Debugging Logic Programs under the Answer Set Semantics

Martin Brain and Marina De Vos[*]

Department of Computer Science
University of Bath
Bath, United Kingdom
{mjb|mdv}@cs.bath.ac.uk

Abstract. This paper discusses the background, algorithms and implementation techniques to support programmers in 'debugging' logic programs under the answer set semantics. We first investigate what constitutes an error in such programs and which classes of errors exist. This is used to motivate techniques and algorithms that respectively eliminate certain classes of errors and explain how an error occurred and why it occurred. Finally, details of the IDEAS (Interactive Development and Evaluation tool for Answer Set Semantics) system are given, a prototype version of which implements all of the techniques described.

1 Introduction

Answer set programming (ASP) is a modern logic programming system, designed for semantic clarity, efficient implementation and ease of use for knowledge representation and declarative problem solving. It utilises the answer set semantics[1] defined on normal logic programs (also referred to as $AnsProlog^*$[3]). Many of the fundamental theoretical questions have already been addressed, the semantics of the language are well defined, the properties of key classes of program are known, algorithms for computing the semantics of programs have been developed as well as a number of robust implementations. Preliminary results[11, 13] suggest that ASP is already as efficient as many comparable systems. However, ASP has yet to find widespread use in many 'real world' applications (although there exist a few[5, 14, 15]). These are vital, not only for the long term future of ASP, but also because they help inform and motivate further theoretical research. ASP is still a very new technology, a lack of teaching material and a lack of awareness within the wider community explain some of the problem but perhaps most critically, large ASP programs are difficult to write. This is a combination of many factors, but even with reasonably well understood 'modules' for encoding some basic concepts[3, 9], no best practise, methodology or standard techniques exist for addressing the over all problem. Critically, no real programmer support tools are available. Programmers using a traditional procedural language such as C have an extensive toolkit of applications to help them develop programs and then find errors

[*] 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.

in them. Graphical integrated development environments, interactive execution shells (commonly called debuggers), library and system call tracing tools and memory profilers are but a few examples. For ASP, excluding solvers (the tools that generate answer sets from a set of rules (program)), there exist a few 'front ends'[7] but no real analysis tools.

This paper discusses what it means for an answer set program (only ground normal logic programs, $AnsDatalog^{\perp}$ are handled, although the results can easily be generalised) to be incorrect and a number of techniques to minimise the number of errors and locate the remaining ones. Critically, these do not depend on any property of the program being debugged. It also discusses a prototype implementation of these techniques in the tool IDEAS[4] (Interactive Development and Evaluation tool for Answer set Semantics).

2 Classification of Errors

The primary question in the design of any debugging or interactive fault finding system is what constitutes an error. The academic fields of software engineering and human computer interaction are awash with classification schemes for errors. Different approaches categorise errors by what causes them to occur, what allows them to occur, what they effect and how they do so. This variety of approaches is not just restricted to the analysis of 'systems', it can be found in classification systems of programming errors. Security implications of incorrect programs are categorised by effect (information leakage, escalation of privileges, denial of service, etc.) while memory use analysers classify errors by causes (uninitialised memory, unallocated memory, etc.). Here we shall use a standard[2] classification of errors in a programming language by what level of the formal specification of the language they appear in.

- **Lexical** :- errors that occur when a part of the statement in the given language is not a valid word. For example in the English language, "The cat is fghjk." contains a lexical error as the fourth word is not a valid English word. In the case of programming languages these can be found by the lexical analysis (or tokeniser) phase of the language compiler / interpreter.
- **Syntactic** :- the words in the statement are valid but they do not form a valid sentences. For example "The the is cat mat on." is made of valid words but not in a correct order. Again these errors can be found automatically in programming languages by the compiler.
- **Semantic** :- the statement is well constructed but does not have a valid meaning. For example "The cat is gaseous" is syntactically valid but does not mean anything. Depending on both the language and the error it may be possible to detect these at compile time or only at run time (if at all).
- **Conceptual** :- the statement is technically correct but not what the user wanted. For example "The dog is brown." is valid English but does not say anything useful about the cat. These are traditionally the hardest type of bugs to find. Software engineering is, in part an attempt to stop this sort of errors by introducing procedures, such as verification against a specification.

In the case of answer set programming, the content of these categories is somewhat different to procedural languages. Object constants, literals and variables are introduced as they are needed, thus there are comparatively few types of lexical error possible. Also, the syntax of the language is very simple so the scope for syntactic errors is also small. Both of these class of error can be found automatically and thus are not of interest. The semantics of $AnsDatalog^{\perp}$ are defined for all syntactically valid programs so in theoretical terms there are no semantic errors. Given that some researchers[3] view logic programs as specifications, or equivalent to them, this categorisation is not entirely surprising. This means new divisions are needed to provide a meaningful classification of errors in answer set programs and that all debugging techniques for answer set programs are subject to the same problems and vagaries as any attempts at the automatic location of logical errors.

The classification of errors in answer set programs is further complicated as there is no clear indication of what is a 'faulty' program. Where as a procedural program that fail to compile, crash with segmentation faults or throws an exception can be reasonably assumed to be faulty in some way, there is no general, automated way of telling, just by examining the answer sets produced, whether an answer set program is behaving incorrectly or not. Looking for programs that produce only contradictions is not satisfactory as this will not catch many more subtle bugs. A number of systems exist that analyse and 'repair' inconsistent programs[16, 17]; although powerful, it is felt that they do not solve the complete problem of debugging.

The impact of this is two fold: firstly, it means that an effect-based classification system of errors is not a viable option and additionally, it is reasonable to assume that the user has some idea of what they wanted, and how that is different from what they have. If they did not they would not be using a debugging tool as they would not realise there was anything wrong with their program[1].

The proposed classification is to divide errors into rule level and program level errors. *Rule level* errors are errors in the answer sets of a program that are caused by a single incorrect rule. For example, accidentally omitting the head of an rule and turning it into a constraint or misspelling the name of a literal in the body. *Program level* errors are errors caused by the interaction of multiple 'correct'[2] rules. For example, a set of rules that specifies an exclusive choice between a number of alternatives might not require that one is picked at all, while the programmer intended this to be the case. Clearly this is a loose and fluid distinction and is intended to be used for intuitive description of the errors that a technique intends to solve rather than a hard classification.

In summary, debugging answer set programs is more a task of supporting a programmer in investigating why a program does not behave as expected, rather than a series of static tests that can be performed. It is also a non-exact process, as it is subject to the potential differences between what the programmer wrote, what the programmer meant to write and what the programmer actually meant.

[1] This raises some interesting questions about how answer set programs should be used in the wider context of problem solving.

[2] Obviously these rules are not correct in the strict sense, this more a reference to rules that are expressed as originally intended.

3 Language Modifications

By altering the language syntax of answer set tools slightly it is possible to eliminate the possibility of making certain types of lexical and syntactic rule level errors. These, however do not completely eliminate conceptual errors at the rule level, a user can still write $a \leftarrow b.$ when they meant $a \leftarrow b, \mathrm{not}\ c.$ and there is really no way of finding this automatically.

– Introduce a notation, (<> is suggested) for the symbol $\perp$ and mandate its use. If rules with $\perp$ in the head are notated without a head then any rule in which the programmer has neglected to fill in a head atom will become a constraint. For example, there is no way of telling automatically if `:- a, b.` is notation for $\perp \leftarrow a, b.$ or $c \leftarrow a, b.$ without the head. With an explicit $\perp$ notation these errors by omission can be found automatically.
– Introduce a notation for declaring the existence of a literal symbol and create a syntax error if a literal has been used before it has been defined. This will catch misspellings and typos in the names of predicates, implementations may also wish to use technology for spelling checking algorithms to suggest replacement predicates in an interactive context. lparse[12] has an option (-w similar) that flags any predicates or variables with similar names that is a step towards this functionality.
– Introduce notation for declaring sets of object constants and require that when literals are declared, the range of each term is specified. As well as making programs clearer this information can be checked by the grounder to ensure that no unintended ground naf-literals can occur. This would also form the basis of a type system which could be used for further static analysis.
– Warning or flagging an error when a predicate is used with different arities will catch some errors. From a theoretical view, these are not errors, however one of the conventions of usage of ASP is that a predicate name should only be used once and have a fixed arity. lparse already implements this feature.

4 Algorithm Based Debugging

The naive approach to creating tools that catch program level errors is to adapt the concepts used in Prolog and procedural interactive debugging tools (such as the GNU Debugger (GDB)[8]) to work with the existing solver algorithms. By adding explicit points of control into the algorithm (for example, between the expand and branch steps of the algorithm, or after each step of the inference function) the user can be given control of how many and what type of steps can be taken, in much the same way that `trace` and `spy` are used in Prolog. Commands to these systems would be broken into two categories, control commands that would allow the computation to continue until a specified point was reached or condition occurred and display commands that would allow the user to query the status of atoms and rules within the current context. The noMoRe system[6] implements this functionality, although with a less conventional interface.

This approach is simple to implement, additions of small amounts of control code can add this functionality to any implementation of an answer set computation algorithm. It provides a conceptual similar interface for programmers who are familiar with Prolog or traditional procedural debugging tools. Beyond debugging, this approach provides the only realistic approach to assessing the efficiency of encoding a problem with particular constructs. Without this sort of technique working out why one program takes longer to compute the solutions to a problem than another is very difficult. If the heuristics of the solver are configurable, this could be used to tune the solver to a particular program. Thus this type of application has scope beyond debugging and provides a useful tools.

However, as simply a debugging tool it has a number of drawbacks. Firstly, it increases the cognitive load on the programmer considerably, as it is very difficult in the general case to work out what information the programmer is not interested in when using this technique. Blocks of rules and atoms could be marked as ‘not of interest’, but this requires intervention from the programmer and introduces the possibility of undershooting (adding time to the debugging process) or overshooting (rendering the results useless). Even if the marking of blocks (similar to the choice of what predicates to `spy` upon) is correct, this will take time. Once the computation process is running it is difficult to ‘go backwards’ without storing large amounts of state within the application, which in turn impacts the efficiency of the solver. The implementation is solver specific and more importantly requires the user to understand the solver algorithm. At the current state of development this is not an unreasonable assumption however with increasingly sophisticated solver algorithms and as ASP reaches a wider, non specialist audience, this becomes increasingly less tenable. Finally, this approach makes little use of the dependencies between literals, information that is very easily extracted from ASP programs and very little use of the formal semantics. It prioritises the computation algorithm over the semantics - ASP becomes a configuration for a search algorithm rather than a formal, logical language. In essence this approach answers the question ‘how did this error occur?’ rather than the (related) question the programmer actually wants answering; ’why did this error occur?’.

5 Query based debugging

This section presents two techniques for answering simple questions about the answer sets of a program and describes how to use these to build a debugging system based around explaining why particular behaviour has occurred.

Given that the programmer has computed the answer sets of the program and found a discrepancy between what they expected and what they received, there are a number of possible scenarios. Each answer set may contain additional, unexpected atoms or may not contain expected atoms. Alternatively, there may be one or more answer sets missing or combinations of atoms not present in any answer set. The first set of questions are restricted to one answer set and can be answered by asking “Why is set S contained in answer set A” where both A and S are sets containing both positive and negative literals (i.e. for every literal l in the Herbrand Base either $l \in A$ or $\mathrm{not}\ l \in S$).

The second scenario asks questions about the whole computation process and can be resolved with the question “Why is set S not contained in any answer set”.

In the examples and pseudo code given, output (the explanations) is given as text. This is clearly only one option, a graphical system of some sort may prove more intuitive and more efficient to use. There are a large number of options from diagrams using showing the relations between the text to entirely graph based approaches to animated illustration of the input program.

5.1 Why is Set S Contained in Answer Set A?

The most obvious question a programmer is likely to ask when faced with an unexpected result[3] is *“Why is atom a in the answer set?”*. The answer is remarkably simple and (crucially) recursive. a is in an answer set if there is a rule that supports it, $\mathrm{not}\ a$ is in an answer set (equivalently a is not in an answer set) if there are no rules to support it. Given the bodies of the rules that support (or fail to support) a given atom are sets of literals, the question may be asked recursively, tracing back the justification of an atom back to facts and unsupported atoms. If the recursion is automated in any fashion, care must be taken to avoid creating chains of explanations with circular justifications. Clearly providing a ‘because ... ’ answer to this type of question will naturally lead to the programmer responding “but what if set T was in the answer set”, which can be resolved using the second algorithm.

```
/* Global Variables */
P    // The program
A    // An answer set of P

/* Arguments */
S    // A non empty set of positive and negative literals contained in A

/* Local Variables */
justified // A variable used to check if a particular conclusion has been
          //  justified

/* Functions */
subset(A,B)            // Returns true if A is a subset of B
contains(atom,set)     // Returns true if atom is in set
support(atom,program)  // Returns the set of rules in the program that have
                       // atom as their head
applied(rule,answerset)// Returns true if rule is applied with respect to
                       // answerset


/* Start of query algorithm */

/* For every element of S */
for s in S {
    /* Need to look at the rules that support s */
    R = support(s,P);

    if (s.positive == true) {
        /* If it is positive */
        print(‘‘s is in A as’’);

        /* Find which of the supporting rules is applied */
        justified = false;
```

[3] This technique also allows unexpected but correct results to be explained.

```
        for r in R {
            if (applied(r,A) == true) {
                print(‘‘r is applied with respect to it’’);
                justified = true;
            }
        }

        /* If no justification has been found there is an inconsistancy */
        if (justified == false) {
            print(‘‘error - s is unsupported’’);
        }
    } else {
        /* If it is negative */
        print(‘‘s is not in A as’’);

        /* Work throught the supporting rules and say *
         * why they are not supported                 */
        if (R.size == 0) {
            print(‘‘there are no rules supporting s’’);
            continue;
        }

        for r in R {
            print(‘‘r supports s but’’);
            justified = false;

            /* See which body literals are not supported */
            for b in R.body {
                if ((b.positive == true) && (contians(b,A) == false)) {
                    print(‘‘b is not in the answer set’’);
                    justified = true;
                } else if ((b.positive == false) && (contians(b,A) == true)) {
                    print(‘‘b is in the answer set’’);
                    justified = true;
                }
            }

            if (justified == false) {
                print(‘‘error - A incomplete’’);
            }

        }
    }
}
/* End of query algorithm */
```

For example given the program:

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

asking[4] “Why is $\{a, \text{not } d\}$ contained in answer set $\{a, b, c\}$?” will produce the following:

> a is in $\{a, b, c\}$ as $a \leftarrow b.$ is applied with respect to it. d is not in $\{a, b, c\}$ as only $d \leftarrow \text{not } c, a.$ supports d but c is not in the answer set.

[4] The user is assumed to have requested the computation of the answer sets of the program before asking this.

5.2 Why is Set S not Contained in any Answer Set?

The corresponding question is somewhat harder to answer as there are a number of possible ways in which a given set of atoms might not occur in an answer set. They could lead to a contradiction, not be supported or require mutually inconsistent conditions to be supported. This algorithm essential extends the set one step in ‘both directions’, adding in the immediate consequences as well as the literals required to support the set. It produces a set of possible situations, each of which is a superset of S representing one of the possible scenarios in which everything in S is consistently supported. The process can then be repeated with each of the scenarios until no possible scenarios are output.

One further complication is the need to build a directed acyclic graph representing how the literals in the set support each other. Without this it is difficult to deal with programs that contain positive justification loops (i.e. $a \leftarrow b. b \leftarrow a.$). When the algorithm is first used this set will be empty. There is one node in the graph for every literal in the Herbrand Base as well as nodes for $\top$ and $\bot$ and links are either positive or negative. Adding a rule to the graph attempts to creates links from each element of the body to the head, it fails if this would introduce a loop or requires adding a negative link from a node that already has positive links to others (or vice versa).

```
/* Global Variables */
P    // The program
Ans  // The set of answer sets of P

/* Arguments */
S    // A non empty set of positive and negative literals
G    // A directed, acyclic graph with nodes for each literals

/* Local Variables */
Imp     // If S was in an answer set then so would everything in Imp
Sit     // A set of sets, each is a possible supporting situation
oldSit  // Used as a temporary variable
tmp     // A temporary set of literals

/* Functions */
subset(A,B)             // Returns true if A is a subset of B
contains(atom,set)      // Returns true if atom is in set
support(atom,program)   // Returns the set of rules in the program
                        // that have atom as their head
applied(rule,answerset) // Returns true if rule is applied with respect to
                        // answerset
inconsistent(set)       // Returns true if set contains an atom and it’s negation
supported(graph)        // Returns all of the atoms supported in the graph
unsupported(graph)      // Returns all of the atoms unsupported in the graph


/* Start of query algorithm */

/* Check that S is not inconsistent */
for s in S {
    if ((s.positive == true) && contains(-s,S)) {
        print(‘‘S is inconsistent’’);
        return;
    }
}

/* Check that it is not in any answer sets */
for A in Ans {
    if (subset(S,A) == true) {
        print(‘‘S is a subset of A’’);
        return;
    }
}

/* Check that S does not directly imply a contradiction */
tmp = S U supported(G)
for r in P {
    if (subset(r.body,tmp) == true) {
        if (r.type == CONSTRAINT) {
            print(‘‘S is not in any answer set as r is
                    applicable with respect to S’’);
            return;
        } else {
            if (addRule(r,G) == false) {
                print(‘‘S is not in any answer set as r is applicable but
                        causes a contradiction.’’);
                return;
            }
        }
    }
}

/* Check there are consistant ways of supporting atoms in S */
Sit.add((emptyset,G));
for s in S {
    if (s.positive == true) {

        /* Find which rules support s */
        R = support(s,P);

        if (R.size == 0) {
            print(‘‘S is not in an answer set as s is unsupported.’’);
            return;
        }
        oldSit = Sit;
        Sit = emptySet;

        /* For every rule that can support s */
        for r in R {
            /* And every situation */
            for Z in oldSit {
                /* Create a new situation with r supporting s */
                if (inconsistent(Z.S U supported(Z.G) U r.body) == false) {
                    addRule(r,Z.G);
                    Sit.add((Z.S U unsupported(Z.G),Z.G));
                }
            }
        }

    } else {
        /* If s is negated in S */
        /* Find which rules support s */
        R = support(s,P);

        if (R.size == 0) {
            continue;
        }

        /* Extend each situation so that not rule supporting s is applicable */
        for r in R {
            oldSit = Sit;
            Sit = emptySet;

            for Z in oldSit {
                tmp = r.body \ (Z.S U supported(Z.G));

                if (tmp == emptyset) {
                    print(‘‘s is supported by r.’’);
                return;
            } else {
                for a in tmp {
                    Sit.add((Z.S U {a}, Z.G));
                }
            }
        }
      }
    }

    /* If there are no situations in which this atoms *
     * is supported, S cannot be an answer set        */
    if (Sit == emptySet) {
        print(‘‘No situations in which s is supported
                that are consistant with the proceeding
                atoms’’);
        return;
    }

}

if (Sit.size == 1) {
    print(‘‘S is a partial evaluation, guessing random atom.’’);
    oldSit = Sit;
    Sit = emptyset;
    Sit.add((oldSit[0].S U chooseAtom(), oldSit[0].G));
    Sit.add((oldSit[0].S U -chooseAtom(), oldSit[0].G));
}

output Sit;
/* End of query algorithm */
```

Actual implementations may wish to display the set of possible situations (`Sit`) after each atom in S has been justified or annotate the atoms in each set in `Sit` with how it is derived and thus provide a list of which rules clash.

This algorithm can also be used to explain the behaviour of inconsistent programs by starting with S as a set of literals each supported by a fact and querying the resulting set until a contradiction is reached.

5.3 Using these Algorithms

Clearly the answers given by by these algorithms allow the questions to be asked recursively. To use these algorithms effectively, the system designer must allow the programmer to control this recursion directly. When justifying why a set of atoms appears in an answer set, only some of the responses will be of interest and there is no automatic way of knowing which without requiring more information from the programmer.

There are a number of possible ways of controlling the recursion. The simplest approach is to implement the algorithms as they are presented and let the programmer handle the recursion at a command level. Any system that has a graphical interface to these algorithms way also provide support for recursive queries (displaying the results in an expanding tree in a similar fashion to graphical file managers is one option). Allowing the programmer to specify a ‘recursion depth’ or a set of atoms which they are confident belong to the answer set (and thus the recursion stops when it reaches one) are other options.

6 Implementation

The techniques outlined in this paper have been implemented in the latest development version of IDEAS (Interactive Development & Evaluation tool for Answer Set programs), an interactive command line based tool for developing and analysing answer set programs. It is licensed under the GNU General Public License (GPL)[10].

7 Conclusions and Future Research

This paper presents techniques and algorithms that provide the first comprehensive debugging support for answer set programs. These can locate many simple errors and support the programmer in working out why a program displays erroneous behaviour. Without more extensive studies into how programmers use ASP, there are further questions to be answered.

However, there are numerous related questions which remain unanswered. Firstly, there is the issue of how to implement these techniques in a programmer friendly, efficient and elegant manner. Extending the algorithms to handle frequently used structures such as sets and lists in an intuitive fashion would be very useful, especially if it could be integrated with 'front end' tools that generate them. There are also a number of questions that overlap into the areas of software engineering and human computer interaction, how do programmers use ASP tools, how do they work from a description of a problem to a logical model of it and what tools are needed to support this, such as how should large amounts of logically structured information be presented to the user and how can formally based technologies such as ASP be made 'more accessible' to non experts.

Acknowledgements

The authors would like to thank Tommi Syrjänen, Ken Satoh, Tom Crick and the participants of the Dagstuhl seminar on Answer Set Programming (05171) for their comments and suggestions.

References

1. M. Gelfond and V. Lifschitz : *The stable model semantics for logic programming*, Logic Programming: Proc. of the Fifth Int'l Conf. and Symp. Pg 1070-1080, MIT Press, 1988
2. A. Aho, R. Sethi and J. Ullman *Compilers - Principles, Techniques and Tools*, ISBN 0-201-10088-6, Addison Wesley, 1988.
3. Chitta Baral. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.
4. M. Brain. Undergraduate dissertation: Incremental answer set programming. Technical Report 2004–05, University of Bath, U.K., Bath, May 2004.
5. M. Nogueira, M. Balduccini, M. Gelfond, R. Watson and M. Barry : *An A-Prolog decision support system for the Space Shuttle*, AAAI Spring 2001 Symposium, Mar 2001.

6. C. Anger, K. Konczak, and T. Linke. NoMoRe: Non-monotonic reasoning with logic programs. In G. Ianni and S. Flesca, editors, Eighth European Workshop on Logics in Artificial Intelligence (JELIA'02), volume 2424 of Lecture Notes in Artificial Intelligence. Springer Verlag, 2002.
7. M. Brain and M. 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, 2003. online CEUR-WS.org/Vol-78/asp03-final-brain.ps.
8. GDB homepage `http://www.gnu.org/software/gdb/`
9. Giovambattista Ianni, Giuseppe Ielpa, Francesco Calimeri, Adriana Pietramala, Maria Carmela Santoro. G. Ianni, G. Ielpa, F. Calimeri, A. Pietramala, M. Carmela Santoro. A System with Template Answer Set Programs. In Proceedings of the 9th European Conference, JELIA 2004, LNCS 3229 Springer Verlag, Lisbon, Portugal, September 27-30, 2004.
10. Free Software Foundation, Inc. GNU Public Licence `http://www.gnu.org/licenses/gpl.html`
11. 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.
12. T. Syrjänen. Implementation of local grounding for logic programs with stable model semantics Technical Report B18, Digital Systems Laboratory, Helsinki University of Technology, October 1998.
13. Hietalahti, M., Massacci, F., and Niemelë, I. DES: a challenge problem for nonmonotonic reasoning systems. In Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, Breckenridge, Colorado, USA
14. Leone N. , Gottlob G. , Rosati R. , Eiter T. , Faber W. , Fink M. , Greco G. , Ianni G. , Kalka E. , Lio V. , Lembo D. , Lenzerini M. , The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. "SIGMOD 2005", Baltimore, Maryland (USA), 2005, 2005
15. Giorgini, P., Massacci, F., Mylopoulos, J. and Zannone, N. Requirements engineering meets trust management: model, methodology, and reasoning. In Proceedings of the Second International Conference on Trust Management (iTrust 2004) (2004), T. Dimitrakos, C. D. Jensen, and S. Poslad, Eds., vol. 2995 of Lecture Notes in Computer Science, Springer-Verlag Heidelberg, ISBN 3-540-21312-0pp. 176-190.
16. Satoh, K., Consistency Management in Software Engineering by Abduction, Proceedings of the ICSE-2000 Workshop on Intelligent Software Engineering, pp. 90 – 99, Limerick, Ireland (2000)
17. T. Syrjänen, On Debugging ASP, draft version.