Debugging ASP using ILP

TINGTING LI

Institute for Security Science and Technology, Imperial College London, UK
E-mail: tingting.li@imperial.ac.uk

MARINA DE VOS, JULIAN PADGET

Department of Computer Science, University of Bath, Bath, UK
E-mail: {mdv,jap}@cs.bath.ac.uk

KEN SATOH

National Institute of Informatics and Sokendai, Japan
Email: ksatoh@nii.ac.jp

TINA BALKE

Centre for Research in Social Simulation, University of Surrey, UK
E-mail: t.balke@surrey.ac.uk

submitted 29 April 2015; accepted 5 June 2015

Abstract

Declarative programming allows the expression of properties of the desired solution(s), while the computational task is delegated to a general-purpose algorithm. The freedom from explicit control is counter-balanced by the difficulty in working out what properties are missing or are incorrectly expressed, when the solutions do not meet expectations. This can be particularly problematic in the case of answer set semantics, because the absence of a key constraint/rule could make the difference between none or thousands of answer sets, rather than the intended one (or handful). The debugging task then comprises adding or deleting conditions on the right hand sides of existing rules or, more far-reaching, adding or deleting whole rules. The contribution of this paper is to show how inductive logic programming (ILP) along with examples of (un)desirable properties of answer sets can be used to revise the original program semi-automatically so that it satisfies the stated properties, in effect providing debugging-by-example for programs under answer set semantics.

KEYWORDS: answer set programming, debugging, inductive logic programming

1 Introduction

Answer Set Programming (ASP) is a declarative programming paradigm for logic programs under answer set semantics. Like all declarative paradigms it has the advantage of describing the constraints and the solutions rather than the writing of an algorithm to find solutions to a problem. In recent years we have seen the rise of ASP applications[1]. As with all programming languages, initial programs contain bugs that need to be eradicated.

[1] In 2009, LPNMR (Erdem et al. 2009) had a special track on successful applications of ASP.

While syntactic errors can easily be picked up by the compiler, interpreter or solver, logical errors are harder to spot and to debug. For imperative languages, debugger tools assist the programmer in finding the source of the bug. In declarative programming, the flow of the program is not necessarily known by the programmer (unless he or she is intimately familiar with the implementation of the solver and its many flags) and probably should not, to be in keeping with the declarative philosophy.

With the development of applications in ASP, the need for support tools arises. In imperative languages, the developer is often supported by debugging tools and integrated development environments. Following the same approach, the first debugging tool was introduced by Brain et al. (2005). Since then, new debugging techniques have been put forward, which range from visualisation tools (Calimeri et al. 2009; Cliffe et al. 2008; K-loimüllner et al. 2013), through meta-language extensions (Gebser et al. 2009) to idealised theoretical models (Denecker and Ternovska 2008). In our opinion Pührer's (Oetsch et al. 2011) stepping technique is the most practical approach. It allows the programmer to select at each choice point which atom(s) should be added to the current candidate answer sets. The tool will then update the candidate with the atoms become true/false as a consequence of this choice. It offers the closest resemblance to the debugging techniques used in imperative languages, so programmers might be more familiar with this approach. Nevertheless, it requires the user to step through the computation of the answer set which causes a cognitive overhead, which ideally should be avoided. It also reveals the procedural aspect of the answer set computation. Another promising approach is that by Mikitiuk et al. (2007), which uses a translation from logic-program rules to natural language in order to detect program errors more easily.

In this paper we propose a debugging technique for normal logic programs under answer set semantics, based on inductive logic programming. By allowing the programmer to specify those features that *should* appear in an answer set (positive examples) and those that *should not* appear in any answer set (negative examples), the inductive logic program can suggest revisions to the existing answer set program such that the positive examples feature in at least one answer set of the revised program, while none exhibit the negative examples. The implementation of the theory revision is done in ASP using abductive logic programming techniques.

The rest of the paper is organised as follows: In Section 2 we provide brief introduction to answer sets programming before defining the debugging task for answer set programs. In Section 3 we start with an intuitive description of our debugging approach (Section 3.1). The most important component – *revision tuples* – of the debugging process is elaborated in Section 3.2. After describing the process formally (Section 3.3), we present its implementation based on *Inductive Logic Programming* (Section 3.4). We conclude this paper with a discussion of the related and future work.

2 Bugs in Answer Set Programming

2.1 Answer Set Programming

The basic component of a normal ASP program is an atom or predicate e.g. `r(X, Y)` denotes XrY. X and Y are variables which can be grounded with constants. For this paper we will assume there are no function symbols with a positive arity. Each ground atom can be assigned the truth value *true* or *false*. ASP adopts *negation as failure* to compute the negation of an atom, i.e. $\mathbf{not}\ a$ is true if there is no evidence to prove a in the current program. *Literals* are atoms a or negated atoms $\mathbf{not}\ a$.

A normal logic program P is a conjunction of rules r of the general form: $a :- b_1, ..., b_m,$ not $c_1, ...,$ not c_n. where $r \in P$, and a, b_i and c_j are atoms from a set of atoms $\mathcal{A}$. Intuitively, this means *if all atoms b_i are known/true and no atom c_j is known/true, then a must be known/true*. a is the head part of the rule $head(r)$ = a , while b_i and not c_j are body parts of the rule $body(r) = \{b_1, ..., b_m, \text{not } c_1, ..., \text{not } c_n\}$. A rule is a *fact rule* if $body(r) = \emptyset$ or a *constraint rule* if $head(r) = \emptyset$, indicating that no solution should be able to satisfy the body.

The finite set of all constants that appears in the program P is referred as the *Herbrand universe*, denoted $\mathcal{U}_P$. Using the Herbrand universe, we can ground the entire program. Each rule is replaced by its grounded instances, which can be obtained by replacing each variable symbol by an element of $\mathcal{U}_P$. The ground program, denoted $ground(P)$, is the union of all ground instances of the rules in P. The set of all atoms grounded over the Herbrand universe of a program is called the *Herbrand base*, denoted by $\mathcal{B}_P$. These are exactly those atoms that will appear in the grounded program.

An interpretation is a subset of the Herbrand base. Those atoms that are part of the interpretation are considered true while the others are false through negation as failure. An interpretation is a model iff for each rule holds that the head is true whenever the body is true. The model M is minimal if no other model N exists that is a subset of M ($N \nsubseteq M$).

The semantics of an ASP program P is defined in terms of *answer sets*, i.e. assignments of true and false to all atoms in the program that satisfy the rules in a minimal and consistent fashion. Let P be a program. The *Gelfond-Lifschitz transformation* of $ground(P)$ w.r.t S, a set of ground atoms, is the program $ground(P)^S$ containing the rules $a :- B$ such that $a :- B, \mathbf{not}\ C \in ground(P)$ with $C \cap S = \emptyset$, and B and C are sets of atoms.
A set of ground atoms $S \subseteq \mathcal{B}_P$ is an *answer set* of P iff S is the minimal model of $ground(P)^S$. A program P may have zero or more answer sets, and Π_P denotes all the possible answer sets of P.

2.2 Debugging ASP programs

In some cases, the obtained answer sets in Π_P might not be consistent with our expectation, although they satisfy all the current rules of the program. In other words, there is a bug. Thus we need to track back to revise the program, in order to produce the expected results only. First of all, we need a way to represent the goal of debugging, i.e. what we expect the program is able to produce or not. The representation needs to be accessible for human inspection, but also be compatible with the semantics of ASP. Therefore, we use sets of ground literals to indicate desirable and undesirable properties of a program. For instance, we use the atom `state(X,Y,N)` to denote that the square on row `X` and column `Y` in a Sudoku puzzle has value `N`. Now if we want to express that it is desirable a particular square has a certain value, or avoids having certain values, such expectations can be directly captured by ground literals of `state(X,Y,N)`.

By expressing a desirable or undesirable property using a set of ground literals, we can define the goals of debugging an ASP program. These goals guide the debugging process to produce necessary revisions for the original program. We also expect the revised program to be as close as possible to the original, so we use a *cost* function to measure the difference between the original and revised program. An ASP debugging task is guided by a set of literals that can be used to revise the program. A revision alternative for a program can only use these literals to add to or delete from existing rules, create new rules or delete entire existing rules.

Definition 1 (*ASP Debugging Task*)

A *logic program debugging task* is denoted as a tuple $\langle P, Lit, \Phi, \Psi, cost\rangle$ where P is a normal logic program, where $Lit \subseteq \mathcal{B}_P$ is the set of literals that can be used for revision, $\Phi, \Psi \subseteq 2^{\mathcal{B}_P \cup \text{not } \mathcal{B}_P}$ are sets of *desirable* and *undesirable* properties of P respectively, such that $\Phi \cap \Psi = \emptyset$. The cost function computes a difference metric between two logic programs. A debugging task is *valid* iff: $\nexists I \in \Pi_P : \forall \phi \in \Phi, I \models \phi$, or $\exists \psi \in \Psi : \exists I \in \Pi_P, I \models \psi$.
A program P' is a revised alternative for P if:
• $\forall r' : A :- B, \textbf{not } C \in P' : \exists r : A :- B', \textbf{not } C' \in P : B \setminus B' \subseteq Lit, B' \setminus B \subseteq Lit, C \setminus C' \subseteq Lit, C' \setminus C \subseteq Lit$ or $A \cup B \cup C \subseteq Lit$
• $\forall r : A :- B, \textbf{not } C \in P : \exists r' : A :- B', \textbf{not } C' \in P' : B \setminus B' \subseteq Lit, B' \setminus B \subseteq Lit, C \setminus C' \subseteq Lit, C' \setminus C \subseteq Lit$ or $A \cup B \cup C \subseteq Lit$

Let $\mathcal{R}_P$ be the set of all alternatives to P. A revised program $P' \in \mathcal{R}_P$ is a solution to the task, iff:
(i) $P' \in argmin\{cost(P, P'') : P'' \in \mathcal{R}_P\}$,
(ii) $\exists I \in \Pi_{P'} : \forall \phi \in \Phi, I \models \phi$,
(iii) $\forall \psi \in \Psi : \forall I \in \Pi_{P'}, I \not\models \psi$.

Given a revised rule r' of the revised program P', comprising head literals A, positive body literals B and negative body literals $\textbf{not } C$ there is a corresponding rule r in the original program P, comprising head literals A, positive body literals B' and negative body literals $\textbf{not } C'$ where the new added literals to B'(or C') or removed literals from B'(or C') must be in the set Lit. Conversely, given a rule r in the original program P, comprising head literals A, positive body literals B and negative body literals $\textbf{not } C$, there is a corresponding rule r' of the revised program P' comprising head literals A, positive body literals B' and negative body literals $\textbf{not } C'$, where the new added literals to B (or C) or removed literals from B(or C') must be in the set Lit.

Thus, the revised program P' is a *debugged* program in terms of the specified properties. As described in the definition, the purpose of debugging an ASP program is to produce a revised program P', which is able to produce at least one answer set with the all specified desirable literals Φ present and none of the answer sets contain any undesirable literals in Ψ. Intuitively, the program P' eliminates all the undesirable properties while preserving the desirable ones. Additionally, the revised program P' has minimal difference with the original one, as measured by the *cost* function. A detailed example of debugging a program is given in Section 3.5.

3 Debugging ASP programs with ILP

3.1 Overview

Given a normal logic program P, and desirable and undesirable properties Φ and Ψ, the debugging task is to revise P to satisfy the specified properties, that is, the revised program P' can produce at least one answer set satisfying all properties in Φ while no answer set contains any component in Ψ. In order to obtain such a solution, we first need to compute all the possible changes we could make to P, which includes adding/removing a literal to/from a rule, or adding or deleting a rule all together. Given a set of literals Lit, we provide structural and content specifications (called *mode declarations*: see Section 3.2) of the rules that can be constructed with Lit, which determine the learning space for the alternatives to the rules currently specified in the program P. By encoding the properties Φ and Ψ as constraints, the debugging task can be converted into a boolean satisfiability problem (SAT) and solved by existing techniques. In this paper, we employ *Inductive Logic Programming*.

Inductive Logic Programming (ILP) (Muggleton 1995) is a machine learning technique used to obtain logic theories by means of generalising (positive and negative) examples with respect to a prior base theory. In this paper we are interested in the revision of logic programs in light of desirable and undesirable properties. We want to support the synthesis of new rules and the deletion or revision of existing ones by means of examples. A positive (negative) example in this context is a collection of ground literals that represents (un)desirable properties for the debugging task. The task then is not learning a new theory, but rather revising an existing one. The space of possible solutions, i.e the revised theories, needs to be well-defined to make the search space as small as possible. This is achieved by a so-called "language bias". Only theories satisfying this bias can be learned. Mode declarations (Muggleton 1995) are one way of specifying this language bias, determining which literals are allowed in the head and body of the rules of the theory. In this paper, we also employ mode declarations to constrain the structure of rules, and we encode mode declarations as *revision tuples* (details are given in Section 3.2), which represent all revision operations we could apply to a program to derive a debugged program satisfying the language bias.

In our system, users are firstly prompted to provide examples characterising properties, and a set of (head and body) literals, from which mode declarations are generated to, in turn, produce all revision tuples. The *ILP debugger* then selects the revision tuples that are able to derive the debugged program that satisfies the stated properties. It is considered preferable (Corapi et al. 2011) that a revised program should be as similar to the original one as possible. This suits the purpose of debugging while maintaining, as much as possible, the aims and design of the program being revised. One measure of minimality, similar to (Wogulis and Pazzani 1993), can be defined in terms of the *number of revision operations*. Revision operations are: (i) deleting a rule (i.e. removing an existing rule), (ii) adding a rule (i.e. forming a new rule), and (iii) adding or deleting body elements (i.e. revising an existing rule). We define a cost function $cost(P, P')$ to determine the cost of revising program P to P'.

The revision process can be applied repeatedly, but the results at each step depend on the set of (un)desired literals that are specified and it is important to note that the revision process is non-monotonic. Specifically, $P \xrightarrow{\mathcal{R}_1} P' \xrightarrow{\mathcal{R}_2} P''$ may not result in the same program as $P \xrightarrow{\mathcal{R}_2} P' \xrightarrow{\mathcal{R}_1} P''$. Thus for incremental revision, the (un)desired literals must be accumulated to ensure that revisions made at one step are not undone at another. This iterative process is driven by the programmer, when she finds more issues with the program she is debugging. At each revision step all the desired and undesired literals are taken into account.

3.2 Mode Declaration and Revision Tuples

In the definition of the ASP debugging task (Definition 1) we defined revision alternatives for a given program and set of literals. In ILP, this is determined by means of mode declarations.

Definition 2 (*Mode Declaration*)
A mode declaration is a tuple of the form $\langle id, pre(l), var(l)\rangle$ with $l \in Lit$, Lit a set of literals and id the unique label of a mode declaration. $pre(l)$ is the predicate associated with the literal l, $var(l)$ the associated variables.
Lit can be divided into head literals Lit^h and body literals Lit^b, such that $Lit = Lit^h \cup Lit^b$. For a set of mode declarations M, we have head mode declarations $M^h = \{m|m = \langle id, pre(l), var(l)\rangle, l \in Lit^h\}$, and body mode declarations $M^b = \{m|m = \langle id, pre(l), var(l)\rangle, l \in Lit^b\}$.

We next define literal compatibility, where a literal might be a constituent of a logic program. Consequently, we define rule compatibility in terms of literal compatibility. Thus, given a set of literals compatible with M, a set of compatible rules can be formed to derive a constrained search space for a debugging task.

Definition 3 (*Literal Compatibility*)
Given a mode declaration m, a literal l' is *compatible* with m iff (i) l' has the *predicate* defined in $pre(l)$ of m, and (ii) l' has the *variables* in $var(l)$ of m.

Definition 4 (*Rule Compatibility*)
Given a rule r formed by a head literal h and several body literals b_i: $h \leftarrow b_0, \ldots, b_n$, the rule r is compatible with the mode declarations M iff: (i) there is a head mode declaration $m \in M^h$ compatible with h, and (ii) there is a body mode declaration $m \in M^b$ compatible with $b_i, i \in [0, n]$.

We have described the purpose of mode declarations in the revision process but to realize a computational solution, we need a representation. That is the purpose of this section and as such, it is just a technical explanation of the means to synthesize the revisions of the rules of a given set of literals. The revision of a rule takes one of two forms: (i) *specialization:* in which a literal is added to the body, thus adding a constraint or (ii) *generalization:* in which a literal is removed from the body, thus removing a constraint. In order to guide this process, we use mode declarations that constrain the search space and lead to the construction of *revision tuples* rev that describe specific rule revisions, i.e. following a revision tuple, a rule can be revised to be another. One or more revision tuples comprise the final solution to the debugging task in the later sections.

The key to forming revision tuples is to generate them for *all possible* revisions that could be applied to the rules of a program. A deletion operation is quite simple, but addition is more complicated, because we need to consider not only which literal to add, but also the relation between existing variables and those carried by the new literal. Each possible relation may result in different rule structures. To operationalise this process, *bound variable tuples*, denoted Ξ_b^h, are defined over a pair of head h and body literal b, where each element of the tuple corresponds to a body variable and collects the indices of head variables whose type is the same as the body variable. For instance, suppose we add a new body literal b to a rule with the head literal h, we have $var(h) = \langle \mathtt{P1}, \mathtt{P2}, \mathtt{Q1} \rangle$ and $var(b) = \langle \mathtt{P3}, \mathtt{Q2} \rangle$, such that P1, P2 and P3 are of one type and Q1 and Q2 are of another. The corresponding Ξ_b^h is then $\langle \langle 0, 1 \rangle, \langle 2 \rangle \rangle$ in which the first inner tuple is a collection of head variable indexes for P3 and the second for Q2. If there is no related variables between the head literal and the new body literal, then the Ξ_b^h is $null$.

Definition 5 (*Revision Tuple*)
A revision tuple rev is the representation of a revision operation in respect of a particular rule: $rev = \langle rId, \Theta, cost \rangle$, where rId is the unique identifier of the rule in a program. Θ denotes the structure of the revised rule. $cost$ is the metric associated with each revision operation. By default, $cost$ is 1 unit. There are two forms of Θ, defining addition and deletion operations, respectively:
1. $\Theta = \langle h!_{id},\ b!_{id},\ \Xi_b^h \rangle$, where $h \in M^h$, $b \in M^b$ and Ξ_b^h is the bound variable tuple. A revision tuple rev with such Θ implies an addition operation which extends the rule with a new body literal b in terms of Ξ_b^h.
2. $\Theta = \langle h!_{id},\ i \rangle$ where $h \in M^h$ and i is the index of an existing body. A revision tuple rev with such Θ implies a deletion operation to remove the *i*th body literal.
A set of revision tuples is denoted by $\mathcal{R}$. The relation $P \overset{\mathcal{R}}{\rightarrow} P'$ indicates a program P is revised to P' by applying revision tuples in $\mathcal{R}$. For now, the cost of a set of revision tuples $\mathcal{R}$ is determined by the sum of the $cost$ of each individual tuple in the set.

The revision tuples express all applicable revisions to the current program. As we employ ASP to implement the whole procedure, we translate revision tuples to corresponding ASP facts. Concrete examples of revision tuples can be found in the case study in Section 3.5. For instance, in line 32 of Fig.2, the revision tuple is encoded as `rev(1,(hNull, bEqNeg,null),1)`, denoting that a new body literal with id `bEqNeg` should be added to rule `1` at cost `1` unit.

3.3 Formal Procedures for Debugging

Now we can formalize the procedures of a program debugging task, building on the presented definitions. We rephrase a debugging task into a theory revision task such that the solution results in a debugged program satisfying the properties. Given a debugging task $\langle P, Lit, \Phi, \Psi, cost \rangle$, the following steps compute the solution:

1. Construct M based on Lit; mode declarations for the given set of literals.
2. Encode C from Φ and Ψ; encode the set of properties as constraint rules.
3. Derive $\mathcal{R}$ from M; all revision tuples compatible to the rules in P.

4. Derive $\widetilde{P}^d$ and $\widetilde{P}^a$ from P; convert a program P to revisable forms by applying certain syntactic transformations. $\widetilde{P}^d$ is to learn any necessary deletion operations and $\widetilde{P}^a$ for addition operations. These revisable forms use abducibles to indicate whether an existing rule remains unchanged or has literals deleted or added.
5. Learn $\Pi_{\mathcal{R}'}$ to form P', where $\forall \mathcal{R}' \in \Pi_{\mathcal{R}'}, \mathcal{R}' \subseteq \mathcal{R}$, $P \overset{\mathcal{R}'}{\to} P'$, $P' \models C$; A set of solutions $\Pi_{\mathcal{R}'}$ is learned. Each solution $\mathcal{R}'$ revises rules in P, giving P' which is a debugged program with regard to the specified properties.
6. Select $\mathcal{R}'_{min}$ to optimise P': select the solution $\mathcal{R}'_{min} \in \Pi_{\mathcal{R}'}$ with minimum cost, resulting the P' with the minimum difference from P.

As defined in Def. 1, the optimisation is guaranteed by the *cost* function, which computes the differences between two programs in terms of the operations needed to revise one to the other. For now, we assign a unit cost to each operation, so the total cost is the number of operations. If there is a desire to weight operations differently, the cost associated with each operation can be customized.

3.4 Implementation

Inductive Logic Programming (ILP), as discussed in Section 3.1, is a symbolic machine learning technique which is capable of generating revisions to a problematic program P in order to satisfy some specified properties expressed by examples (i.e. ground literals). The solutions preserve the desirable properties while eliminating undesirable properties. In our case, properties reflect our debugging goals, by which some ground literals are expected or not expected to appear in the resulting answer sets. All the solutions produced by the ILP debugger are guaranteed to satisfy those properties. Furthermore, we use the *cost* criterion to make the revised program to as close as possible to the original, for convenience of human inspection and to minimize the cost of grounding and runtime.

3.4.1 Obtaining the Revision Tuples

When given a program and a set of (un)desirable literals, our aim is to find out which rules, or more precisely which *parts* of some rules, need to be revised. The revisions typically involve *deletion* of existing body atoms, or *addition* of new body atoms. Therefore, we need to adapt the existing rules to be ready for searching for possible changes. The following adaptations are designed for such purpose: we first use the `try/3` predicate to label each body atom of the existing rules. Next we collect all possible revision tuples for deletion. Each `try/3` literal is extended by a pair of use-delete rules, from which we can construct two variants: one with the atom referenced in the `try/3` literal while the other with the to-be-deleted atom. The two variants will be then examined later against the specified properties in the resulting answer sets. If the variant with the to-be-deleted atom succeeds, it implies a deletion operation is needed. The relevant revision tuples are also generated as part of this process and are used to capture the required operations. Next we need to obtain all the revision tuples for addition. We use the predicate *extension/2* to label each head atom of the rules, and extend the *extension/2* rules with other valid literals to derive different variants. If any of those variants satisfy all the properties, then an addition operation is required.

By means of these syntactic transformations, a normal program P is converted into its revisable forms $\widetilde{P}^d$ and $\widetilde{P}^a$. More importantly, all possible revision operations for a program are explored, with the corresponding revision tuples collected in the set $\mathcal{R}$. Now we move to the abductive stage to learn solutions using $\mathcal{R}$.

3.4.2 Abduction and Optimisation

Both inductive and abductive learning take *examples* as inputs, from which inductive learning aims to learn a rule to generalise the example, while abductive learning seeks explanations/causes for the example. In the our ASP debugger, we achieve inductive learning by abductive learning, through learning revision tuples for rules that invalidate the examples. In our case, examples are the literals characterising desirable and undesirable properties.

Given a set of desirable and undesirable literals, a set of revision tuples collected in $\mathcal{R}$ and the revisable forms $\widetilde{P}^d$ and $\widetilde{P}^a$ of the program P, we are ready for abductive learning. All solutions are computed by the ILP debugger, which is the program comprising: $B \cup \widetilde{P}^d \cup \widetilde{P}^a \cup \mathcal{R} \cup C$. B is the fixed parts of P, including grounding and other domain facts. The set of constraint rules C captures desirable and undesirable properties. Each generated answer set represents a solution to the problem, i.e. a set of revision tuples denoting alternative suggestions to revise the original program to satisfy the stated properties.

As stated earlier, we are only interested in the revision with the minimum cost between P and P'. The total difference in cost between P and P' is the number of revision operations stated in a solution. In order to find the solution resulting in the minimum cost, we take advantage of the *aggregate* statement provided by CLINGO (Gebser et al. 2007), specifying a lower and an upper bound by which the weighted literals can be constrained, $:-\mathbf{not}\ [\mathtt{rev}(\_,\_,\mathtt{Cost}) = \mathtt{Cost}]\mathtt{Max}$. We apply an incremental strategy for the variable $\mathtt{Max}$, for example, if no solution can be found when $\mathtt{Max} = \mathtt{1}$, then we continue incrementally until a solution is found. A direct encoding in ASP using minimise statements or weak constraints, would require the grounding of more eleborate revisions which may not be needed. Our incremental approach only grounds what is necessary.

As the cost `Max` increases, the computation time increases accordingly, but the cost is bounded because the whole search space of rules is finite, so the number of possible operations is therefore bounded. Thus, there is a maximum cost `Max`. While this guarantees that the program terminates, it does not guarantee that the program always returns a solution, as the user might have provided unsatisfiable properties.

3.5 Example: Debugging Sudoku Solver

In this section, we present a simple but illustrative example to demonstrate the proposed debugging approach. In the example, a Sudoku solver P is implemented of which Fig.2 lists part in lines 1–11. *Rule 1* (line 6-7) is a constraint specifying that a digit *cannot* appear twice in a column. However, we intentionally omitted the body condition $\mathbf{not}\ \mathtt{equal}(\mathtt{XA},\mathtt{XB})$ to give us a problematic solver P. *Rule 2* (line 9 to 10) states a digit cannot appear twice in a row. We omit the constraint rules for sub-square and other fact rules for grounding to save space. The most important atom is $\mathtt{state}(\mathtt{X},\mathtt{Y},\mathtt{N})$, which is true when the number `N` is placed at the square X in row Y.

3	2	4	1
4	1	2	3
1	4	3	2
2	***?***	***?***	***?***

(a)

3	2	4	1
4	1	2	3
1	4	3	2
2	***3***	***1***	***4***

(b)

Fig. 1. Example: (a) Sudoku puzzle 4×4; (b) solution

Case Study: a Sudoku solver debugging task $\langle P, Lit, \Phi, \Psi, cost\rangle$, where:
the original (partial) program for the solver P is given as below:

```
%--- 4 possible locations and vaules
position(1 .. 4). value(1 .. 4).
%------ Each square may take any value
1{state(X,Y,N) : value(N) }1 :- position(X), position(Y).
%--- Rule 1: a number can appear twice in a column
:- state(XA,Y,N), state(XB,Y,N), position(XA), position(XB),
   position(Y), value(N).
%--- Rule 2: a number cannot appear twice in a row
:- state(X,YA,N), state(X,YB,N), position(X), position(YA),
   position(YB),value(N), not equal(YA, YB).
......
```

Desirable Φ and undesirable properties Ψ are converted into constraint rules:

```
:- not state(4, 1, 2).
:- state(4,2,2).   :- state(4,2,1).   :- state(4,2,4).
:- state(4,3,4).   :- state(4,3,2).   :- state(4,4,3).
:- state(4,4,1).   :- state(4,3,3).   :- state(4,4,2).
```

Revisable forms of *Rule 1*:

```
:- try(1, 1, state(XA,Y,N)), try(1, 1, state(XB,Y,N)),
   extension(1, (XA, XB)).

% keep or not the body literal state(XA,Y,N)
try(1, 1, state(XA,Y,N)) :- state(XA,Y,N).
try(1, 1, state(XA,Y,N)) :- not state(XA,Y,N), rev(1, (1, b1), 1).
% keep or not the body literal state(XB,Y,N)
try(1, 1, state(XB,Y,N)) :- state(XB,Y,N).
try(1, 1, state(XB,Y,N)) :- not state(XB,Y,N), rev(1, (2, b2), 1).

% add new body literal equal(A,B)
extension(1,(XA, XB)) :- equal(XA,XB),rev(1,(hNull,bEqPos,null),1).
% add new body literal not equal(A, B)
extension(1,(XA, XB)) :- not equal(XA,XB),rev(1,(hNull,bEqNeg,null),1).
% nothing needs to add
extension(1,(XA, XB)) :- rev(1,(hNull,null,(0,0)),1).
```

One of the optimised solutions (represented by revision tuples) is:

```
rev(1,(hNull,bEqNeg,null),1)
```

which suggests to add a new body literal **not** `equal(XA, XB)` to the *Rule 1*:

```
:- state(XA,Y,N), state(XB,Y,N), position(XA), position(XB),
   position(Y), value(N), not equal(XA, XB).
```

Fig. 2. Debugging the Sudoku Solver

A 4×4 puzzle is given with known digits for the first three rows in Fig.1(a). The current solver produces 24 possible guesses for the last row, but only one of them (as in Fig.1(b)) is correct according to Sudoku rules. Next we can declaratively specify desirable and undesirable properties to navigate the debugging for the solver in order to produce the expected answer sets only. Desirably, the first square on row 4 has the number 2 (`state(4, 1, 2)`). Undesirably, the second square on row 4 cannot be 2, 1 or 4, which have already appeared in other squares on the same column. Similar undesirable properties are specified for the other squares on row 4. Both desirable and undesirable properties are encoded as constraint rules in lines 12–15 in Fig.2. Using *Rule 1* as an example, we then demonstrate how to convert the rule into its revisable form and collect relevant revision tuples as `rev/3` atoms. We assume each revision tuple has unit cost. The produced optimal solution comprises a single revision tuple `rev(1, (hNull, bEqNeg, null), 1)`, indicating that *Rule 1* needs a new body literal `bEqNeg`, which corresponds to **not** `equal(XA, XB)`. Consequently, a debugged Sudoku solver is obtained with the revised *Rule 1* as shown in lines 33–34 in Fig.2.

4 Related and Future Work

In this paper we present an open-loop debugger and automatic reviser for normal logic programs under answer set semantics. We use inductive logic programming with an abductive implementation in ASP to revise the original program based on the desirable and undesirable properties specified by the user/programmer.

The approach we have outlined is both less and more than the classic reference on the algorithmic debugging of logic programs (Shapiro 1983). Less, in that it is coarse-grained, because ASP is a batch process from input to output, rather than a steppable, proof tree creation process, through which the contribution of different program fragments can be observed. But more, in that it offers whole program debugging through the specification of input and (un)desired output. While algorithmic debugging depends upon an oracle (aka the programmer) to say whether the input/output relation is correct for a *single* case, in our approach the developer provides *several* positive/negative examples against which to test the program at the same time. Algorithmic debugging isolates the fragment of code that demonstrates an incorrect input/output relation in the case of the current input, but working out how to fix the code is the programmer's responsibility. In contrast, the mechanism described here proposes changes to fix the code with respect to all the examples supplied.

In earlier work (Corapi et al. 2011; Athakravi et al. 2012), we showed that the use of ILP, implemented as an abductive logic program under answer set semantics, can be used to revise a subclass of answer set programs. Those programs were the result of an automatic translation of an action language in the domain of normative multi-agent systems and demonstrated a very small range of variation in grammatical structure. Furthermore, that revision process only handles negative examples and each revision is just one negative example. In this paper, we extend the approach to be able to deal with general answer set programs and allow each revision to take as many positive and negative examples as possible into account.

As discussed in the introduction, a number of other debugging tools for answer set programming already exist (Brain et al. 2005; Calimeri et al. 2009; Cliffe et al. 2008; Kloimüllner et al. 2013; Gebser et al. 2009; Oetsch et al. 2011; Mikitiuk et al. 2007). They all offer in some form a mechanism to locate where the solving process goes wrong, but provide little support for fixing it. (Oetsch et al. 2011) is probably the most advanced in terms of language support, notably being capable of dealing with aggregates.

Arguably, the mode of development we have set out has more in common with (Lieberman 1981), in that it encourages a form of "programming-by-example" through the user's provision of inputs and examples of (un)satisfactory outputs, than with (Shapiro 1983), which directs the user's knowledge at identifying which *part* of a program is behaving (in)correctly. However, that approach may be more comprehensible because it focusses attention on localised fragments of the program, whereas revision is a whole program process. This raises the question of how to make whole program debugging accessible given the volume of data involved. (Calimeri et al. 2009) takes some steps in this direction, but how well suited this might be to understanding the revision process is less clear, while a more general purpose approach to visualising inference (Lieberman and Henke 2015) seems potentially more appropriate.

Our current system can improve in a number of ways. The user support at present is minimal. We plan to provide the tool as plug-in for SEALION (Busoniu et al. 2013), an IDE for answer set programming. While users can specify (un)desirable properties of the program, they are currently very basic. Even with the current implementation it would be relatively easy to extend them from sets of literals to sets of rules. The language bias determines the search space for the revised program. At the moment, the bias is fixed through our mode declaration including all literals of the program. In future versions, we want it to be possible for the programmer to steer the revisions if he/she knows were the problem might lie. This will reduced the run-time of the system, but the resulting revisions will be the same, asssuming that the programmer has specified an appropriate bias. Equally, based on the properties provided, the language bias can be adjusted to for example those literals that could influence the properties provided.

A major and more complex future task is to extend the capability of the system to deal with aggregates. The abstract semantics provided by (Pührer 2014) can make a good starting point for our system.

As programs evolve over time, we might still want them to adhere to the same properties as the ones the program started off with. Unit testing for answer set programming (Janhunen et al. 2010; Janhunen et al. 2011; Febbraro et al. 2011; De Vos et al. 2012) is not new. The approach presented in this paper allows for the specification of properties the system has to adhere to, and it will revise the program when the system is no longer satisfying the properties. In that respect, one might forese using this approach as an automated test-driven development method for answer set programming.

Our current system is using minimal number of revisions steps for selecting the most appropriate revision to the original programme. We plan to investigate whether for an evolving system, this results in the best possible programs.

References

ATHAKRAVI, D., CORAPI, D., RUSSO, A., DE VOS, M., PADGET, J. A., AND SATOH, K. 2012. Handling change in normative specifications. In *Declarative Agent Languages and Technologies X - 10th International Workshop, DALT 2012, Valencia, Spain, June 4, 2012, Revised Selected Papers*, M. Baldoni, L. A. Dennis, V. Mascardi, and W. W. Vasconcelos, Eds. Lecture Notes in Computer Science, vol. 7784. Springer, 1–19.

BRAIN, M., WATSON, R., AND DE VOS, M. 2005. An Interactive Approach to Answer Set Programming. In *ASP05: Answer Set Programming: Advances in Theory and Implementation*, M. De Vos and A. Provetti, Eds. Research Press International, Bath, UK, 190–202. Also available from http://CEUR-WS.org/Vol-142/files/page190.pdf.

BUSONIU, P., OETSCH, J., PÜHRER, J., SKOCOVSKY, P., AND TOMPITS, H. 2013. Sealion: An eclipse-based IDE for answer-set programming with advanced debugging support. *TPLP 13,* 4-5, 657–673.

CALIMERI, F., LEONE, N., RICCA, F., AND VELTRI, P. 2009. A visual tracer for DLV. *Answer Set Programming*, 79.

CLIFFE, O., DE VOS, M., BRAIN, M., AND PADGET, J. 2008. Aspviz: Declarative visualisation and animation using answer set programming. In *International Conference of Logic Programming*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 724–728.

CORAPI, D., RUSSO, A., DE VOS, M., PADGET, J. A., AND SATOH, K. 2011. Normative design using inductive learning. *TPLP 11,* 4-5, 783–799.

DE VOS, M., KIZA, D., OETSCH, J., PÜHRER, J., AND TOMPITS, H. 2012. Annotating answer-set programs in LANA. *Theory and Practice of Logic Programming 12,* 4-5, 619–637.

DENECKER, M. AND TERNOVSKA, E. 2008. A logic of nonmonotone inductive definitions. *ACM transactions on computational logic (TOCL) 9,* 2, 14.

ERDEM, E., LIN, F., AND SCHAUB, T., Eds. 2009. *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*. Lecture Notes in Computer Science, vol. 5753. Springer.

FEBBRARO, O., LEONE, N., REALE, K., AND RICCA, F. 2011. Unit testing in ASPIDE. In *Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers*, H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds. Lecture Notes in Computer Science, vol. 7773. Springer, 345–364.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. Conflict-Driven Answer Set Solving. In *Proceeding of IJCAI07*. 386–392.

GEBSER, M., PÜHRER, J., SCHAUB, T., TOMPITS, H., AND WOLTRAN, S. 2009. SPOCK: A debugging support tool for logic programs under the answer-set semantics. In *Applications of Declarative Programming and Knowledge Management*, D. Seipel, M. Hanus, and A. Wolf, Eds. Lecture Notes in Computer Science, vol. 5437. Springer Berlin Heidelberg, 247–252.

JANHUNEN, T., NIEMELÄ, I., OETSCH, J., PÜHRER, J., AND TOMPITS, H. 2010. On testing answer-set programs. In *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, H. Coelho, R. Studer, and M. Wooldridge, Eds. Frontiers in Artificial Intelligence and Applications, vol. 215. IOS Press, 951–956.

JANHUNEN, T., NIEMELÄ, I., OETSCH, J., PÜHRER, J., AND TOMPITS, H. 2011. Random vs. structure-based testing of answer-set programs: An experimental comparison. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, 242–247.

KLOIMÜLLNER, C., OETSCH, J., PÜHRER, J., AND TOMPITS, H. 2013. Kara: A system for visualising and visual editing of interpretations for answer-set programs. In *Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th*

Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers, H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds. LNCS, vol. 7773. Springer, 325–344.

LIEBERMAN, H. 1981. Tinker: Example-based programming for artificial intelligence. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81), Vancouver, BC, Canada, August 1981*, P. J. Hayes, Ed. William Kaufmann, 1060.

LIEBERMAN, H. AND HENKE, J. 2015. Visualizing inference. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, B. Bonet and S. Koenig, Eds. AAAI Press, 4280–4281.

MIKITIUK, A., MOSELEY, E., AND TRUSZCZYNSKI, M. 2007. Towards debugging of answer-set programs in the language PSpb. In *Proceedings of the 2007 International Conference on Artificial Intelligence, ICAI 2007, Volume II, June 25-28, 2007, Las Vegas, Nevada, USA*, H. R. Arabnia, M. Q. Yang, and J. Y. Yang, Eds. CSREA Press, 635–640.

MUGGLETON, S. 1995. Inverse entailment and progol. *New generation computing 13*, 245–286.

OETSCH, J., PÜHRER, J., AND TOMPITS, H. 2011. Stepping through an answer-set program. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR2011*, J. P. Delgrande and W. Faber, Eds. LNCS, vol. 6645. Springer, 134–147.

PÜHRER, J. 2014. Stepwise debugging in answer-set programming: Theoretical foundations and practical realisation. Ph.D. thesis, Vienna University of Technology, Vienna, Austria.

SHAPIRO, E. Y. 1983. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA.

WOGULIS, J. AND PAZZANI, M. J. 1993. A methodology for evaluating theory revision systems: Results with Audrey II. In *IJCAI*. 1128–1134.