Towards Incremental Computation of the Answer Set Semantics: Preliminary Report

Martin Brain Marina De Vos[*]
Department of Computer Science
University of Bath
Bath, United Kingdom
{mjb,mdv}@bath.ac.uk

Abstract

In this paper a new algorithm for computing answer sets in an evolving environment is presented. Currently, each time any new information is 'learnt', all answer sets must be recomputed, this algorithms reduces the amount of recomputation needed, making interactive, learning applications possible. Rather than computing completely from scratch, our incremental answer solver, IDEAS, stores information, i.e. the answer sets, of previous computations. This is intended for application domains in which the $AnsProlog^*$ program is developed over the course of the program but answer sets are required throughout. Apart from being the first incremental solver, IDEAS is also the first parallel one. In this paper we discuss both the theoretic aspects as the implementation side of our system.

1 Introduction

$AnsProlog^*$[2] or answer set programming is a new approach to the ideas of knowledge representation, real world reasoning and declarative problem solving that has been under development for the past 10-15 years. It has a variety of powerful and useful features such as supporting non monotonic reasoning, handling of multiple possible world views, both classical and intuitistic negation and the ability to characterise and reason about partial and incomplete information. A variety of extensions to the core language support constructs such as preference (both weighted[15] and ordered[4, 5, 9]), choices (exclusive[8] and inclusive[6]) and belief operators[12].

Critically the semantics of $AnsProlog^*$ are simple, expressive and computable. Each set of rules gives a (possibly empty) set of answer sets. These may be thought of as world views that are consistent with the information given; inconsistency yields no answer sets. Over the past 5 or so years a number of answer set solvers (software tools that take text based representation of an $AnsProlog^*$ program and compute the answer sets), have been implemented. The most popular and widely used ones are Smodels[10], Helsinki University of Technology, and DLV[14], created at the Technical University of Vienna and the University of Calabria. These can be run using standard commodity computer hardware and already have comparable (if not better)

[*]Contacting author. This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging technologies under the IST-2001-37004 WASP project.

performance to older and more established logic programming systems[16] both in terms of computation time and scalability.

However, very few of the potential application areas of knowledge representation technology are static. For example a diagnostic system will not only be used once on any given problem. An initial computation will give a variety of potential scenarios, additional test can then add to the knowledge base and determine the exact status of the system. In this scenario static problem solving is of interest but of much less use than strategies able to evolve. Thus, many of the applications of $AnsProlog^*$ involve computing the answer sets of a sequence of similar rule sets. The current solution is to simply run each rule set through a traditional answer set solver independently. However this approach takes time, making interactive applications with larger rule sets diffi cult and cumbersome to use.

This document presents, up to our knowledge, the fi rst results on incremental answer set solvers. For this paper, we restrict to algorithms dealing with updates of adding or subtracting one rule. Critically in many cases this is considerably simpler than just computing the answer sets of the new program. This allows the $AnsProlog^*$ program to develop and grow in the way conventional databases or bases of expertise would. The fi rst release of an implementation of these ideas as an interactive development and evaluation environment is available from `http://www.cs.bath.ac.uk/~mdv/incremental/`
It features the fi rst parallel solver implementation and lazy (on demand) computation along with slack time computation to give the fastest interactive response times.

It is important to note that the presented algorithms in no way changes the syntax or the semantics of the programs. The answer sets of a new rule set given by the incremental algorithm is exactly the same as the one given by a traditional answer set solver. Also the algorithms do not attempt to produce a solution method of lower theoretical complexity; in the worst case scenario all answer sets have to be recomputed.

2 Answer Set Programming

Only a brief overview is given here, readers are advised to refer to [2] for a more in depth coverage of the motivation, semantics, properties and computation of $AnsProlog^*$ programs. What is discussed here is also a syntactic and functional subgroup of $AnsProlog^*$, referred to as $AnsDatalog^{\perp}$[1].

An $AnsDatalog^{\perp}$ program is made up of a series of rules. Each rule has the form:
$A_0 \leftarrow A_1, \ldots, A_n, \mathbf{not}\ A_{n+1}, \ldots, \mathbf{not}\ A_m$.

Where A_0 is an atom or $\perp$ and A_i for $i \in [1, m]$ are atoms. A_0 is the head of the rule, denoted $H(r)$ for rule r and $\{A_1, \ldots, A_m\}$ is the body, denoted $B(r)$. The intuition for this rule is that if all of $A_1, \ldots, A_n$ are known and none of $A_{n+1}, \ldots, A_m$ are known then A_0 is considered to be known (in the case that A_0 is $\perp$, this indicates a contradiction).

When speaking about the status of rules with respect to a given set of atoms the terms *applicable* and *applied* are used. A rule is said to be applicable with respect to

[1]The theory also holds for $AnsDatalog^{\neg,\perp}$ as classical negation can easily be replaced.

a set if all of $A_1, \dots, A_n$ and none of $A_{n+1}, \dots, A_m$ are in the set. It is applied if it is applicable and A_0 is also in the set.

A wide variety of different semantics for logics of this form have been developed[7] as well as a variety of characterisations of the semantics of $AnsProlog^*$. In this paper we shall use the characterisation given by [13]. This is divided into two sections, the semantics of programs that do not contain negation and a semantic criterion and reduct for removing negation.

Programs without negation (also referred to as $AnsDatalog^{-\mathbf{not}}$) each have one answer set which is given by the logical closure of the rule set, i.e. starting with the facts (rules that have no body and are thus not dependent on anything), recursively build a set of anything that can be concluded using a rule who's body is in the set.

To remove negations the Gelfond-Lifschitz reduct (or transformation)[11] is used. To reduce a program with respect to a set of atoms S:

- Removing every rule that contains $\mathbf{not}\ p$ in the body if $p \in S$
- Removing all remaining negative literals (i.e. $\mathbf{not}\ q$) from the rules

The answer sets of the program are the sets of atoms S such that S is the answer set of the reduced program.

In short the answer sets of a program can be thought of as all of the possible world views that can be supported by the rules. For example, program $P = \{a \leftarrow b\ ;\ c \leftarrow \mathbf{not}\ d, a\ ;\ d \leftarrow \mathbf{not}\ c, a\ ;\ b\ ;\ e \leftarrow d\}$ has two answer sets $\{a, b, c\}$ and $\{a, b, d, e\}$. When reduced with respect to $\{a, b, c\}$, only one rule is removed given the program $\{a \leftarrow b\ ;\ c \leftarrow a\ ;\ b\ ;\ e \leftarrow d\}$ which has the answer set $\{a, b, c\}$ (thus making it an answer set of P). Note that e is not included in the answer set of the reduced program as there is no way of concluding d and so the rule giving e cannot be used. On the other hand if P is reduced by $\{a, b, e\}$ then the following program is obtained $\{a \leftarrow b\ ;\ c \leftarrow a\ ;\ d \leftarrow a\ ;\ b\ ;\ e \leftarrow d\}$ which has the answer set $\{a, b, c, d, e\}$, which is not the same as the set used to perform the reduct and thus not an answer set of P.

Notice that every rule in a program is either applied or not applicable with respect to its answer sets. The converse is not true, consider the set $\{a, b, c, e\}$ and the program P.

3 Theoretic Foundation

This section presents the foundations of the incremental algorithms. An outline of the additive algorithm is provided in the next section. Full details of the theory can be found in [3].

The fi rst key point is that the complexity and effi ciency of computing alterations to answer sets, as versus recomputing varies considerably with the rule that is being added and what it is being added to. For example, using the program P from the previous section; adding $e \leftarrow b$ will only add e to every answer set that doesn't already contain it, as e is not used in the body of any rule. Likewise removing $e \leftarrow d$ will have a minimal effect. Adding $\perp \leftarrow e, d$ top the initial program would simply remove the second answer set. However removing the first rule, $a \leftarrow b$, would cause major changes as the exclusive choice between c and d will not be made.

Definition 1 *Let P be a set of rules and a an atom then. We define support and usage as follows:*

- $support(a, P) = \{r \in P | H(r) = a\}$
- $usage(a, P) = \{r \in P | a \in B(r) \lor \textbf{not } a \in B(r)\}$

Intuitively the support of an atom in a given set of rule is the set of rules which could support it within a given answer set. Usage is the number of rules in which it appears. They can be used to easily infer some basic facts; for example if $support(a, P) = \emptyset$ then a will not appear in any answer set.

Unfortunately adding rules to a rule set is not as simple as just adding the head atom in answer sets where the rule is applicable. Adding atoms to an answer set may in turn make or stop other rules being applicable and thus require other atoms to be added or removed. These are the implications of adding / removing the original atom. The positive implications are the atoms that have to be added and the negative implications are the atoms that need to be removed.

Definition 2 *Let P be a program, A a set of atoms and $a \notin A$ is the atom to be added into A. Let $P_{A,+} \subset P$ be the set of rules that are applicable with respect to A and $P_{A,-} = P \setminus P_{A,+}$ (The complementary set of rules that are not applicable).*

The positive implications I^+ and negative implications I^- are defined as:

- $I(+, P, A, +a) = \{H(r) | \exists r \in usage(a, P) \cdot r \in P_{A,-} \land r \in P_{A \cup \{a\},+}\}$
- $I(-, P, A, +a) = \{H(r) | \exists r \in usage(a, P) \cdot r \in P_{A,+} \land r \in P_{A \cup \{a\},-}\}$

Intuitively these are the set of heads of any rule that becomes applicable by adding a and anything that can only be concluded by a rule depending on $\mathrm{not\ a}$.

The implications of removing an atom are defined in an analogous manner.

For compactness the following notation is introduced for denoting the implications of adding and subtracting an atom:

- $I(P, A, +a) := (\ I(+, P, A, +a), I(-, P, A, +a)\ )$
- $I(P, A, -a) := (\ I(+, P, A, -a), I(-, P, A, -a)\ )$

Handling implications is a non trivial problem as each one can give rise to further implications and they may also be linked so order of resolution can matter. For example, consider the program $P = \{a \leftarrow c, \textbf{not } b\ ;\ b \leftarrow c, \textbf{not } a\ ;\ c \leftarrow d\}$ which has only one answer set $A = \emptyset$. Adding the rule $d \leftarrow$ gives the following sequence of implications:

- $I(P, \emptyset, +d) = (\{c\}, \emptyset)$
- $I(P, \{d\}, +c) = (\{a, b\}, \emptyset)$

Adding d leads to a further implication, c must be added. However adding c leads to a more complex situation. Adding a will stop b being an implication and vica versa. The final answer sets of $P \cup \{d \leftarrow\}$ are $\{a, c, d\}$ and $\{b, c, d\}$. Implications are the main complexity in incremental algorithms, handling implications effectively is mostly solving the incremental problem.

Dependency graphs showing the relation between rules are a common tool in answer set semantic research. Past uses have included the theoretical basis for stratified logic programs and some other functional[2] subclasses of $AnsProlog^*$ and as the basis for computing answer sets[1]. They have been characterised in a variety of ways with different degrees and displays of aggregation of information[3], thus we give a definition of the exact type required here rather than referencing a standard definition.

Definition 3 *Let P be a program, its dependency graph $D = (N, L)$ where N is a set of nodes and $L \subset N \times N \times P$ is a set of directed links annotated with rules, is constructed as follows:*

- $N = \{n_a | a$ *an atom*$\}$
- $L = \{(n_a, n_b, r) | \forall r$*usage*$(a, P) \cap$ *support*$(b, P)\}$

Intuitively a link is created from a to b if there is a rule that has a or **not** a in the body and b in the head. Nodes are annotated with atoms [4]. This means that for every rule r there will be $|B(r)|$ links to $H(r)$. Self referential rules can easily be removed without the changing the semantics. Therefore, we will assume that they have been removed before starting the solving process, and thus no node links directly to itself.

For these graphs to be of any use, there have to be ways of extracting information from them. Thus a definition of reachability is presented.

Definition 4 *Let $D = (N, L)$ be the dependency graph of program P and let a be an atom. The reachability set after n steps from a is notated as $R^n(a)$ and defined as*

- $R^1(a) = \{b \in \mathcal{B}_P | \exists (n_a, n_b, r) L\}$
- $R^n(a) = \{b \in R^1(c) | c \in R^{n-1}(a)\} \cup R^{n-1}(a)$

This is the obvious definition, the set of nodes which are reachable after travelling along n links. The following propositions give basic properties of the reachability function and link it to the concept of implication.

Proposition 1 *Let P be a program, D its dependency graph and a and atom then there exists n_a s.t. $R^{n_a}(a) = R^{n_a+1}(a)$*

For clarity some simple notation is introduced. In the context of the preceding proposition $R^{n_a}(a)$ is notated as $R^\omega(a)$. If $b \in R^1(a)$, b is said to be directly reachable from a, while if $b \in R^\omega(a)$ it is just said to be reachable from a.

[2] As opposed to syntactic subclasses such as $AnsProlog^{\neg,\perp}$

[3] The link between rules, heads and bodies is essentially three non orthogonal directions of information, graphs are essentially a projection of this information onto a two dimensional paradigm, hence the number of fundamentally equivalent ways of expressing the relation. The choice between them is based entirely on which way the information is to be accessed.

[4] atoms will be used interchangeably with ‘the node representing the atom’

Proposition 2 *Let P be a program, $D = (N, L)$ it's dependency graph, a an atom and A a set of atoms then:*

- $I(+, P, A, +a) \subset R^1(a)$
- $I(+, P, A, -a) \subset R^1(a)$
- $I(-, P, A, +a) \subset R^1(a)$
- $I(-, P, A, -a) \subset R^1(a)$

Thus all of the implications of altering atom a and all of their knock on implications will be contained within R^ω, giving a technique for bounding the possible changes caused by adding a rule to a program. To use this and to infer that atoms cannot be altered and thus their status propagated to new answer sets results that relate the concept of reachability to the answer sets of the corresponding program are needed.

Definition 5 *Let P be a program and a, b are atoms. Then a may effect the status of b in an answer set $\Leftrightarrow$ there exists a chain of rules $(r_1, r_2, \ldots, r_n)$ with $r_1, \ldots, r_n \in P$ such that $(a \in B(r_1)) \vee (\textbf{not } a \in B(r_1))$, $(H(r_n) \in B(r_{n+1})) \vee (\textbf{not } H(r_n) \in B(r_{n+1}))$ and $H(r_n) = b$.*

Although complex to write down this defi nition is conceptual simple. If there are rules $(r_1, \ldots, r_n)$ such that a can effect whether r_1 is applicable, r_n can effect whether b appears in an answer set and each rule can influence the next then it is fair to say that a can influence whether b is in an answer set. After applying the reduct, this chain may allow the immediate consequence operator to conclude b if the status of a is known. This allows the concepts of the dependency graph (and thus implications) to be related to answer sets.

Proposition 3 *Let P be a program and a, b are atoms.*

$b \in R^\omega(a) \Rightarrow a$ *may effect the status of* b

Perhaps more usefully, the 'opposite' can be concluded

Theorem 1 *Let P be a program and a, b are atoms.*

$b \notin R^\omega(a) \quad \Rightarrow \quad \neg(a$ *may effect the status of* $b)$

Thus there exists a link between the maximum implications of a change and the reachability set from the changed atom. Also shown is the equivalence of being reachable and being able to effect another atoms existence in an answer set. This allows the conclusion that if an atom cannot be reached by the changed atom then it will have the same status in the new answer set that it had in the original.

4 Addition Algorithm

In the previous section we introduced the theory behind our algorithms. In this section we take a closer look at the actual algorithms. The subtracting algorithm is very similar to the addition one. Due to page limit restrictions, we will only deal with addition. Full details of the both algorithms and their sound and completeness proofs can be found in [3].

The algorithm for handling adding rules is divided into three phases. The first section removes all cases where the new rule can be easily shown to have no effect, or it's effect is trivial. After this phase, if there are answer sets which require non trivial changes, the second phase takes over to further reduce the program. It is possible that the first phase handles all answer sets in which case the algorithm finishes. The second phase of the algorithm works by 'bounding' the changes caused by adding the rule, if it can be shown that a particular conclusion is not effected by the addition of the new rule then it will be the same in the new answer set as it was in the old one. Again if there are still sets that need to be handled after this phase is completed the program can be handed over to the final phase can be used to complete the answer sets. Again it is possible that phase two handles all cases and there is no more work to be done. The third section of the algorithm is a modified version of the Smodels[10] algorithm which dynamically cuts out sections of the program if it is clear that they will not be affected by the changes.

Phase One From the definition of answer sets it is clear that adding a rule r to an answer set S will not change anything if r is either applied or not applicable with respect to S. In these cases the rule will not alter the eventual conclusion of the direct consequence operator or be removed by the reduct or contain unsupported atoms respectively. Only if the rule is applicable but not applied does it stop A being an answer set, i.e. cause a change. Phase one of the algorithm reduces the problem to cases where r is applicable but not applied and handles some edge cases (i.e. the introduction of new atoms to the system, constraints and self referential rules). From this point on the algorithm branches per answer set to phase two.

Phase Two Phase two of the algorithm focuses on cases when the rule to be added is applicable but not applied with respect to a particular answer set of the program. In this case the conditions for the rule to be 'true' hold and what the rule concludes is not already known - it actually makes a difference. This stage aims to resolve this difference in simple cases and to bound the maximum effect it can have before handing to phase three if necessary. This is done by computing the implications that will occur and cutting the dependency graph so only the the part that may be changed remain. If no implications are found the algorithm terminates.

Phase Three Phase three of the algorithm resolves the implications and dependency graph that have been discovered in phase two, to produce the answer sets of the modified program. It uses of a modified version of the standard bound / reduce / branch approach. When atoms within the program are shown to be unmodified by the changes caused by adding rule r, it dynamically cuts the graph of atoms that need to be resolved, thus reducing the size of the problem further. Sections of the graph that become unreachable from any of the nodes that can still be changed take the same value they had in the original answer set; there is no longer any way that they could be anything else. This dynamic cutting of the dependency graphs improves the efficiency significantly.

5 Implementation

The implementation of the incremental algorithms is divided into two sections. IASAI (Incremental Answer Set Algorithm Infrastructure) provides an abstract framework for developing answer set solver algorithms and applications that use them. IDEAS (Interactive Development & Evaluation tool for Answer Set programs) uses the IASAI interface to provide a text based, interactive system for manipulating, developing and using answer set programs.

The focus of IASAI is creating a common interface for different answer set algorithms. Modules that implement the IASAI interface[5] are under development for the algorithm presented in this paper, the Smodels algorithm[10] and a generic external answer set solver (aimed at compatibility with DLV[14]). It provides basic operations on rule sets such as adding and subtracting rules, calculating answer sets, etc. as well as functional descriptions of the representations of rules and atoms. As well as providing support for use of answer set programming with incremental knowledge bases it is hoped that this will give a uniform interface for implementing any logic engine that uses answer set semantics without needing to tailor it for a particular system or algorithm.

IDEAS is a command line tool that provides a user interface to the functionality of IASAI. It is intended to be used as a stand alone tool as well as being easily integrated into larger systems for rapid prototyping of logical systems that use these semantics. As well as running in a traditional interactive fashion it also supports scripting in the style of Unix command shells.

The incremental IASAI back end is designed to perform the minimum amount of work required to satisfy the request made to it. This is implemented as a FIFO list of task structures. Each task structure contains an indication of which function should be used (phase two, phase three, conventional solver, etc.) and the arguments. Where the algorithms would call these key functions (such as handing off between the phases and branching in the conventional solver) a task is added to the list. When an answer set is requested the structures are in turn removed from the list and executed until an answer set has been generated. This allows the computation to effectively be 'paused' after each answer set has been discovered, thus only doing work when it is required.

The division of work into task has been carefully chosen so that the only interaction between them is consequence. Executing a task may give more tasks but will not effect any existing tasks and the order of execution is non critical. This allows a custom version of the task dispatch function to be used which assigns tasks to one of a set of threads. Existing answer set solver implementations[10, 14] use recursion or lists to

[5] It is not fully in described in this document as it is still under development.

handle branching, however this approach allows the simultaneous execution of both branches if run on suitable hardware. Given the increasing trend towards thread and process based parallelism in modern CPU design, this sort of approach is required to make maximum use of computing resources and drastically cut the real time of such computations.

The just in time computation approach outlined above clearly helps the initial response time of an interactive program. Returning after the computation of the first answer set will give less delay than computing all answers sets before returning control to the user (assuming a non categorical rule set). However if a second answer set is requested the response time of the just in time method will be less. Slack time computation is the implementation work around to solve this problem.

Slack time computation allows tasks to be dispatched while the front end is waiting and processing user input. IASAI back ends have a function that dispatches tasks while a given semaphore remains clear. To make use of this feature front ends are required to use two threads, a main thread and a slack thread. Only one of these threads will make use of the IASAI back end or in fact heavy use of the CPU at any given time making it suitable for single processor hardware. Before the main thread is about to start processing user input (for an interactive program this is clearly very light work for the system) it wakes up the slack thread, which then calls the slack function from the IASAI back end. This dispatches tasks within the back end while the main thread is processing the user input. When the user input functions returns, the main thread marks the semaphore controlling the slack function, this causes it to finish computation and return, the slack thread of the front end then sleeps and the main thread continues as normal. When the system is being used interactively this gives the a better distribution of the compute load across the running time of the program and minimises the latency of commands to the program.

6 Future Research

This paper presents a complete solution for handling the addition (and subtraction) of single rules to an $AnsDatalog^{\neg,\perp}$ programs. However in terms of the overall topic this is only the beginning. In this section some of the theoretical questions and implementation areas raised by this work are presented.

Theory As this work treats changes in rule sets as single, atomic operations no consideration of the pattern of rule modification has been made. For example adding an exclusive choice between two new atoms requires at least 2 rules and is potentially quite computationally expensive as the second rule will require a near complete recomputation. Likewise care must be taken to add constraints before the rules than generate large numbers of options. One area of interest is to look at ways of adding multiple rules simultaneously, allowing a new concept or block of data to be added in one operation with considerable potential savings. An alternative approach to the same problem would be to develop some form of criterion or heuristic for when to handle a series of changes using the presented algorithm and when it is more efficient to recompute completely.

Such issues lead naturally to considering a modular approach to answer set and logic programming. Modular programming is a well accepted technique and a powerful abstraction mechanism, using modifi ed versions of the presented algorithms and techniques for making several simultaneous changes it may well be possible to provide this for answer set programming. This raises more possibilities, from pre-computation of fi xed blocks of rules to distributed computation to mixing rule sources (for example using databases as sources of facts) to the possibilities of choosing logical paradigms on a module by module basis (using preference based choice formalisms such as OCLP[9] for human interaction and $AnsProlog^*$ for the actual computation). All of these would be further step towards providing a modern programming environment for answer set computation.

Finally nothing has been presented on the addition of rules in any of the formalisms that extend $AnsDatalog^{\neg,\perp}$. Those that can be mapped or reduced to $AnsDatalog$ could be converted quite easily, although the nature of such mappings may signifi - cantly reduce the value of such algorithms. However logic systems such as $AnsProlog^{or}$ with a higher computational complexity and logic system which include function symbols are a much more interesting issues. Clearly the presented algorithms provide part of, but not a complete solution.

Implementation As well as a number of theoretical questions, there are also a large number of implementation issues raised by this work.

Firstly the IDEAS front end is still only in the alpha development stages. There are a large number of features that need to be implemented before it reaches the standard of a fi nished product. Examples are lexical matching to spot typographical errors, support for a wider variety of input formats and portability to more software platforms are more implementation specifi c.

Although not necessarily of massive theoretical signifi cance, other IASAI front ends would provide interesting applications of answer set solvers in other problem domains. A front end that supported PROLOG syntax (or as much of it as is reasonable) could be a useful teaching tool. Another option is an SQL front end which would provide a more intelligent approach to complex data mining applications while maintaining compatibility with a large array of existing tools.

For comparative and benchmarking purposes it may well be advantageous to implement IASAI back ends that integrate with and / or implement some of the other answer set solver algorithms.

Finally the incremental IASAI back end has considerable scope for future development. To support larger input programs the task list and dispatch infrastructure could be extended to distribute not only across threads but between separate machines, allowing the computing power of high performance clusters and distributed systems to be used. Finally the slack time infrastructure could be used to perform speculative execution if there are no tasks remaining. At the most basic level this could be calculating which atoms / answer set combinations give no implications. However as implications are only dependent on the head atom of the rule being added it may well be possible to handle non trivial implications. Given enough time between rule set alterations it would be possible to compute all possible atom / alteration / answer set combinations so handling alteration would simply require looking up the appropriate results. In computational terms this is a wasteful approach as a large number of the results will never be used and will have to be discarded after the rule set has been changed, however in application areas in which this compute time would simply be wasted this is not a problem. Further work may develop ways of modifying the data produced by speculative execution rather than simply discarding it. It might also be possible to produce constructive rather than analytic heuristics for what information is to be added; the system could give a list of rules that would fulfi l or aid certain criteria on the answer sets (i.e. that there is only one answer set / the program is categorical, the answer sets allow certain queries to be answered defi nitively). This is envisaged to have signifi cant potential for learning based knowledge representation applications, for example in autonomous, intelligent agent systems.

References

[1] C. Anger, K. Konczak, and T. Linke. NoMoRe: Non-monotonic reasoning with logic programs. In *Eighth European Workshop on Logics in Artificial Intelligence (JELIA'02)*, volume 2414 of *Lecture Notes in Artificial Intelligence*, 2002.

[2] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.

[3] M. J. Brain. Undergraduate dissertation: Incremental answer set programming. Technical Report 2004–05, University of Bath, U.K., Bath, May 2004.

[4] Martin Brain and Marina De Vos. Implementing OCLP as a front-end for Answer Set Solvers: From Theory to Practice. In *ASP03: Answer Set Programming: Advances in Theory and Implementation*. Ceur-WS, September 2003. online CEUR-WS.org/Vol-78/asp03-fi nal-brain.ps.

[5] Gerhard Brewka and Thomas Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109(1-2):297–356, April 1999.

[6] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and weak constraints in disjunctive datalog. In Jurgen Dix, Ulrich Furbach, and Anil Nerode, editors, *4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in Computer Science*, pages 2–17, Dagstuhl, Germany, July 1997. Springer Verslag.

[7] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)*, 33(3):374–425, 2001.

[8] Marina De Vos and Dirk Vermeir. On the Role of Negation in Choice Logic Programs. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Logic Programming and Non-Monotonic Reasoning Conference (LPNMR'99)*, volume 1730 of *Lecture Notes in Artificial Intelligence*, pages 236–246, El Paso, Texas, USA, 1999. Springer Verslag.

[9] Marina De Vos and Dirk Vermeir. A Logic for Modelling Decision Making with Dynamic Preferences. In *Proceedings of the Logic in Artificial Intelligence (Jelia2000) workshop*, number 1999 in Lecture Notes in Artifi cial Intelligence, pages 391–406, Malaga, Spain, 2000. Springer Verslag.

[10] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system `dlv`: Progress report, comparisons and benchmarks. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann, San Francisco, California, 1998.

[11] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of fifth logic programming symposium*, pages 1070–1080. MIT PRESS, 1988.

[12] Michael Gelfond. Strong Introspection. In *AAAI'91*, pages 386–391, 1991.

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

[14] I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, Berlin, July 28–31 1997. Springer.

[15] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable Model Semantics of Weight Constraint Rules. In *Proceedings of the Eight International Conference on Logic and Nonmonotomic Reasoning*, Lecture Notes in Computer Science, pages ???–???, El Paso, Texas, US, December 1999. Springer-Verslag.

[16] N Pelov, E De Mot, and M. Denecker. Logic programming approaches for representing and solving constraint satisfaction problems : a comparison. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 225–239, Reunion Island, France, November 2000.