Approximating Extended Answer Sets

Davy Van Nieuwenborgh[1] and **Stijn Heymans**[2] and **Dirk Vermeir**[3]

Abstract. We present an approximation theory for the extended answer set semantics, using the concept of an approximation constraint. Intuitively, an approximation constraint, while satisfied by a "perfect" solution, may be left unsatisfied in an approximate extended answer set. Approximations improve as the number of unsatisfied constraints decreases. We show how the framework can also capture the classical answer set semantics, thus providing an approximative version of the latter.

1 Introduction

The idea behind the answer set semantics for logic programs is both intuitive and elegant. Given a program P and a candidate answer set M, one computes a reduct program P_M of a simpler type for which a semantics $P_M^\star$ is known. The reduct P_M is obtained from P by taking into account the consequences of accepting the proposed truth values of the literals in M. The candidate set M is then an answer set just when $P_M^\star = M$, i.e. M is "self-producible".

In [17], the same reduction technique is applied to define the extended answer set semantics for simple logic programs, i.e. programs containing only classical negation. In contrast with the classical answer set semantics, extended answer sets are able to handle contradictory programs by allowing rules to be defeated, i.e. to leave certain rules unsatisfied.

The answer set semantics has been implemented in state-of-the-art solvers such as DLV[11] or SMODELS[14]. While these solvers perform well in many cases, they implement an "all or nothing" strategy, i.e. they do not provide the user with "partial" or "approximate" answer sets if there are insufficient resources for computing an exact solution. Clearly, given the inherent complexity of the problems tackled by such solvers, it would be desirable to have such an approximation facility, e.g. in cases where the available resources are limited. Furthermore, for many application areas obtaining such a "good enough" solution within a certain time limit, may be attractive, e.g. in diagnostic reasoning [18].

In this paper, we propose to provide an approximation theory for the extended answer set semantics, based on so-called approximation constraints. Intuitively, an approximation constraint is a constraint that should be satisfied by a perfect solution to a problem, but which may be left unsatisfied by an approximate solution. The more approximation constraints an approximate extended answer set satisfies, the "better" it approximates an exact solution. We will show that the proposed framework can also be used to approximate classical answer sets[4], which may lead to the development of "anytime" approximate answer set solvers.

The computation of extended answer sets for programs without constraints is very efficient, i.e. it takes quadratic time in the size of the program to compute an extended answer set for such programs, which implies that a first approximation can be computed very quickly. Although grounding adds an exponential[5] factor [6] to the computation in the worst case, the extended answer set semantics should perform much better in the average case. Moreover, the semantics allows for a "ground when needed" implementation, in contrast to solvers such as DLV [11] or SMODELS [14] which first completely ground the program, before computing a solution.

The above mentioned efficiency implies that the basic extended answer set semantics is not able to handle problems located above $\mathcal{P}$. However, adding approximation constraints to the semantics lifts the complexity up to include $\mathcal{NP}$-complete problems, placing the proposed semantics at the same level as the classical answer set semantics.

The remainder of the paper is organized as follows: Section 2 introduces the extended answer set semantics, while Section 3 presents the framework of approximation constraints. In Section 4 the relationship with classical answer programming is explored, while we briefly discuss some related work in Section 5. Finally, we conclude and give some directions for future research in Section 6. Due to space restrictions, all proofs are omitted, they can be found in the technical report [16].

2 Extended Answer Set Semantics

In this section, we introduce the extended answer set semantics for simple logic programs, i.e. logic programs with only classical negation, no disjunction in the head of rules and no constraints.

A *term* is a constant or a variable, where the former will be written lower-case and the latter upper-case. An *atom* is of the form $p(t_1, \ldots, t_n)$, $0 \leq n < \infty$, where p is an n-ary[6] predicate name and t_i, $1 \leq i \leq n$, are terms. A *literal* is an atom a or a classically negated atom $\neg a$.

Definition 1 *A **simple logic program** (SLP) is a countable set P of rules of the form $a \leftarrow \alpha$ where $\{a\} \cup \alpha$ is a finite set of literals*[7].

[1] Supported by the Flemish Fund for Scientific Research (FWO-Vlaanderen).

[2] Digital Enterprise Research Institute (DERI), University of Innsbruck, Austria. Email: stijn.heymans@deri.org

[3] Departement of Computer Science, Vrije Universiteit Brussel (VUB), Pleinlaan 2, B1050 Brussels, Belgium. Email: {dvnieuwe,dvermeir}@vub.ac.be

[4] Conversely, Theorem 4 in [17] shows that classical answer set programming can be used to compute extended answer set semantics.

[5] If we restrict the arities of predicates to small numbers, e.g. binary or ternary, it is shown in [8] that the additional complexity of grounding only adds one level of the polynomial hierarchy in contrast to the exponential factor in general.

[6] We thus allow for 0-ary predicates, i.e., *propositions*.

[7] Note that constraints, i.e. rules with empty heads, are not allowed for the moment.

A *ground* atom, literal, rule, or SLP does not contain variables. Substituting every variable in a SLP P with every possible constant in P yields the ground SLP $gr(P)$.

Example 1 *Grounding the SLP*

$$\neg p(c,a) \leftarrow \qquad p(X) \leftarrow \neg q(X,b) \qquad q(X) \leftarrow \neg p(X,a)$$

yields the SLP[8].

$$\begin{array}{lll} p(a) \leftarrow \neg q(a,b) & p(b) \leftarrow \neg q(b,b) & p(c) \leftarrow \neg q(c,b) \\ q(a) \leftarrow \neg p(a,a) & q(b) \leftarrow \neg p(b,a) & q(c) \leftarrow \neg p(c,a) \\ \neg p(c,a) \leftarrow & & \end{array}$$

In the rest of the paper we always assume ground SLPs and ground literals; to obtain the definitions for ungrounded SLPs, replace every occurrence of a SLP P by $gr(P)$, e.g., an extended answer set of an ungrounded SLP P is an extended answer set of $gr(P)$.

For a set of literals X, we use $\neg X$ to denote the set $\{\neg p \mid p \in X\}$ where $\neg\neg a \equiv a$. Further, X is said to be *consistent* if $X \cap \neg X = \emptyset$, i.e. X does not contain contradictory literals a and $\neg a$.

Definition 2 *The **Herbrand base** $\mathcal{B}_P$ of a SLP P is the set of all ground atoms that can be formed using the language of P. The set of all literals that can be formed with P, i.e. $\mathcal{B}_P \cup \neg\mathcal{B}_P$, is denoted by $\mathcal{L}_P$. An **interpretation** I of P is any consistent subset of $\mathcal{L}_P$.*

*A rule $r = a \leftarrow \alpha$ is **satisfied** by an interpretation I, denoted $I \models r$, if $a \in I$ whenever $\alpha \subseteq I$, i.e., if r is **applicable** ($\alpha \subseteq I$) then it must be **applied** ($\alpha \cup \{a\} \subseteq I$). The rule r is **defeated** w.r.t. I iff there exists an applied **competing rule** $\neg a \leftarrow \alpha'$ in P, such a rule is said to **defeat** r.*

Intuitively, in an extended answer set (see below), a rule $r\colon a \leftarrow \alpha$ may not be left unsatisfied, unless one accepts the opposite conclusion $\neg a$ which must itself be motivated by a competing applied rule $\neg a \leftarrow \alpha'$ that defeats r.

Example 2 *Consider the SLP P_1 containing the rules $\neg a \leftarrow$, $\neg b \leftarrow$, $a \leftarrow \neg b$, and $b \leftarrow \neg a$. For the interpretation $I = \{\neg a, b\}$ we have that I satisfies all rules in P_1 but one: $\neg a \leftarrow$ and $b \leftarrow \neg a$ are applied while $a \leftarrow \neg b$ is not applicable. The unsatisfied rule $\neg b \leftarrow$ is defeated by $b \leftarrow \neg a$.*

For a set of rules P, we use $P^\star$ to denote the unique minimal [15] model of the positive logic program consisting of the rules in P, where negative literals $\neg a$ are considered as fresh atoms. We can compute $P^\star$ using the immediate consequence operator $T_P(X) = \{l \in \mathcal{L}_P \mid l \leftarrow \beta \in P \wedge \beta \subseteq X\}$. Clearly, the operator T_P is monotonic, and $T_P^\infty(\emptyset) = P^\star$.

For the program of Example 2, we have that $P_1^\star = \{a, b, \neg a, \neg b\}$ is inconsistent. The following definition allows us to not apply certain rules when computing a consistent interpretation for programs such as P_1.

Definition 3 *The **reduct** $P_I \subseteq P$ of a SLP P w.r.t. a set of literals I contains just the rules satisfied by I, i.e. $P_I = \{r \in P \mid I \models r\}$.*

*An interpretation S is an **extended answer set** of P iff $P_S^\star = S$, i.e., S is **founded**, and all rules in $P \setminus P_S$ are defeated w.r.t. S.*

We use $\mathcal{ES}(P)$ to denote the set of all extended answer sets of P.

[8] Note that a variable X in a rule should be grounded with the same constant in that rule (either with a, b or c), while it may be grounded with other constants in other rules, i.e., the variables in a rule are considered local to the rule. One can, e.g., replace the above SLP by the equivalent program

$$\neg p(c,a) \leftarrow \qquad p(X) \leftarrow \neg q(X,b) \qquad q(Y) \leftarrow \neg p(Y,a)$$

Thus, the extended answer set semantics deals with inconsistencies in a simple yet intuitive way: when faced with contradictory applicable rules, just select one for application and ignore (defeat) the other. In the absence of extra information (e.g., a preference relation for satisfying certain rules at the expense of others [17]; or the approximation constraints introduced in this paper), this seems a reasonable strategy for extracting a consistent semantics out of inconsistent programs.

Reconsidering Example 2, it is easy to verify that P_1 has three extended answer sets, i.e. $M_1 = \{\neg a, b\}$, $M_2 = \{a, \neg b\}$ and $M_3 = \{\neg a, \neg b\}$. Note that e.g. $P_{1\,M_1} = P_1 \setminus \{\neg b \leftarrow\}$, i.e. $\neg b \leftarrow$ is defeated w.r.t. M_1.

While Definition 3 allows P_M, with M an extended answer set, to be a strict subset of P, it still maximizes the set of satisfied rules w.r.t. an extended answer set.

Theorem 1 *Let P be a SLP and let M be an extended answer set for P. Then P_M is maximal w.r.t. set inclusion among the reducts of founded interpretations of P.*

The reverse of the above theorem does not hold in general, as witnessed by the following example.

Example 3 *Consider the program P_2 containing the following rules.*

$$\neg a \leftarrow \qquad b \leftarrow \qquad \neg b \leftarrow \neg a$$

The interpretation $N = \{b\}$ is founded with $P_{2\,N} = \{b \leftarrow, \neg b \leftarrow \neg a\}$ which is clearly maximal since $P_2^\star$ is inconsistent. Still, N is not an extended answer set because $\neg a \leftarrow$ is not defeated.

The extended answer set semantics is universal.

Theorem 2 *Every SLP has extended answer sets.*

Moreover, the extended answer sets can be efficiently computed using an immediate consequence operator:

$$T_P^e(S) = \begin{cases} S \cup \{l\} & \text{if } r : l \leftarrow \alpha \in P \text{ and } \alpha \subseteq S \text{ and } \neg l \notin S \\ S & \text{otherwise} \end{cases}.$$

Clearly, the above operator is non-deterministic as the result depends on the order in which rules are applied. However, it can be shown that ${T_P^e}^\infty(\emptyset)$ always results in an extended answer set of P. The non-deterministic behavior of T_P^e greatly influences the complexity of its implementation. In the worst case an algorithm for T_P^e will run in time quadratic[9] in the size of the program, but on average, depending on the order in which rules get applied, the running time will be much better[10].

Finally, note that the above implies that an implementation that uses T_P^e does not need a fully grounded version of the input program but may instead operate on a "ground (a rule) as needed" basis, as illustrated by the following example.

[9] Note that, by a well-known result [6], grounding of the program may add an exponential (worst-case) factor to the total cost of computing an extended answer set for an ungrounded program. However, if the arities of the predicates are bounded by a small number, [8] shows that this exponential factor reduces to one level of the polynomial hierarchy.

[10] Also in the non-grounded case the average complexity will be much better than the exponential worst-case scenario as, depending on the sequence in which rules get applied, certain parts of the program need not to be (fully) grounded.

Example 4 *Consider the following non-grounded program.*

$$\begin{array}{cc} p(a) \leftarrow & q(a) \leftarrow \\ p(b) \leftarrow & q(b) \leftarrow \\ \neg q(X) \leftarrow p(X) & \neg p(X) \leftarrow q(X) \end{array}$$

A possible computation of $T_P^{e\,\infty}(\emptyset)$ could result in the extended answer set $\{p(a), p(b), \neg q(a), \neg q(b)\}$, e.g. by starting the computation with applying the fact rule $p(a) \leftarrow$, i.e. $T_P^e(\emptyset) = \{p(a)\}$. As a consequence, we can make a ground instantiation $\neg q(a) \leftarrow p(a)$ of the rule $\neg q(X) \leftarrow p(X)$, which an implementation can choose to apply in computing $T_P^e(\{p(a)\}) = \{p(a), \neg q(a)\}$. If we now do a similar reasoning for the fact $p(b) \leftarrow$ to obtain $T_P^e(\{p(a), \neg q(a)\}) = \{p(a), \neg q(a), p(b)\}$ and $T_P^e(\{p(a), \neg q(a), p(b)\}) = T_P^{e\,\infty}(\emptyset) = \{p(a), p(b), \neg q(a), \neg q(b)\}$, it is easy to see that an implementation does not need any grounded version of the rule $\neg p(X) \leftarrow q(X)$, while classical implementations that first ground everything will obtain two grounded versions $\neg p(a) \leftarrow q(a)$ and $\neg p(b) \leftarrow q(b)$.

Note that we do not conjecture that our approach is better in the worst-case scenario. We only argue that by interleaving the grounding and computation process, one can get better results, both in time and space, in the average case.

3 Approximation Constraints

Because of Theorem 2, checking whether a SLP has an extended answer set can be done in constant time, and thus the semantics is not very expressive when compared to e.g. classical answer set semantics (for which the problem is $\mathcal{NP}$-complete). In this section, we increase the expressiveness by allowing for constraints to appear in programs.

Intuitively, an *approximation constraint* is like a traditional constraint, i.e. a condition that a solution of a program should satisfy. However, approximation constraints may be left unsatisfied, e.g. due to lack of computation time. This induces a partial order on such "approximate" solutions, where better approximations satisfy more constraints. Combined with the results from Section 2, it then becomes possible to devise an incremental extended answer set solver that combines both the grounding process and the computation, i.e. a "ground when needed" computation[11].

Definition 4 *A **simple approximation logic program** (SALP) is a tuple (P, C), where P is a SLP and C is a set of **approximation constraints**, i.e. rules of the form $\leftarrow \alpha$, with α a finite set of literals*[12].

*For an interpretation I of P, an approximation constraint $c \in C$ is said to be **satisfied**, denoted $I \models c$, if $\alpha \not\subseteq I$; otherwise it is said to be **violated**, denoted $I \not\models c$. We use I_v^C to denote the set of approximation constraints in C that are violated w.r.t. I, i.e. $I_v^C = \{c \in C \mid I \not\models c\}$.*

*Extended answer sets of P are **approximate extended answer sets** of (P, C). An approximate extended answer set S of (P, C) is an **extended answer set** of (P, C) iff $S_v^C = \emptyset$, i.e. all approximation constraints are satisfied w.r.t. S.*

In the rest of this paper, we will use $\mathcal{ES}_n(P, C)$, with $0 \leq n \leq |C|$, to denote the set containing the approximate extended answer sets of (P, C) that violate less than $n + 1$ approximation constraints in C, i.e. $\mathcal{ES}_n(P, C) = \{S \in \mathcal{ES}(P) \mid |S_v^C| \leq n\}$. Clearly, we have that $\mathcal{ES}_0(P, C)$ corresponds to the extended answer sets of (P, C); and $\mathcal{ES}_0(P, C) \subseteq \mathcal{ES}_1(P, C) \subseteq \cdots \subseteq \mathcal{ES}_{|C|}(P, C) = \mathcal{ES}(P)$.

Example 5 *Consider the following SALP (P, C) with P*

$$\begin{array}{ccc} \neg a \leftarrow & \neg b \leftarrow & \neg c \leftarrow \\ b \leftarrow \neg a & a \leftarrow \neg b & c \leftarrow a \end{array}$$

and the following approximation constraints C

$$\leftarrow \neg a, \neg b \qquad \leftarrow a \qquad \leftarrow c$$

We have four extended answer sets for P, i.e. $\mathcal{ES}(P) = \{S_1, S_2, S_3, S_4\}$, with $S_1 = \{\neg a, \neg b, \neg c\}$, $S_2 = \{\neg a, b, \neg c\}$, $S_3 = \{a, \neg b, \neg c\}$ and $S_4 = \{a, \neg b, c\}$. One can verify that $\mathcal{ES}_2(P, Q) = \{S_1, S_2, S_3, S_4\}$, $\mathcal{ES}_1(P, C) = \{S_1, S_2, S_3\}$ and $\mathcal{ES}_0(P, C) = \{S_2\}$. Thus S_2 is the only extended answer set of the above SALP (P, C).

In general, we may consider a partial order $\preceq$ on extended answer set approximations which is such that better approximations are smaller. i.e. $S_1 \prec S_2$[13] iff S_1 is a better approximation of an extended answer set than is S_2. It seems reasonable to demand that any such preference order satisfies $S_1 \preceq S_2 \Rightarrow |{S_1}_v^C| \leq |{S_2}_v^C|$, i.e. better approximations satisfy more approximation constraints; and that "full" extended answer sets correspond to $\prec$-minimal approximations.

Two obvious (others are possible) candidates that satisfy this requirement are:

- cardinality, i.e. $S_1 \preceq S_2$ iff $|{S_1}_v^C| \leq |{S_2}_v^C|$; and
- subset, i.e. $S_1 \preceq S_2$ iff ${S_1}_v^C \subseteq {S_2}_v^C$

which are also used in other problem areas, such as diagnostic systems [13], that deal with preference on candidate solutions.

Note that, if $\mathcal{ES}_0(P, C) = \emptyset$, i.e. there are no extended answer sets, and depending on the preference relation used, a $\prec$-minimal approximate extended answer set S may be useful, e.g. to provide insight, via the set of violated constraints S_v^C, about possible problems with the program. In this way a semantical debugging scheme for (extended) answer set programming can be defined, a concept which has already been explored in the context of prolog [3, 7, 12], but still needs better exploration in the former setting.

The following result sheds some light on the complexity of reasoning with approximation constraints. As $\mathcal{ES}_{|C|}(P, C) = \mathcal{ES}(P)$, we will only consider the cases $\mathcal{ES}_n(P, C)$ with $n < |C|$ (note that this implies that $|C| \geq 1$).

Theorem 3 *Let (P, C) be a grounded SALP and take i such that $0 \leq i < |C|$. Deciding whether there exists an approximate extended answer set $S \in \mathcal{ES}_i(P, C)$ is $\mathcal{NP}$-complete.*

Again an exponential factor [6] or, in case of small bounded predicate arities, an additional level of the polynomial hierarchy [8] has to be added in the case of non-ground SALPs, i.e. either NEXPTIME-complete or Σ_2^P-complete respectively.

The above theorem implies that the semantics already gets its full computational complexity with a single constraint, which is not a surprise as we can replace each constraint $\leftarrow \alpha \in C$ with a rule $inconsistent \leftarrow \alpha$ in P and take $C = \{\leftarrow inconsistent\}$. However, the above complexity result is a worst-case scenario (just as

[11] Note that current answer set solvers lack this support for "ground when needed" (see Section 4) and compute first the complete grounding, instead.

[12] Note that we allow non-ground constraints. However, grounding a SALP can be done by computing $gr(P \cup C)$ and splitting up the result in two sets, i.e. the ground rules and the ground constraints. Again, we assume grounded SALPs in the rest of the paper.

[13] As usual, $S_1 \prec S_2$ iff $S_1 \preceq S_2$ and not $S_2 \preceq S_1$.

with grounding), i.e. in an implementation we may expect, in the average case, that more constraints require more computation time.

Finally, the framework developed above can be used to implement an incremental extended answer set solver, i.e. a solver that produces an approximate solution and then, as long as resources permit, produces successively better approximations. Given enough resources, this approach will eventually result in an extended answer set, i.e. an approximation that satisfies all constraints.

Intuitively, such an implementation will start by computing an extended answer set S along the lines of the immediate consequence operator presented in Section 2. Afterwards, a backtracking algorithm will try to improve the approximation by selecting a constraint to satisfy, while maintaining the partial ordering defined on approximations.

4 Classical Answer Set Approximation

Theorem 3 indicates that the semantics presented in Section 3 is computationally equivalent to the classical answer set semantics [9] for non-disjunctive programs containing negation as failure in the body of rules. Here we show how the classical answer set semantics can be effectively translated to approximate extended answer sets, thus making it possible to compute classical answer sets via successive approximations.

We briefly introduce the answer set semantics for semi-negative programs. Such programs are build using atoms and naf-atoms, i.e. for an atom a, we use *not* a to denote its negation-as-failure (naf) version, which intuitively means that *not* a is true when a is not true. A semi-negative logic program P is a countable set of rules of the form $a \leftarrow \beta$ where a is an atom and β is a finite set of (naf-)atoms. An interpretation I for a semi-negative program P is a subset $I \subseteq \mathcal{B}_P$. An atom a is satisfied w.r.t. I, denoted $I \models a$ if $a \in I$; while a naf-atom *not* a is satisfied w.r.t. I, i.e. $I \models$ *not* a, when $I \not\models a$. The answer set semantics for semi-negative programs is defined in two steps.

First, consider programs without naf-atoms. For such a naf-free program P, an interpretation I is an answer set iff $P^\star = I$. Next, for semi-negative programs, we use the Gelfond-Lifschitz transformation [9]. The idea behind this transformation is, for a given program P and a candidate solution S, to remove all naf constructs w.r.t. S and then check if the candidate solution is supported by the reduct program. Formally, for a semi-negative program P and a candidate solution S, the GL-reduct of P w.r.t. S, denoted P^S, is obtained from P by (a) removing each rule $a \leftarrow \beta$ with *not* $b \in \beta$ and $b \in S$, and (b) remove all naf-atoms from the remaining rules. Clearly, the program P^S is free from negation as failure. An interpretation S is an answer set of a semi-negative program P iff S is an answer set of P^S, i.e. $P^{S\star} = S$. We use $\mathcal{AS}(P)$ to denote the set containing all answer sets of P.

Example 6 *Consider the following semi-negative program.*

$$a \leftarrow \textit{not } b \qquad b \leftarrow \textit{not } a$$
$$c \leftarrow a \qquad d \leftarrow b$$

Intuitively, the above program represents an exclusive choice between a or b and depending on this choice, the program derives an additional atom c or d resp. Indeed, one can verify that the above program has two answer sets, i.e. $S_1 = \{a, c\}$ and $S_2 = \{b, d\}$. E.g., the reduct P^{S_1} contains the rules $\{a \leftarrow \quad c \leftarrow a \quad d \leftarrow b\}$, for which $P^{S_1\star} = \{a, c\} = S_1$.

We construct, for a semi-negative logic program P, a SALP $L(P) = (Q, C)$ such that the extended answer sets of $L(P)$, i.e. elements of $\mathcal{ES}_0(Q, C)$, are in one-to-one correspondence with the answer sets of P.

In the following definition we use α' to denote the set of literals obtained from a set α where each naf-atom *not* a from α is replaced by $\neg a$. The notation is further extended to rules and programs.

Definition 5 *Let P be a semi-negative program. The SALP $L(P) = (Q, C)$ is defined by $Q = P' \cup \{\neg a \leftarrow \mid a \in \mathcal{B}_P\}$ and $C = \{\leftarrow \beta', \neg a \mid a \leftarrow \beta \in P\}$.*

Intuitively, we simulate negation as failure in two steps. First, we introduce negation as failure explicitly in Q using classical negation by introducing a fact $\neg a$ for each atom $a \in \mathcal{B}_P$. Secondly, we assert approximation constraints in C to enforce the satisfaction of the original rules in the program P: a rule $a \leftarrow \beta' \in Q$ (i.e. $a \leftarrow \beta \in P$) is only satisfied by an interpretation iff the approximation constraint $\leftarrow \beta', \neg a \in C$ is satisfied w.r.t. that interpretation.

Example 7 *Reconsider the program from Example 6. Its SALP version $L(P) = (Q, C)$ is defined by the SLP Q*

$$\neg a \leftarrow \qquad \neg b \leftarrow \qquad \neg c \leftarrow \qquad \neg d \leftarrow$$
$$a \leftarrow \neg b \qquad b \leftarrow \neg a \qquad c \leftarrow a \qquad d \leftarrow b$$

and the set of constraints C

$$\leftarrow \neg b, \neg a \qquad \leftarrow \neg a, \neg b \qquad \leftarrow a, \neg c \qquad \leftarrow b, \neg d$$

Consider the following extended answer sets of Q, i.e. $S'_1 = \{a, \neg b, c, \neg d\}$, $S'_2 = \{\neg a, b, \neg c, d\}$ and $S'_3 = \{a, \neg b, \neg c, \neg d\}$. One can check that $\mathcal{ES}_0(Q, C) = \{S'_1, S'_2\}$ and clearly $S_1 = S'_1 \cap \mathcal{B}_P$ and $S_2 = S'_2 \cap \mathcal{B}_P$. On the other hand, $S'_3 \in \mathcal{ES}_1(Q, C)$, as $c \leftarrow a \in C$ is violated. However, one can easily see that $S_3 = S'_3 \cap \mathcal{B}_P$ is an answer set of the program $P \setminus \{c \leftarrow a\}$.

The behavior outlined in the previous example is confirmed in general by the following theorem.

Theorem 4 *Let P be a semi-negative logic program and consider $L(P) = (Q, C)$ as defined in Definition 5. Then, for $S \subseteq \mathcal{B}_P$,*

$$S \cup \{\neg l \mid l \in \mathcal{B}_P \setminus S\} \in \mathcal{ES}_0(Q, C) \Longleftrightarrow S \in \mathcal{AS}(P) \ ,$$

i.e. the answer sets of P are in one-to-one correspondence with the extended answer sets of $L(P)$.

Further, for i, with $0 < i \leq |C|$, we have, with

$$S \cup \{\neg l \mid l \in \mathcal{B}_P \setminus S\} \in (\mathcal{ES}_i(Q, C) \setminus \mathcal{ES}_{i-1}(Q, C)) \Leftrightarrow S \in \mathcal{AS}(P \setminus S_v^C) \ ,$$

where, abusing notation, $P \setminus S_v^C$ denotes the subset of rules in P for which the corresponding approximation constraint in C is satisfied w.r.t. S.

The above result shows that the approximation framework for extended answer sets is able to handle the classical answer set semantics in the sense that, each time a better approximation for $L(P)$ is computed, it is also an answer set for a larger subset of the program P. Interestingly, the algorithm does not rely on negation as failure, but uses only classical negation and the ability to leave rules unsatisfied, i.e. defeated.

Finally, the above results may help to resolve another issue regarding current answer set solvers such as DLV [11] and SMODELS [14]. Indeed, these solvers require that a program first be completely grounded before the actual answer set computation starts. With approximations based on extended answer sets, this is probably not necessary, since the computation of an extended answer set may operate on a "ground (a rule) as needed" basis (Section 2).

5 Related Work

The idea of an approximation theory in logic and logic programming is not new. A good example of this is the "anytime" family of reasoners for propositional logic [4]. Intuitively, such a system consists of a sequence $\vdash_0, \vdash_1, \ldots, \vdash_c$ of inference relations such that each $\vdash_i$ is at least as good[14] as $\vdash_{i-1}$ and there exists a complete inference relation $\vdash_c$ for the problem. Using such an anytime reasoner, one starts the computation with the inference relation $\vdash_0$ and as long as there is computation time left, the inference relations $\vdash_1, \vdash_2, \ldots$ are applied to obtain better approximations. When the complete reasoner $\vdash_c$ is reached during the computation, one had enough resources available to compute a completely correct solution.

This idea is e.g. applied in [5] to obtain an anytime reasoner for Boolean Constraint Propagation, which is used in [18] to obtain an approximation theory for diagnostic reasoning.

In [1] a first attempt is made to devise a framework for interactive answer set programming. The idea is to find which part of a program needs to be recomputed in order to find a new answer set in case a single rule is added to the program. Using this framework, one can obtain a similar approximative version of answer set programming as the one presented in Section 4 by starting the computation with a single rule and then subsequently adding new rules. However, the framework presented in this paper is more suitable in the sense that we can start the computation with the whole program, not just a single rule, and obtain a first approximation that in the average case will satisfy more than one rule in the program under consideration.

Weak constraints were introduced in [2] as a relaxation of the concept of a constraint. Intuitively, a weak constraint is allowed to be violated, but only as a last resort, meaning that one tries to minimize the set of violated constraints (e.g. using either subset minimality or cardinality minimality). The main difference of this approach with ours, is that we use approximation constraints to find better approximations, as long as we have resources available, of a final solution that satisfies all approximation constraints (and which will be reached if we have enough time); while the weak constraints in [2] serve to differentiate between the answer sets of the program without weak constraints, by posing additional conditions on those answer sets that do not necessarily have to be satisfied. Also note that the weak constraints from [2] have a much higher complexity than the framework presented in this paper, i.e. Δ_2^P for non-disjunctive programs, or the second level of the polynomial hierarchy.

6 Conclusions and Directions for Further Research

We presented a first attempt at an approximation theory, by means of approximation constraints, for the extended answer set semantics of programs containing only classical negation. Further, we showed how classical answer sets can be approximated using the proposed framework.

In future work, we will implement an approximate extended answer set solver that "grounds when needed". In our implementation, we will try to incorporate some of the ideas developed in the Platypus system [10], i.e. a system to compute answer sets in a distributed environment, to obtain an implementation that can be used in a distributed setting. One topic for further research in this context is e.g. how on the fly rule instantiations (which result from the "ground when needed" process) have to be propagated to the other participants in the distributed computation process.

[14] Note that each $\vdash_i$ has to be sound and tractable, but not necessarily complete.

Furthermore, we also plan to look into the related problems of profiling (finding out which parts of a program are hard to compute) and debugging (finding out which parts of a program are wrong) for (extended) answer set programming. Finally, the present approach could be generalized to support user-defined preference relations ("hints") on approximation constraints which could be used to influence the approximation process.

REFERENCES

[1] M. Brain, R. Watson, and M. De Vos, 'An interactive approach to answer set programming', in *Proc. of the 3rd Intl. Workshop on ASP: Advances in Theory and Implementation (ASP05)*, volume 142 of *CEUR Workshop Proceedings*, pp. 190–202, (2005).

[2] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo, 'Strong and weak constraints in disjunctive datalog', in *Proc. of the 4th Intl. Conference on Logic Programming (LPNMR '97)*, pp. 2–17, (1997).

[3] M. Calejo and L.M. Pereira, 'Declarative source debugging', in *Proc. of the 5th Portuguese Conf. on AI (EPIA91)*, volume 541 of *LNCS*, pp. 237–249. Springer, (1991).

[4] Mukesh Dalal, 'Anytime families of tractable propositional reasoners', in *Proc. of the 4th Intl. Symp. on Artificial Intelligence and Mathematics (AI/MATH96)*, pp. 42–45, (1996).

[5] Mukesh Dalal, 'Semantics of an anytime family of reasoners', in *Proceedings of the 12th European Conference on Artificial Intelligence*, pp. 360–364. John Wiley and Sons, (1996).

[6] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, 'Complexity and Expressive Power of Logic Programming', *ACM Computing Surveys*, **33**(3), 374–425, (2001).

[7] Mireille Ducassé, 'A pragmatic survey of automated debugging.', in *Proc. of the Intl. Workshop on Automated and Algorithmic Debugging (AADEBUG93)*, volume 749 of *LNCS*, pp. 1–15. Springer, (1993).

[8] Thomas Eiter, Wolfgang Faber, Michael Fink, Gerald Pfeifer, and Stefan Woltran, 'Complexity of model checking and bounded predicate arities for non-ground answer set programming', in *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004)*, pp. 377–387. AAAI Press, (2004).

[9] Michael Gelfond and Vladimir Lifschitz, 'The stable model semantics for logic programming', in *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pp. 1070–1080, Seattle, Washington, (August 1988). The MIT Press.

[10] Jean Gressmann, Tomi Janhunen, Robert E. Mercer, Torsten Schaub, Sven Thiele, and Richard Tichy, 'Platypus: A platform for distributed answer set solving.', in *Logic Programming and Nonmonotonic Reasoning, 8th International Conference (LPNMR 2005)*, volume 3662 of *LNCS*, pp. 227–239. Springer, (2005).

[11] N. Leone, G. Pfeifer, W. Faber, F. Calimeri, T. Dell'Armi, T. Eiter, G. Gottlob, G. Ianni, G. Ielpa, C. Koch, S. Perri, and A. Polleres, 'The dlv system', in *Proc. of the Eur. Conf. on Logics in AI (JELIA2002)*, volume 2424 of *LNCS*, pp. 537–540. Springer, (2002).

[12] Lee Naish, 'A declarative debugging scheme', *Journal of Functional and Logic Programming*, **1997**(3), (1997).

[13] Raymond Reiter, 'A theory of diagnosis from first principles', *Artificial Intelligence*, **32**(1), 57–95, (1987).

[14] Patrik Simons, Ilkka Niemelä, and Timo Soininen, 'Extending and implementing the stable model semantics', *Artificial Intelligence*, **138**(1-2), 181–234, (2002).

[15] M. H. van Emden and R. A. Kowalski, 'The semantics of predicate logic as a programming language', *Journal of the Association for Computing Machinery*, **23**(4), 733–742, (1976).

[16] Davy Van Nieuwenborgh, Stijn Heymans, and Dirk Vermeir. Approximating extended answer sets. Technical report, Vrije Universiteit Brussel, Dept. of Computer Science, 2006, http://tinf2.vub.ac.be/~dvnieuwe/ecai2006technical.ps.

[17] Davy Van Nieuwenborgh and Dirk Vermeir, 'Preferred answer sets for ordered logic programs', in *Proc. of the Eur. Conf. on Logics in Artif. Intell. (JELIA2002)*, volume 2424 of *LNCS*, pp. 432–443. Springer, (2002).

[18] A. Verberne, F. van Harmelen, and A. ten Teije, 'Anytime diagnostic reasoning using approximate boolean constraint propagation', in *Proc. of the 7th Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR 2000)*, pp. 323–332. Kaufman, (2000).