$. We write $a \rightarrow_1 b$ iff $a \rightarrow_p b$ or $a \rightarrow_n b$. The dependency graph of $\mathcal{P}$, $DG_\mathcal{P}$ is the graph $\langle L, \rightarrow_1 \rangle$, where $L \subseteq Lit(\mathcal{P})$ is the set of classical literals appearing in $\mathcal{P}$. We denote by $\rightarrow$, the transitive closure of $\rightarrow_1$. A *strongly connected component* (SCC) of $DG_\mathcal{P}$ is a maximal set of classical literals $S \subset Lit(\mathcal{P})$ such that $a \rightarrow b$ for every $a, b \in S$ satisfying $a \neq b$.

Using the SCC analysis, we can split the program into *program components*, where each component can be evaluated in a layered manner based on repeated application of the splitting set theorem. Given the dependency graph $DG_\mathcal{P}$ of a program $\mathcal{P}$ and its set of SCCs, we say that PC_S is the *program component* of $\mathcal{P}$ *associated* with an SCC S of $DG_\mathcal{P}$ iff PC_S is the maximal set of rules $r \in \mathcal{P}$ such that for every $r \in PC_S$, $Lit(r) \subseteq S$. We denote by $Comp(\mathcal{P})$, the set of all program components in $\mathcal{P}$. For any two program components PC_S and PC_T, we define the *dependency relation between program components* as follows: PC_S depends on PC_T ($PC_S \rightharpoonup PC_T$) iff there are two literals $a \in S$ and $b \in T$ such that $a \rightarrow b$ in $DG_\mathcal{P}$.

Definition 2.7. The *program component dependency graph* $CG_\mathcal{P}$ of $\mathcal{P}$ is the graph defined by $\langle Comp(\mathcal{P}), \rightharpoonup \rangle$.

By its definition, the program component dependency graph of a program $\mathcal{P}$ is a directed acyclic graph (DAG). Using the splitting set theorem, we may evaluate a logic program in a layered manner according to the structure of its program component dependency graph. The answer sets of a program can be composed from the answer sets of its program components.

Example 2.10. Suppose P_6 is the following program:

$$p.$$
$$a \vee b \leftarrow p$$
$$c \vee d \leftarrow p$$
$$x \leftarrow a, c.$$
$$y \leftarrow b, d.$$

We have the following 5 strongly connected components in the dependency graph DG_{P_6}:

$$C_1 = \{p\}$$
$$C_2 = \{a, b\}$$
$$C_3 = \{c, d\}$$
$$C_4 = \{x\}$$
$$C_5 = \{y\}$$

with the following program components associated with them:

$$PC_1 = \{p\}$$
$$PC_2 = \{a \vee b \leftarrow p\}$$
$$PC_3 = \{c \vee d \leftarrow p\}$$
$$PC_4 = \{x \leftarrow a, c\}$$
$$PC_5 = \{y \leftarrow b, d\}$$

We have that $ANS(PC_1) = \{\{p\}\}$. Going up along CG_{P_6}, we compute the answer sets of PC_2 and PC_3 using $ANS(PC_1)$ as an input. We obtain $ANS(PC_2) = \{\{a, p\}, \{b, p\}\}$ and $ANS(PC_3) = \{\{c, p\}, \{d, p\}\}$. Each possible combination of the answer sets of PC_2 and PC_3 gives a possible input for the evaluation of PC_4 and PC_5, for example: with the input $\{a, c, p\}$, PC_4 has one answer set $\{x, a, c, p\}$. In the end, the set of all the answer sets of P_6 is $\{\{a, c, p, x\}, \{b, c, p\}, \{a, d, p\}, \{b, d, p, y\}\}$.

2.2.6 Answer Set Programming Solvers

Several systems have been developed and implemented that performs evaluation on input of logic programs based on the answer set semantics. We refer to these systems as Answer Set Programming (ASP) solvers, or simply answer set solvers. Several well-known ASP solvers are DLV [Leone et al., 2006], clasp/claspD [Gebser et al., 2009, Drescher et al., 2008], SMODELS [Simons et al., 2002] and ASSAT [Lin and Zhao, 2002]. We review these systems and point out specific restrictions and extensions that they provide.

DLV

The DLV system[2] was developed as a joint work of the University of Calabria and Vienna Technical University and is still being actively maintained. It is an efficient engine for

---

[2]http://www.science.at/proj/dlv/

computing answer sets and accepts as core input language the disjunctive logic programs as defined in Section 2.2.1. DLV also extends the language of logic programs with the following syntactic and semantic extensions:

(1) **Weak constraints** The DLV system provides an extension called *weak constraints* which allows for computation of *optimal answer sets* w.r.t to a certain *penalization* and *prioritization.* We refer the reader to [Buccafurri et al., 1997, 2000] for further discussion on the syntax and semantics of weak constraints in DLV.

(2) **Built-in predicates** To provide a more intuitive syntax for the commonly used comparison predicates, DLV provides the built-in predicates "$X < Y$", "$X <= Y$","$X > Y$", "$X >= Y$", and "$X! = Y$" with the obvious semantic of "less-than", "less-than-or-equal-to", "greater-than", "greater-than-or-equal-to" and "not-equal-to", respectively. Comparison predicates can be used on string literals as well as integer values, so long as they appear on the positive bodies of the rules (or constraints). However, full support for integer arithmetic has not been developed yet. Several built-in predicates such as $A = B + C$ and $A = B * C$ for integer addition and multiplication are provided to emulate arithmetic operation, in conjunction with the use of the predicate "$\#int(X)$" which holds for all nonnegative integers, up to a (user-defined) limit.

(3) **Aggregate predicates** Taking the idea from the database query languages, DLV also provides the so-called *aggregate predicates*[Dell'Armi et al., 2003]. Aggregate predicates allow for expressing properties over a set of elements, such as *count*, *minimum* and *maximum.* They can appear in bodies of the rules and constraints, possibly negated using the negation-as-failure (*not*). The use of aggregate predicates can make the encoding of problems in a more concise and natural way, by minimizing the use of auxiliary predicates and recursive programs. In terms of efficiency, encoding using aggregate predicates often outperforms those without by reducing the size of the ground instantiation of the program. For a brief example of the syntax and semantic of aggregate predicates, consider the following construct:

$$1000 \leq \#max\{X : employee(Y), monthlySalary(Y, X)\} \leq 2000$$

The above aggregate predicate evaluates to true for all interpretations where the maximum salary for any employee is between 1000 and 2000.

Lparse and Smodels

SMODELS [Simons et al., 2002] is a system for solving logic programs under the answer set semantics developed by the researchers at the Helsinki University Technology. SMODELS itself cannot directly receive an input in the syntax of the logic programs defined so far, as is the case with DLV, but it uses a front-end called Lparse (developed by the same researchers) to perform preprocessing and grounding/instantiation. The resulting output from Lparse is a concise numeric representation of ground program computed from the input program, which is read by SMODELS to generate the answer sets. Initially, SMODELS was not built with the capability to evaluate disjunctive logic programs, however, an extended version called GnT [Janhunen et al., 2006] has been developed to handle disjunctive logic programs.

SMODELS requires a stricter safety condition than the one described in Definition 2.1. The input program for SMODELS must satisfy the condition of being *domain-restricted*: any variables appearing in a rule must appear in a so-called *domain predicate* in the positive body literals of the rule. Intuitively, the domain predicates of a program are predicates appearing in the program which are defined non-recursively, for details, see [Niemela et al.,

2000]. This restriction does not actually reduce the expressivity of the language itself. However, the weaker safety condition imposed by DLV does allow for a more concise and natural encoding of certain problems.

SMODELS allows for an extension of the logical syntax by supporting *cardinality constraint* rules. The main idea is that a cardinality constraint such as:

$$1\{a, b, not\, c\}2$$

holds in a model if at least 1 but at most 2 of the literals $\{a, b, not\,\}$ are satisfied. This allows for a compact representation of problems where such cardinality constraint rules are applicable.

Postdam Answer Set Solving Collection

The Postdam Answer Set Solving Collection (Potassco) is a collection of answer set solving tools comprising of several answer sets solvers and grounders, developed by researchers at University of Postdam. The main answer solving system is Potassco is clasp [Gebser et al., 2009], for solving normal/HCF program, and claspD [Drescher et al., 2008] for solving disjunctive programs. clasp and claspD receives as input ground programs in the format of Lparse. Hence, similar to SMODELS, clasp and claspD must be used in conjunction with a grounder, such as Lparse. The Potassco project also provides their own grounder system, called GrinGo. It uses virtually the same input-output format as Lparse, and can be interchanged with Lparse as a grounder.

The Potassco project received a special attention recently, as it won the 2009 ASP Competition, scoring highest among other ASP solver systems. To achieve its performance level, clasp and claspD use sophisticated techniques such as: conflict-driven *nogood* learning, backjumping, restarts and unit propagations.

ASSAT

The system ASSAT takes a different approach at implementing the answer set semantics. The main idea behind its operational semantics lies on the concept of *program completion* and *loop formulas* (see [Lin and Zhao, 2002] for details). For an input consisting of a normal logic program, logical formulas corresponding the program and its completion are constructed. Afterwards, for each loop appearing in the program, a formula is constructed and added to the set of formulas obtained thus far in a selective manner. The resulting logical theory is then solved using a SAT solver to obtain the answer sets of the input program. As such, ASSAT can be viewed as a translator, transforming the problem of computing the answer sets of a logic program into a satisfiability problem solvable using any standard SAT solver.

As is the case with SMODELS, ASSAT does not have its own grounding/instantiation module and relies on Lparse to provide an input of ground logic programs. To perform the actual satisfiability checking, some of the available SAT solvers such as: Chaff2, Walksat, GRASP, Satz and SATO has been used and tested, with Chaff2 being the one to perform best.

2.3 Computational Complexity

We review the topics in computational complexity and provides summary of the complexity results of the reasoning tasks for classes of logic programs. We assume that the reader

is already familiar with the basic concepts in computational complexity, such as: Turing machine, problem reductions, completeness and determinism vs nondeterminism. For a complete treatment of the subjects, see e.g., [Papadimitriou, 1994, Garey and Johnson, 1979].

2.3.1 Complexity Classes

Recall that the complexity class P, resp. NP is the class of decision problems ("yes" or "no" problems) which can be computed on a deterministic, resp. nondeterministic Turing Machines in polynomial time. For a complexity class C, the complementary class is denoted by coC. The Polynomial Hierarchy PH is defined recursively as follows: $\Sigma_0^P = \Pi_0^P = \Delta_0^P = \mathsf{P}$, while

$$\Sigma_k^P = \mathsf{NP}^{\Sigma_{k-1}^P}$$
$$\Pi_k^P = co\Sigma_{k-1}^P$$
$$\Delta_k^P = P^{\Sigma_{k-1}^P}$$

for $k \geq 1$, and finally $\mathsf{PH} = \bigcup_{k \geq 0} \Sigma_k^P$. Furthermore, $\mathsf{DP} = \{L \cap L' \mid L \in \mathsf{NP}, L' \in \mathsf{coNP}\}$. The complexity class PSPACE is the class of problems computable on deterministic Turing Machines with polynomial space, while $\mathsf{NEXPTIME}$ and $\mathsf{NEXPSPACE}$ denote the classes of problems decidable by nondeterministic Turing Machines in exponential time, resp. space. We recall that the following relations hold

$$\mathsf{NP} \subseteq \mathsf{DP} \subseteq \mathsf{PH} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NEXPTIME}$$

It is generally believed that these inclusions are strict, and that PH is a true hierarchy of classes with increasing complexity. A canonical problem for the classes under PSPACE is the problem of deciding whether a quantified Boolean formula (QBF) $\exists x1 \ldots \exists x_m \forall x_{m+1} \ldots \forall x_n E$, $n > m \geq 1$, is valid, where E is a propositional formula built from atoms $x_1, \ldots, x_n$. The unrestricted version of QBF itself is known to be PSPACE-complete, however, under the restriction of $m = 1$ and $n = 2$,[3] QBF falls under the complexity class Σ_2^P [Garey and Johnson, 1979].

2.3.2 Complexity of Logic Programming

We provide a brief summary to the complexity results of the reasoning tasks in logic programming. For a comprehensive survey to the complexity results of various classes of logic programs, we refer to [Dantsin et al., 2001]. We start by considering the propositional (variable-free) fragment of logic programs. As presented in [Eiter and Gottlob, 1993], we have the following results:

Theorem 2.11. Given a finite propositional EDLP, deciding whether it has an answer set is Σ_2^P-complete.

Proof. Membership in Σ_2^P follows from the fact that for a propositional EDLP P, deciding whether an interpretation S is a consistent answer set of P is in coNP. A guess for a consistent answer set of P can be verified with a call to an NP-oracle, hence the problem is in $\mathsf{NP}^{\mathsf{NP}}$. The main idea to prove hardness is a reduction from the problem of deciding if an instance of QBF is valid, which is known to be in Σ_2^P. This is done by encoding

---

[3]The QBFproblem under this restriction is also called 2QBF

an instance of QBF, Q in an EDLP P_Q and showing that Q is satisfiable iff P_Q has an answer set. We refer the reader to [Eiter and Gottlob, 1993] for the exact details of this encoding.   □

In fact, the result above remains true even under the conditions that negation-as-failure has a single occurrence in P and each rule head contains at most two atoms. The following result shows the complexity of brave reasoning in EDLP.

Theorem 2.12. Given a propositional EDLP P and a set of atoms S, deciding whether S occurs in any answer set of P is Σ_2^P-complete. The result remains valid even under the conditions that: P is positive, each rule head in P contains one or two atoms and S is a single atom.

Proof. Membership follows from the fact that a guess for a consistent answer set A of P such that $S \subseteq A$ can be done in polynomial time using an NP-oracle. Hardness can be proved by considering a positive program P, and the program $P' = P \cup \{a \leftarrow\ not\ a\}$. It is easy to verify that in this case, it holds that the statements: (i) P' has an answer set and (ii) a is contained in any answer set of P, are equivalent to each other. Since deciding whether (i) is true or not is known to be Σ_2^P-hard, the result follows.   □

It is a quite intuitive result that the complexity of deciding answer set existence and brave reasoning fall in the same class, since both problems basically boils down to asking for existence of answer set satisfying certain properties. Cautious reasoning, on the other hand, reflect the complementary problem of deciding whether no answer set with certain properties exists. Naturally, we would expect such problem to be in the complementary class of Π_2^P. The result presented in [Eiter and Gottlob, 1993] confirms this expectation.

Theorem 2.13. Given a propositional EDLP P and a set of atoms S, deciding whether S is contained in every answer set of P is Π_2^P-complete, even for the restrictions that: "*not*" has a single occurrence in P, each rule head in P has one or two atoms and S consists of only one atom.

Proof. Membership in Π_2^P follows from the fact that a guess for an answer set A of P such that $S \not\subseteq A$ can be verified in polynomial time using an NP-oracle. Hence, the complement problem of cautious reasoning is in Σ_2^P, which proves membership of cautious reasoning in Π_2^P. To prove hardness, without loss of generality, consider the case that S has only one element, $S = \{a\}$. Consider the program $P' = P \cup \{\leftarrow a\}$. Then the following statements are equivalent: (i) P' has a consistent answer set, (ii) S is not contained in every answer set of P. Since (i) is known to be Σ_2^P-hard in Theorem 2.12, the complement of (ii) is Π_2^P-hard, which proves the result.   □

For HCF disjunctive programs, as shown in [Ben-Eliyahu and Dechter, 1994], reasoning tasks fall under the lower complexity class coNP-complete, and thus, such programs are transformable to (disjunction-free) normal logic programs under the stable model semantics.

Complexity results for various classes of the non-ground EDLP have been derived, see e.g., [Eiter et al., 1997, 1998]. In general, the complexity of the reasoning tasks for disjunctive logic programs for the non-ground case is one exponential higher compared to the corresponding reasoning tasks for the propositional case. In particular, reasoning tasks with complexity in Σ_2^P-complete for the propositional case becomes NEXP^{NP} for the non-ground case. Intuitively, we can see that the process of grounding a logic program itself might already be exponential, even for programs without negation, a fact which contributes to the exponentially higher complexity for the reasoning tasks of non-ground logic programs.

3

Programs with Bounded Predicate Arities

In this chapter, we focus our attention to the class of logic programs with bounded predicate arities. To state it more formally, we are considering the class of all logic programs $\mathcal{P}$ such that there exist a fixed bound k, with every predicate arity a in $\mathcal{P}$ satisfying $a \leq k$. As it turns out cf. [Eiter et al., 2007], the complexity of the reasoning tasks in this class of programs is much lower than for programs where the arity of the predicates are unbounded. Specifically, it has been shown that the reasoning tasks have complexity well within the polynomial hierarchy, and is thus far cheaper to evaluate than unrestricted programs.

The restriction of bounded predicate arities in logic programs is actually very relevant in practice, since most problems usually expressible using a set of arities with fixed arities to encode relationships between objects. In fact, in such areas as graph theory, planning, knowledge representation and many other areas where logic programming is applicable, most of the interesting problems are expressible using programs with predicate arities of at most 3. Often, complex relationships requiring higher predicate arities can be broken up into predicates with smaller arities. Most importantly, representations of problems in logic programs that require the predicate arities to vary as one of the parametric input of the problem specification are quite rarely encountered in real world applications.

We start this chapter by providing a summary and discussion on the complexity results of programs with bounded predicate arities. We will then a look at how the competitive ASP solver systems performs on such programs, specifically with regard to the space requirements during the grounding steps. As it turns out, each of the systems performs a grounding step on the input program which requires an exponential space, contrary to the results that programs with bounded predicate arities have complexity in the polynomial hierarchy. We will conclude the chapter with a discussion on an approach that has been suggested to overcome this problem (using a *meta-interpretation* technique) and the limitations it has.

3.1 Complexity results

Following [Eiter et al., 2007], we would like denote the taxonomy of the classes of logic programs in a succinct notation. We define the classes $DL[L]$ where $L \subseteq \{not_s, not, \vee_h, \vee\}$. The set L is used to denote the (possibly combined) admission of:

- not_s: (default) negation; the program remains stratified

- not: unrestricted default negation

- $\vee_h$: disjunction; the program remains HCF

- $\vee$: unrestricted disjunction

Thus, we have for instance, that $DL[\vee_h, not_s]$ contains all HCF stratified programs and $DL[\vee, not\,]$ is the full EDLP as defined in Section 2.2.1.

At this point, we would like to review the complexity results for propositional logic programs, as this would give a baseline for comparison against the results for non-ground programs with bounded predicate arities. Using the notation above, we can summarize the complexity of the three canonical reasoning tasks: answer set existence, brave reasoning and cautious reasoning on propositional logic programs for each class in the taxonomy as displayed in Table 3.1.

Existence/brave/cautious	{}	{not_s}	{not}
{}	P/P/P	P/P/P	P/NP/coNP
{$\vee_h$}	NP/NP/coNP	NP/NP/coNP	NP/NP/coNP
{$\vee$}	NP/Σ_2^P/coNP	Σ_2^P/Σ_2^P/Π_2^P	Σ_2^P/Σ_2^P/Π_2^P

Table 3.1: Complexity results for the propositional fragment of logic programs

We now turn to the complexity results for non-ground programs with bounded predicate arities. We will examine each of three reasoning tasks, and discuss how these results compare to the propositional case.

3.1.1 Answer set existence

The complexity results of answer set existence for programs with bounded predicate arities are summarized in Table 3.1.1.

Answer set existence	{}	{not_s}	{not}
{}	coNP	Δ_2^P	Σ_2^P
{$\vee_h$}	Σ_2^P	Σ_2^P	Σ_2^P
{$\vee$}	Σ_2^P	Σ_3^P	Σ_3^P

Table 3.2: Complexity results of answer set existence for programs with bounded predicate arities

Informally, we can understand these results as follows. First, we note that guessing an interpretation $I \subseteq HB_{\mathcal{P}}$ and deciding whether I satisfies $\mathcal{P}^I$ is in Σ_2^P. To decide whether I is a minimal model of $\mathcal{P}$, we can use an oracle to guess $I' \subset I$ and check whether I' satisfies $\mathcal{P}^I$. Such oracle would have Σ_2^P membership, since it first needs to guess a subset $I' \subset I$ (with complexity NP) and for each of this I', it needs to check that I satisfies $\mathcal{P}^I$ (also in NP). This means that the complexity of answer set existence is in Σ_3^P for the general case. For $DL[\vee]$, to decide answer set existence, we observe that existence of a (not necessarily minimal) model implies the existence of an answer set. Thus, minimality check can be omitted, which implies that answer set existence stays within Σ_2^P for this class. For $DL[not\,]$ programs, we can ensure minimality by not only guessing I, but also a (polynomial size) set of founded proofs for I with respect to $\mathcal{P}^I$. A founded proof of a literal a is a sequence of rule applications $r_1\theta_1 \ldots r_k\theta_k$ which derives a from scratch. The

proofs can be checked in polynomial time, and constitute a witness that the corresponding set of ground literal must be in any answer set of the program. Applying this analysis to check answer set existence for the classes of $DL[]$, $DL[not_s]$ and $DL[not\,]$ programs, we obtain the respective complexity results of coNP, Δ_2^P and Σ_2^P. From the discussion on the equivalence of HCF programs and normal programs by way of the shifting operators (cf. Section 2.2.4), we can infer also that answer set existence for $DL[\vee_h]$ is in Σ_2^P as well. For detailed formal proofs, including hardness property for each result, we refer to [Eiter et al., 2007].

In comparison with the complexity results for the propositional case, answer set existence for bounded predicate arities moves one level up in the polynomial hierarchy for every class, except for the class of $DL[not_s]$, where it jumps from tractable to the second level of PH. In this case, we cannot do better than checking stratum by stratum, which requires a polynomial number of NP-oracle calls, thus the Δ_2^P result.

3.1.2 Brave and cautious reasoning

The complexity results for brave and cautious reasoning for programs with bounded predicate arities obtained in [Eiter et al., 2007] are summarized in Table 3.1.2.

Brave/cautious reasoning	{}	{not_s}	{not}
{}	DP^1/NP	Δ_2^P	Σ_2^P/Π_2^P
{$\vee_h$}	Σ_2^P/Π_2^P	Σ_2^P/Π_2^P	Σ_2^P/Π_2^P
{$\vee$}	Σ_3^P/Π_2^P	Σ_3^P/Π_3^P	Σ_3^P/Π_3^P

Table 3.3: Complexity results of brave and cautious reasoning for programs with bounded predicate arities

The DP result for Horn programs comes from the fact that brave reasoning for this class of programs requires a conjunction of two independent decisions, one is in NP and the other is in coNP. The former refers to the test of minimality, while the latter is to show that no contradiction is derivable (which is not needed in the case of definite Horn programs). For normal programs with stratified negation, again we have a slightly higher complexity, since we must a (polynomial) number of NP problems according to the strata of the program, i.e two NP oracle calls are needed per stratum. The other results are quite a natural change compared to the respective results for the propositional case, where the complexity of brave and cautious reasoning for each program class is one level higher in the polynomial hierarchy. Also, as is the case for propositional programs, the complexity of brave reasoning for $DL[\vee]$ is one level higher than cautious reasoning, due to the fact that minimality is needed for the former, but not for the latter.

3.2 Evaluation on current systems

One of the important implications of the complexity results summarized in the previous section is that reasoning tasks on programs with bounded predicate arities is feasible under polynomial space. However, experimental results given in [Eiter et al., 2007] show that the current ASP solver systems do not respect this complexity bound, the reason being that they create a ground program which is equivalent to the input, which in general has exponential size, even for programs with bounded predicate arities. It should be noted, however, that these systems may employ optimization techniques to minimize the cost of a

full grounding step, but in general, exponential space behavior is still observable. We refer the reader to [Eiter et al., 2007] for the complete experimental results on this subject.

We now focus our attention on three systems: DLV, GrinGo/ with clasp/claspD and Lparse with SMODELS. In evaluating an input program, these systems perform three main steps: i) grounding, ii) model generation and iii) model checking. The grounding step produces an equivalent ground program. A partial evaluation may be carried out towards parts of the program. Specifically, the normal-stratified rules of the program can be evaluated efficiently using a variant of an algorithm called the *generalized semi-naive* evaluation technique [Ullman, 1989]. In DLV, this step is performed by a module called *Instantiator/Intelligent-Grounding*(IG)[Leone et al., 2006]. DLV provides the facility to perform only instantiation/grounding, instead of fully evaluating the program, by specifying the command line option `-instantiate`. The output of this instantiation is formatted in a syntactically correct (ground) logic program. In Lparse-SMODELS, the grounding step is performed by Lparse, which also performs semi-naive evaluation to the normal-stratified portion of the program. The output produced by Lparse, however, uses a different format using a numeric representation for the literals appearing in the program. The use of such format allows for a more concise representation of the ground program.

In DLV's Instantiator/IG, the grounding step initially computes the set of ground literals $I = I^T \cup I^{PT}$ which contains all "relevant" ground literals in the input program. The set I^T is the set of *true* literals, in the sense that, they have been determined to be true in every answer set of the program, while the set I^{PT} is the set of *possibly true* literals, which may or may not be in some answer sets of the program. To fully evaluate the literals in I^{PT}, further computation using a guess and check approach needs to be done, which is done by the other modules in DLV: Model Generator and Model Checker.

The set I^T is first initialized with the facts (EDB) of the input program. It is then extended with the head of the (ground) rules instantiated from a rule r which is simplified into facts using the following simplification steps:

- If a positive body q is in $B^+(r)$, and $q \in I^T$, then delete q from $B^+(r)$.

- If a negative body literal *not q* is in $B^-(r)$ and $q \notin I$, then delete *not q* from $B^-(r)$.

- If a negative body literal *not q* is on $B^-(r)$ and $q \in I^T$, then delete the instantiated rule.

In order to exploit stratifications and increase efficiency, evaluation of the extension of the predicates is performed according to a *topological ordering* in the dependency graph. This ordering ensures that, if a rule with a body containing predicate q then the rules for predicate q has been instantiated before and its true instances are already in I. In particular, for disjunction-free stratified program, all generated rule instances will be simplified to facts or deleted, leaving only $I = I^T$ as the (single) answer set.

For rules that contain disjunction or unstratified negation, the instantiated head of each of these rules can be true/false in an answer set. Each of these (ground) head literals is being put into I^{PT}, the "possibly true" literals. All the (ground) literals which appear in an instantiated rule that contains a literal already in I^{PT} will also be put in I^{PT}. We refer to [Leone et al., 2006, Perri et al., 2007, Leone et al., 2001] for more thorough discussions on DLV's instantiation and grounding techniques.

Example 3.1. Let P_1 be:

$$d(0).d(1).$$
$$e(X) \leftarrow d(X). \tag{3.1}$$
$$p(0, X) \leftarrow e(X), not\ q(1, X). \tag{3.2}$$
$$q(1, X) \leftarrow e(X), not\ p(0, X). \tag{3.3}$$
$$r \leftarrow p(Y, X1), p(Y, X2), e(Y), e(X1), e(X2). \tag{3.4}$$

For program P_1, DLV computes I^T by first initializing it with $\{d(0), d(1)\}$. The second rule can then be evaluated, giving the extensions $\{e(1), e(0)\}$, so that $I^T = \{d(0), d(1), e(0), e(1)\}$. $p/2$ and $q/2$ appears in unstratified rules, hence their extensions will be put into I^{PT}. Rules (3) and (4) have only one positive body literals, $e(X)$. Instantiating this literal with the known extensions $e(0)$ and $e(1)$, we obtain $I^{PT} = \{p(0, 0), p(0, 1), q(1, 0), q(1, 1)\}$. Finally, r is added to I^{PT}, after instantiating the body of rule (5) with any of the known extensions for $p/2$ and $e/1$ which are already in I^{PT}.

A literal with predicate q is *solved* if: 1) q is defined solely by non-disjunctive rules, 2) q does not depend (even transitively) to any unstratified predicate or disjunctive predicate. The simplification steps described above will be able to fully evaluate *solved* literals, and determine its truth values. However, for unsolved literals, there will be rules that cannot be further simplified and the evaluation of these literals need to be done using model generation and model checking.

Consider program P_1 in Example 3.1 above. Literals $e(0), e(1)$ are solved and thus can be completely evaluated, and is known to be true in all answer sets of the program. Literals $p(1, 0), p(1, 1), q(0, 0)$ and $q(0, 1)$ are also solved, since no defining rules are instantiated for them, and they are known to be false in all answer sets of P_1. However, literals $p(0, 1), p(0, 0), q(0, 1), q(0, 0)$ and r are not *solved*, because they depend on unstratified rules. The simplification steps described above will result in the so-called *residual ground rules*. In Example 3.1, the resulting residual ground rules are:

$$p(0, 0) \leftarrow not\ q(1, 0).$$
$$p(0, 1) \leftarrow not\ q(1, 1).$$
$$q(1, 0) \leftarrow not\ p(0, 0).$$
$$q(1, 1) \leftarrow not\ p(0, 1).$$
$$r \leftarrow p(0, 0).$$
$$r \leftarrow p(0, 1), p(0, 0).$$
$$r \leftarrow p(0, 1).$$

This instantiated ground rules will need to be submitted into the model generator and model checker to be fully evaluated to obtain the answer sets of the program.

A careful observation to the method described above will reveal the fact that, for certain classes of programs, even with bounded predicate arities, the resulting residual ground program may have exponentially many rules.

Example 3.2. Consider the class of programs P_k, where $k \geq 1$ as follows:

$$a(0).a(1).a(2).b(0).$$
$$c(X) \leftarrow a(X), not\, b(X)$$
$$p \leftarrow a(X1), c(X1), \dots, a(Xk), c(Xk)$$
$$d(X) \vee e(X) \leftarrow c(X)$$
$$q \leftarrow a(X1), d(X1), \dots, a(Xk), d(Xk)$$

Instantiation technique as described above will recognize that the following portion of the program P_k is stratified:

$$a(0).a(1).a(2).b(0).$$
$$c(X) \leftarrow a(X), not\, b(X)$$
$$p \leftarrow a(X1), c(X1), \dots, a(Xk), c(Xk)$$

Hence, the instantiation will evaluate this part of the program completely, obtaining the set $I^T = \{a(0), a(1), a(2), b(0), c(1), c(2), p\}$ and leaves no residual ground rules. However, continuing the evaluation to the next rules, we encounter the rule $d(X) \vee e(X) \leftarrow c(X)$. The ground rule $d(0) \vee e(0) \leftarrow c(0)$ can be omitted, since $c(0) \notin I$. On the other hand, the ground rules $d(1) \vee e(1) \leftarrow c(1)$ and $d(2) \vee e(2) \leftarrow c(2)$ can only be simplified by deleting the body literals $c(1)$ and $c(2)$, giving the ground rules $d(1) \vee e(1)$ and $d(2) \vee e(2)$. At this point, the instantiation step puts $d(1)$, $d(2)$, $e(1)$ and $e(2)$ into I^{PT}. Finally, the last rule is instantiated to 2^k residual ground rules, generated by substituting each of the variables Xi, $1 \leq i \leq k$ with constant symbols 1 and 2. The simplification step only deletes the body literals $a(0)$ and $a(1)$ from the rules. The complete residual program consisting of $2^k + 2$ rules is given below:

$$d(1) \vee e(1)$$
$$d(2) \vee e(2)$$
$$q \leftarrow d(0), \dots, d(0)$$
$$\vdots$$
$$q \leftarrow d(1), \dots, d(1)$$

Lparse and GrinGo perform instantiation in a similar but slightly different manner than DLV's Instantiator/IG, due to the stricter safety condition in the language of Lparse/GrinGo, which allows only domain-restricted rules in the input program. Each domain predicate in the program will be fully evaluated, and the extensions computed from it must be true in every answer set of the program. The property of being domain restricted for a predicate is transitive, meaning that if, for a predicate p, all the rules defining p contain only domain predicates which occur in the positive body literals of the rules, then p is also a domain predicate. Simplification can then be performed similarly as in DLV's Instantiator/IG to obtain the ground residual program. As it is the case in DLV's instantiation, the ground program obtained from Lparse/GrinGo may contain exponentially many rules.

3.3 Meta-Interpreter Approach

As a possible solution to overcome the exponential space behavior of the instantiation/grounding step on the current ASP solver systems, [Eiter et al., 2007] suggested

the use of a *meta-interpreter* approach. Informally, this approach takes an input program P and encodes/represents it as facts. A set of rules is then written to generate polynomially bounded subsets of $Ground(P)$. A meta-interpretation logic program is then defined, which basically follows the technique used in [Eiter et al., 2003] to perform evaluation on these subsets. The resulting program should produce a polynomially sized grounding using systems such as DLV and Lparse.

We describe this approach in a more detailed manner. For simplicity, we consider the only the sublanguage of logic program containing only Horn rules. The meta-interpreter approach defines the several modules as explained in the following descriptions.

Program Table. The *program table* is a set of facts representing the input program. To allow such representation of the program, first rewrite the program into a new program which contains only one predicate symbol. This can be done by selecting a fresh predicate symbol and incorporating the original predicate symbols as its arguments, and filling unused arguments with new constant symbols. The resulting program should have the same intended meaning with the original program.

The program table is then written to encode the occurrences of literals in the rules. Each head literal and body literal is encoded in a fact using the predicate $tabH$ and $tabB$, respectively. To identify the rule in which a literal occurs, a new constant symbol denoting the label for each rule is specified as an argument of the predicates $tabH$ and $tabB$. To allow encoding of a variable X, a new constant symbol $varX$ is used. A simple illustration is given below:

Example 3.3. Consider the program P_1 below:

$$p(X_1, X_5) \leftarrow e(X_1, X_2), \ldots, e(X_4, X_5)$$
$$reachable(X, Y) \leftarrow p(X, Y)$$
$$reachable(X, Y) \leftarrow reachable(X, Z), p(Z, Y)$$
$$e(v_1, x_1).e(v_1, y_1).e(x_1, v_2).e(y_1, v_2).$$
$$e(v_2, x_2).e(v_2, y_2).e(x_2, v_3).e(y_2, v_3).$$

Program P_1 can be rewritten into program P_1' which uses only the predicate symbol q as follows:

$$q(p, X_1, X_5) \leftarrow q(e, X_1, X_2), \ldots, q(e, X_4, X_5)$$
$$q(reachable, X, Y) \leftarrow q(p, X, Y)$$
$$q(reachable, X, Y) \leftarrow q(reachable, X, Z), q(p, Z, Y)$$
$$q(e, v_1, x_1).q(e, v_1, y_1).q(e, x_1, v_2).q(e, y_1, v_2).$$
$$q(e, v_2, x_2).q(e, v_2, y_2).q(e, x_2, v_3).q(e, y_2, v_3).$$

The program table for P_1 is then written as follows:

$tabH(r1, p, varX_1, varX_5).tabB(r1, varX_1, varX_5).\ldots.tabB(r1, varX_4, varX_5).$
$tabH(r2, reachable, varX, varY).tabB(p, varX, varY).$
$tabH(r3, reachable, varX, varY).tabB(r3, reachable, varX, varZ).tabB(r3, p, varZ, varY).$
$tabH(r4, e, v_1, x_1).tabH(r5, e, v_1, y_1).tabB(r6, e, x_1, v_2).tabB(r7, e, y_1, v_2).$
$tabH(r8, e, v_2, x_2).tabH(r9, e, v_2, y_2).tabH(r10, e, x_2, v_3).tabH(r11, e, y_2, v_3).$

We will refer to the program table obtained from a program P as P^{table}.

Program labels. *Program label* is a set of facts which encodes the properties of the input program, such as the labels for the rules, the occurrences of the variable symbols in the rules and the occurrences of the constant symbols. Moreover, a set of new constant symbols is defined to denote the labels of the selected ground rules during the selection and valuation step (defined afterwards). We know that we need at most $num\_p * num\_c^a$ ground rules selected, where num_p, num_c and a denote the number of predicate symbols, the number of constant symbols and the (maximum) arity of the input program, respectively.

Example 3.4. For program P_1 in Example 3.3, we have 7 constants, 3 predicates, and arity 2. Hence, we have $3 * 7^2 = 147$ the following program labels:

$$rule(r1)....rule(r11).$$
$$rulevar(r1, varx1).\quad rulevar(r1, varx2).\quad rulevar(r1, varx3).$$
$$rulevar(r1, varx4).\quad rulevar(r1, varx5).\quad rulevar(r2, varx).$$
$$rulevar(r2, vary).\quad rulevar(r3, vary).\quad rulevar(r3, vary).$$
$$rulevar(r3, varz).$$
$$const(v_1).\quad const(v_2).\quad const(v_3).$$
$$const(x_1).\quad const(x_2).\quad const(y_1).\quad const(y_2).$$
$$label(l1)....label(l147).$$

We refer to the program label obtained from a program P as P^{label}.

Instance selection. The *instance selection* module is a set of of rules which performs nondeterministic instantiation to the rules in the input program and selection of the ground rules. Using the information encoded in the program label, we write the instance selection as follows:

$$sel(L, R) \vee nsel(L, R) \leftarrow\ label(L), rule(R)$$
$$\leftarrow\ sel(L, R1), sel(L, R2), R1 <> R2$$
$$val(L, V, C) \vee nval(L, V, C) \leftarrow\ sel(L, R), rulevar(R, V), const(C)$$
$$nval(L, V, C1) \vee nval(L, V, C2) \leftarrow\ sel(L, R), rulevar(R, V),$$
$$const(C1), const(C2), C1 <> C2$$
$$valued(L, V) \leftarrow\ val(L, V, \_)$$
$$\leftarrow\ label(L), rule(R), sel(L, R),$$
$$rulevar(R, V), not\ valued(L, V)$$

The first rule non-deterministically select a rule for each label, and the second rule makes sure that no label is used in two different rules. The third rule non-deterministically valuate each variable in a rule with a constant. The fourth rule eliminates the models for which a variable is valuated with two different constants. The last two rules makes sure that at each label, all the variables in the rules selected for that label are instantiated with a constant. We refer later to this program as *Sel*.

Program Input. To provide a representation of the ground rules which can be used as an input for the meta-interpreter, we define a module called *program input*. First, to encode the ground literals occurring in a ground rule, we define a set of facts which map each

possible combination of valuation of a literal with the available constant symbols. This can be done using a set of facts of the form $map(p, a_1, a_2, \ldots, a_k, id\_n)$ where p is a predicate symbol and a_i, $i = 1, \ldots, k$ are constant symbols occurring in the program, while id_n is a new constant symbol, unique for each mapping fact.

Using this mapping facts, we can represent the ground rules as facts obtained from the rules generated by the following steps:

- Each facts can be directly included as an input for the meta-interpreter, since there are no variables in facts, so no selection and valuation are necessary. For a fact F with literal $p(a_1, a_2, \ldots, a_k)$ we create a rule:

$$head(A, F) \leftarrow rule(F), map(p, a_1, \ldots, a_k, A)$$

- For a literal $p(X_1, X_2, \ldots, X_k)$ in the head of a rule R, where each X_i is a variable, we create a rule:

$$head(A, L) \leftarrow sel(L, R), tabh(R, X_1, \ldots, X_k), val(L, X_1, Val_1), \ldots,$$
$$val(L, X_k, Val_k), map(p, Val_1, \ldots, Val_k, L)$$

- For a literal $p(X_1, X_2, \ldots, X_k)$ in the body of a rule R, where each X_i is a variables, we create a rule:

$$pbl(A, L) \leftarrow sel(L, R), tabB(R, X_1, \ldots, X_k), val(L, X_1, Val_1), \ldots,$$
$$val(L, X_k, Val_k), map(p, Val_1, \ldots, Val_k, L)$$

Example 3.5. Referring to the program P_1 in Example 3.3, we define the mapping facts as follows:

$$map(e, v_1, v_1, id\_1).$$
$$map(e, v_1, v_2, id\_2).$$
$$\vdots$$
$$map(reachable, y_2, y_2, id\_147).$$

For the facts in program P_1, the program input module for P_1 will contain the following rules:

$$head(A, r4) \leftarrow rule(r4), map(e, v1, x_1, id\_1)$$
$$\vdots$$
$$head(A, r11) \leftarrow rule(r11), map(e, y2, v_3, id\_11)$$

assuming that $id\_1, \ldots, id\_11$ are the constant symbols used in the mapping facts for each of these facts. For the head literal $p(X_1, X_5)$ in the first rule of P_1, we create the following rule:

$$head(A, L) \leftarrow tabH(r1, p, varX_1, varX_5), val(L, X_1, Val_1),$$
$$val(L, X_5, Val_2), map(p, Val_1, Val_2, A)$$

We refer to the program input module for a program P as P^{input}.

Meta-interpreter. Finally, having defined the the representation for the ground rules, we are ready to define the *meta-interpreter* module itself. It basically follows the format given in [Eiter et al., 2003], except for some small details.

$$
\begin{aligned}
pos\_body\_exist(R) &\leftarrow pbl(\_, R) \\
pos\_body\_true(R) &\leftarrow label(R), not\, pos\_body\_exist(R) \\
pbl\_inbetween(X, Y, R) &\leftarrow pbl(X, R), pbl(Y, R), pbl(Z, R), X < Z, Z < Y \\
pbl\_notmax(X, R) &\leftarrow pbl(X, R), pbl(Y, R), X < Y \\
pbl\_notmin(X, R) &\leftarrow pbl(X, R), pbl(Y, R), Y < X \\
pos\_body\_true\_upto(R, X) &\leftarrow pbl(X, R), not\, pbl\_notmin(X, R), in\_AS(X) \\
pos\_body\_true\_upto(R, X) &\leftarrow pos\_body\_true\_upto(R, Y), pbl(X, R), in\_AS(X), Y < X, \\
&\quad\ not\, pbl\_inbetween(Y, X, R) \\
pos\_body\_true(R) &\leftarrow pos\_body\_true\_upto(R, X), not\, pbl\_notmax(X, R) \\
neg\_body\_false(R) &\leftarrow nbl(X, R), in\_AS(X). \\
in\_AS(X) &\leftarrow head(X, R), pos\_body\_true(R), not\, neg\_body\_false(R).
\end{aligned}
$$

Note that the meta-interpreter itself is also designed to handle evaluation of normal logic programs with default negation. However, at the moment, the previous modules: program table and program input assumes that the input program is Horn. This restriction may be removed later on with a more suitable representation in these modules. We will refer to this meta-interpretation program as $Meta$.

The meta-interpretation approach works by combining all the modules described above into a single program. The answer sets of this new program will correspond to the answer set of the answer set of the original program. Formally, if P denotes the input program, then we construct the program $P^t = P^{table} \cup P^{label} \cup P^{input} \cup Sel \cup Meta$. Without going to the formal proof, we may intuitively see that for each answer set $A \in AS(P)$, there is an answer set $A^t \in AS(P^t)$ which corresponds to A. Specifically, if $A = \{a_1, \ldots, a_k\}$, then the corresponding A^t will be of the form $\{in\_AS(m_1), \ldots, in\_AS(m_k)\}$, where each m_i is the label for the mapping of a_i, $1 \leq i \leq k$ as defined in P^{input}.

The meta-interpreter approach described here has been pursued in an earlier work, and some extensions of this approach has been considered. To allow default negation in the rules, modifications on the program table, program label and program input has been defined. Furthermore, since the answer sets produced by the meta-interpreter is computed only from a subset of the original program, an additional step of checking that the answer set does satisfy every rule in the original program is needed, if default negation occurs. Some modifications on the instance selection modules have also been considered which prunes the number of subsets of the ground program evaluated by the meta-interpreter.

To obtain a concrete picture of the performance of this meta-interpretation technique in practical cases, an implementation of this method has been built. The implementation works as a sort of translator, receiving an input program P and outputs the program P^t which can then be fed into an ASP solver. DLV was chosen as the backed solver for this purpose. Experimental test results on the implementation showed that the meta-interpreter approach does not perform quite as well as expected. Even though theoretically the output program P^t should use polynomial amount of space, some memory overhead caused by the meta-interpretation technique proved to be quite big. Specifically, the instance selection module, which contains two disjunctions to perform the selection and valuation of the rules, generates many choice points along the way.

Experimental results confirmed that the meta-interpreter approach is inefficient in practice. Space consumption remains high, even though in theory, it is still polynomial with respect to the input size. Furthermore, evaluation time increases exponentially with increasing input size and is quite high even for small input programs. In the next chapter, we describe a different approach to evaluate programs with bounded predicate arity.

4

Basic Evaluation Methods

In this chapter we propose three different methods for evaluating an input program $\mathcal{P}$ with bounded predicate arities which stay under polynomial space. The first two methods assume that $\mathcal{P}$ is normal or *head-cycle-free* (HCF) disjunctive. It follows from the discussion in Section 2.2.4 that a HCF program can be rewritten into an equivalent normal program using the shifting operation as defined in Definition 2.5. As a result, we can define the same evaluation method for HCF programs as for normal programs. However, this property does not apply to non-HCF programs. We hence propose another method to handle non-HCF programs.

Before describing these evaluation methods, we present several definitions and observations used in the methods.

Definition 4.1. For a rule r, define $pos(r)$ as the rule derived from r by deleting every negative body literals in r, $pos(r) = H(r) \leftarrow B^+(r)$. For a program $\mathcal{P}$, define $pos(\mathcal{P})$ to be the program consisting of $pos(r)$ where r is a rule in $\mathcal{P}$ which is not a constraint, or more formally $pos(\mathcal{P}) = \{pos(r) \mid r \in \mathcal{P}, H(r) \neq \emptyset\}$.

Definition 4.2. Let $\mathcal{P}$ be any logic program. Then $pos(shift(\mathcal{P}))$ is definite Horn, and has exactly one answer set. Define $S_\mathcal{P}$ to be the answer set of $pos(shift(\mathcal{P}))$. To be more precise, $pos(shift(\mathcal{P}))$ might not have an answer set, due to the occurrence of a pair of complementary atoms a and $\neg a$ in different answer sets of $\mathcal{P}$. In such cases, we allow $S_\mathcal{P}$ to be inconsistent.

We make the following observation:

Proposition 4.1. Every answer set A of $\mathcal{P}$ is a subset of $S_\mathcal{P}$.

Proof. Recall that every answer set A of $\mathcal{P}$ is the minimal model of $\mathcal{P}^A$. Now, let A_p denote the answer set of the program $pos(shift(\mathcal{P}^A))$.[1] We will show that $A \subseteq A_p$. Suppose, in the contrary, we have that $A \nsubseteq A_p$. Consider the interpretation $A' = A_p \cap A$. We have that $A' \subsetneq A$. Let r be any rule in $\mathcal{P}^A$. We claim that $A' \models r$. First, observe that r is positive, and that any $r' \in pos(shift(r))$ satisfies $B^+(r') = B^+(r)$ and $H(r') \subseteq H(r)$. If we have $B^+(r) \subseteq A'$, then since $A' \subseteq A_p$, we can choose a rule r' from $pos(shift(r)) \subseteq pos(shift(\mathcal{P}^A))$ such that $A' \cap H(r') \neq \emptyset$. Since $H(r') \subseteq H(r)$, we must have also that $A' \cap H(r) \neq \emptyset$, thus proving that $A' \models r$ for any $r \in \mathcal{P}^A$. This contradicts the fact that A is the minimal model of $\mathcal{P}^A$. This contradiction proves that $A \subseteq A_p$.

---

[1]Similar to the definition of $S_\mathcal{P}$, we allow A_p to be inconsistent also.

Now, observe that $\mathcal{P}^A \subseteq pos(\mathcal{P})$. Hence, we have that $shift(\mathcal{P}^A) \subseteq shift(pos(\mathcal{P}))$ and also $pos(shift(\mathcal{P}^A)) \subseteq pos(shift(pos(\mathcal{P}))) = pos(shift(\mathcal{P}))$. Since both $pos(shift(\mathcal{P}^A))$ and $pos(shift(\mathcal{P}))$ are positive, we can infer that the answer set of $pos(shift(\mathcal{P}^A))$, i.e., A_p, is a subset of the answer set of $pos(shift(\mathcal{P}))$, i.e., $S_\mathcal{P}$. As a conclusion, we have that $A \subseteq A_p \subseteq S_\mathcal{P}$. $\square$

Example 4.2. Take P as the following program:

$$a \vee b \leftarrow \ not\, c$$
$$d \leftarrow \ a$$
$$\neg d \leftarrow \ b$$

We have that $pos(shift(P))$ is the following program:

$$a \leftarrow$$
$$b \leftarrow$$
$$d \leftarrow a$$
$$\neg d \leftarrow b$$

and $S_P = \{a, b, d, \neg d\}$. It is easy to verify that every answer set of P is a subset of S_P.

Another observation we make at this point, is that any answer set A of $\mathcal{P}$ must contain the facts of $\mathcal{P}$, plus all the atoms derivable from these facts and the definite rules of $\mathcal{P}$. Formally, we define the following:

Definition 4.3. For a program $\mathcal{P}$, define $Def(\mathcal{P})$ as the set of all definite Horn rules in $\mathcal{P}$. Define $D_\mathcal{P}$ as the single answer set of $Def(\mathcal{P}) \cup EDB(\mathcal{P})$.

We have the following result:

Proposition 4.3. Every answer set $A \in \mathcal{ANS}(\mathcal{P})$ satisfies $A \supseteq D_\mathcal{P}$.

For any logic program $\mathcal{P}$, the sets $D_\mathcal{P}$ and $S_\mathcal{P}$ provide the lower and upper bound estimates for the answer sets of $\mathcal{P}$. This is one of several key concepts used in the evaluation methods we are about to describe.

4.1 Evaluation Methods for Normal and HCF Programs

In this section, we assume that the input program $\mathcal{P}$ is HCF. After performing the shifting operation, if needed, we obtain an equivalent normal program. To compute the answer sets of $\mathcal{P}$, we make the following observations. Let A be any answer set of $\mathcal{P}$, then every atom $a \in A$ must be derivable using exactly one ground rule, because $\mathcal{P}$ is normal. Since the size of A is polynomially bounded if $\mathcal{P}$ has bounded predicate arities, we can conclude that every answer set A of $\mathcal{P}$ is computable using a polynomially bounded subset of $\mathcal{P}$. In fact, the size of this subset can be set exactly to $|A|$.

This motivates us to pursue the following approach: to compute all answer sets of the normal program $\mathcal{P}$ with bounded predicate arities, we consider some polynomially bounded subsets of $\mathcal{P}$ and then generate *candidate answer sets* from these subsets. Each candidate answer set need not necessarily be an answer set of $\mathcal{P}$, since it is derived only from a subset of $\mathcal{P}$ and might not satisfy every rule in $\mathcal{P}$. We call such candidate answer sets, *local answer sets* on the ground that they are computed from only a subset of the input program. Formally, if $\mathcal{P}' \subseteq \mathcal{P}$ then we say that any $A \in ANS(\mathcal{P}')$ is a local answer set of $\mathcal{P}$. Lemma 4.4 characterizes the property of the local answer sets which are also answer sets of the program.

Lemma 4.4 (Locality lemma)**.** Let $\mathcal{P}$ be any logic program and $\mathcal{P}' \subseteq \mathcal{P}$. An answer set A of $\mathcal{P}'$ is also an answer set of $\mathcal{P}$ if and only if for each rule $r \in \mathcal{P}$ holds that $A \models r$.

Proof. The "only-if" part is trivial since every answer set A of $\mathcal{P}$ must satisfy $A \models r$ for any $r \in \mathcal{P}$. Now, assume that $A \models r$ for all $r \in \mathcal{P}$. Then $A \models \mathcal{P}^A$ as well. Suppose that A is not an answer set of $\mathcal{P}$, then there must exists a proper subset $A' \subset A$ such that $A' \models \mathcal{P}^A$. However, since $\mathcal{P}' \subseteq \mathcal{P}$, it must also holds that $A' \models \mathcal{P}'^A$. But this means that A is not a minimal model of $\mathcal{P}'$, a contradiction. $\square$

Hence, to ensure soundness of this approach, we must check whether each of the candidate answer sets satisfies every rule of the program. For this purpose, we can use a simple rewriting technique and applying it to the rules in $\mathcal{P}$ to produce a set of constraints for checking satisfiability against the local answer set.

Definition 4.4. For any $r \in \mathcal{P}$ of the form 2.1, define $cons(r)$ as the following the constraint:

$$\leftarrow b_1 \dots, b_m, not\ c_{m+1}, \dots, not\ c_n, not\ a_1, \dots, not\ a_k$$

Also, define $cons(\mathcal{P}) = \{cons(r) \mid r \in \mathcal{P}\}$

We claim the following proposition.

Proposition 4.5. For an interpretation I and program $\mathcal{P}$, it holds that $I \models \mathcal{P}$ iff $cons(\mathcal{P}) \cup I$ is consistent

Proof. For the "only-if" part: suppose that $cons(\mathcal{P}) \cup I$ is not consistent. Then there must be a rule $r' \in \mathcal{P}$ such that the body of $cons(r')$, $B(cons(r'))$ evaluates to true. Since $cons(\mathcal{P}) \cup I$ consists of only facts from I and constraints, it must be the case that $I \models B(cons(r'))$. By the definition of $cons(r')$, it can be concluded that $I \models B(cons(r'))$ implies $I \models B(r')$ but $I \not\models H(r')$, which implies that $I \not\models r'$, in contradiction with the assumption that $I \models \mathcal{P}$. Conversely, if $cons(\mathcal{P}) \cup I$ is consistent, then it follows that for every $r \in \mathcal{P}$, $I \not\models B(cons(r))$. This means that either $I \not\models B(r)$ or $I \models H(r)$, each of which implies that $I \models r$. $\square$

Hence, to check every local answer set I for satisfiability against the program $\mathcal{P}$, we can proceed by checking consistency for the program $cons(\mathcal{P}) \cup I$.

Example 4.6. Let P_1 be the program:

$$p(X) \vee q(X) \leftarrow\ e(X)$$
$$r \leftarrow\ e(X_1), p(X_1), \dots, e(X_k), p(X_k)$$
$$e(0).e(1)$$

Program P_1 is semantically equivalent to the following normal program P_1', obtained from P_1 by shifting operation:

$$p(X) \leftarrow\ e(X), not\ q(X)$$
$$q(X) \leftarrow\ e(X), not\ p(X)$$
$$r \leftarrow\ e(X_1), p(X_1), \dots, e(X_k), p(X_k)$$
$$e(0).e(1).$$

$Ground(P'_1)$ contains $2^k + 4$ rules and two facts. However, it can be easily verified that each answer set of P_1 can be computed from a subset of $Ground(P'_1)$ which contains at most 3 rules (not including the facts). For example, the answer set $\{e(0), e(1), p(0), q(1), r\}$ of P_1 is an answer set of the following subset of $Ground(P'_1)$:

$$
\begin{aligned}
p(0) &\leftarrow e(0), not\ q(0) \\
q(1) &\leftarrow e(1), not\ p(1) \\
r &\leftarrow e(1), p(1) \\
e(0).e(1)&.
\end{aligned}
$$

Furthermore, according to Lemma 4.4, each candidate answer set A obtained from evaluating a subset of $Ground(P'_1)$ is also an answer set of P_1 iff it satisfies every rule in P_1. Checking this condition can be done by constructing the following program $cons(P'_1)$:

$$
\begin{aligned}
&\leftarrow e(X), not\ q(X), not\ p(X) \\
&\leftarrow e(X_1), p(X_1), \ldots, e(X_k), p(X_k), not\ r \\
&e(0).e(1).
\end{aligned}
$$

and checking whether $cons(P'_1) \cup A$ is consistent.

We summarize the approach for computing answer sets of a normal or HCF disjunctive program $\mathcal{P}$ as follows:

(i) Perform shifting operation on $\mathcal{P}$ to get an equivalent normal program $\mathcal{P}_N$.

(ii) Compute local answer sets from polynomially bounded subsets of $\mathcal{P}_N$.

(iii) For every local answer set I obtained in step (iii), check whether I satisfies all the rules in $\mathcal{P}$

(iv) Output every local answer set I which satisfies the check in the previous step.

We will now present two different methods to compute the local answer sets of a program.

4.1.1 Method 1

We start by making the assumption that program $\mathcal{P}$ is normal. The main idea of this method is to compute local answer sets by considering the minimum sized subsets of $Ground(\mathcal{P})$ which might produce an answer set. Since we know that any answer set $A \in \mathcal{ANS}(\mathcal{P})$ satisfies $A \subseteq S_{\mathcal{P}}$, we can intuitively reason that A can be obtained by considering the subsets of $Ground(\mathcal{P})$ such that the set of head literals in the rules is exactly $S_{\mathcal{P}}$. We formalize this idea by making the following definition:

Definition 4.5. Call a set of ground rules $R_{\mathcal{P}} \subseteq Ground(\mathcal{P})$, a *minimal subset guess* of $\mathcal{P}$, iff it satisfies the following conditions:

(i) $EDB(\mathcal{P}) \subseteq R_{\mathcal{P}}$.

(ii) For every literal $l \in S_{\mathcal{P}}$, there is exactly one rule $r \in R_{\mathcal{P}}$ such that $H(r) = \{l\}$.

(iii) There is no rule $r \in R_{\mathcal{P}}$ for which $H(r) \not\subseteq S_{\mathcal{P}}$.

(iv) Every positive body literal b in each rule $r \in R_{\mathcal{P}}$ satisfies $b \in S_{\mathcal{P}}$.

Clearly, since $|R_{\mathcal{P}}| = |S_{\mathcal{P}}|$ and $S_{\mathcal{P}}$ is polynomially bounded, we can conclude that every minimal subset guess of $\mathcal{P}$ is also polynomially bounded. We claim the following result:

Proposition 4.7. Every answer set A of a normal program $\mathcal{P}$ is an answer set of at least one minimal subset guess $R_{\mathcal{P}}$ of $\mathcal{P}$

Proof. We will construct the required minimal subset guess from A. Since A is an answer set of $\mathcal{P}$, it holds that $A = Least(\mathcal{P}^A)$. Consider the following sequence of interpretation deriving A using the $\mathcal{T}_{\mathcal{P}^A}$ operator: $A_0 = \emptyset$, and for $i \geq 0$, $A_i = \mathcal{T}_{\mathcal{P}^A}(A_{i-1})$. Since $A = \bigcup_{i \geq 0} A_i$, for each $a \in A$, we can choose the smallest j such that $a \in A_j$. For such j, define $\rho(a)$ as any rule in $\mathcal{P}^A$ such that $H(\rho(a)) = a$. Let $\chi(a)$ be the rule in $Ground(\mathcal{P})$ corresponding to $\rho(a)$, that is, $\rho(a)$ is the result of deleting the negative body literals in $\chi(a)$. Furthermore, we define $\chi(A) = \{\chi(a) \mid a \in A\}$. Also, for any $b \in S_{\mathcal{P}} \setminus A$, define $\chi(b)$ as any rule in $Ground(\mathcal{P})$ such that $H(\chi(b)) = b$, and finally define $R = \{\chi(a) \mid a \in S_{\mathcal{P}}\}$. By its definition, R satisfies all the conditions of a minimal subset guess of $\mathcal{P}$

Now, observe that $\mathcal{P}^A \subseteq R^A$. However, we also have that $R^A \subseteq \mathcal{P}^A$, since $R \subseteq \mathcal{P}$. Hence, it must be the case that $R^A = \mathcal{P}^A$. Since $A \in \mathcal{ANS}(\mathcal{P})$, it must be the case that $A \in \mathcal{ANS}(R)$ also. Hence, we can take R as the required minimal subset guess for A. This completes the proof. $\qquad\square$

Proposition 4.7 guarantees that by considering all minimal subset guesses of $\mathcal{P}$, every answer set A of $\mathcal{P}$ will be produced by Method 1. Together with Lemma 4.4, we have proved that evaluation of logic programs under answer set semantics using Method 1 is sound and complete. We state this result formally as follows:

Proposition 4.8. For a normal/HCF program $\mathcal{P}$, every answer set produced by Method 1 is an answer set of $\mathcal{P}$. Conversely, if A is an answer set of $\mathcal{P}$, then Method 1 will produce A.

Example 4.9. Consider program P_1 from Example 4.6 with $k = 2$. We have that $S_{P_1} = \{e(0), e(1), p(0), p(1), q(0), q(1), r\}$. Consider the following minimal subset guess of P_1:

$$
\begin{aligned}
p(0) &\leftarrow e(0), not\ q(0) \\
q(0) &\leftarrow e(0), not\ p(0) \\
p(1) &\leftarrow e(1), not\ q(1) \\
q(1) &\leftarrow e(1), not\ p(1) \\
r &\leftarrow e(0), p(0) \\
&e(0).e(1).
\end{aligned}
$$

This program has four answer sets, namely:

$$
\begin{aligned}
A_1 &= \{e(0), e(1), p(0), p(1), r\} \\
A_2 &= \{e(0), e(1), p(0), q(1), r\} \\
A_3 &= \{e(0), e(1), q(0), p(1)\} \\
A_4 &= \{e(0), e(1), q(0), q(1)\}
\end{aligned}
$$

which are local answer sets for the original program P_1. One of these four local answer sets (A_3) is actually not an answer set of P_1. Indeed, A_3 does not satisfy the last rule of P_1.

There is only one other answer set of P_1 which has not been accounted for using the above minimal subset guess: $\{e(0), e(1), q(0), p(1), r\}$. This answer set can be obtained by considering the following minimal subset guess:

$$p(0) \leftarrow e(0), not\ q(0)$$
$$q(0) \leftarrow e(0), not\ p(0)$$
$$p(1) \leftarrow e(1), not\ q(1)$$
$$q(1) \leftarrow e(1), not\ p(1)$$
$$r \leftarrow e(1), p(1)$$
$$e(0).e(1).$$

To ensure completeness of Method 1, all possible minimal subset guess would have to be considered. However, we can see from the example above, different minimal subset guess might have common answer sets. Furthermore, since there may be exponentially many answer set, we cannot store previously found answer sets to avoid recomputing them again because this would lead to exponential space requirement. Hence, using Method 1 to generate candidate answer sets might cause some answer sets to be recomputed several times.

Example 4.10. For program P_1 above, it can be verified that $\{e(0), e(1), p(0), p(1), r\}$ is an answer set of every minimal subset guess of P_1.

4.1.2 Method 2

The main idea used in Method 2 to produce a local answer set is by observing that since any answer set/local answer set A of $\mathcal{P}$ satisfies $A \subseteq S_{\mathcal{P}}$, we may find A by iterating over the subsets of $S_{\mathcal{P}}$ and checking each subset for minimality. Furthermore, based on the result stated in Proposition 4.3, we know that only the subsets of $S_{\mathcal{P}}$ satisfying $D_{\mathcal{P}} \subseteq S_{\mathcal{P}}$ need to be considered.

Suppose now, that we have an interpretation A such that $D_{\mathcal{P}} \subseteq A \subseteq S_{\mathcal{P}}$. To prove minimality of A, we need to find a set of ground rules $\mathcal{P}'$ such that A is an answer set of $\mathcal{P}'$. To this purpose, we define the following:

Definition 4.6. A subset $R_{\mathcal{P},A}$ of $Ground(\mathcal{P})$ is called a *supporting minimal subset guess* of $\mathcal{P}$ w.r.t A iff it satisfies the following properties:

(i) $EDB(\mathcal{P}) \subseteq R_{\mathcal{P},A}$.

(ii) For each $a \in A$, there exist exactly one $r \in R_{\mathcal{P},A}$ such that $H(r) = \{a\}$.

(iii) There is no rule $r \in R_{\mathcal{P},A}$ for which $H(r) \not\subset A$.

(iv) For every rule $r \in R$, every positive body literal $p \in B^+(r)$ satisfies $p \in A$.

(v) No rule $r \in R$ has a negative body literal $not\ n \in B^-(r)$ satisfying $n \in A$.

Clearly by the above definition, we have $|R_{\mathcal{P},A}| = |A|$. Since $A \subseteq S_{\mathcal{P}}$ and $S_{\mathcal{P}}$ is polynomially bounded if $\mathcal{P}$ has bounded predicate arities, we conclude that the supporting minimal subset guesses of $\mathcal{P}$ for any A are also polynomially bounded.

If no supporting minimal subset guesses of $\mathcal{P}$ w.r.t A exists, then we can safely conclude that A is not an answer set of $\mathcal{P}$. If a supporting minimal subset guess $R_{\mathcal{P},A}$ of $\mathcal{P}$ w.r.t A does exist, then proving the minimality of A can be done by showing that A is an answer set of $R_{\mathcal{P},A}$. In order to describe the approach we take to achieve this purpose, we need the following definitions:

Definition 4.7. The function π is defined recursively as follows:

- For an atom $a = p(t_1, \ldots, t_k)$, we define $\pi(a) = p'(t_1, \ldots, t_k)$, where p' is a new predicate symbol. Similarly, for a literal $not\ a$, $\pi(not\ a) = not\ \pi(a)$.

- For a set of literals $L = \{l_1, \ldots, l_k\}$, define $\pi(L) = \{\pi(l) \mid l \in L\}$

- For a rule r, define $\pi(r) = H(r) \leftarrow B^+(r), \pi(B^-(r))$

- For a program R, define $\pi(R) = \{\pi(r) \mid r \in R\}$

Definition 4.8. For a set of literals L, define $\tau(L)$ to be the set of constraints

$$\tau(L) = \{\leftarrow\ not\ l, \pi(l) \mid l \in L\}$$

Consider the program $R'_{\mathcal{P},A} = \pi(R_{\mathcal{P},A}) \cup \pi(A) \cup \tau(A)$. We have the following result:

Lemma 4.11. $R'_{\mathcal{P},A}$ is consistent iff A is an answer set of $R_{\mathcal{P},A}$.

Proof. Since $\pi(A) \subseteq R'_{\mathcal{P},A}$, every answer set M of $R'_{\mathcal{P},A}$ must satisfy $\pi(A) \subseteq M$. Furthermore, since we have the constraints $\tau(A) \subseteq R'_{\mathcal{P},A}$, it must be the case that $A \subseteq M$. Consider the program $\pi(R_{\mathcal{P},A})$. It is easy to see that $\pi(R_{\mathcal{P},A})^A = R^A_{\mathcal{P},A}$. If $R'_{\mathcal{P},A}$ is consistent, then it must be the case that A is the answer set of $\pi(R_{\mathcal{P},A})^A$. This implies that A is also the answer set of $R^A_{\mathcal{P},A}$, and consequently also of $R_{\mathcal{P},A}$. Conversely, if $R'_{\mathcal{P},A}$ is inconsistent, then A is not the answer set of $\pi(R_{\mathcal{P},A})^A$, which implies that A is also not the answer set of $R^A_{\mathcal{P},A}$. $\square$

Example 4.12. Consider again program P_1 from Example 4.6:

$$
\begin{aligned}
p(X) &\leftarrow\ e(X), not\ q(X) \\
q(X) &\leftarrow\ e(X), not\ p(X) \\
r &\leftarrow\ e(X_1), p(X_1), \ldots, e(X_k), p(X_k) \\
e(0).&e(1).
\end{aligned}
$$

We have that $S_{\mathcal{P}} = \{e(0), e(1), q(0), q(1), p(0), p(1), r\}$. For the choice of $A = \{e(0), e(1), p(0), q(1), r\}$, we have the following supporting minimal subset guess $R_{\mathcal{P},A}$ for A:

$$
\begin{aligned}
p(0) &\leftarrow\ e(0), not\ q(0) \\
q(1) &\leftarrow\ e(1), not\ p(1) \\
r &\leftarrow\ e(0), p(0) \\
e(0).&e(1).
\end{aligned}
$$

To check whether A is an answer set of $R_{\mathcal{P},A}$, we construct the program $R'_{\mathcal{P},A}$ as follows:

$$e'(0).e'(1).$$
$$p'(0).q'(1).$$
$$r'.$$
$$p(0) \leftarrow e(0), not\ q'(0)$$
$$q(1) \leftarrow e(1), not\ p'(1)$$
$$r \leftarrow e(0), p(0)$$
$$e(0).e(1).$$
$$\leftarrow p'(0), not\ p(0)$$
$$\leftarrow q'(1), not\ q(1)$$
$$\leftarrow r', not\ r$$

Program $R'_{\mathcal{P},A}$ is consistent, with A as its single answer set. Hence, we conclude that A is indeed an answer set of $R_{\mathcal{P},A}$.

Once we established the fact that there is a supporting minimal subset guess R such that A is an answer set of R, we proceed similarly as in Method 1 to check that A satisfies every rule in $\mathcal{P}$. We summarize the steps performed in Method 2 as follows:

(i) Compute the set of literals $S_{\mathcal{P}}$.

(ii) Enumerate interpretation A such that $D_{\mathcal{P}} \subseteq A \subseteq S_{\mathcal{P}}$.

(iii) For each A computed in step (ii), find a supporting minimal subset guess R of A such that A is an answer set of R.

(iv) For each A satisfying the condition in step (iii), check whether A satisfies every rule in $\mathcal{P}$. If it does, then A is an answer set of $\mathcal{P}$.

It is clear that since we begin the computation of a candidate answer set by first guessing the interpretation, and then followed by finding the justification (the supporting subset), no answer sets will be recomputed, as is the case when Method 1 is used.

Finally, to complete our discussion on Method 2, we state the soundness and completeness property this method:

Proposition 4.13. For a program $\mathcal{P}$, any output produced by Method 2 from evaluating $\mathcal{P}$ is an answer set of $\mathcal{P}$. Conversely, if A is an answer set of $\mathcal{P}$, then Method 2 will produce A.

Proof. If the evaluation using Method 2 on program $\mathcal{P}$ produces the answer set A, then it must be the case that A is an answer set of some supporting minimal subset guess $R_A \subseteq \mathcal{P}$. Moreover, we must have $A \models \mathcal{P}$. By Lemma 4.4, A must be an answer set of $\mathcal{P}$.

Suppose now that A is an answer set of $\mathcal{P}$, then A must satisfy $D_{\mathcal{P}} \subseteq A \subseteq S_{\mathcal{P}}$. Hence Method 2 will visit A during the subset generation step. Moreover, $R_{\mathcal{P},A} = \mathcal{P}^A$ is a supporting minimal subset guess of A, and clearly A is an answer set of $R_{\mathcal{P},A}$. Hence, Method 2 will produce A as an output. $\square$

4.2 Evaluation Method for non-HCF Disjunctive Programs

For non-HCF disjunctive programs, the property that each answer set A of a program $\mathcal{P}$ can be computed from a subset of $Ground(\mathcal{P})$ with size equals to $|A|$ no longer holds.

Example 4.14. Let $\mathcal{P}_2$ be the program

$$a \vee b \leftarrow$$
$$a \leftarrow b$$
$$b \leftarrow a$$

The only answer set of $\mathcal{P}_2$ is $\{a, b\}$. Furthermore, no proper subset of $\mathcal{P}_2$ is sufficient to derive $\{a, b\}$.

In general, for non-HCF programs, it is not possible to employ the strategy being used in the previous section, by which we try to find answer sets of a program by computing candidate answer sets from subsets of the program. It is generally not true that only one rule is needed to derive an atom in an answer set, hence we cannot know the size of the subset of the program sufficient to compute the answer sets. To evaluate non-HCF programs with bounded predicate arities under polynomial space, we need to use a different approach.

The approach we use will consists of two main steps:

(i) Generate *sufficient* number of models of the input program $\mathcal{P}$. Not all models of $\mathcal{P}$ need to be considered in order to find the answer sets of $\mathcal{P}$. Specifically, following the results in Proposition 4.1 and Proposition 4.3, we need only to consider models I of $\mathcal{P}$ such that $D_{\mathcal{P}} \subseteq I \subseteq S_{\mathcal{P}}$.

(ii) For each model I of $\mathcal{P}$ considered in step (i), we check whether I is the minimal model of the GL-reduct of $\mathcal{P}$ w.r.t I, $\mathcal{P}^I$. By definition, each such model is an answer set of $\mathcal{P}$

We will refer to this method as Disjunctive Method. The following explains the two steps of Disjunctive Method in a more detailed manner:

4.2.1 Generating models

Consider the set $PT_{\mathcal{P}} = S_{\mathcal{P}} \setminus D_{\mathcal{P}}$. We can generate models I of $\mathcal{P}$ satisfying $D_{\mathcal{P}} \subseteq I \subseteq S_{\mathcal{P}}$, using the following program:

$$G = \{a \vee \neg a \mid a \in PT_{\mathcal{P}}\} \cup D_{\mathcal{P}} \cup cons(\mathcal{P})$$

Intuitively, the program $\{a \vee \neg a \mid a \in PT_{\mathcal{P}}\} \cup D_{\mathcal{P}}$ represents a "guess" of an interpretation I such that $D_{\mathcal{P}} \subseteq I \subseteq S_{\mathcal{P}}$, while $cons(\mathcal{P})$ makes sure that the answer set of G satisfies $\mathcal{P}$.

However, we realize that G may have unbounded size rules which depend on disjunctions, and evaluating such program using the current ASP solvers may cause exponential space. The following approach is used to avoid this problem. First, we split the program $\mathcal{P}$ into two parts, $\mathcal{P} = Small(\mathcal{P}) \cup Big(\mathcal{P})$ as follows:

- $Small(\mathcal{P}) = \{r \in \mathcal{P} \mid |Ground(r)| \leq B(P)\}$, where $B(P)$ is a polynomially bounded number w.r.t the size of P.

- $Big(\mathcal{P}) = \mathcal{P} \setminus Small(\mathcal{P})$

The program $Small(\mathcal{P})$ represents the subset of $\mathcal{P}$ which contains the *small rules*, whereas $Big(\mathcal{P})$ represents the *big rules*, which may contain exponentially many ground rules.

We perform a two-step computation: generate *candidate models* using the guessing rules as in G, but use only $cons(Small(\mathcal{P}))$, instead of $cons(\mathcal{P})$. Afterwards, we check these candidate models using the rules is $Big(\mathcal{P})$. Formally, let

$$G_\mathcal{P} = \{a \vee \neg a \mid a \in PT_\mathcal{P}\} \cup D_\mathcal{P} \cup cons(Small(\mathcal{P}))$$

If $I = A \cap S_\mathcal{P}$ for some answer set A of $G_\mathcal{P}$, then I must satisfy $D_\mathcal{P} \subseteq I \subseteq S_\mathcal{P}$. Hence, if the program $I \cup cons(Big(\mathcal{P}))$ is consistent, then I must satisfy $I \models Small(\mathcal{P}) \cup Big(\mathcal{P}) = \mathcal{P}$.

Example 4.15. Let $\mathcal{P}_3$ be the following program:

$$
\begin{aligned}
p(X) \vee q(X) &\leftarrow e(X) \\
p(X) &\leftarrow q(X) \\
q(X) &\leftarrow p(X) \\
r &\leftarrow p(X_1), q(X_1), p(X_2), q(X_2), p(X_3), q(X_3) \\
e(0).&e(1).
\end{aligned}
$$

Let us assume, for the sake of the illustration, that $Small(\mathcal{P}_3)$ is

$$
\begin{aligned}
p(X) \vee q(X) &\leftarrow e(X) \\
p(X) &\leftarrow q(X) \\
q(X) &\leftarrow p(X) \\
e(0).&e(1).
\end{aligned}
$$

and $Big(\mathcal{P}_3)$ is

$$r \leftarrow p(X_1), q(X_1), p(X_2), q(X_2), p(X_3), q(X_3)$$

We have that $S_{\mathcal{P}_3} = \{e(0), e(1), p(0), p(1), q(0), q(1), r\}$, $D_{\mathcal{P}_3} = \{e(0), e(1)\}$. Hence, $G_{\mathcal{P}_3}$ is the folowing program:

$$
\begin{aligned}
p(0) &\vee \neg p(0) \\
p(1) &\vee \neg p(1) \\
q(0) &\vee \neg q(0) \\
q(1) &\vee \neg q(1) \\
r &\vee \neg r \\
&\leftarrow e(X), not\ p(X), not\ q(X) \\
&\leftarrow p(X), not\ q(X) \\
&\leftarrow q(X), not\ p(X) \\
e(0).&e(1).
\end{aligned}
$$

There are exactly two answer sets of $G_\mathcal{P}$:

(i) $A_1 = \{e(0), e(1), \neg r, q(0), q(1), p(0), p(1)\}$ and

(ii) $A_2 = \{e(0), e(1), r, q(0), q(1), p(0), p(1)\}$.

The corresponding $I_1 = A_1 \cap S_{\mathcal{P}_3}$ and $I_2 = A_2 \cap S_{\mathcal{P}_3}$ are:

(i) $I_1 = \{e(0), e(1), q(0), q(1), p(0), p(1)\}$ and

(ii) $I_2 = \{e(0), e(1), r, q(0), q(1), p(0), p(1)\}$.

The program $Big(\mathcal{P}_3)$ has only one rule, which gives $cons(Big(\mathcal{P}_3))$ the following single rule:

$$\leftarrow \ not\, r, p(X_1), q(X_1), p(X_2), q(X_2), p(X_3), q(X_3)$$

It is easy to see that among I_1 and I_2, the only I such that $I \cup cons(Big(\mathcal{P}_3))$ is consistent, is $I = I_2$. Hence, the only model of $\mathcal{P}_3$ that needs to be considered, in order to find the answer sets of $\mathcal{P}_3$, is I_2.

4.2.2 Checking minimality of the models

We recall that an interpretation I is a minimal model of $\mathcal{P}$ iff I is the subset minimal set satisfying $\mathcal{P}^I$. Thus, to show that I is the minimal model of $\mathcal{P}$, we can proceed by enumerating the subsets I' of I and ensure that no I' satisfies $\mathcal{P}^I$. To achieve this goal, we describe the following strategy. First, we construct the following program:

$$\mathcal{P}^{min}(I) = cons(shift(\pi(\mathcal{P})) \cup \pi(I)$$

Then, the following proposition holds:

Proposition 4.16. The program $\mathcal{P}^{min}(I) \cup I'$ is consistent iff I' satisfies $\mathcal{P}^I$.

Proof. Suppose $I' \not\models \mathcal{P}^I$. Then, there is a rule $r \in \mathcal{P}$ such that $I \models B^-(r)$, $I' \models B(r)$ but $I' \not\models H(r)$. Consider the constraint $c = cons(shift(\pi(r)))$. Clearly, $c \in \mathcal{P}^{min}(I)$. Since $I' \models B(r)$, we have that $I' \models B^+(c)$. Also, since $I' \not\models H(r)$, $\pi(I) \cup I' \models B^-(c)$. Putting these two facts together, we conclude that $\pi(I) \cup I' \models B(c)$, and hence we have found an instantiation of the constraint c which evaluates to true, implying the inconsistency of $\mathcal{P}^{min}(I) \cup I'$. Conversely, if $\mathcal{P}^{min}(I) \cup I'$ is inconsistent, we can find a constraint $c \in \mathcal{P}^{min}(I)$ which is violated. Reasoning analogously to the steps above, the reader can verify that for the rule $r \in \mathcal{P}$ such that $c = cons(shift(\pi(r)))$, it holds that $I' \not\models r$. This completes the proof. $\qquad\square$

Example 4.17. Consider program P_4 as follows:

$$a \vee b \leftarrow \ not\, c$$
$$c \leftarrow \ d, not\, a$$
$$d.$$

Let us take two models of P_4: $I_1 = \{b, c, d\}$ and $I_2 = \{a, d\}$. First, consider a proper subset of I_1: $I'_1 = \{c, d\}$. The program $P_4^{min}(I_1) \cup I'_1$ is :

$$c'.d'.$$
$$\leftarrow \ not\, a', not\, b', not\, c$$
$$\leftarrow \ not\, c', d', not\, a$$
$$b.c.d.$$

This program is consistent, which means that $I'_1 \models P_4^{I_1}$, and consequently, $\{b, c, d\}$ is not a minimal model of P_4. On the other hand, it can be verified that for every proper subset I'_2 of I_2, the program $P_4^{min}(I_2) \cup I'_2$ is inconsistent, thus proving minimality of I_2.

We summarize the evaluation method for non-HCF disjunctive programs, as follows: for an input program $\mathcal{P}$:

(i) Split the program into $Small(\mathcal{P})$ and $Big(\mathcal{P})$.

(ii) Construct the program $G_{\mathcal{P}}$ to generate candidate models

(iii) Find an answer set I of $G_{\mathcal{P}}$ such that $I \cup cons(Big(\mathcal{P}))$ is consistent

(iv) For each model I obtained from the previous step, find a proper subset $I' \subset I$ such that $\mathcal{P}_I^{min} \cup I'$ is consistent. If no such I' exists, then output I as an answer set of $\mathcal{P}$.

5

Evaluation Framework

In this chapter, we propose a framework for efficiently evaluating a program with bounded predicate arities using the three methods proposed in the previous chapter. In designing the architecture for the evaluation framework, we recognize that the task of evaluating a logic program under answer set semantics has been studied and implemented which results in several ASP solvers, such as DLV, claspD, SMODELS and ASSAT. Many optimizations techniques have been applied to these solvers to obtain a good evaluation performance. However, as we have seen in the previous chapters, these ASP solvers currently do not have a special evaluation techniques for logic programs with bounded predicate arities that perform under polynomial space complexity.

In the previous chapter, we have described three evaluation methods for answer set semantics which stays within polynomial space for programs with bounded predicate arities. During the steps in these methods, we need to perform evaluations on certain subsets of the program, as well as checking consistencies on some programs constructed from several transformation procedures. On each of these steps, the program to be evaluated or checked for consistency is either stratified or has a polynomially bounded number of ground rules. Current ASP solvers are already capable of performing these types of reasoning tasks in polynomial space with good performance, hence it would be beneficial to use these solvers for these tasks.

As discussed in Chapter 2, Method 1 and Method 2 require generating subsets of the ground input program which satisfy certain required conditions. This task basically amounts to finding instantiations of the rules in a program, which can actually be performed using standard Prolog resolution. We propose to use Prolog to perform this tasks, since many Prolog implementations exist which have relatively good performance for unification and query answering. Using Prolog will allow us to avoid constructing special purpose functions to perform the ground program subsets enumeration, which may not perform quite efficient.

Another aspect of the evaluation framework that will be examined in this chapter is the strategy to exploit the modularity property of the programs. Our evaluation framework will be based on the previous works in programs modularity and strongly connected component (SCC) analysis, while the evaluation strategy will be designed such that it does not require exponential space.

To explain the details of the framework, we will examine the components which will be the building blocks for its architecture. Each component takes a specific type of input, performs a certain task and returns an output. Each component may be composed of other

(smaller) components which make up its functionality. The global view of the architecture will present an overall picture of the interaction between the components to achieve the goal of the evaluation framework. We first present the basic components of the framework.

5.1 Basic Framework Components

5.1.1 External ASP Solver

The architecture of the framework assumes that an external ASP solver is available, which satisfies the following conditions:

(i) The ASP solver is capable of evaluating any stratified logic program under polynomial space

(ii) The ASP solver is capable of evaluating any logic program (possibly unstratified) under polynomial space, provided that the program has only polynomially many ground rules.

(iii) The ASP solver provides a mechanism to interact with it in such a way that the resulting answer sets from the evaluation can be read in a streaming way, i.e one answer set at a time.

The conditions (i) and (ii) are required for the reason that the methods described in the previous chapter requires steps which consist of evaluation of stratified logic programs and programs with polynomially bounded size of ground rules. These two requirements are satisfied by many of the current ASP solvers. The third condition is required in order to avoid the implementation from having to store all answer sets of a program obtained from the external ASP solver, since doing it clearly requires exponential space. This mechanism can be easily realized by implementing a pipeline connection to the ASP solver, which can be used to read answer sets from the ASP solver one answer set at a time.

We will refer to this component as ASPSolver. It receives two inputs: IDB, which is a set of rules and EDB which is a set of facts. It provides two functionalities: computing the answer sets of $IDB \cup EDB$ and checking whether $IDB \cup EDB$ is consistent. To achieve these functionalities, the ASPSolver component is equipped with two functions, which we will denote by getNextAnswerSet and answerSetsLeft. The first call for getNextAnswerSet returns the first answer set of $IDB \cup EDB$ and subsequent calls will return the next answer sets, until no more answer sets are available. Of course, the order by which the answer sets are returned does not matter.

The functions answerSetsLeft returns the value **true** or **false** based on the condition whether all answer sets have been returned using getNextAnswerSet or not. It should be clear also that if $IDB \cup EDB$ is inconsistent, then answerSetsLeft will always return false.

Example 5.1. Let IDB be the following rules:

$$b \leftarrow a, not\, c$$
$$c \leftarrow a, not\, b$$
$$d \leftarrow b$$

and let $EDB = \{a\}$. Then on the first call, ASPSolver(IDB, EDB).getNextAnswerSet may return $\{a, b, d\}$, and the next call returns $\{a, c\}$. At this point, since all answer sets have been returned, a call to ASPSolver(IDB, EDB).answerSetsLeft will return **false**.

5.1.2 Prolog Engine

The architecture assumes the availability of a Prolog engine to perform queries on Prolog programs that can be used in computing subsets of a logic program. This will be provided by a component called PrologEngine. Component PrologEngine receives an input in the form of a Prolog program, and provides functionality to perform queries as well as to add and delete instances into the Prolog database.

The query functionality will be denoted by the function doQuery. This function takes as an input a Prolog query and sends the query to the Prolog engine. To retrieve the answers of the query, we define the functions getNextAnswer and answersLeft, which work in a similar way to ASPSolver's getNextAnswerSet and answerSetsLeft. Each answer returned is a set of variables bindings satisfying the query specified by the previous call to doQuery w.r.t the current program. We also define a function called checkQuery for this framework component, used to perform a ground query to Prolog and returns a Boolean value **true** or **false**.

To provide a way to add and delete instances to the Prolog database, the PrologEngine component defines two functions: addFacts and delFacts. Each function takes as input a set of facts, and performs Prolog's assert or retract to each fact in the input.

Example 5.2. Let P be the following Prolog program:

$$p(X) \leftarrow q(X), not\, r(X)$$
$$q(X) \leftarrow s(X)$$

Let us first call the function PrologEngine(P).addFacts($\{s(0), s(1)\}$). Then, set up a query $p(X)$ using a call to PrologEngine(P).doQuery($p(X)$). The first answer which may get by calling PrologEngine(P).getNextAnswer might be $X/0$, followed by $X/1$. If we now make a call to PrologEngine(P).addFacts($\{r(1)\}$) and perform the query $p(X)$ once more, then retrieving the answer using PrologEngine(P).getNextAnswer will return only $X/0$. At this point, PrologEngine(P).answersLeft will return **false**.

5.1.3 Global Model Checker

In the evaluation step of both Method 1 and Method 2, there is a step where an interpretation (the candidate answer set) is checked whether it satisfies all the rules of a program. We describe here a simple component to encapsulate this functionality. We will call this framework component ModelChecker. It receives as input a set of rules IDB. Only one function is provided by this component, which is called checkModel. This function takes as an input an interpretation I and return the value **true** or **false** depending whether I models IDB or not.

To realize this functionality, first ModelChecker rewrite the rules in IDB into the set of constraints $cons(IDB)$, and then, using the component ASPSolver, it checks whether $cons(IDB) \cup I$ is consistent. This simple process is described in Algorithm 1.

5.1.4 Model Generator

This component, which we will call ModelGen, is used to generate models of a program which may potentially be an answer set of the program. It receives two input data: a set of rules IDB and a set of facts EDB and outputs some models of $IDB \cup EDB$. This functionality is provided using two functions called getNextModel and modelsLeft.

To generate the required models from $IDB \cup EDB$, we first apply the shifting operator IDB. Let IDB_s be the resulting program after applying this operation. Next, we take the

---
Algorithm 1 Algorithm for ModelChecker

Input: IDB a set of rules, I an interpretation
Output: value **true** or **false**
 1: Construct $cons(IDB)$ from IDB
 2: **if** ASPSolver$(cons(IDB), I)$.answerSetsLeft $=$ **true then**
 3:      **return true**
 4: **else**
 5:      **return false**
 6: **end if**

---

positive program obtained by deleting all the negative body literals in IDB_s. Let IDB_s^+ be the resulting program, which is a Horn program. We now need evaluate the program $IDB_s^+ \cup EDB$ under the paraconsistent answer set semantics to obtain a single answer set PT. To emulate paraconsistent evaluation, we can simply rename each classically negated atom $\neg a$ appearing in IDB or EDB into neg_a, and rename all the atoms having the form neg_a in PT back into $\neg a$.

After obtaining PT, we start to enumerate all the possible subsets of $S \subseteq PT$ such that $EDB \subseteq S$. For each such S, we check whether the program $cons(IDB) \cup S$ is consistent. If it is consistent, then S will be one of the output returned by getNextModel. The Algorithm 2 specifies how the models are generated in ModelGen.

Example 5.3. Let IDB be the following program:

$$a \vee b \leftarrow c, not\, d$$
$$p \leftarrow e, not\, a$$

Then IDB_s^+ is the program:

$$a \leftarrow c$$
$$b \leftarrow c$$
$$p \leftarrow e$$

and $cons(IDB)$ is the following program:

$$\leftarrow not\, a, not\, b, c, not\, d$$
$$\leftarrow not\, p, e, not\, a$$

Let $EDB = \{c, e\}$. Then $PT = \{c, e, a, b, p\}$, and the models obtained using ModelGen are: $\{c, e, a\}$, $\{c, e, a, p\}$, $\{c, e, b, p\}$, and $\{c, e, a, b, p\}$. Each of these models can be obtained by invoking the function ModelGen(IDB, EDB).getNextModel.

It should be clear that not all models of $IDB \cup EDB$ are admitted as an output of ModelGen. For example, in the program given above, the Herbrand base for $IDB \cup EDB$: $\{a, b, c, d, p, q\}$ (which of course, is a model of the program) is not admitted as one of the models obtained from ModelGen(IDB, EDB).getNextModel. However, from the discussions in Section 4.1.1, we can conclude that all answer sets of the input program will be admitted as outputs of this component.

5.1.5 Disjunctive Model Generator

Apart from the component ModelGen defined above, we also define another framework component to generate models which is intended to be used during the evaluation of

---

Algorithm 2 Generating models in ModelGen

Input: IDB a set of rules, EDB a set of facts
Output: some models of $IDB \cup EDB$
 1: $IDB_s \leftarrow$ Shift(IDB)
 2: $IDB_s^+ \leftarrow$ Positive(IDB_s)
 3: $PT \leftarrow$ ASPSolver(IDB_s^+, EDB).getNextAnswerSet
 4: **for each** $S \subseteq PT$ such that $EDB \subseteq S$ **do**
 5:     **if** ModelChecker(IDB).checkModel(S) = **true then**
 6:         Output S
 7:     **end if**
 8: **end for**

---

non-HCF programs. Let us call this component DisjunctiveModelGen. It receives as input a program P and outputs models of the P. Similarly as component ModelGen, DisjunctiveModelGen provides two functions to interface with, called getNextModel and modelsLeft. The method by which the DisjunctiveModelGen performs its functionalities follows the steps outlined in Section 4.2.1. It splits P into $Small(P)$ and $Big(P)$. It then construct the program G_P and compute the answer sets. Each answer set I from G_P is then used to construct the program $I \cup cons(Big(P))$. If $I \cup cons(Big(P))$ is consistent, then I is returned as an output.

5.1.6 Program Subset Generators

In the steps of the evaluation methods for normal and HCF programs described in Section 4.1, we need to generate some subsets of the grounded input program. Given a set of normal rules IDB, a set of facts EDB and a set of atoms PT, in Method 1 we need to generate the *minimal subset guesses* of $IDB \cup EDB$ w.r.t PT, while in Method 2, we need to generate the *supporting minimal subset guesses* of $IDB \cup EDB$ w.r.t PT. These will be performed by two components called SubsetGen1 and SubsetGen2, respectively.

The main processing steps employed in SubsetGen1 and SubsetGen2 consists of rewriting the rules in IDB into Prolog programs to obtain all the substitutions required to instantiate the rules in IDB to obtain the desired subset of the grounded program. For SubsetGen1, this rewriting is performed as follows: for each $r \in IDB$, where r is of the form

$$h \leftarrow B^+(r), B^-(r)$$

we define the following transformation[1]:

$$\pi_1(r) = gr(h, r\_id, val(\overline{X})) \leftarrow B^+(r)$$

where r_id is a unique symbol assigned for each rule $r \in IDB$, and $\overline{X}$ is a vector of all the variables appearing in r. If the rule r contains no variable, we simply put the Prolog anonymous variable (the underscore '_') in place of $val(\overline{X})$. We define the Prolog program $IDB_{\pi_1} = \{\pi_1(r) \mid r \in IDB\}$. We load the program IDB_{π_1} into Prolog together with facts EDB. To obtain the *minimal subset guesses* of $IDB \cup EDB$ w.r.t PT, we first add the facts in PT and then perform the following conjunctive query on the Prolog database:

$$Q_{PT} = gr(a_1, R_1, V_1) \wedge \ldots \wedge gr(a_k, R_k, V_k)$$

---

[1]Note that r is normal, so it contains only one head atom

where $PT = \{a_1, \ldots, a_k\}$. Each answer to this query will be a set of substitutions $Sub = \{Sub_1, \ldots, Sub_k\}$, where each Sub_i will be of the form $\langle R_i/r\_i, val(c_1, \ldots, c_k)\rangle$, which encodes the information to instantiate rule r_i using substitutions $X_1/c_1, \ldots, X_k/c_k$. Each answer contains all the information needed to instantiate the rules in IDB to obtain the required subset of $Ground(IDB)$.

Using the previously defined evaluation component PrologEngine, we can describe all the steps performed in SubsetGen1 in Algorithm 3, where in this algorithm, we have defined the following functions:

- createQuery is a function that takes as input a set of facts and produces the conjunctive Prolog query as described above, and

- Instantiate is a function that takes as input a set of rules IDB and a set of substitutions Sub, and outputs the set of ground rules obtained by instantiating the rules in IDB using the substitutions in Sub.

Algorithm 3 Generating program subsets in SubsetGen1

Input: IDB a set of normal rules, EDB a set of facts, PT an interpretation
Output: A set of ground rules
 1: Construct program IDB_{π_1} from IDB
 2: PrologEngine($IDB_{\pi_1} \cup EDB$).addFacts(PT)
 3: $Q \leftarrow$ createQuery(PT)
 4: PrologEngine($IDB_{\pi_1} \cup EDB$).doQuery(Q)
 5: **while** PrologEngine($IDB_{\pi_1} \cup EDB$).answersLeft $=$ **true do**
 6:     $Sub \leftarrow$ PrologEngine($IDB_{\pi_1} \cup EDB$).getNextAnswer
 7:     $CurrentSubset \leftarrow$ Instantiate(IDB, Sub)
 8:     Output $CurrentSubset$
 9: **end while**

Example 5.4. Let IDB be the following program:

$$p(X) \leftarrow e(X), not\, q(X)$$
$$q(X) \leftarrow e(X), not\, p(X)$$
$$r \leftarrow e(X_1), e(X_2), p(X_1), p(X_2)$$

and $EDB = \{e(0), e(1)\}$. Then IDB_{π_1} is the following Prolog program:

$$gr(p(X), r_1, val(X)) \leftarrow e(X)$$
$$gr(q(X), r_2, val(X)) \leftarrow e(X)$$
$$gr(r, r_3, val(X_1, X_2)) \leftarrow e(X_1), e(X_2), p(X_1), p(X_2)$$

For $PT = \{p(0), p(1), q(1), r\}$, the corresponding conjunctive query is:

$$Q = gr(p(0), R_1, V_1) \wedge gr(p(1), R_2, V_2) \wedge gr(q(1), R_3, V_3) \wedge gr(r, R_4, V_4)$$

After adding the facts in PT into the Prolog database, we obtain the following answers for the query Q:

1. $Sub_1 = \langle R_1/r_1, V_1/val(0), R_2/r_1, V_2/val(1), R_3/r_2, V_3/val(1), R_4/r_3, V_4/val(0,0)\rangle$.
   This answer corresponds to the following ground program $Subset_1$:

$$p(0) \leftarrow e(0), not\, q(0)$$
$$p(1) \leftarrow e(1), not\, q(1)$$
$$q(1) \leftarrow e(1), not\, p(1)$$
$$r \leftarrow e(0), p(0), e(0), p(0)$$

2. $Sub_2 = \langle R_1/r_1, V_1/val(0), R_2/r_1, V_2/val(1), R_3/r_2, V_3/val(1), R_4/r_3, V_4/val(1,0)\rangle$.
   This answer corresponds to the following ground program $Subset_2$:

$$p(0) \leftarrow e(0), not\, q(0)$$
$$p(1) \leftarrow e(1), not\, q(1)$$
$$q(1) \leftarrow e(1), not\, p(1)$$
$$r \leftarrow e(1), p(1), e(0), p(0)$$

3. $Sub_3 = \langle R_1/r_1, V_1/val(0), R_2/r_1, V_2/val(1), R_3/r_2, V_3/val(1), R_4/r_3, V_4/val(0,1)\rangle$.
   This answer corresponds to the following ground program $Subset_3$:

$$p(0) \leftarrow e(0), not\, q(0)$$
$$p(1) \leftarrow e(1), not\, q(1)$$
$$q(1) \leftarrow e(1), not\, p(1)$$
$$r \leftarrow e(0), p(0), e(1), p(1)$$

4. $Sub_4 = \langle R_1/r_1, V_1/val(0), R_2/r_1, V_2/val(1), R_3/r_2, V_3/val(1), R_4/r_3, V_4/val(1,1)\rangle$.
   This answer corresponds to the following ground program $Subset_4$:

$$p(0) \leftarrow e(0), not\, q(0)$$
$$p(1) \leftarrow e(1), not\, q(1)$$
$$q(1) \leftarrow e(1), not\, p(1)$$
$$r \leftarrow e(1), p(1), e(1), p(1)$$

It can be seen that the programs $Subset_1, Subset_2, Subset_3$ and $Subset_4$ are exactly the *minimal subset guesses* of $IDB \cup EDB$ w.r.t PT.

SubsetGen2 works in a manner similar to SubsetGen1. However, since we want only *supporting* rules w.r.t a set of facts S, we need to filter out the rules which contain any atom $a \in S$ in the negative body literals. This amounts to putting the negative body literals in the original rule as a NAF literal in the Prolog rule. Specifically, let $r \in IDB$ be a rule of the form

$$h \leftarrow B^+(r), B^-(r)$$

then we define $\pi_2(r)$ as the following Prolog rule:

$$r = gr(h, r\_id, val(\overline{X})) \leftarrow B^+(r), B^-(r)$$

and we define $IDB_{\pi_2} = \{\pi_2(r) \mid r \in IDB\}$. The procedure to generate the *supporting minimal subset guesses* of $IDB \cup EDB$ w.r.t to an interpretation S is performed similarly

---
Algorithm 4 Generating program subsets in SubsetGen2

Input: IDB a set of normal rules, EDB a set of facts, S an interpretation
Output: A set of ground rules
 1: Construct program IDB_{π_2} from IDB
 2: PrologEngine($IDB_{\pi_2} \cup EDB$).addFacts(S)
 3: $Q \leftarrow$ createQuery(S)
 4: PrologEngine($IDB_{\pi_2} \cup EDB$).doQuery(Q)
 5: **while** PrologEngine($IDB_{\pi_2} \cup EDB$).answersLeft = **true do**
 6: $\quad$ $Sub \leftarrow$ PrologEngine($IDB_{\pi_2} \cup EDB$).getNextAnswer
 7: $\quad$ $CurrentSubset \leftarrow$ Instantiate(IDB, Sub)
 8: $\quad$ Output $CurrentSubset$
 9: **end while**
10: PrologEngine($IDB_{\pi_2} \cup EDB$).delFacts(S)

---

as in SubsetGen1. However, since we need to generate these subsets w.r.t to several different interpretations, we need to delete the previous interpretation S from the database before starting to generate the subsets for the next interpretation S. Algorithm 4 details the steps performed by SubsetGen2.

As a way of interfacing with the components SubsetGen1 and SubsetGen2, each component is equipped with two functions: getNextSubset and subsetsLeft. The first returns the next subset generated while the second returns the value **true** of **false**, depending on whether any subset is left to be returned or not.

5.1.7 Answer Set Verifier

During the steps of Method 2, we perform a verification to determine whether or not an interpretation is an answer set of a set of ground normal rules. Let A be the interpretation an R be a set of ground normal rules. To verify that A is an answer set of R, we construct the program R'_A as described in Section 4.1.2 and then check whether R'_A is consistent.

We denote the component to perform this functionality as ASVerifier. It receives as inputs a set of ground normal rules R and an interpretation A and has a function called VerifyAS. This function takes no input parameter and return the value **true** or **false** depending on whether A is an answer set of R or not. This component utilizes the component ASPSolver to check consistency of R'_A. The mechanism is summarized in Algorithm 5.

Algorithm 5 Verifying an answer set using ASVerifier

Input: R a set of normal ground rules, A an interpretation
Output: value **true** or **false**
 1: Construct program R'_A from R and A
 2: **if** ASPSolver($R'_A, \emptyset$).answerSetsLeft = **true then**
 3: $\quad$ **return true**
 4: **else**
 5: $\quad$ **return false**
 6: **end if**

5.1.8 Minimality Checker

In evaluating a non-HCF disjunctive program, one of the step required is to check the minimality of a model. Given, an input program P and a model I of P, we need to

determine that I is a minimal model of P, or equivalently, that there exist no proper subset $I' \subset I$ such that $I' \models P^I$. Following along the steps outlined in Section 4.2.2, the check is performed by enumerating the proper subsets of $I' \subset I$, and constructing the program $P_{I'}^{min}$. If the program $P_{I'}^{min} \cup I$ is consistent, then $I' \models P^I$, and we conclude that I is not a minimal model of P.

As an optimization step, we can skip the subsets of I which does not contain the facts of P, since any model of P must contain this set of facts. Furthermore, we can extend the set of facts using the definite rules of P, since any atoms derived from the facts and the definite rules must also be in any model of P. To put this idea formally: let $Def(P)$ denotes the definite rules of P, $EDB(P)$ be the set of facts in P and D be the (single) answer set of $Def(P) \cup EDB(P)$. If I is a model of P, then I is a minimal model of P iff no proper subsets of $I' \subset I$ which satisfies $D \subseteq I'$, satisfy $I' \models P^I$. In other words, we need only to check those subsets of I, I' such that $D \subseteq I'$.

Let us define the component MinChecker that performs the checking for model minimality. This component receives as input a program P and a model I of P. The framework component shall provide a function called checkMin() which returns the value **true** or **false**, depending on whether or not I is a minimal model of P. We propose two approaches on how minimality checking is performed under the architecture framework. The first approach uses Prolog, while the second one uses an ASP solver.

Minimality Checking with Prolog

Assume that we have a model I of a program P. We first compute the answer set of $EDB(P) \cup Def(P)$, call this D. To check minimality for P, first we represent the following sets as sorted Prolog lists:

- the set I, represented by the Prolog fact $model(L_I)$, where L_I is a sorted Prolog list corresponding to I.

- the set D, represented by the Prolog fact $definite(L_D)$, where L_D similarly contains D in a sorted Prolog list.

To allow classically negated atoms in Prolog, we simply rename an atom $\neg a$ into neg_a, provided that neg_a is a new symbol unused in the input program.

We first define the Prolog clauses to generate proper subsets of a set (represented in a sorted list) as follows:

$$gen\_sub([\ \ ],[\ \ ]).$$
$$gen\_sub([H|T],[H|M]) :- gen\_sub(T,M).$$
$$gen\_sub([\_|T],S) :- gen\_sub(T,S).$$
$$gen\_propsub([H|T],[H|M]) :- gen\_propsub(T,M).$$
$$gen\_propsub([\_|T],S) :- gen\_sub(T,S).$$

Assuming the availability of the commonly used Prolog predicate $append/3$[2], we construct the clauses for generating the proper subsets $I' \subset I$ which satisfies $D \subseteq I'$:

$$model\_subset(I,IP) :- model(I), definite(D), append(D,S,I),$$
$$gen\_propsub(S,SP), append(D,SP,IP).$$

---

[2]It is assumed here that $append(A,B,C)$ means that appending A to B results in C.

To emulate the constraints in the program $\mathcal{P}^{min}(I)$ introduced on Section 4.2, we define the set of prolog rules V as follows: $V = \{violated \leftarrow \ B(c) | c = cons(shift(\pi(r))), \forall r \in P\}$. For example: if $P = \{a \vee b \leftarrow \ c, d \leftarrow \ not\, e, c\}$ then V is the following Prolog program:

$$violated: - c, not\, a, not\, b.$$
$$violated: - not\, e', not\, d.$$
$$violated: - not\, c.$$

To facilitate assertion and retraction on each atom member of a list L, we assume that we have defined the predicates $assert\_all/1$ and $retract\_all/1$. For a subset IP of I represented as a Prolog list, we define the predicate $satisfy/1$ as follows:

$$satisfy(IP): - assert\_all(IP), not\, violated,$$
$$retract\_all(IP).$$

Note that retracting the atoms in IP is required since we may perform another checking afterwards with a different IP. Finally, we define the minimality property of a model I using the following Prolog clauses:

$$notmin(I): - model\_subset(I, S), satisfy(S).$$
$$min(I): - model(I), not\, notmin(I).$$

Denote by $P_{PL}^{min}(I)$, the Prolog program obtained by putting all the clauses above together. Minimality checking for I can then be performed by first asserting the facts $\pi(L_I)$, and then performing the query $min(L_I)$ with the program $P_{PL}^{min}(L_I)$ on the Prolog system.

We formalize the procedures of minimality checking using Prolog in the context of the framework architecture we have defined so far. The framework component implementing minimality checking using Prolog will be called PrologMinChecker. Algorithm 6 shows how PrologMinChecker performs its functionality.

Algorithm 6 Checking for minimality using PrologMinChecker

Input: Program I and model I of P
Output: value **true** or **false**
  1: Construct Prolog program $P_{PL}^{min}(I)$ as described above
  2: PrologEngine($P_{PL}^{min}(I)$).addFacts($\pi(I)$)
  3: **if** PrologEngine($P_{PL}^{min}(I)$).checkQuery($min(I)$) = **false then**
  4:     $minimal = false$
  5: **end if**
  6: Output $minimal$

Minimality Checking with ASP Solver

Checking minimality can also be done using an ASP solver. The main idea works similarly as in the methods used to generate models, where subsets of an interpretation are generated and checked for satisfaction. In this case, we are generating (proper) subsets of a model I and checking whether one of the subsets satisfies the GL-reduct of the program with respect to I.

Let P be the input program, I the model which we will check for minimality and D be the answer set of $EBD(P) \cup Def(P)$. Generating the proper subsets $I' \subset I$ which satisfies $D \subseteq I$ can be performed using the following program

$$P_g(I) = \{a \vee \neg a \mid a \in I \setminus D\} \cup D \cup c_I$$

where c_I is the following constraint:

$$\leftarrow a_1, \ldots, a_k$$

if $I = \{a_1 \ldots, a_k\}$. The constraint c_I is used just to make sure that we do not accept I itself to be considered as a proper subset. If A is an answer set of $P_g(I)$, then $I' = A \cap I$ is a proper subset of I satisfying $D \subseteq I'$. Conversely, if I' satisfies $I' \subset I$ and $D \subseteq I'$, then it is easy to see that there must be an answer set A of $P_g(I)$ such that $I' = A \cap I$.

We recall the definition of $P^{min}(I)$ for a program P and a model I described in Section 4.2.2, and the result obtained in Proposition 4.16: that minimality of I in P can be proved by showing that there is no proper subset $I' \subset I$ for which $P^{min}(I) \cup I'$ is consistent. Using program P_g to generate the subsets I' of I, we may try to check minimality of I by checking whether $P_g(I) \cup P^{min}(I)$ is consistent or not. Unfortunately, evaluating this program using the current ASP solvers directly may cause exponential space grounding, since $P_g(I)$ contains disjunctions and the rules in $P^{min}(I)$ may have unbounded number of variables.

We use a technique similar as in the procedure of generating models: split the program $P^{min}(I)$ into two parts, the small part and the big part. The small part can be evaluated along with $P_g(I)$, and the resulting answer sets can then be checked against the big part. Using the functions $Small$ and Big as defined in Section 4.2.1, we describe minimality checking using ASP solvers as follows:

- Construct program $P_g(I)$.

- Let $P_1 = P_g(I) \cup Small(P)^{min}(I)$.

- Let $P_2 = Big(P)^{min}(I)$.

- Find an answer set I' of P_1 such that $P_2 \cup I'$ is consistent. If such I' exists, I is not minimal. Otherwise, I is minimal.

By the definition of the function $Small$, P_1 will have polynomially many ground rules, and hence evaluating P_1 using an ASP solver does not cause exponential space grounding. Program P_2 contains only constraints. Hence, the program $P_2 \cup I'$ is stratified and evaluating it using an ASP solver will not cause exponential space grounding as well.

We summarized the procedure described here by defining a framework component which implements minimality checking using ASP solvers using the steps explained above. We call this framework components DisjunctiveModelChecker. Algorithm 7 shows how this framework component is implemented in the context of the evaluation framework.

5.2 Evaluation Components

We describe three evaluation components that implement the three evaluation methods which have been described in the previous chapter. Each of them will have a common set of inputs and a common set of functions to interface with them, but each has different

---

Algorithm 7 Minimality checking using external ASP solver

Input: IDB a set of rules, EDB a set of facts, model I of $IDB \cup EDB$
Output: true or **false** indicating whether I is a minimal model
1: $isMinimal \leftarrow$ **true**
2: $D = $ ASPSolver$(Def(IDB), EDB)$.getNextAnswerSet
3: $P_g(I) \leftarrow \{a \vee \neg a \mid a \in I \setminus D\} \cup D \cup c_I$
4: $P_1 \leftarrow P_g(I) \cup Small(P)^{min}(I)$
5: **while** ASPSolver$(P_1, \emptyset)$.answerSetsLeft $=$ **true do**
6:      $I_p \leftarrow$ ASPSolver$(P_1, \emptyset)$.getNextAnswerSet
7:      **if** ASPSolver$(Big(P)^{min}(I), I_p)$.answerSetsLeft **then**
8:          $isMinimal \leftarrow$ **false**
9:          **break**
10:      **end if**
11: **end while**
12: *Output isMinimal*

---

algorithms to implement their evaluation procedures based on which evaluation method it corresponds to. The common inputs are IDB, a set of rules and EDB a set of facts, while the set of common interface functions are called getNextAnswerSet and answerSetsLeft. The function getNextAnswerSet returns the next answer set computed by the component from $IDB \cup EDB$ while answerSetsLeft signals whether the component still has more answer sets to be returned. We describe these components in details in the following sections.

5.2.1 Evaluation Component for Method 1

Following the steps in Section 4.1.1, we present a framework component called EvalMethod1 to perform evaluation using Method 1. The main steps performed by this evaluation component are: generate minimal subset guesses using the component **SubsetGen1**, compute the answer sets of the subsets using **ASPSolver**, and check that the answer sets satisfies the input program, using **ModelChecker**. This process is summarized in Algorithm 8.

Algorithm 8 Evaluating HCF programs using Method 1

Input: IDB a set of HCF rules, EDB a set of facts
Output: answer sets of $IDB \cup EDB$
1: $IDB_s \leftarrow$ Shift(IDB)
2: $IDB_s^+ \leftarrow$ Positive(IDB_s)
3: $PT \leftarrow$ ASPSolver(IDB_s^+, EDB).getNextAnswerSet
4: **while** SubsetGen1(IDB, EDB, PT).subsetsLeft $=$ **true do**
5:      $CurrentSubset \leftarrow$ SubsetGen1(IDB, EDB, PT).getNextSubset
6:      **while** ASPSolver$(CurrentSubset, EDB)$.answerSetsLeft $=$ **true do**
7:          $CandidateAS \leftarrow$ ASPSolver$(CurrentSubset, IDB)$.getNextAnswerSet
8:          **if** ModelChecker(IDB).checkModel$(CandidateAS) = $ **true then**
9:              Output $CandidateAS$
10:          **end if**
11:      **end while**
12: **end while**

5.2.2 Evaluation Component for Method 2

Method 2 for answer set evaluation starts in manner similar to with Method 1: we compute the set of possibly true atoms PT from IDB and EDB. Then, we enumerate the possible subsets S of PT such that $EDB \subseteq PT$. For each such subset, we find a *supporting minimal subset guess R*, and we verify that S is indeed an answer set of R. If S is an answer set of R, the last step to be performed is to check whether S satisfies IDB. However, we could also modify these steps such that this satisfaction checking is performed *before* we generate the *supporting minimal subset guesses* for S. This would allow us to filter out non-models from the subsets of S and avoid the unnecessary step of generating *supporting minimal subset guesses* for these non-models.

Let us define a framework component to realize this evaluation method. We call this framework component EvalMethod2. EvalMethod2 uses the previously defined components to perform its functionalities. The framework component ModelGen generates the models of $IDB \cup EDB$ from PT. For each such model, we find a *supporting minimal subset guess* such that the model is an answer set of the subset. This can be performed using the components SubsetGen2 and ASVerifier defined previously. If we succeed in finding such subset, then the model is indeed an answer set of $IDB \cup EDB$, and we return it as an output. This process is summarized in Algorithm 9.

Algorithm 9 Evaluating HCF programs using Method 2

Input: IDB a set of HCF rules, EDB a set of facts
Output: answer sets of $IDB \cup EDB$
 1: **while** ModelGen(IDB, EDB).modelsLeft = **true do**
 2:     $I \leftarrow$ ModelGen(IDB, EDB).getNextModel
 3:     $found \leftarrow$ **false**
 4:     **while** $found = $ **false** and SubsetGen2(IDB, EDB, I).subsetsLeft = **true do**
 5:         $CurrentSubset \leftarrow$ SubsetGen2(IDB, EDB, I).getNextSubset
 6:         **if** ASVerifier($CurrentSubset, I$).verifyAS = **true then**
 7:             $found \leftarrow$ **true**
 8:         **end if**
 9:     **end while**
10:     **if** $found = $ true **then**
11:         Output I
12:     **end if**
13: **end while**

5.2.3 Evaluation Component for the Disjunctive Method

For non-HCF disjunctive programs, we define the evaluation component of the framework which performs the evaluation of the programs using the method outline in Section 4.2. The method starts by generating models of the input program, and then check each model for minimality. The first step of this process can be performed using the component DisjunctiveModelGen already defined previously. The second step is performed using the minimality checker component MinChecker.

We denote the framework component implementing the evaluation of disjunctive programs EvalDisjunctive. Algorithm 10 shows the processing performed by EvalDisjunctive.

Algorithm 10 Evaluating a non-HCF program

Input: IDB a set of disjunctive rules, EDB a set of facts
Output: answer sets of $IDB \cup EDB$
 1: **while** DisjunctiveModelGen($IDB \cup EDB$).modelsLeft = **true do**
 2:     $I \leftarrow$ DisjunctiveModelGen(IDB, EDB).getNextModel
 3:     **if** MinChecker($IDB \cup EDB$, I).checkMin = **true then**
 4:         Output I
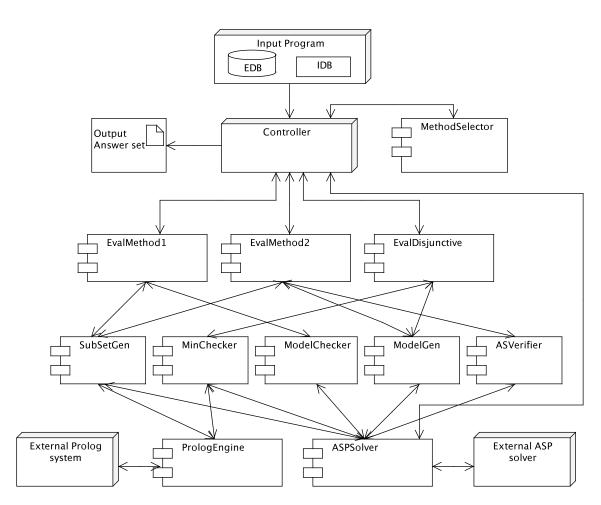 5:   **end if**
 6: **end while**

Figure 5.1: Overall view of the architecture

5.2.4 Overall view of the architecture

Figure 5.1 provides an overall view on the architecture described in the previous sections and shows how each framework component interact with each other. The Controller component represents the main control flow of the architecture and is in charged of accepting input, performs preprocessing (parsing, safety checking etc.), invoking other components to perform their tasks, and finally outputting the results. The controller invokes the MethodSelector component (described below in Section 5.3.1) to determine the appropriate method to use, and calls the selected method. Each evaluation method component performs its task of computing answer sets and returns the results back to the controller.

Both EvalMethod1 and EvalMethod2 employs program subset generator component SubGen for generating ground program subsets. EvalMethod1 uses ModelChecker for checking whether the local answer sets computed is a model of the input program. EvalMethod obtains candidate answer set from ModelGen, the model generator. EvalMethod2 also uses ASVerifier to verify that the model obtained from ModelGen is an answer set of the ground program subset generated by SubsetGen. EvalDisjunctive also uses ModelGen to obtain candidate answer sets. It then invokes MinChecker to test minimality of the candidate.

Both SubsetGen and MinChecker use the component PrologEngine to perform their tasks, while ModelChecker, ModelGen and ASVerifier use ASPSolver. Finally, PrologEngine, resp. ASPSolver perform the actual communication to the external Prolog, resp. ASP solver system.

5.3 Program Modularity Analysis

In this section, we aim to provide an application of the methods in modularity analysis of logic programs in the context of the evaluation framework of logic programs with bounded predicate arities. Specifically, we intend to use the split program and SCC analysis to build a more efficient evaluation framework for logic programs with bounded predicate arities. One of the advantages of applying modularity analysis in evaluating a program is that it allows us to confine the evaluation to a subset of the program (program component). This certainly will provide an improvement on the efficiency of the evaluation, because program components naturally have a smaller size than the whole program. In fact, for certain programs, applying SCC analysis and splitting programs into program components alone will allow us to avoid exponential space, which will otherwise be present when the evaluation of the program is performed using the classical approach of computing the full ground program in the first step.

Example 5.5. Take as an example, program P_1 as follows:

$$\begin{aligned}
p(X) \vee q(X) \leftarrow &\ d(X) \\
r \leftarrow &\ e(X_1), p(X_1), \ldots, e(X_k), p(X_k) \\
d(0).d(1). &
\end{aligned}$$

Evaluating P_1 using the classical approach will cause exponential space. However, using SCC analysis, we can split the program into two program components: one component consists of the first rule, and another containing the second rule. Evaluating the first program component, we get 4 answer sets. Each of this answer set is then fed to the second component, which basically is just a Horn rule; evaluating it should not need exponential space.

In the context of evaluating programs with bounded predicate arities using the three methods we described, we can reason about the advantages of applying modularity analysis in the evaluation framework on each methods. One of the possible sources of the significant portion of the time consumptions during the evaluation using Method 1 is in the enumeration of the ground program subset. By applying modularity analysis, we may get smaller number of ground program subsets to be considered in each program component, thus reducing the overall evaluation time. In the second and third method, time consumption mainly depends on the number of the guesses that have to be made to obtain the models. In general, the number of guesses is exponential in the size of an interpretation. By applying modularity analysis, we can confine the evaluation on each program component, thus reducing the number of guesses that have to be made.

5.3.1 Evaluating a program component

Before we begin with the description of the strategy of evaluating the program components, let us focus on the task of evaluating one program component, since evaluating a program component can be done independently from the other program components. One question that might come up is, which evaluation method is more appropriate to use for a particular program component.

First, we observe that the property of being HCF/non-HCF for a program component is preserved by the process of splitting a program during the SCC analysis. This can be understood easily by recalling that the property of HCF/non-HCF is determined by the existence of a cycle in the positive dependencies between atoms appearing in the head of the program. Since splitting program into SCCs retains all the cycles of dependencies (including cycles of positive dependencies), the property of being HCF/non-HCF is preserved, and in fact confined to each program component. This allows us to test for HCF/non-HCF condition in each program component, and use the appropriate evaluation method accordingly.

Another possibility that we may consider is that the program component might actually be stratified or having only polynomially many ground rules. In this case, it is not necessary to use the evaluation methods that we have described for programs with bounded arities. Since we are making the assumption that an external ASP solver is available that can perform the evaluation for stratified programs or programs with polynomially many ground rules under polynomial space efficiently, we can just use this external ASP solver to evaluate such program components directly.

To go further along this line of reasoning, for program components which are normal/HC but having exponentially many rules, we might consider to have a heuristic to select between using Method 1 or Method 2, based on a certain criteria. Such criteria might be: the type of the evaluation (generating all answer sets vs checking consistency), some syntactical properties of the program component, or some other criteria which might be relevant in the decision of using Method 1 or Method 2.

Ideally, one would want to have a certain mechanism that allows choosing the appropriate methods according to a certain preference. However, as a guideline, the following principles may be used by such heuristic:

(i) If the program component is stratified or if it has polynomially many ground rules, then it is preferable to evaluate it directly using the external ASP solver.

(ii) If the program component is non-HCF, then there is no choice other than using the evaluation methods for non-HCF programs.

(iii) If the program component is HCF and has a small set of possibly true atoms, then using Method 2 might be preferable, since evaluation using Method 2 does not cause an answer set to be recomputed several times.

(iv) On the other hand, if the set of possibly true atoms is big, then using Method 1 might be beneficial, since using it allows us to avoid having to enumerate a great number of subsets.

(v) If the goal of the computation is only to check for consistency, where finding *all* answer sets is not required, than using Method 1 might be preferable, because it yields a better efficiency in this case.

To continue on our formalization of the framework, let us define a framework component called MethodSelector that performs the task of selecting the appropriate or preferred method of evaluation on a certain program component. This framework component takes as input a program P and returns the selected method for evaluating component. Let us denote the function to use to return the method from MethodSelector, getMethod. The output of the function can either be *Direct* (which signifies that the selected method is a direct evaluation by the external ASP solver), $Method1$, $Method2$ or $Disjunctive$.

We can now describe a framework component that perform the evaluation of one program component. Let us call this framework component EvalComponent. It receives as inputs: a set of rules IDB and a set of facts EDB in a program component and returns the answer sets of the $IDB \cup EDB$. Similarly as the previous framework components, the functionality of EvalComponent is represented by two functions: answerSetsLeft and getNextAnswerSet. Algorithm 11 gives a formalization of this framework component. Basically, this component just queries the MethodSelector component to obtain the preferred method to use, and then perform the evaluation using the appropriate evaluation component.

Algorithm 11 Evaluating a program component using EvalComponent

---

Input: IDB a set of disjunctive rules, EDB a set of facts
Output: answer sets of $IDB \cup EDB$
1: $SelectedMethod \leftarrow$ MethodSelector$(IDB \cup EDB)$.getMethod
2: **if** $SelectedMethod = Direct$ **then**
3:     **while** ASPSolver(IDB, EDB).answerSetsLeft = **true do**
4:         $CurrentAS \leftarrow$ ASPSolver(IDB, EDB).getNextAnswerSet
5:         Output $CurrentAS$
6:     **end while**
7: **else**
8:     Let EvalM be the framework evaluation component corresponding to $SelectedMethod$
9:     **while** EvalM(IDB, EDB).answerSetsLeft = **true do**
10:        $CurrentAS \leftarrow$ EvalM(IDB, EDB).getNextAnswerSet
11:        Output $CurrentAS$
12:    **end while**
13: **end if**

5.3.2 Evaluation strategy

Given a decomposition of a program into program components and the dependency graph between them, a simple approach that one might consider when evaluating the program is the following:

(i) We start by evaluating the bottom components. Evaluation is done completely, meaning that we obtain all the answer sets of these bottom components, and store them for further use. These bottom components can then be discarded away.

(ii) Each stored answer set is then fed to the upper components, and evaluation on the program components proceeds according to the order of the program components in the dependency graph. All answer sets of each component is stored and the components that has been evaluated is discarded away.

(iii) When we reach the top components, evaluation is complete, and all answer sets has been stored. We just need to output each of these answer sets.

However, in the context of evaluating programs with bounded predicate arities, this simple approach is not appropriate to be used with respect to the goal of preserving polynomial space computation. One can easily see that the number of answer sets of a program may be exponential, even for programs with polynomially many ground rules. Thus, we need to devise an evaluation strategy which avoids storing all answer sets of each program component.

The proposed strategy to be used is to allow evaluation in each program component to retrieve one answer set at a time and feed this answer set to the upper components until we reach the top component. Intuitively, we can view this evaluation strategy as a process of streaming answer sets from the bottom components until the top components one answer set at a time. Once we finish the streaming process for one answer set, we will need to backtrack to the previous components in order to obtain the next answer sets. In order to perform such strategy, we need first to arrange the program component in a topological sort. Recall that, for a directed graph $G = \langle V, E \rangle$, a *topological sort* of G is a partial linear ordering $\prec$ of V such that for any $v, w \in V$, $v \prec w$ iff there is a directed path from v to w. Recall also that if G is acyclic, G always has at least one topological sort.

Note that the component dependency graph is always acyclic, hence, the existence of a topological sort for the program components is guaranteed. By sorting the program components in a topological sort, we ensure that when a program component is evaluated, all the program components on which the current program component depends will have already been evaluated.

Let us state this strategy in a more formal way. Suppose we have a topologically sorted program components: $C_1, \ldots, C_n$. We provide an array data structure of size n, called *GlobalAS*. Each *GlobalAS*[i] will store the currently obtained answer set for component C_i. We proceed by evaluating each C_i in order of the sort, obtaining the current answer set a_i for each C_i and storing it in *GlobalAS*[i]. When we reach C_n, we construct the union of all the answer sets stored in each *GlobalAS*[i], and output the result. We then backtrack to the last component which still has answer sets left, and proceed the evaluation in this manner until the first components has no more answer sets left.

Let us call a function by the name Evaluate that performs the process of evaluating program components using the above strategy in a recursive manner. This function takes as input the program components in a topological ordering and an index number that allows for recursive calls to Evaluate. The evaluation is started using the call to Evaluate with input index 1. Algorithm 12 shows how such function works, utilizing the previously defined framework component EvalComponent.

Algorithm 12 Evaluating program components

Input: $C[1 \ldots n]$ (program components), j (index)

Output: global answer sets of the components

1: $EDB_j \leftarrow \bigcup_{i \leq j} Global\, AS[i]$
2: **if** $j = n$ **then**
3:     **while** EvalComponent(C_j, EDB_j).answerSetsLeft **do**
4:         $AS \leftarrow$ EvalComponent(C_j, EDB_j).getNextAnswetSet
5:         Output AS
6:     **end while**
7: **else**
8:     **while** EvalComponent(C_j, EDB_j).answerSetsLeft **do**
9:         $AS \leftarrow$ EvalComponent(C_j, EDB_j).getNextAnswetSet
10:         $Global\, AS[j] \leftarrow AS$
11:         Evaluate$(C[1 \ldots n], j + 1)$
12:     **end while**
13: **end if**

6

Implementation and Experimental Results

To provide a concrete implementation of the methods and framework defined in the previous chapters, we have developed an application called BPA, for Bounded Predicate Arities. BPA receives as input of logic programs and computes the answer sets of the programs under answer set semantics using the three previously described evaluation methods. We also performed an experimental testing using several problem instances to obtain a concrete benchmark on how the methods compare against current ASP systems and to verify that the methods and framework we described indeed perform the evaluation within polynomial space. The experiment measures the performance of program evaluation using BPA against evaluations using DLV and GrinGo/claspD, both with respect to time and space consumptions.

   We will start this chapter by first describing the implementation details of BPA, including the choices of data structures, underlying algorithms and other technical choices made during its development. This description should be thorough enough for those who want to understand the internal workings of this system by studying the source code of the system. We will then describe the details of the benchmark experiment performed as well as provide a brief analysis of the results obtained from the experiment.

6.1 Architecture of BPA

As described in the previous chapter, the architectural framework we have designed for performing evaluation of logic programs requires the use of an external ASP solver as a back-end component, as well as a Prolog system to compute variable bindings during the process of generating subsets of the ground rules. For this particular implementation, we have selected DLV as the back-end ASP solver, due to its relatively efficient performance, the author's familiarity with it as well as for its ease of use (due to DLV having an integrated grounder and model generator/model checker). The Prolog system we chose is XSB [Sagonas et al., 1994], with the reason being that it has an easy to use, yet quite powerful Application Programing Interface (API) to simplify the interaction and communication between XSB and external applications.

   BPA is written using C++ and follows the principles of Object Oriented Programming (OOP) and software design patterns, to allow it to be flexible and easily extendable/modifiable. In particular, each framework component described in Chapter 5 is represented by an abstract class, where each function/procedure provided by the framework component is represented by a pure virtual method of the abstract class. The exact details

of how each function is performed is left to be defined in the subclasses, allowing for an easy way of modifying the details of the workings of each of the framework components. For example, the framework component for generating models, ModelGen, is represented by an abstract class `ModelGenerator` and the function for computing the models get-NextModel is represented by the pure virtual function `getNextModel()` defined in the class `ModelGenerator`. The exact details of how this function is implemented is defined in a subclass of `ModelGenerator` called `DisjunctiveModelGenerator`. If one needs to replace the algorithm for generating models defined in `DisjunctiveModelGenerator` with another algorithm, then one just needs to define a new class derived from `ModelGenerator`, implementing all the functionalities of a model generator as described in Section 5.1.4. After that, one only needs to replace the instantiations of `ModelGenerator` in the classes defining the framework evaluation components for Method 2 and the Disjunctive Method, with the instantiations for the newly defined subclass. The polymorphism principle of OOP should make it possible for the newly defined component to be used in the other parts of the system without changing their source codes.

To speed up development of the system, we have used quite heavily some of the freely available libraries for C++ programming language, including the Standard Template Library[1] (STL) and Boost[2] to perform lower level algorithms and utility functions, such as parsing, string manipulations, graph algorithms; and to provide basic data structures to build upon. To keep the description brief, we will omit the explanations for describing these libraries, and only focus on the relevant discussions with respect to the internal workings of BPA itself.

6.1.1 Parser and Data structures

BPA accepts the core of the syntax of the logic programs as defined in Section 2.2.1, with an extension of also allowing binary infix comparison predicates, such as: "=","<", " != " etc. Arithmetic operations have not been addressed, since it requires additional semantics not yet considered in this work. Furthermore, to focus on defining the core semantics of logic programs with bounded predicate arities, we have refrained from considering the extensions of syntax and semantics of logic programs defined by current ASP solvers, such as weak constraints and aggregates in DLV, and cardinality constraints defined in Lparse/GrinGo. In essence, the syntax accepted by BPA lies in the common subset of the language accepted by these ASP solvers.

Parsing in BPA is performed using the functionality provided by Boost Spirit, a library for performing scanning and parsing included in Boost. Spirit allows for an easy specification of the accepted syntax in a BNF-like C++ construct. For example, the following is part of the syntax specification defined in BPA:

```
constant = symbol|stringconst|integer;
anon = leaf_node_d[ch_p('_')];
variable =  leaf_node_d[upper_p >> *(alnum_p|ch_p('_'))];
term = constant | variable | anon;
```

which can be intuitively understood as follows: a constant is made of either a symbol, a string constant (constants such as: "Logic Program") or an integer. An anonymous variable is represented by an underscore "_", and a named variable by a string of alphanumeric characters beginning with an uppercase letter (possibly also containing the character "_"

---

[1] http://www.sgi.com/technology/stl/
[2] http://www.boost.org

int the middle/end). A term is then defined to be either a constant, a (named) variable or an anonymous variable. Using this construct, one can easily build syntax specification quite rapidly and easily, compared to using traditional parsers/lexical analyzers such as YACC/GNU Bison. However, it does come with a small performance overhead. But since parsing the input program is done only once, we can safely ignore this little performance overhead for most input programs.

During parsing, safety checking of the rules is also performed, following the safety condition defined in Definition 2.1. If the rules/facts pass all the syntax specifications and safety checking, they will be stored in the corresponding data structures to represent a set of rules and a set of facts. An additional step of standardization apart is performed for the rules containing variables: the variable names in the rule are renamed into "`VAR_0`", "`VAR_1`",...,"`VAR_N`" to simplify further processing.

This parsing functionality is implemented in the abstract class `ProgramParser` and its subclass `SpiritProgramParser`. The safety checking procedure is implemented in a class called `SafetyChecker`, which is one of the members of `ProgramParser`. If one needs to replace this parsing algorithm with another algorithm, one can define a new subclass of `ProgramParser` and replace the instantiation of `ProgramParser` in the main procedure of BPA with the newly defined class. Finally, the class `ProgramParser` provides the methods `getEDB()` and `getIDB()` to return the facts and the rules, respectively, of the input program (assuming they have been correctly parsed and are safe).

To represent the constructs occurring in a logic program, such as: terms, atoms and literals, we employ data structures which are modelled following those used in DLVHEX [Schindlauer, 2006], with some additional functions needed to implement the methods used in this work. Each basic data structure is implemented as a C++ class equipped with the methods needed to manipulate the data contained inside it. The underlying storage itself uses C++ STL containers, such as: `set`, `vector` and `map`. Furthermore, to simplify the low-level mechanisms, such as: memory allocations and pointer management, BPA uses a component of Boost Library called Shared Pointers, which helps avoid memory-related problems in the use of program data structures, e.g., memory leaks and faulty pointer arithmetic. The following described these basic data structures briefly.

Term

The class `Term` represents the concept of a term in the input program. A term can be a constant symbol (such as: `p`, `q`), an integer constant, a (quoted) string constant (e.g., `''Foo Bar''`) or a variable (indicated by an uppercase letter in the beginning). The class `Term` also defines utilities functions, such as: `bindVariable()` to substitute a variable with a constant to compute the ground instantiations of an atom, a literal or a rule. The arguments of an atom (including its predicate name) can then be represented as a vector of `Term`'s.

Atom

An atom in a program is represented using the class `Atom`. As previously mentioned, an atom can be modelled simply as a vector of terms. Thus, to store the data inside an atom, we can use a `vector` container storing previously defined `Term` objects. The class `Atom` itself acts as a wrapper of this vector of `Term` objects and provides methods needed for further manipulations, such as: binding the variables inside an atom using a set of variable substitutions, obtaining the predicate arity of the atom, and negating an atom. A `boolean`

member of the class is used to signify whether the atom occurs positively or negatively (in the classical sense).

A special type of `Atom`, called `InfixAtom` is used to represent the occurrences of infix binary comparison predicates between atoms, e.g.,: ">", "<=" and "!=". BPA does not, in itself, assign any semantic values for these comparison predicates, and leaves it to the external ASP solver to evaluate the semantics of the comparison predicates. In fact, they are treated equally as any other atoms, except for the fact that, during safety condition checking, it is also required that any variables occurring in a comparison predicate must also occur in the positive body literals of the rule. In other words, an atom consisting of a comparison predicate is treated similarly as an atom occurring in the head and the negative body literals, with regard to the safety condition.

Literal

A literal is simply an atom with an additional information of whether it occurs positively/negatively[3] in the body of a rule. The class `Literal` provides a representation of a literal in a program. It consists mainly of an `AtomSet` object and `boolean` member indicating whether the literal occurs positively or negatively.

AtomSet

The class `AtomSet` represents the concept of a set of atoms, which in the context of answer set semantics, encompasses constructs such as a set of facts, a model/interpretation and an answer set. In these cases, an `AtomSet` consists only of ground atoms. However, an `AtomSet` can also represent a set of non-ground atoms, such as the set of of atoms appearing in the head of a rule. The class is equipped with methods required to analyze and manipulate an atom set, such as: checking whether it is consistent, getting its size, finding its intersection with other `AtomSet`, etc.

Rule

The class `Rule` represents a rule in a program. The data structure composing a `Rule` object consists of an `AtomSet` object to represent the head literals, and a set of `Literal` objects to represent the body literals of the rule. Several methods are defined for the class to perform analysis of rule properties. For example, a method `isDefinite()` is used to indicate whether the rule represented by the `Rule` object is definite or not. Another function called `isSmall()` is used to determine whether the rule is considered to be *small* or *big*, relative to a predetermined parameter. The parameter used in BPA is the maximum number of ground rules which may be generated from the rules, and can be set/adjusted according to the user's input. These utility functions will make it easier to write the code implementing the methods and algorithms described in the previous chapters.

Program

Finally, to represent an input program, we define a class `Program`, which mainly consists of a set of `Rule` objects. To be exact, the class `Program` represents the set of rules (intensional database) appearing in a program, whereas the set of facts (extensional database) in the program is represented by an `AtomSet`. As the underlying storage for the rules, the STL's `set` container is used to store the pointers to the `Rule` objects. The use of a `set` container,

---

[3]In the sense of default negation.

coupled with a standardization-apart preprocessing during the parsing step, allows us to store efficiently a set of semantically equivalent rules, such as:

$$p(X, Y) \leftarrow q(X, Y), r(Y, Z)$$
$$p(A, B) \leftarrow q(A, B), r(B, \_)$$

as one rule, to avoid the unnecessary overhead.

The class `Program` defines, among others, the following methods needed for implementing the three evaluation methods described in Chapter 4:

- `doShift()`: performs the shifting operation on a program, as defined in Definition 2.5.

- `getCons()`: computes the result of the function $cons(P)$ for a program P as defined in Definition 4.4.

- `getPrimed()`: implements the function $\pi(P)$ for a program P as defined in Definition 4.7.

- `getDefinite()`: returns the definite rules of the program.

- `getSmall()` and `getBig()`: returns the *small* and *big* rules of the program, respectively, as described in Section 4.2.1

6.1.2 Framework Components

Following the framework architecture defined in Chapter 5, BPA defines a set of abstract and derived classes representing each of the framework components. These classes are briefly discussed in the following.

ASP Solver

The abstract class `ASPSolver` represents the framework component ASPSolver for communicating with an external ASP solver system. The functions provided by the framework component ASPSolver as specified in Section 5.1 are declared as virtual methods of the abstract class `ASPSolver`. For example, `getNextAnswerSet()` is used for returning the next answer set read from the ASP solver. The implementation of these methods are then defined in a derived subclass of ASPSolver specifying the details of how communication with the particular external ASP solver is performed.

BPA uses DLV as the external ASP Solver and the mechanism for communication is detailed in a subclass of ASPSolver called DLVASPSolver. Communication between BPA and DLV mainly consists of setting up a Unix style pipe data channel for interprocess communication, taking advantage of DLV's command line switch "`--`". Using pipe, it is possible to read answer sets from DLV in a streaming fashion, i.e., one answer set at a time. This is a necessary condition to stay in the polynomial space limit for the computation, since storing all answer sets from a program (or program component) at once may potentially require exponential space.

To use a different ASP solver in BPA, one needs to define a new subclass of ASPSolver implementing the details of how communication with the external answer set can be performed, including also: converting (serializing) the program objects into the right textual representation accepted by the ASP solver, as well as parsing the output produced by it. As soon as this is done, one can just replace the line instantiating the ASP solver in the main program to use the newly defined class, and all the other parts should work correctly. The polynomial space bound should still be maintained, as long as all the conditions specified in Section 5.1.1 are satisfied by the chosen ASP solver.

Prolog Engine

Similar to the ASP solver component, the framework component for interfacing with an external Prolog engine in BPA is defined in an abstract base class, called `PrologEngine`, with the particular implementation of the interface defined in a subclass of this class. BPA uses XSB as the Prolog engine to compute subsets of the ground rules of a program. The details of how to interface with XSB are specified in a subclass of `PrologEngine` called `XSBPrologEngine`.

In Section 5.3.2, we described an evaluation strategy of program components along the dependency graph. The evaluation strategy consists of a backtracking procedure for evaluation of each of the program components, with each program component possibly requiring the availability of a Prolog engine to compute the grounding rules. Since we certainly do not want the facts/rules from one program component to interfere with an active query to compute subsets of ground rules in another program component, there needs to be a separate instance of the Prolog engine for each program component.[4] This can be performed either by using a process/thread creation generally provided by the operating system to call each required Prolog instance (which may be cumbersome), or using the Prolog engine's built-in multithreading feature (if it is available). Fortunately, XSB supports multithreading, using the C language API functions called `xsb_ccall_thread_create()` and `xsb_kill_thread()` to create and destroy the threads, respectively. Each created thread is a separate full-fledged Prolog instance capable of storing its own databases and performing queries. The class `XSBPrologThread` details the implementation of how each thread perform common functions, such as: loading a program, asserting facts, sending a query, retrieving answers from a query and so on.

Program Subset Generator

The Program Subset Generator components defined in Section 5.1.6 is defined in the abstract class `ProgramSubsetGenerator` and implemented in a subclass of `ProgramSubsetGenerator` called `PrologProgramSubsetGenerator`. Evaluation using both Method 1 and Method 2 can use this class to compute subsets of the ground rules of a program. The only difference on how ground program subsets are generated in Method 1 and Method 2 (as defined in Definition 4.5 and Definition 4.6) is that ground program subsets in Method 2 are computed by considering also the negative body literals in the rules, to make sure that only the ground rules supporting the models are produced. The class `PrologProgramSubsetGenerator` is designed to handle both cases accordingly.

Model Generator

The framework component for generating models in BPA is defined in the abstract base class `ModelGenerator`. This class only provides the declaration of the functions needed in generating models, as defined in Section 5.1. The particular implementation of how to generate models in BPA follows the outline of the method defined in Section 4.2, and is given in a derived class called `DisjunctiveModelGenerator`.

As explained in Section 4.2, generating models is done by first computing the set of possibly true atoms PT, and proceeds by generating subsets of this set, using classical guessing rules $a \vee \neg a \leftarrow$ for every $a \in PT$, plus some constraints obtained from the *small* rules. This guessing program is submitted to the back-end ASP solvers, which will return

---

[4] To be more precise, only program components which are going to be evaluated using Method 1 and Method 2 require a Prolog engine.

back some candidate models. These candidate models are read one by one, and each one is checked against the *big* rules, to ensure that it is indeed a model of the program.

The criteria for including a rule as *small* or *big* is determined by the previously mentioned method of the class `Rule` called `isSmall()`, which in turn is determined by a parameter setting adjustable during the invocation of BPA. The exact mechanism can be briefly described as follows: suppose a rule has V many variables, and the program has C many constants. First, if V is less than a certain predetermined constant, V_{max}, then the rule is considered to be small. Otherwise, for a predetermined parameter N_{max}, if C^V is larger than N_{max}, then the rule is considered to be big, otherwise it is small. In BPA, the default values for V_{max} is 4, while N_{max} defaults to 10000. Both of them can be adjusted accordingly at run time. The smaller the values for V_{max} and N_{max} are, the smaller the space requirement for BPA will tend to be. However, computation might take longer, since there tend to be less rules to filter out the candidate models during the guessing step, and more candidate models (which turns out to be not a model) are produced.

Minimality Checker

Section 5.1.8 describes two approaches to minimality checking in the context of the evaluation framework defined in Chapter 5, one uses an external ASP solver, and the other uses a Prolog engine, to perform minimality checks. BPA provides both types of minimality checking procedures, defined in the classes `DisjunctiveMinChecker` and `PrologMinChecker`, respectively. These classes are defined as subclasses of the abstract base class `MinChecker`, which provides a common interface method `checkMin()` to perform minimality checking for both subclasses. The common interface makes it possible, if needed, to switch between the two types of minimality checkers without affecting other parts of the system.

After some early experimental testings, we found that the performance of minimality checking using Prolog as described in Algorithm 6 tend to be poor. For some input programs, it performs much worse than the minimality checking using an external ASP solver as defined in Algorithm 7. The reason for this is that the enumeration process in the algorithm does not permit any pruning of the subsets of the model that needs to be checked. If the model being checked is actually minimal, the algorithm will have to go through all (proper) subsets of the model first, before finally concluding that it is minimal[5]. Once the size of the model to be checked for minimality reaches a certain number, enumerating all these subsets is no longer feasible.

On the other hand, the approach for minimality checking using ASP solvers works better for some input programs, because it allows pruning of some of the subsets of the model by putting the small rules together with the rules used to generate the subsets. As a simple illustration, consider the following program, P:

$$a \lor b \leftarrow$$
$$b \leftarrow a$$
$$a \leftarrow b$$

and the model $I = \{a, b\}$. To check minimality of I, the approach suggested in Algorithm 6 will require Prolog to generate all the proper subsets of I: $\emptyset$, $\{a\}$ and $\{b\}$ and performing a query to check each subsets before concluding that each of them does not satisfy P^I. On

---

[5]More precisely, only subsets of the model containing the answer set of the definite part of the program are considered. However, this is a minor and irrelevant detail for this discussion.

the other hand, Algorithm 7 will construct the following program:

$$a \vee \neg a \leftarrow$$
$$b \vee \neg b \leftarrow$$
$$\leftarrow a, b$$
$$\leftarrow not\ a, not\ b$$
$$\leftarrow a, not\ b$$
$$\leftarrow b, not\ a$$
$$a'.$$
$$b'.$$

in an attempt to generate a proper subset of I satisfying P^I. However, the program above is inconsistent, and the algorithm concludes directly that I is a minimal model.

Due to this performance reason, we have selected Algorithm 7 as the default algorithm to use in BPA.

Evaluation Components

The three evaluation components described in Section 5.2.1, 5.2.2 and 5.2.3 are represented by the abstract base class `EvalComp`. This abstract class captures the concept of evaluation of a program (or program component) using a particular selected method. Each of the three methods themselves are then implemented as subclasses of `EvalComp`, called `EvalMethod1`, `EvalMethod2` and `EvalDisjunctive`. Another subclass, called `EvalDirect` is used to specify the evaluation procedure that simply feeds the program to the external ASP solver, which is more appropriate for stratified or small programs/program components.

A special function declared in `EvalComp` and implemented in each of its subclasses, called `doEval()`, is used to perform the actual evaluation step to obtain the answer sets of the program, one at a time. For example, `doEval()` in `EvalMethod2` consists of the steps for getting the next model of the program (utilizing a `ModelGenerator` object), computing a supporting minimal subset of the ground rules (using a `ProgramSubsetGenerator` object) and verifying that the current model is an answer set of the generated ground rules. If it is, then it will be stored in an internal representation of the class, ready to be returned through the invocation of the method `getNextAnswerSet()`. Otherwise, `doEval()` will keep on searching for an answer set, until there are no more models obtained from `ModelGenerator`.

A program code that needs to perform evaluation of a program using a certain method simply instantiates the object of the appropriate method that it requires, and use the methods `answerSetsLeft()` and `getNextAnswerSet()` on the object to loop through all answer sets. The following code illustrates this:

```
EvalComp* eval;
eval = new EvalMethod1(/*input list*/) /* Or method 2, 3, direct */
while(eval->answerSetsLeft())
{
        AtomSet as = eval->getNextAnswerSet();
        /* Do some stuff with as */
        .....
}
```

The base class for the evaluation component, `EvalComp` is abstract enough such that one can extend the system, including introducing a new evaluation method by defining a

subclass of `EvalComp`, following the conventions used in how the functions are named in the class. The newly defined evaluation class can then be plugged into the main program and called by program codes in other parts of the system.

6.1.3 Dependency Information and SCC Evaluation

Dependency graph representation and SCC computation in BPA are modelled after the one of DLVHEX. To store dependency information between atoms/literals in the program and to find program SCCs, the following data structures are defined in BPA:

Dependency

The class `Dependency` stores a dependency information between two different atoms appearing in a program. Dependency information is taken at the non-ground level, and considers also dependency between two unifying atoms. Four types of dependency between atoms are considered:

- Positive head-body dependency, for example: in $p(X) \leftarrow q(X)$, $p(X)$ depends on $q(X)$ positively.

- Negative head-body dependency, for example: in $q(X) \leftarrow d(X), not\ r(X)$, $q(X)$ depends on $r(X)$ negatively.

- Disjunctive dependency, for example: in $a \vee b \leftarrow$, a and b depend on each other through disjunctive dependencies.

- Unifying dependency, for example: in the following program

$$p(X,Y) \leftarrow e(X,Y)$$
$$r(X,Y) \leftarrow p(X,Z), e(Z,Y)$$

  we have that $p(X,Y)$ and $p(X,Z)$ depend on each other through unifying dependencies.

In each `Dependency` object, a `Rule` object for which the dependency comes from is associated and stored.

AtomNode

The class `AtomNode` represents the concept of one node in the dependency graph of the program. It stores the `Atom` object plus some additional information, including: the set of dependencies the atom has, as well as the set of rules for which the atoms are associated with. The function `getRules()` is used to retrieve these rules.

GraphBuilder

The task of building the dependency graph from the input program itself is performed by the class `GraphBuilder`. The class does not actually has any data structure/storage purposes, and its sole use is to build the `AtomNode` objects and `Dependency` objects associated with them, according to the input program.

ComponentFinder

The class `ComponentFinder` defines an abstract class for performing graph analysis to find the strongly connected components of the program. The strategy BPA uses is to utilize the functions provided by Boost Graph Library (BGL) for graph algorithms. First, the dependency information contained in the `AtomNode` and `Dependency` objects are used to create a copy of the dependency graph in BGL format. We can then use BGL's function `string_components()` to find the SCCs of the graph. The details of this process are implemented in a subclass of `ComponentFinder` called `BoostComponentFinder`.

ProgramComponent

The class `ProgramComponents` represents a program component as defined in Definition 2.6. It stores a set of `AtomNode` objects associated with it, a set of other `ProgramComponents` for which it depends on, and another set of `ProgramComponents` which depend on it. These stored data allow for an easy navigation between `ProgramComponents` through their dependency relation. A function called `getBottom()` returns a set of rules (in the form of `Program` object) which is associated with the program component, by calling each `AtomNode`'s `getRules()` function.

DependencyGraph

The class `DependencyGraph` is responsible for invoking the `ComponentFinder` to find SCCs, and stores the resulting `ProgramComponent` objects and the dependency between them. A `DependencyGraph` object created from an input program P is identical to the definition of program component dependency graph defined in Definition 2.7. All the information needed to perform evaluation along the SCCs are contained in `DependencyGraph` and once it is set up, we are ready to perform the evaluation as described in Section 5.3.2. A final task performed by `DependencyGraph` is to perform the topological sorting of the program components. This is done in the member function `getComponents()`. The function returns a vector of `ProgramComponents` already in a topological sort. The sorting procedure is also provided by the Boost Graph Library, and we can simply use the function called `topological_sort()` in BGL to obtain the sorted components.

HCFDetector

Head-cycle detection is performed using the dependency information obtained in the previous steps. In this case, negative dependencies do not play any role, and are ignored. Since partitioning the dependency graph into SCCs will not break any head-cycles occurring in the program, detecting head-cycle can be done at each program component, after SCC analysis is completed. Head-cycle detection then proceeds similarly as the process of finding SCCs, since a cycle is also an SCC. Boost Graph Library is again employed to find these cycles. This results in a set of cycles involving `AtomNode` objects. At each cycle, a step is performed that will check whether any two members of the cycle depend on each other through disjunctive dependencies. If no such pair exists, then the program component is declared to be HCF. Otherwise, it is non-HCF. The class `HCFDetector` and `BoostHCFDetector` define the implementation of this head-cycle detection process.

MethodSelector

As explained in Section 5.3.1, a mechanism is needed to make a decision of which method will be used to evaluate each program component. At the very least, the mechanism should decide that a non-HCF program should be evaluated using Disjunctive Method, while a HCF-program can be evaluated with any of Method 1 or Method 2, according to a certain predetermined criteria. In fact, it may also be the case that a normal/HCF program will be evaluated more efficiently using the Disjunctive Method. Another simple optimization efforts would be to detect whether the particular program component can already be evaluated efficiently using the external ASP solver, without using any of three methods described, thereby reducing any unnecessary overhead.

   At this point, we do not specify in detail, how this decision should be made. In BPA, an abstract class called `MethodSelector` is used to abstract away this decision process. Any procedure implementing this decision process can then be specified in a subclass of this class. In BPA, a simple method selection procedure is included, specified in the class `SimpleMethodSelector`. There is certainly no claim that the method selection procedure performed by this class is the optimal one, since its purpose is merely to illustrate how one can write a method selection procedure in the context of the architecture of BPA, and to provide BPA with a default decision of which method is to be used, in case the user does not specify any preference.

GraphProcessor

The final piece in the architecture of BPA is the `GraphProcessor` class, which takes all the information built from the previous data structures and performs the evaluation of the input program according to the strategy laid out in Section 5.3.2. `GraphProcessor` starts by obtaining the program components which are already topologically sorted from a `DependencyGraph` object. It starts the computation (detailed in the function `run()`) by receiving the facts of the input program as the initial input of the evaluation procedure. It then follows the algorithm described in 5.3.2, using a recursive function named `eval()` which implements the backtracking procedure of Algorithm 12. This function is also responsible for printing out the answer sets, as well as controlling the execution and termination of the overall evaluation. For example, it keeps track the number of answer sets printed thus far, so that if a limit on the number of answer sets printed is supplied in the beginning, execution can be stopped as soon as this limit is reached.

   An overview of the relationships of the data structures and architecture components in BPA, showing the flow of information between components, is shown in Figure 6.1.

6.1.4 Using BPA

BPA is written using the standard features of the C++ language, and uses only portable libraries such as STL and Boost. Moreover, both DLV and XSB can be installed and run in similar ways across different computer architectures and operating systems. Hence, it should be possible to install and run BPA on different systems. However, there are certain specific implementation details in BPA, such as the pipe data channel used to communicate with the external ASP solver, which might not work in the same way for non-Unix-based system, such as Windows. Also, due to time constraints, we have not been able to ensure BPA to build and run on Windows systems. Nonetheless, it should be possible to adapt the code for such implementation details to conform to any system infrastructures without many problems.
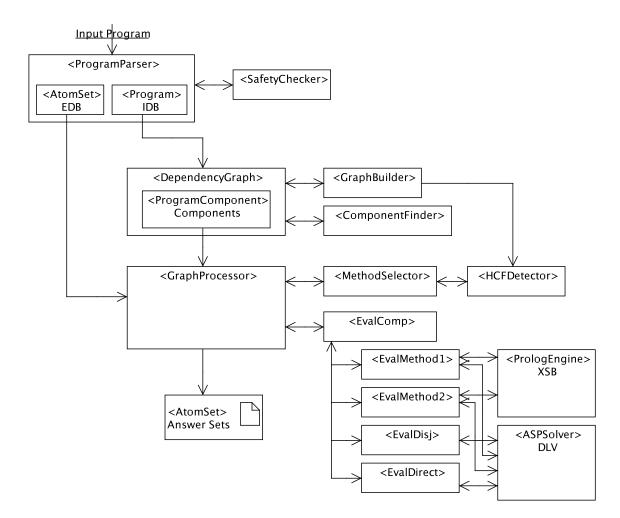
Input Program

<ProgramParser>

<AtomSet>
EDB

<Program>
IDB

<SafetyChecker>

<DependencyGraph>

<ProgramComponent>
Components

<GraphBuilder>

<ComponentFinder>

<GraphProcessor>

<MethodSelector>

<HCFDetector>

<EvalComp>

<EvalMethod1>

<PrologEngine>
XSB

<EvalMethod2>

<EvalDisj>

<ASPSolver>
DLV

<EvalDirect>

<AtomSet>
Answer Sets

Figure 6.1: Architecture and flow of information in BPA

For Unix/Unix-based systems, such Linux or OSX, compiling and installing BPA should be a relatively easy process. In this section, we provide a brief guide to compiling, installing and running BPA, which should work (perhaps with little adjustments) across all Unix-based systems.

Installing BPA

Before installing BPA, we have to make sure that the requirements for compiling BPA are met. At the very basic, the system should have a C++ preprocessor, compiler and linker available, as well basic system libraries. Most Unix-based systems have these preinstalled. Next, we need to install Boost development library, which is required by BPA. In Linux distributions, these are usually provided as a ready-to-install all-in-one package[6] comprising of all the components of Boost library. One can also choose to install individual components of Boost. At the very minimal, the following components should be present: Boost Spirit, Boost String Algorithms, Boost Graph Library and Boost Shared Pointer Library. BPA has been tested to work with Boost library version 1.35 and 1.36.

BPA uses DLV and XSB by default, so they need to be installed prior to compiling BPA. Installing DLV is simple enough and needs no explanations. For XSB, one thing to note is that BPA requires the multithreading feature of XSB, which is not activated by default. To activate this feature, one must compile XSB with the compile option "`--enable-mt`". Further information can be obtained in XSB's manual. BPA has been known to work with XSB version 3.1 and 3.2.

The full source code for BPA is available at *http://www.kr.tuwien.ac.at/research/systems/bpa*. To build the code, one can issue the following commands in the top-level directory of the source code tree:

```
$   ./configure --with-xsb=PATHTOXSB --with-dlv=PATHTODLV
$   make
```

where `PATHTOXSB` and `PATHTODLV` specify the *full* path to where XSB (the top-level directory) and DLV (the executable file) are located, respectively. If DLV's executable file is already in the system's search path[7], then one can omit the configuration option "`--with-dlv`". If the configuration option "`--with-xsb`" is omitted, then it is assumed that XSB is installed in `/home/$USER/XSB`.

Once the compilation is done, and no errors occurred, there should be an executable called `bpa` inside the `src` directory. To make it easy to use it, one can then copy this executable to a preferred location, or make symbolic links to it.

Running BPA

The syntax for calling BPA is the following:

```
bpa [OPTIONS] [FILENAME]
```

If no filename is given, BPA will read from the standard input. Otherwise, it will read its input from the specified filename. If more than one filename is given, only the first is read, and the rest are ignored. The following is the list of options accepted by BPA:

- "`-n=N`": stop after printing N answer sets. If N is 0 or not specified, BPA will compute all answer sets and print them all. Note that, since Method 1 may produce

---

[6]In Debian-based distribution, it is called `libboost-dev`.
[7]Usually specified by the environment variable `$PATH`.

the same answer set more than once, it cannot be guaranteed that these N answer sets printed are actually N distinct answer sets. There is currently no possibility to tell BPA to output exactly N distinct answer sets.

- "`-m=M`": prefer method M to perform evaluation, where M is 1, 2 or 3, indicating the use of Method 1, Method 2 or the Disjunctive Method, respectively. Program components which are non-HCF will always be evaluated using the Disjunctive Method, no matter what values are given in this option. Likewise, program components which are identified as small or stratified will always be evaluated by directly sending them to the external ASP solver. If this option is not given, a simple built-in heuristics for selecting evaluation method in BPA will be used to determine which evaluation method is preferred for each program component. A (non-official) value of 0 for this option will force BPA to not use these three evaluation methods, and simply pass each program component to the external ASP solver. This might be useful if one is not interested in avoiding exponential space, and simply wants to use BPA's SCC/modularity feature to break down the input program into program components and evaluate them separately.

- "`-filter=p1[,p2[...]]`": filter the answer sets produced to contain only the extensions of the predicates p1, p2,.... Similar to DLV's `-filter` option.

- "`-vmax=V`": sets the maximum number of variables a rule can have to be considered *small*. If a rule has less than, or equal to V variables, it will be considered as small, and will be included as constraints during the first step of model generation in Method 2 and the Disjunctive Method, and also during minimality checking in the Disjunctive Method. If this option is not given, the default value for V is 4. If the rule has more than V variables, then the decision of whether it is considered big or small is determined by the parameter given in the command line option, "`-grmax`".

- "`-grmax=N`": sets the maximum number of possible ground rules a rule can have to be considered to be small. If a rule has V variables, and V is greater than the value given in the "`-vmax`" option, and if the program contains C number of constant symbols, then the rule is considered to be small iff C^V is less than, or equal to $N * 10000$. Otherwise, it is considered to be big. The default value for N, when "`-grmax`" is not specified is taken to be 1; that is, by default C^V is compared to 10000.

6.2 Experiments with BPA

To obtain a measure of the performance of BPA in comparison with the current ASP solvers, we have conducted a benchmarking experiment using BPA, DLV and GrinGo/claspD. We selected 6 problem classes, which we will describe in details shortly. These problems are selected and are encoded in such a way that when they are evaluated using current ASP solvers such as DLV and GrinGo/claspD, we can observe exponential space grounding behavior. As a consequence, some of the encodings may not be the most efficient encodings one can use for these problems. However, we note that the purpose of this experiment is merely to show that BPA (and by extension, the three methods proposed) does indeed perform its evaluation within polynomial space, even when other ASP solvers do not. We start by describing the problem classes we have used in this experiment.

6.2.1 Test Problems

2QBF

Description. We recall that the problem of **2QBF** is the problem of deciding whether a quantified Boolean (QBF) $\Phi = \exists X \forall Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = C_1 \vee \ldots \vee C_k$ is a DNF formula over $X \cup Y$, is valid.

Encoding. A **2QBF** problem is usually encoded in a propositional program (for example, see e.g., [Leone et al., 2006]). However, we refrain from doing this, since in a propositional program no grounding is necessary, and the program would be irrelevant for the purpose of this experiment. We therefore construct a non-propositional encoding of **2QBF**. We first rephrase the problem as follows. Deciding whether $\Phi = \exists X \forall Y \phi$ is valid is equivalent to deciding whether it is not the case that, for all assignments of X, there exist an assignment for Y such that ϕ is false. This way, we may encode this problem as a constraint satisfaction problem, by checking whether a rule encoding the truth value of $\neg\phi$ is violated or not.

We first represent $\neg\phi$ in CNF. If ϕ is given by

$$\phi = \bigvee_{i=1}^{k} D_i$$

where each $D_i = l_{i,1} \wedge l_{i,2} \wedge l_{i,3}$, then $\neg\phi$ is given by

$$\neg\phi = \bigwedge_{i=1}^{k} C_i$$

where each $C_i = \neg l_{i,1} \vee \neg l_{i,2} \vee \neg l_{i,3}$. We may encode the truth value of each possible combinations of of negative/positive occurrences of the literals in a clause C_i using facts of the predicates $c_{p,n}/3$, with $p + n = 3$ intuitively mean that the clause has p positive literals and n negative literals. For example, if a clause has exactly two of its literals occur positively, say $c = x_1 \vee \neg y_2 \vee y_3$, then the following set of facts

$$c_{2,1}(1,1,1).$$
$$c_{2,1}(1,1,0).$$
$$c_{2,1}(1,0,1).$$
$$c_{2,1}(1,0,0).$$
$$c_{2,1}(0,1,1).$$
$$c_{2,1}(0,1,0).$$
$$c_{2,1}(0,0,0).$$

which contains each possible valuation of $c_{2,1}/3$ using constants 1 and 0, except $c_{2,1}(0,0,1)$, captures all the instances for which the clause is true. There are 7 facts for each of the predicates $c_{3,0}/3$, $c_{2,1}/3$, $c_{1,2}/3$ and $c_{0,3}/3$, giving a total of 28 facts. We call this set of facts F_c. Using F_c, we may represent each clause $c \in C$ in the body of a rule with the corresponding predicate $c_{p,n}/3$. For example, the clause $c = x_1 \vee \neg y_2 \vee y_3$ may be represented by a literal $c_{2,1}(L1, L2, L3)$, where $L1$, $L2$, and $L3$ respectively binds the values of x_1, y_3 and y_2.

To check the satisfaction of the formula in each possible valuation of the variables $x_i \in X$, we add the following guessing clauses

$$val(x_i, 0) \vee val(x_i, 1) \leftarrow \qquad\qquad (6.1)$$

Finally, the constraint to check the truth value of ϕ is written as

$$\leftarrow \; val(x_1, X_1), \dots, val(x_m, X_m), c_{\sigma(1)}(L_{1,1}, L_{1,2}, L_{1,3}), \dots, c_{\sigma(k)}(L_{k,1}, L_{k,2}, L_{k,3}) \quad (6.2)$$

where each $c_{\sigma(i)}(L_{i,1}, L_{i,2}, L_{i,3})$, $1 \leq i \leq k$ is the corresponding representation of clause C_i in $\neg\phi$, using the predicates $c_{3,0}/3$, $c_{2,1}/3$, $c_{1,2}/3$ or $c_{0,3}/3$, as described previously. As an illustration, consider the following example:

Example 6.1. Let the QBF be $\Phi = \exists X \forall Y \phi$ and ϕ is

$$\phi = (x_1 \wedge \neg x_2 \wedge y_1) \vee (\neg x_2 \wedge y_1 \wedge \neg y3)$$

We have that

$$\neg\phi = (\neg x_1 \vee x_2 \vee \neg y_1) \wedge (x_2 \vee \neg y_1 \vee y3)$$

There are only two X-variables, x_1 and x_2. Thus, we have the following two guessing rules:

$$val(x_1, 0) \vee val(x_1, 1) \leftarrow$$
$$val(x_2, 0) \vee val(x_2, 1) \leftarrow$$

Finally, the constraint is written as

$$\leftarrow \; val(x_1, X_1), val(x_2, X_2), c_{1,2}(X_2, X_1, Y_1), c_{2,1}(X_2, Y_3, Y_1)$$

Exponential space behavior is potentially observed when the solver is instantiating the constraint, with increasing size of X.

Parameters and Instances. The parameters of the problem are the size of the sets X and Y, as well as the number of clauses k. In total, we have selected 40 instances of **2QBF**, with $|X|$ ranges from 5 to 22, $|Y|$ from 6 to 25 and k from 5 to 32. The distribution of the X and Y variables, as well as the sign of each variables occurring in the clauses have been generated randomly. Since the goal of the experiment is merely to compare performances of the three systems, the instance generator was not designed to produce "hard" instances.

Modified Strategic Companies

Description. The original **Strategic Companies** problem is described as follows: suppose that there is a set of companies $C = \{c_1, \dots, c_n\}$ owned by a holding and a set $P = \{p_1, \dots, p_k\}$ of products. For each c_i, $1 \leq i \leq n$, we have a set $P_i \subseteq P$ of products produced by c_i, and a set $R_i \subset C_i$ of companies controlling (owning) c_i. We assume that the holding produces all the products in P, that is $\bigcup_{c_i \in C} P_i = P$. Each company may have more than one controlling set. A subset of the companies $C' \subseteq C$ is called a *production-preserving* set if the following conditions hold:

- the companies in C' produce all the products in P, i.e., $\bigcup_{c_i \in C'} P_i = P$, and

- The companies in C' are closed under the controlling relation, i.e., if $R_i \subseteq C'$ for some i, $1 \leq i \leq n$, then $c_i \in C'$.

A subset minimal set C', which is production-preserving, is called a *strategic set*. A company is called strategic iff $c_i \in C'$, for some strategic set C'. We refer to e.g.,[Cadoli et al., 1997] for more discussions on the problem.

Despite the seemingly lengthy definition of the problem, it can actually be represented and solved using a fairly simple logic program encoding. First, we adopt the restriction used in [Cadoli et al., 1997, Leone et al., 2006] where each product is produced by at most two companies (i.e., for each $p \in P$, $|\{c_i \mid p \in P_i\}| \leq 2$). Moreover, each company is controlled by at most three other companies (i.e., for each i, $1 \leq i \leq n$, $|R_i| \leq 3$). These restrictions do not reduce the complexity of the problem(which is Σ_2^P-complete).

Encoding. We encode first the production capability of each company, as well the control relation between companies using the following set of facts:

(i) $product\_by(p, c_i, c_j)$, if $\{c_k \mid p \in P_k\} = \{c_i, c_j\}$, where c_i and c_j may coincide.

(ii) $controlled\_by(c_i, c_j, c_k, c_l)$, if $R_i = \{c_j, c_k, c_l\}$, where c_j, c_k and c_l may not be distinct.

Then, the following program solves the **Strategic Companies** problem[Leone et al., 2006]:

$$strat(X) \vee strat(Y) \leftarrow product\_by(P, X, Y)$$
$$strat(W) \leftarrow controlled\_by(W, X, Y, Z)$$

We observe however, that under this encoding, solving **Strategic Companies** using current ASP solvers will not trigger the exponential space behavior that we are interested in. We therefore add an additional requirement to this problem, as follows: given k companies $c_1, \ldots, c_k$, such that c_i controls c_{i+1} for $1 \leq i \leq k - 1$, if $c_1, c_2, \ldots, c_{k-1}$ is strategic, then c_k must also be strategic. We allow k to vary in different instances.

This additional requirement can be encoded by the following set of rules:

$$controlled(C, X_1) \leftarrow controlled\_by(C, X_1, X_2, X_3)$$
$$controlled(C, X_2) \leftarrow controlled\_by(C, X_1, X_2, X_3)$$
$$controlled(C, X_3) \leftarrow controlled\_by(C, X_1, X_2, X_3)$$
$$strat(X_k) \leftarrow controlled(X_2, X_1), controlled(X_3, X_2), \ldots, controlled(X_k, X_{k-1}),$$
$$strat(X_1), \ldots, strat(X_{k-1})$$

With this additional rules, exponential space behavior is potentially triggered when an ASP solver grounds the last rule. We refer to this new problem as **ChainStratComp**, to reflect the existence of a "chain query" in the last rule given above. As a last modification, we require that at least two of the companies be strategic. Without loss of generality, let the two companies selected be $c0$ and $c1$.

$$\leftarrow not\ strat(c0)$$
$$\leftarrow not\ strat(c1)$$

Parameters and Instances. Parameters set up for this problem are the number of companies c, the total number of products, p and the size k used in the last rule. In total, we have selected 44 instances of **ChainStratComp**, with the number of companies ranging from 4 to 10, the number of products from 8 to 20 and k from 5 to 15. The control relations and company-product relations have been generated randomly.

Layered non-HCF program

Description. We formulate a class of simple non-HCF programs to test how the Disjunctive Method performs. The programs in the class do not actually corresponds to any real problem/scenario. However, the intention of using such class of programs in this experiment

is merely to see how the evaluation of non-HCF programs (using Disjunctive Method) compares to the evaluation of such programs using current ASP solvers.

We construct each program in this class as a layered program, with each layer consisting of a set of rules containing a head cycle between literals appearing in the head of one of the rules. At each layer, a rule is formulated that will potentially generate an exponential number of ground rules. We will refer to this class of programs as **Simple-NHCF**.

Encoding. The program encoding consists of a set D of n facts of the form $d_1(i)$, $1 \leq i \leq n$. For each layer j, $1 \leq j \leq l$, the layer L_j consists of the rules:

$$p_i(1, X) \vee \ldots \vee p_i(k, X) \leftarrow d_i(X)$$
$$p_i(2, X) \leftarrow p_i(1, X)$$
$$\vdots$$
$$p_i(1, X) \leftarrow p_i(k, X)$$
$$r_i \leftarrow d_i(X_1), p_i(Y, X_1), \ldots, d_i(X_k), p_i(Y, X_k)$$
$$d_{i+1}(X) \leftarrow r_i, d_i(X)$$

The complete program is

$$P = \bigcup_{i=1}^{l} L_i \cup D.$$

An exponential number of ground rules can potentially be generated by instantiating the rule $r_i \leftarrow d_i(X_1), p_i(Y, X_1), \ldots, d_i(X_k), p_i(Y, X_k)$ in each layer i.

Parameters and Instances. Three parameters are used: the size of the initial set of facts $|D|$, the length of the head-cycle (also the length of the rule deriving r_i) k, and the number of layers to build, l. We generated 48 instances of **Simple-NHCF** with n ranging from 4 to 10, k from 5 to 10 and l chosen to be either 1 or 2.

Clique

Description. The problem **Clique** is stated as follows: given a graph $G = \langle V, E \rangle$, and a positive integer $k \leq |V|$, does G contain a clique of size k or more (i.e., a subset $V' \subseteq V$ with $|V| \geq k$, such that for each $v, w \in V'$, $(v, w) \in V$)?

Encoding. The input graph G is given as a set of facts of the form $edge(v, w)$, indicating that there is an edge between v and w. We assume that the graph is undirected and that only one of $edge(v, w)$ or $edge(w, v)$ is enough to represent the edge relation between v and w. The following rules perform the guess on a clique:

$$node(X) \leftarrow edge(X, \_)$$
$$node(X) \leftarrow edge(\_, X)$$
$$in(X) \vee out(X) \leftarrow node(X)$$
$$\leftarrow in(X_1), in(X_2), X_1 \neq X_2, not\, edge(X_1, X_2), not\, edge(X_2, X_1)$$

To express the existence of clique of size k, we use the following two rules:

$$ok \leftarrow node(X_1), \ldots, node(X_k), in(X_1), \ldots, in(X_k), X_1 \neq X_2, \ldots, X_{k-1} \neq X_k$$
$$\leftarrow not\, ok$$

where the inequalities range over all possible pairs X_i, X_j with $i < j$. Exponentially many ground rules may be obtained by instantiating the rule to derive ok.

Parameters and Instances. To obtain the input graph for **Clique** instances, we generate randomly graphs with n nodes, where n ranges from 5 to 23. At each node, 3 edges connecting the node to other (possibly not distinct) nodes are randomly generated. From these graphs, we then construct a set of **Clique** instances, where the size of the clique k, is chosen to be between $n/2$ to $n/4$. In total, we have generated 32 instances of **Clique**.

Set Packing

Description. The problem of **Set-Packing** is stated as follows: given a collection C of sets and a positive integer $k \leq |C|$, does C contains at least k mutually disjoint sets?

Encoding. The input collection C is given as a set of facts $set(s_1), \ldots, set(s_n)$, where $s_1, \ldots, s_n \in C$. The members of each s_i can then be specified with a set of facts $member(s_i, m_j)$, for each $m_j \in s_i$. The rules used to generate a guess on a subcollection of mutually disjoint sets from C are:

$$in(S) \vee out(S) \leftarrow set(S)$$
$$\leftarrow in(S_1), in(S_2), S_1 \neq S_2, member(S_1, X), member(S_2, X)$$

To express the existence of at least k mutually disjoint sets in C, we use the following rules:

$$ok \leftarrow set(S_1), in(S_1), \ldots, set(S_k), in(S_k), S_1 \neq S_2, \ldots, S_{k-1} \neq S_k$$
$$\leftarrow not\, ok$$

where the inequalities range over all possible pairs S_i, S_j with $i < j$. Exponentially many ground rules can be obtained as a result of instantiating the rule to derive ok.

Parameters and Instances. We have generated 87 test instances of **Set-Packing**, with the number of sets ranging from 7 to 15, the maximum number of members each set ranges from 2 to 7, and the size of the set packing between 2 to 11.

Layered n-Reachability

Description. Given a directed graph $G_0 = \langle V, E \rangle$, we define the property of *n-reachability* for any pair of vertexes $v, w \in V$ as follows: w is n-reachable from v iff there is a set of $n + 1$ vertexes $s_1 = v, s_2, \ldots, s_{n+1} = w$ such that for each $i, 1 \leq i \leq n$, $(s_i, s_i + 1) \in E$. We write $v \rightarrow_n w$ if w is n-reachable from v. The transitive closure of $\rightarrow_n$ is denoted by $\rightarrow_n^+$. The problem of **n-Reachability** is stated as: given a graph G_0, compute all pairs of $v, w \in V$ such that $v \rightarrow_n^+ w$.

To increase the complexity of **n-Reachability**, we opted to make the following modifications:

- First, in order to allow unstratified negation/disjunction in the problem encoding, we formulate the problem so that the computed n-reachability is performed on the subgraphs of the input graph. This would require an additional step of "guessing" a subgraph $G_0' \subseteq G_0$.

- We also require that there should be at least one pair of vertexes v and w such that $v \rightarrow_n^+ w$ in G_0', since otherwise, the problem would be trivial.

- Additionally, no pair of vertexes x, y can be such that $x \rightarrow_{n+1} y$ in G'. This constraint makes the problem to be even harder.

Furthermore, to test the effect of the application of SCC and modularity analysis, we design a "layered computation" of n-reachability, as follows. For each i, $i \geq 0$, we define a new graph $G_{i+1} = \langle V, E' \rangle$ which is constructed by creating an edge for all pairs of v, w such that $v \rightarrow_n w$ in G'_i (i.e., $(v, w) \in E'$ iff $v \rightarrow_n w$ in G'_i). Computation of n-reachability is continued with the new graph G_{i+1}, and so on up to a certain number "layers".

Encoding. The input graph is specified as a set of facts G_F which encodes the edges of G using the predicate $edge/2$: $edge(v, w) \in G_F$ iff $(v, w) \in E$. The computation starts by specifying that $G_0 = G$, using the rule $e_0(X) \leftarrow edge(X)$. Then, for each i, $1 \leq i \leq l$, we define the rules for layer i, L_i as follows:

$$
\begin{aligned}
s_i(X, Y) \vee ns_i(X, Y) &\leftarrow e_{i-1}(X, Y) \\
p_i(X_1, X_{n+1}) &\leftarrow s_i(X_1, X_2), \dots, s_i(X_n, X_{n+1}) \\
r_i(X, Y) &\leftarrow p_i(X, Y) \\
r_i(X, Z) &\leftarrow r_i(X, Y), p_i(Y, Z) \\
e_i(X, Y) &\leftarrow r_i(X, Y) \\
exists_i &\leftarrow p_i(X, Y) \\
&\leftarrow not\ exists_i \\
&\leftarrow p_i(X, Y), s_i(Y, Z)
\end{aligned}
$$

The complete program is given by $P_1 = G_F \cup \{e_0(X) \leftarrow edge(X)\} \cup L$ where

$$
L = \bigcup_{i=1}^{l} L_i.
$$

Exponential number of ground rules (w.r.t. the input size) are potentially generated by instantiating the rule $p_i(X_1, X_{n+1}) \leftarrow s_i(X_1, X_2), \dots, s_i(X_n, X_{n+1})$ on each layer i.

Parameters and Instances. We generate the graphs randomly, with the number of vertexes n ranging from 5 to 15. Each vertex is connected with two other vertexes by two randomly generated edges. The length of the reachability is chosen to vary between 2 until $n/2$, and the number of layers to build is at most 2. In total, we generated 42 test instances for **Layered n-Reachability**.

6.2.2 Experiment Settings

We set up the experiment with a goal of measuring both time and space consumption of each of DLV, GrinGo/claspD and BPA in deciding whether the input program instance is consistent (i.e., has an answer set) or not. We believe that the three methods proposed in this work should be able to perform quite reasonably well in finding *one* answer set of a program, but not necessarily so when it needs to enumerate *all* answer sets. So, the evaluation of the problem instances by the three systems will be only to decide that the problem instances is consistent or not. In both DLV and BPA, we can limit the number of answer sets printed to 1 by adding the command line switch "–n=1". GrinGo/claspD by default prints at most one answer set.

The experiment is performed on a 64-bit machine with 4 cores of Intel Xeon @ 3.00GHz and 16 GB of RAM running OpenSUSE 11.0 with Linux kernel version 2.6.25.20 SMP. The DLV version used is the version available at its home page[8] at the time this experiment is

---

[8] http://www.dbai.tuwien.ac.at/proj/dlv/

conducted, showing the build code `BEN/Oct 11 2007`. We also used the latest version of
GrinGo and claspD available at this time, with GrinGo version 2.0.3 and claspD version 1.1[9].
For every problem instance, we allowed a maximum execution time of 2 hours (7200s) and
a maximum of 2GB memory.

For BPA, if the input problem is normal/HCF, we may select between using Method
1 and Method 2. We performed both methods to see how they compare. Except for
ChainStratComp and **Simple-NHCF** which are non-HCF, we arrange the experiment
so that evaluation on each problem instance by BPA is performed using both Method 1 and
Method 2, by passing the command line switch "`-m=1`" and "`-m=2`". No other command
line options are used.

We measure both time and space consumption for each evaluation by the three systems.
In most Unix systems, measuring a process' time consumption can be done in a simple way:
most systems have a `time` command to do this. Measuring (maximum) memory usage
by a process is, however, not so simple. Since memory usage can increase and decrease
during the lifetime of a process, one must keep a record of the current maximum usage,
until the process is terminated. The POSIX standard actually defines a system call named
`getrusage()` which measures the resource usage of a process. One of the resources being
monitored is the maximum size of virtual memory ever used during the execution of the
process. Some BSD systems have implemented this, but unfortunately, Linux (which is
the operating systems we use for the experiment) does not have a complete support for all
components of `getrusage()`. Specifically, it does not yet support getting the maximum
memory usage of a process.

We came up with a solution to the problem by writing a small program which first
executes the ASP solvers as its child process, using the `fork()` and `exec()` system calls.
It then measures the memory usage of the child process by periodically checking the child
process' status, which in Linux is exposed in the `/proc` filesystem. The period of the
checking is set to 10 microseconds, which we assume to be frequent enough to make sure
that it does not miss any important changes in memory usage. Several testing cases show
that the utility program works reasonably good, with a quite accurate measurement.

6.2.3 Experiment Results

Here we present the experiment results. A note on the graphs: some of the plots presented
shows an "oscillating" (*zig-zag*) pattern, instead of a monotonic pattern, as one might
expects. This is due to the way the data for the instances are sorted, before plotted. Since
each test problem has several parameters, we had to select one of them as the parameter
on which to sort the data. Therefore, an increase in the X-axis does not strictly mean an
increase in the "real-size" of the problem.

2QBF

From 40 test instances, DLV managed to complete the evaluation under the allowed resource
limit for 36 instances, while GrinGo/claspD could solve only 34 instances. BPA (using both
Method 1 and Method 2) managed to solve all 40 instances. Both DLV and GrinGo/claspD
did not complete execution on the failed instances because they exceeded the allowed space
limit.

The time usage of DLV, GrinGo/claspD and BPA, using Method 1 and Method 2 is given
in the graph of Figure 6.2. On all of the instances (except some small instances) BPA
managed to complete execution faster than both DLV and GrinGo/claspD. The are two

---

[9]http://potassco.sourceforge.net/

peculiar instances where BPA took more time than the other systems. The first is the instance with $|X| = 14$, $|Y| = 19|$ and $|C| = 28$, where BPA Method 1 took 297 seconds and BPA Method 2 took 306 seconds, compared to DLV's 56 seconds and claspD's 1 second. The second instance has $|X| = 16$, $|Y| = 21$ and $|C| = 32$, where BPA took more than 1800 seconds, claspD took 27 seconds and DLV failed to complete within the resource limit. It is interesting to note that these two instances are two of the biggest inconsistent instances of the 40 instances tested.

The reason why BPA takes more time to solve these inconsistent instances may be understood with the following reasoning. Method 1 and Method 2 basically work by enumerating subsets ground rules and/or subsets of possibly true atoms. When the program has an answer set, it may be enough to visit small number of these subsets in order to find the answer sets. However, if the program is inconsistent, it may be the case that both Method need to perform full enumeration of those subsets, before concluding that the program has no answer sets. Moreover, if the program is inconsistent, the SCC evaluation performed by BPA, which follows Algorithm 12, may need to go back and forth, backtracking between *lower stratum* program components and *higher stratum* program components many times before concluding that the program has no answer set.

For our encoding of **2QBF**, the "guessing rules" given in 6.1 acts as the lower strata program components which generates candidate answer sets, while the constraint given in 6.2 act as the higher stratum program component which eliminates candidate answer sets that do not satisfy the problem. If the program is inconsistent, Algorithm 12 will go back and forth between the rules 6.1 and constraint 6.2 until no more candidate answer set is produced. This can take a long time depending on the number of candidate
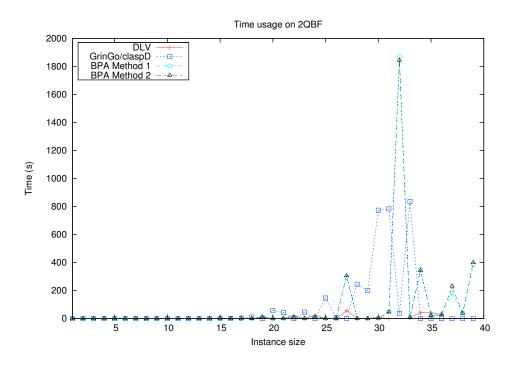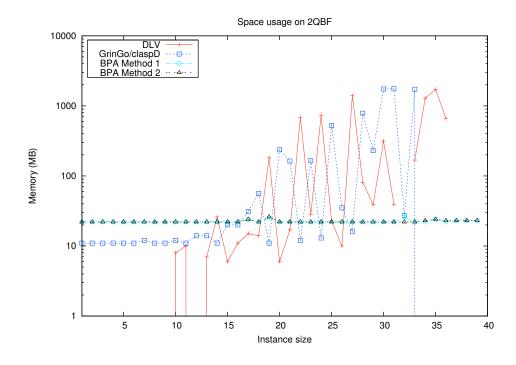

Figure 6.2: Time usage on **2QBF**

Figure 6.3: Space usage on **2QBF**

answer sets generated by the *lower strata* program components. ASP solvers such as claspD and DLV may infact have a more sophisticated backtracking procedure which can detect inconsistencies much earlier, while the backtracking procedure used in Algorithm 12 certainly does not.

The space consumptions of the three systems for **2QBF** instances are given in Figure 6.3. From the presented graphs, we can easily see that BPA, using either Method 1 or Method 2, can evaluate all 40 instances with relatively small space consumptions, ranging between 22-26 MB, compared to DLV and GrinGo/claspD, which exceeded the allowed space limit of 2GB for some instances. This confirms our expectation that both Method 1 and Method 2 should be able to evaluate the input programs under polynomial space.

ChainStratComp

From the 44 instances of **ChainStratComp**, DLV managed to complete evaluation for only 30 instances; it failed to stay under the allowed memory limit for most of the instances with parameter k (length of the "chain-query") greater than, or equal to 12. GrinGo/claspD managed to complete all of the test instances, but barely managed to evaluate the last instance using 1951 MB of memory and almost half an hour execution times. BPA (using Method 3) again completed all test instances with smaller memory usage and (most of the cases) also less execution time. It is worth noting that the program used to encode **ChainStratComp** is such that BPA cannot take any significant advantage of using SCC/modularity evaluation, since all of the three "important" rules in the program are

actually in one program component:

$$strat(X) \lor strat(Y) \leftarrow product\_by(P, X, Y)$$
$$strat(W) \leftarrow controlled\_by(W, X, Y, Z)$$
$$strat(X_k) \leftarrow controlled(X_2, X_1), controlled(X_3, X_2), \ldots, controlled(X_k, X_{k-1}),$$
$$strat(X_1), \ldots, strat(X_{k-1})$$

The results in time usage for the three systems for **ChainStratComp** shows the clear advantage in execution time for BPA over DLV and GrinGo/claspD, especially for bigger instances where the effect of performing full exponential grounding/instantiations in DLV and GrinGo/claspD becomes much more significant. Figure 6.4 shows the time usage of the three systems on **ChainStratComp**.

The space consumptions of the three system for **ChainStratComp** instances are shown in Figure 6.5. BPA started out with a slightly higher memory consumption for the small instances, compared to the other systems. However, memory consumption in BPA stayed between about 20-21 MB for all the rest of the instances, proving that it does not perform full instantiations of the program. The other two systems, on the other hand, clearly showed that their memory consumptions grows exponentially with increasing input size.

Simple-NHCF

The instances of **Simple-NHCF** provide a more extreme view on how the methods we proposed (in this case, Disjunctive Method, which is the relevant method to use since all the instances are non-HCF) can improve the efficiency of the evaluation of a logic program
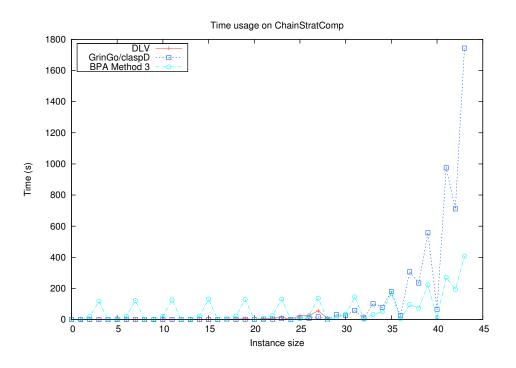

Figure 6.4: Time usage in **ChainStratComp**

with bounded predicate arities compared to other ASP solvers. From the 48 instances tested, DLV managed to complete evaluation for only 14 instances, while GrinGo/claspD completed 34 of the 48 instances. BPA completed all the test instances, and even more than that, it evaluated every instance in a very short time (less than a second) and almost constant space. Figure 6.6 and 6.7 shows the time and space usage on **Simple-NHCF**

To understand how BPA is able to perform so well for these programs, let us consider the following instance of **Simple-NHCF**, P:

$$d0(1).$$
$$d0(2).$$
$$p0(1, X) \vee p0(2, X) \vee p0(3, X) \leftarrow d0(X)$$
$$p0(2, X) \leftarrow p0(1, X), d0(X)$$
$$p0(3, X) \leftarrow p0(2, X), d0(X)$$
$$p0(1, X) \leftarrow p0(3, X), d0(X)$$
$$r0 \leftarrow d0(X1), p0(Y, X1), d0(X2), p0(Y, X2), d0(X3), p0(Y, X3)$$
$$d1(X) \leftarrow r0, d0(X)$$

We recall again the method used for generating models as explained in Section 4.2.1 and Section 5.2.3, as well as the method used for checking minimality described in Section 4.2.2 and Section 5.1.8. Recall the definition of $S_{\mathcal{P}}$ in Definition 4.2. We have that, for program P,

$$I = S_P = \{d0(1), d0(2), p0(1, 1), p0(1, 2), p0(2, 1), p0(2, 2), p0(3, 1), p0(3, 2), r0, d1(1), d1(2)\}$$
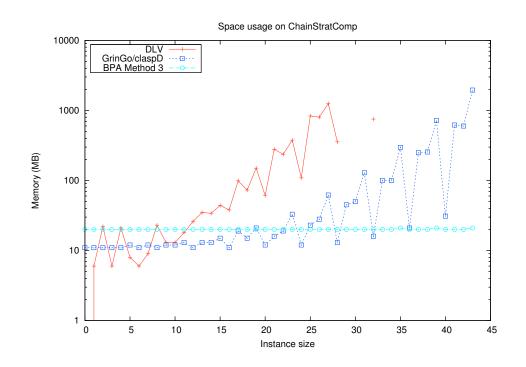

Figure 6.5: Space usage on **ChainStratComp**

The Disjunctive Method will generate the models of P which are subsets of I. However, it is easy to see that, due to the four rules

$$
\begin{aligned}
p0(1, X) \vee p0(2, X) \vee p0(3, X) &\leftarrow d0(X) \\
p0(2, X) &\leftarrow p0(1, X), d0(X) \\
p0(3, X) &\leftarrow p0(2, X), d0(X) \\
p0(1, X) &\leftarrow p0(3, X), d0(X)
\end{aligned}
$$

the set $\{p0(1, 1), p0(1, 2), p0(2, 1), p0(2, 2), p0(3, 1), p0(3, 2)\}$ must be contained in every model of P. The atoms $r0$, $d1(1)$ and $d1(2)$ follows immediately, and we are left with only one model to consider, I itself. No interpretation smaller than I is found to be a model on the model generation step of Disjunctive Method.

We consider now the minimality checking using the method described in Section 5.1.8. Minimality checking is performed by first constructing the program $P_g(I) \cup Small(P)^{min}(I)$, in attempt to produce a model smaller than I which satisfies P^I. Note that P (and every other instance of **Simple-NHCF**) is positive, which means that $P^I = P$. We may assume that the four rules above are considered small, since each of them always contains one variable in all instances of **Simple-NHCF**. We can see that the program $P_g(I) \cup Small(P)^{min}(I)$ is inconsistent, since no proper subset I' of I satisfies the above four rules. Therefore, the algorithm immediately concludes that I is minimal.

The program instances in **Simple-NHCF** represent the classes of program for which the Disjunctive Method can consider as "easy". Each instance of Simple-NHCF has only one answer set, and the Disjunctive Method is able to arrive at the answer set quickly, without causing exponential space consumption. In contrast to this, evaluations on the other ASP
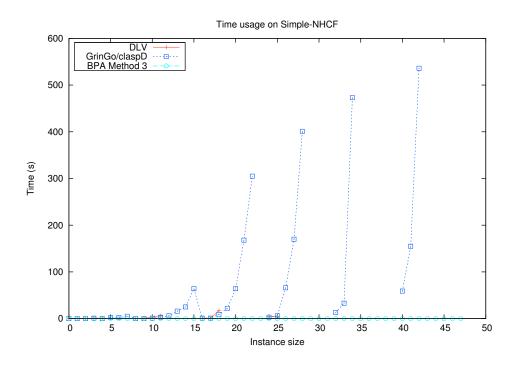

Figure 6.6: Time usage for **Simple-NHCF**

solvers have to wait for the grounders to compute exponentially many ground rules first, because the grounders themselves cannot evaluate completely the rules containing the head-cycle. Not only does this grounding step consume unnecessarily exponential amount of space, but it also becomes the bottleneck for the evaluation of the programs.

Set Packing

From the 87 instances of **Set-Packing** tested, DLV completed 60 of them under the allowed resource limit. GrinGo/claspD managed to complete 75 of them, while BPA completed all test instances successfully. All the failed instances by the two ASP solvers are due to the computation exceeding allowed space limit. Figure 6.8 and 6.9 shows the resource consumptions of the three systems for **Set-Packing**.

Results for the time consumption of the three systems are fairly consistent with the previous test problems: BPA is able to complete most of the test instances in shorter time compared to the other two systems. Maximum time usage for an instance by GrinGo/claspD was more than 400 seconds, while BPA uses only at most 14 seconds using Method 1 and Method 2. DLV failed on most of the bigger instances, and the highest time consumption by DLV was 41 seconds.

However, the space consumptions for the **Set-Packing** instances give us a bit different view from the previous test problems. Even though we can see that the space consumption by BPA still stays within limit, it is unexpectedly high; around 200 MB for some of the instances. For most of the larger test instances, these values are still much smaller than the space consumptions of the other two systems. However, the fact does raise a question on why BPA requires so much memory, if it only stores a few ground rules at any given time.
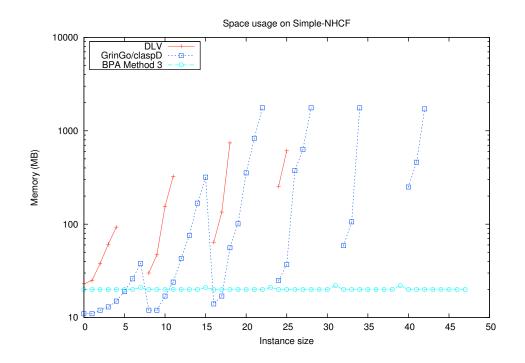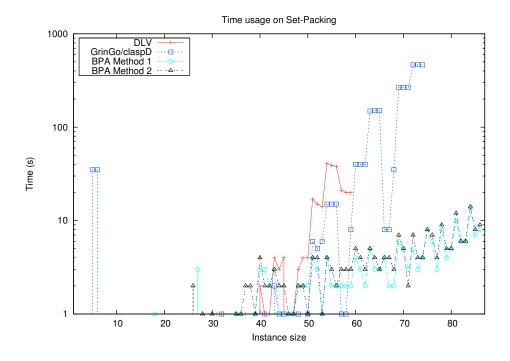

Figure 6.7: Space usage for **Simple-NHCF**

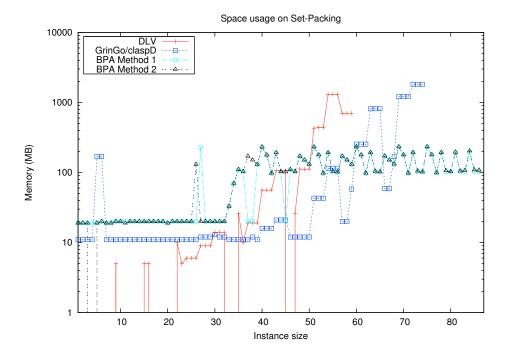Figure 6.8: Time usage for **Set-Packing**

Figure 6.9: Space usage for **Set-Packing**

After some careful examinations, we discovered that this has something to do with a technical issue related to XSB thread management used in BPA. Each Prolog thread in BPA is started using an C API call `xsb_ccall_thread_create()` to XSB. This call creates a new XSB thread and allocates a memory block for use by the thread. When the Prolog thread is no longer needed, BPA calls the function `xsb_kill_thread()` to destroy the thread. In theory, the function call should also cause the Prolog thread to release the allocated memory block. However, in some cases, this does not seem to happen. This causes the seemingly higher memory usage, even though technically, most of the reserved memory blocks are actually no longer used. Despite our sincerest efforts, this bug has not been resolved. However, we note that this is only a technical issue which does not necessarily invalidate the conclusion that the three methods used in BPA perform evaluation under polynomial space, assuming that the input program has bounded predicate arities.

Clique

For the test problem **Clique**, we found that under the allowed resource limit, DLV completed 17 of the 32 instances, GrinGo/claspD completed 23 of them, while BPA with Method 1 and Method 2 completed all 32 test instances. For growing size of test instances, BPA using Method 1 and Method 2 generally managed to complete the evaluation faster than both other systems, with Method 1 performing slightly faster than Method 2. Figure 6.10 depicts the time consumptions of the three systems for the **Clique** instances. Note that, in this plot, some of the data points for DLV and GrinGo/claspD are missing, especially for the larger instances, because these systems failed to complete the evaluation under the allowed resource limit.
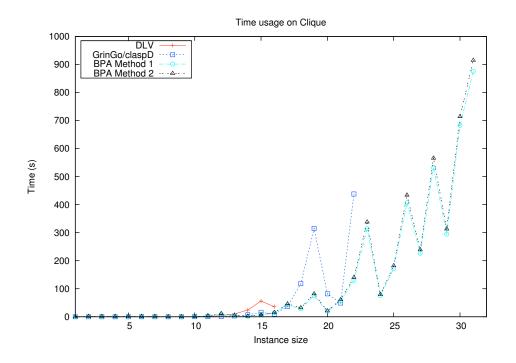

Figure 6.10: Time usage for **Clique**

Turning our attention to the results of the space consumption by the three systems, we see a similar pattern as observed in the results for **Set-Packing**. Even though BPA used lower amounts of space for most of the test instances, compared to the exponentially high amounts of space usages by the other two systems, we notice that BPA still used unnecessarily high amount of memory. For the largest test instance, BPA used more than 1 GB of memory (the other two systems exceed the allowed resource limit of 2 GB for this instance). It seemed that the effects of the technical bug in the implementation of the XSB interface are much bigger for **Clique** than they are for **Set-Packing**. We assume this might be due to the much larger size of the Herbrand base of **Clique**, compared to **Set-Packing**.

Layered n-Reachability

The last test problem we considered is the layered **n-Reachability** problem. Unfortunately, here we encounter another technical problem involving XSB. For the instances where the input graph contains 9 nodes or more, XSB could not complete its computation and complained about a memory corruption. We could not find the reason for this problem, and the problem still remains unsolved. We concede that perhaps this problem appears only because of the way communication between BPA and XSB is implemented, and is not necessarily caused by XSB itself. In summary, from the 42 instances generated, GrinGo/claspD successfully completed 25 instances, DLV completed 21 instances, while BPA only managed to complete 7 smaller instances for which the memory problem did not occur. For the bigger instances, BPA always fails due to the same memory-related problem.

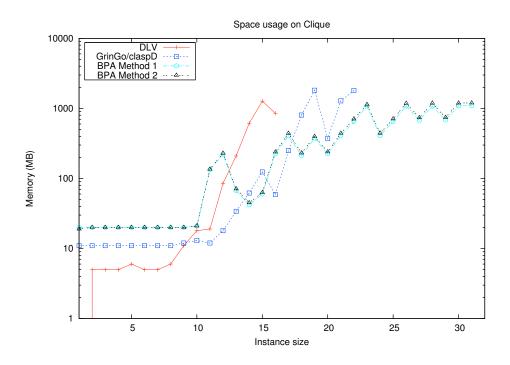This technical problem has prevented us from collecting enough data needed to make

Figure 6.11: Space usage for **Clique**

any comparison on how BPA performs against the other systems for the **n-Reachability** problem. Nevertheless, we would like to stress the point that the poor results obtained in this test problem are due to technical issues, and do not necessarily mean that the evaluation methods used by BPA cannot perform well for the test problem. With a better implementation, we believe that BPA, using Method 1 or Method 2, can perform reasonably well also for this test problem. Unfortunately, due to time constraints, we have not been able to pursue this matter any further.

6.2.4 Concluding Remarks

We have developed a system called BPA which implements the three methods described in Chapter 4, and was designed following the architecture framework described in Chapter 5. It uses DLV as its external ASP solver and XSB as the Prolog engine used to compute subsets of the ground rules. We have also performed experimental tests on BPA using several test problems, and compared its performance against DLV and GrinGo/claspD. The goal of the experiment is to measure time and space consumptions on all three systems for the task of deciding answer set existence.

The results obtained suggest that BPA does confirm our expectation of staying within polynomial space for all the test instances, while also managed to finish evaluation of most test instances in less time than the other two systems. As an exception, some inconsistent program instances require BPA to complete its evaluation longer than DLV or GrinGo/claspD, especially when using Method 1 or Method 2. This is quite understandable, since if a program is inconsistent, it might be the case that Method 1 or Method 2 will have to visit many ground subsets of the input program before deciding that none of them produces an answer set. BPA using Disjunctive Method might not have the same problem, and might be more efficient for inconsistent programs.

Despite the good results given by BPA, we have also seen that it still has some problems, mainly technical ones, which prevent it from being able to efficiently evaluate (if at all) several program instances used in the experiment. The technical problems mainly rise from the use of XSB and how interaction between BPA and XSB is managed. With more efforts and time, it should be possible to overcome this problem.

7

Conclusion

We studied the properties and complexity results of logic programs with bounded predicate arities as presented in [Eiter et al., 2007]. From these results, we have learned that reasoning tasks for logic programs with bounded predicate arities have lower complexities, and in particular, that it is possible to perform the reasoning tasks within polynomial space. We have observed that current ASP solvers do not yet take advantage of this fact, and that specifically, evaluations of logic programs using current ASP solvers may still cause exponential space grounding, even for programs with bounded predicate arities. This manifests as the "grounding bottleneck" problem which can reduce efficiency, with respect to both time and space.

We have pursued an approach as suggested in [Eiter et al., 2007], based on meta-interpretation technique, which reduces an input program into a meta-interpreted form such that its evaluation in current ASP solvers will not cause exponential space grounding. However, we have discovered that the approach does not perform efficiently enough in practice. We thus explored different approaches, leading to the results we presented in this thesis.

The main result consists of three methods for evaluating logic programs with bounded predicate arities which stays within polynomial space. The first two methods deal with normal/HCF programs, and works in similar manners. The main principle used by the two methods is that, for a normal/HCF program with bounded predicate arities, it is possible to compute answer sets of the program by considering polynomially bounded subsets of the ground rules and compute *local answer sets* from these subsets. A local answer set can then be admitted as a global answer set of the program if it satisfies all the rules in the program, i.e., if it is closed under the rules. Using these methods, it can bee seen that full grounding of the input program at once is unnecessary, thus avoiding exponential space requirement associated with it.

We presented also an evaluation method to deal with non-HCF disjunctive programs with bounded predicate arities. The method consists of two main steps: 1) generate some models of the program which can potentially be an answer set and 2) check the models produced by the first step for minimality. Any model found by step 1) and satisfying the minimality check in step 2) is then returned as an answer set of the program. We have showed how these two steps can be reduced into the task of evaluating several programs such that the evaluation does not cause exponential space grounding when using current ASP solvers.

A framework architecture has been designed and presented for performing logic program evaluation using the three methods proposed. The framework allows for an easy understanding on how the methods might be implemented by decomposing them into smaller subtasks, each of which is performed by one framework component. We provided a detailed algorithmic description of each framework component and also described how the framework components cooperate with each other to achieve the overall goal. We have also studied how program modularity analysis and strongly-connected components decomposition are incorporated into the framework architecture to increase the overall efficiency of logic program evaluation using the three proposed methods.

Finally, we have developed an implementation of the evaluation methods which is designed according to the framework architecture. It uses DLV as an external ASP solver and utilized XSB as the Prolog engine to perform tasks related to program subsets generation. We tested the performance of the implementation in deciding answer set existence, and we compared it against claspD and DLV, two of the most efficient ASP solvers available at the moment. The experiment results have shown that the proposed methods can indeed perform evaluation under polynomial space, in contrast to the two ASP solvers, which clearly show exponential space grounding behavior. Moreover, we learned also that using the methods and framework architecture proposed can considerably increase time efficiency for many of test cases, as a result of avoiding the exponential space grounding. However, we concede that for some test cases, performance with respect to time consumption for our prototype system can be worse than the respective performance of claspD and DLV. Such cases are observed especially when the input program is inconsistent. Whereas claspD and DLV might be able to detect inconsistencies earlier using sophisticated techniques and heuristics, our evaluation methods does not provide any means for pruning the search space to arrive at inconsistency faster. They instead have to proceed in visiting many subsets of the ground rules and/or many local answer sets, before concluding that the program has no answer set.

7.1 Future Work

We believe we have achieved a promising result from this work, and especially, that the main goal of the research has been accomplished: to present evaluation methods for bounded predicate arities which stay within polynomial space, and which can perform reasonably well in some practical situations. However, there are still some issues left unaddressed, which can pave the way for further research into the topic. We list some of these in the following:

1. As the experiment results have shown, performance of the methods with respect to time consumption can be worse than the performance of current ASP solvers on the same input program, especially for the case where the input program is inconsistent, or having only few and relatively difficult to find answer sets. This might be due to the way ground program subsets and local answer sets are generated in the methods. If the program is inconsistent, then in the worst case, all possible subsets of the ground rules will be visited. This is of course, more expensive (time-wise) than generating all the ground rules at once, and perform evaluation directly on the generated ground rules. Future works might consider ways in which one can detect inconsistency earlier and avoid generating the ground program subsets, or possibly reducing the number of subsets that need to be visited.

2. The three methods proposed are, in one sense, more appropriate for decision-type problems, where one only needs to know whether a program has an answer set (is consistent) or not (is inconsistent). If the input program is consistent, most of the time, the three methods manage to find an answer set in less time and using less space than current ASP solvers. As pointed out in 1), it might not be the case when the program is inconsistent. However, for optimization-type problems, the situation might be even worse, since this type of problems always requires to have all answer sets visited and checked for optimality.

3. We have intentionally refrained from considering some of the extensions to the core language of logic programs commonly found in current ASP solvers. In particular, one of the basic extensions available in virtually all the current ASP solvers is the support for integer arithmetics. We do not consider how rules with integer arithmetics can be evaluated using the three methods proposed, since grounding a literal containing integer arithmetics requires a special treatment, which we could not fit nicely into the schema of our evaluation methods. In contrast to this, binary comparison predicates are easier since they follow the safety condition in Definition 2.1, and do not require special treatment during the grounding process.

   There are still other, more interesting extensions, such as: weak constraints [Buccafurri et al., 1997], aggregates [Dell'Armi et al., 2003] and cardinality constraints [Simons et al., 2002] which we have not considered as well. It should be an interesting research topic to consider how one might integrate evaluation methods, such as the ones we propose, into systems capable of performing evaluation of programs with these extensions.

4. Regarding implementation details, we have seen some problems surfacing during the experiment. Most of these are due to the less-than-perfect interface with the back-end Prolog system, XSB. For an implementation-oriented research, one can investigate better methods at interfacing with such Prolog systems, to allow more efficient and more reliable memory management and to increase overall (time) performance. One can also investigate the possibilities of using other Prolog systems, or perhaps using other than Prolog (ASP solvers) to perform the tasks related to ground program subsets generation.

5. Finally, a more ambitious topic to consider would be on how to integrate the proposed methods into current ASP solvers themselves, as one of their native optimization procedures. The framework architecture given in Chapter 5 assumes that the three evaluation methods are to be implemented *externally* from current ASP solvers. This approach suffers from the overhead of communications between the system and the external ASP solver. However, there is nothing to prevent us from incorporating the evaluation methods into one of the current ASP solvers, allowing the solver to avoid exponential space grounding by itself. This system will be more likely to have better performance since no communication overhead is imposed.

Bibliography

K. R. Apt, H. A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Inc., Washington DC, 1988. 2.2.3

R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.*, 12(1-2):53–87, 1994. 1.3, 1.3, 2.2.4, 2.5, 2.2.4, 2.3.2

F. Buccafurri, N. Leone, and P. Rullo. Adding Weak Constraints to Disjunctive Datalog. In *Proceedings of the 1997 Joint Conference on Declarative Programming APPIA-GULP-PRODE'97*, Grado, Italy, June 1997. 1, 3

F. Buccafurri, N. Leone, and P. Rullo. Enhancing disjunctive datalog by constraints. *IEEE Trans. Knowl. Data Eng.*, 12(5):845–860, 2000. 1

M. Cadoli, A. Giovanardi, and M. Schaerf. Experimental analysis of the computational cost of evaluating quantified boolean formulae. In *AI*IA '97: Proceedings of the 5th Congress of the Italian Association for Artificial Intelligence on Advances in Artificial Intelligence*, pages 207–218, London, UK, 1997. Springer-Verlag. ISBN 3-540-63576-9. 6.2.1

K. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978. 1.2

E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/502807.502810. 1.3, 2.3.2

T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003*, pages 847–852, Acapulco, Mexico, Aug. 2003. Morgan Kaufmann Publishers. ISBN 0-127-05661-0. 3, 3

J. Dix, J. U. Dix, G. Gottlob, W. Marek, and C. Rauszer. Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundamenta Informaticae*, 28:87–100, 1996. 1.3, 2.2.4, 2.2.4

C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub. Conflict-driven disjunctive answer set solving. In *KR*, pages 422–432, 2008. 2.2.6, 2.2.6

T. Eiter and G. Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In *Proceedings of the International Logic Programming Symposium (ILPS)*, pages 266–278. MIT Press, 1993. 1.3, 2.3.2, 2.3.2, 2.3.2

T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22:364–418, 1997. 1.3, 2.2.5, 2.3.2

T. Eiter, N. Leone, and D. Saccá. Expressive power and complexity of partial models for disjunctive deductive databases. *Theor. Comput. Sci.*, 206(1-2):181–218, 1998. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(97)00129-1. 1.3, 2.3.2

T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Computing preferred answer sets by meta-interpretation in answer set programming. *Theory Pract. Log. Program.*, 3(4):463–498, 2003. ISSN 1471-0684. doi: http://dx.doi.org/10.1017/S1471068403001753. 3.3, 3.3

T. Eiter, W. Faber, M. Fink, and S. Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):123–165, 2007. ISSN 1012-2443. doi: http://dx.doi.org/10.1007/s10472-008-9086-5. 1, 1.4, 2, 3, 3.1, 3.1.1, 3.1.2, 3.2, 3.3, 7

M. R. Garey and D. S. Johnson. *Computers and intractability*. Freeman, 1979. 1.5, 2.3, 2.3.1

M. Gebser, B. Kaufmann, and T. Schaub. The conflict-driven answer set solver clasp: Progress report. In *LPNMR*, pages 509–514, 2009. 2.2.6, 2.2.6

A. V. Gelder. Negation as failure using tight derivations for general logic programs. *J. Log. Program.*, 6(1&2):109–133, 1989. 2.2.3

M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *ICLP*, pages 579–597, 1990. 1.3, 2.1, 2.2

M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991. 1.3, 2.2

M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press. URL http://citeseer.ist.psu.edu/gelfond88stable.html. 1.3, 2.1, 2.2

T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J.-H. You. Unfolding partiality and disjunctions in stable model semantics. *ACM Trans. Comput. Logic*, 7(1):1–37, 2006. ISSN 1529-3785. doi: http://doi.acm.org/10.1145/1119439.1119440. 2.2.6

N. Leone, S. Perri, and F. Scarcello. Improving ASP instantiators by join-ordering methods. In *LPNMR '01: Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 280–294, London, UK, 2001. Springer-Verlag. ISBN 3-540-42593-4. 3.2

N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3): 499–562, 2006. 2.2.6, 3.2, 6.2.1, 6.2.1, 6.2.1

V. Lifschitz and H. Turner. Splitting a logic program. In *Principles of Knowledge Representation*, pages 23–37. MIT Press, 1994. 2.2.5, 2.8

F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. In *Eighteenth national conference on Artificial intelligence*, pages 112–117, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence. ISBN 0-262-51129-0. 2.2.6, 2.2.6

J. McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationary Office. URL `http://citeseer.ist.psu.edu/mccarthy68programs.html`. 1.1

J. L. McCarthy. Epistemological problems of artificial intelligence. In *IJCAI*, pages 1038–1044, 1977. 1.2

J. Minker. On indefinite databases and the closed world assumption. In D. W. Loveland, editor, *CADE*, volume 138 of *Lecture Notes in Computer Science*, pages 292–308. Springer, 1982. ISBN 3-540-11558-7. 1.3, 2.2

I. Niemela, P. Simons, and T. Syrjanen. Smodels: A system for answer set programming, 2000. URL `http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0003033`. 2.2.6

E. Oikarinen and T. Janhunen. Achieving compositionality of the stable model semantics for smodels programs. *Theory Pract. Log. Program.*, 8(5-6):717–761, 2008. ISSN 1471-0684. doi: http://dx.doi.org/10.1017/S147106840800358X. 2.2.5

C. H. Papadimitriou. *Computation Complexity*. Addison-Welsey, 1994. 2.3

S. Perri, F. Scarcello, G. Catalano, and N. Leone. Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence*, 51(2-4): 195–228, 2007. ISSN 1012-2443. doi: http://dx.doi.org/10.1007/s10472-008-9090-9. 3.2

T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *J. Autom. Reasoning*, 5(2):167–205, 1989. 2.2.3, 2.2.5

T. C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424, 1991. 1.3, 2.2, 2.2.3

R. Reiter. A logic for default reasoning. *Artif. Intell.*, 13(1-2):81–132, 1980. 1.2

K. A. Ross. Modular stratification and magic sets for datalog programs with negation. In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 161–171, New York, NY, USA, 1990. ACM. ISBN 0-89791-352-3. doi: http://doi.acm.org/10.1145/298514.298558. 2.2.5

K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 442–453, New York, NY, USA, 1994. ACM. ISBN 0-89791-639-5. doi: http://doi.acm.org/10.1145/191839.191927. 6.1

R. Schindlauer. *Answer-Set Programming for the Semantic Web*. PhD thesis, Technische Universität Wien, 12 2006. 2.2.5, 6.1.1

P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002. 2.2.6, 2.2.6, 3

J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989. ISBN 0-7167-8162-X. 3.2