Debugging Inconsistent Answer Set Programs

Tommi Syrjänen[*]
Helsinki University of Technology, Dept. of Computer Science and Eng.,
Laboratory for Theoretical Computer Science,
P.O.Box 5400, FIN-02015 HUT, Finland
Tommi.Syrjanen@tkk.fi

Abstract

In this paper we examine how we can find contradictions from Answer Set Programs (ASP). One of the most important phases of programming is debugging, finding errors that have crept in during program implementation. Current ASP systems are still mostly experimental tools and their support for debugging is limited. This paper addresses one part of ASP debugging, finding the reason why a program does not have any answer sets at all. The basic idea is to compute diagnoses that are minimal sets of constraints whose removal returns consistency. We compute also conflict sets that are sets of mutually incompatible constraints. The final possible source of inconsistency in an ASP program comes from odd negative loops and we show how these may also be detected. We have created a prototype for the ASP debugger that is itself implemented using ASP.

Introduction

One of the most important phases in computer programming is always debugging; no matter how much care is used in program writing, some errors will creep in. For this reason a practical Answer Set Programming (ASP) system should have support for program debugging. It is not possible to detect all errors automatically since a construct may be an error in one case but correct code in another.

The current ASP systems (Niemelä, Simons, & Syrjänen 2000; Dell'Armi *et al.* 2001; East & Truszczyński 2001; Anger, Konczak, & Linke 2001; Babovich 2002; Lin & Zhao 2002) are still on experimental level and their support for debugging is limited. In this paper we examine how we can debug one class of program errors, namely finding the contradictions in a program. We have developed a prototype debugger implementation for the SMODELS input language but the same principles are applicable for most ASP systems.

Program defects can be roughly divided into two classes (Aho, Sethi, & Ullman 1986):

- *syntax errors*: the program does not conform with the formal syntax of the language; and
- *semantic errors*: the program is syntactically correct but does not behave as the programmer intended.

[*]This research has been funded by the Academy of Finland (project number 211025).

In this discussion we leave out syntactical errors since they are generally easy to remedy: the ASP system notes that the program is not valid and outputs an error message telling where the problem occurred.

The semantical errors are more difficult to handle. In the context of ASP, they too can be roughly divided into two classes:

- *typographical errors* such as misspelling predicate or variable names, using a constant in place of a variable or vice versa; and
- *logical errors* where a rule behaves differently from what was intended.

The intuition of the division is that an error is typographical if it is caused by a simple misspelling of a single syntactical element. For example, using $\mathit{corect}(X)$ instead of $\mathit{correct}(X)$. On the other hand, a logical error is one where the programmer writes a rule that does not do what he or she expects it to do. For example, a programmer writing an encoding for a planning problem might want to state the constraint that an object may be at one place at a time by using the rule:

$$\leftarrow \mathit{at}(O, L_1, I), \mathit{at}(O, L_2, I).$$

The problem is that the values of L_1 and L_2 are not constrained and may take the same value. Thus, for each object o, location x, and time step i, there will be a ground instance:

$$\leftarrow \mathit{at}(o, x, i), \mathit{at}(o, x, i).$$

which causes a contradiction no matter where the object is. In this case the programmer should have added a test $L_1 \neq L_2$ to the rule body.

Our experience is that finding the reason for a contradiction is one of the most laborious tasks in ASP debugging. Currently the most practical approach is to remove rules from the program until the resulting program has an answer set and then examining the removed rules to see what caused the error.

In this paper we examine how we can automate this process using ASP meta-programming. When we have a contradictory program, we create several new ASP programs based on it such that their answer sets reveal the possible places of error.

We borrow our basic idea from the model-based diagnosis (Reiter 1987) field. There we have a system that does not behave like it should and a diagnosis is a set of components whose failure explains the symptoms. In our approach a *diagnosis* is a set of rules whose removal returns consistency to the program. However, we do not attempt construct a standard diagnostic framework. The reason for this is pragmatic: our aim is to create a practical tool that helps answer set programmers to debug their programs. It is not reasonable to expect that a programmer would have an existing system description that could be analyzed since that would in effect be a correct program. On the other hand, we are not willing to leave the debugger completely without of formal semantics. One of the strengths of ASP is that all programs have declarative semantics so it seems natural that also their diagnoses have one. Thus, we construct our own formal framework that shares some features with model-based diagnosis but is different in other areas.

When we construct diagnoses, we are interested in minimal ones. There are several possible ways to define minimality and we will use *cardinality minimality*: a diagnosis is minimal if there is no diagnosis that contains fewer rules than it. Another possibility would be *subset minimality* where a diagnosis is minimal if it does not contain another diagnosis as its subset. We chose cardinality minimality mainly because it was easier to implement in the prototype and also because it is possible that smaller diagnoses are easier to handle in practical debugging.

Not all minimal diagnoses are equally good for debugging purposes. For example, consider the program:

$$\{a\}\ . \tag{1}$$
$$b \leftarrow a. \tag{2}$$
$$c \leftarrow \text{ not } a. \tag{3}$$
$$\leftarrow 1\ \{b, c\}\ . \tag{4}$$

Here (1) says that a may be true or false, (2) tells that b is true if a is true, (3) that c is true if a is not, and finally (4) is a constraint stating that it is an error if either b or c is true.

No matter what truth value we choose for a, either b or c is true, so we have a contradiction. The minimum number of rules that we have to remove to repair consistency is one: removing either (2), (3), or (4) results in a consistent program. Removing (4) gives the most information to the programmer since neither $b \leftarrow a$ nor $c \leftarrow \text{not } a$ can cause the contradiction by themselves. On the other hand, (4) is a constraint telling that its body should not become true so the connection to the contradiction is immediate.

We take the approach that we include only constraints in minimal diagnoses. Examining just them is not enough since a contradiction can arise also from an *odd loop*. An odd loop is a program fragment where an atom depends recursively on itself through an odd number of negations. The simplest example is:

$$a \leftarrow \text{ not } a.$$

This rule causes a contradiction since if a is set to false, we have to conclude that a is true. On the other hand, if a is set to true, the body of the rule is not satisfied so we do not have a justification for a and we have to set it false.

Not all odd loops are errors since they may be used to prune out unwanted answer sets. Since it is difficult to determine which odd loops are intentional and which are errors, we take the approach that all odd loops are considered to be errors.

This means that the programmer has to use some other construct to replace the odd loops. In SMODELS the alternative approach is to first generate the possible model candidates using *choice rules* of the form:

$$\{\mathit{head}\}\ \leftarrow \mathit{body}.$$

Here the intuition is that if *body* is true, then *head* may be true but it may be also false. The pruning is then done using *constraints* of the form:

$$\leftarrow \mathit{body}.$$

A constraint asserts that the *body* must be false. Note that a constraint is actually an odd loop in a disguise: we could replace a constraint by the equivalent rule:

$$f \leftarrow \mathit{body}, \text{not} f.$$

In general, a program may have a number of different minimal diagnoses. In many cases some constraints occurring in them are related to each other. For example, in program:

$$\{a\}\ .$$
$$\leftarrow a. \tag{1}$$
$$\leftarrow \text{ not } a. \tag{2}$$
$$\leftarrow \text{ not } b. \tag{3}$$

there are two different diagnoses: $\{1, 3\}$ and $\{2, 3\}$. Here the constraints (1) and (2) both depend on the value of a. If a is chosen to be true, then (1) fails, if not, (2) fails. In effect, we can have either (1) or (2) in the program, but not both. The constraint (3) is independent from the other two and it always fails.

A *conflict set* is a way of formalizing the concept of related constraints. The intuition is that a set of constraints is a conflict set if every diagnosis of the program contains exactly one member from the set.[1] We use the conflict sets to give more information to the programmer. In the above program the two conflict sets are $\{2, 3\}$ and $\{4\}$. In general, if two rules belong in the same conflict set, the truth values of the literals that occur in their bodies depend on same truth values of same atoms: choosing one value leads to one contradiction and choosing the other leads to another. Grouping them together may lead the programmer to the place of error faster.

Note that there are programs whose constraints cannot be divided into conflict sets. In those cases we cannot use conflict sets to help debugging and have to use other methods. Fortunately, those cases seem to be quite rare in practice.

[1]Note that conflict sets are different from conflicts. In model-based diagnosis a conflict is a set of components that contains at least one malfunctioning component.

Related Work

Brain et. al. (Brain, Watson, & De Vos 2005) presented an interactive way for computing answer sets. A programmer can use the interactive system as a debugging aid since it can be used to explain why a given atom is either included in an answer set or left out from it. Their approach is very similar to our method of computing explanations for diagnoses.

The NoMoRe system (Anger, Konczak, & Linke 2001) utilizes blocking graphs that can be used to examine why a given rule is applied or blocked and thus they provide a visual method for debugging ASP programs.

The consistency-restoring rules of Balduccini and Gelfond (Balduccini & Gelfond 2003) are another related approach. They define a method that allows a reasoning system to find the best explanation for conflicting observations. The main difference between our approaches is that we do not try to fix the contradictory program but instead try to help the programmer to find the places that are in error.

There has been a lot of previous work on the properties of odd and even cycles in a program (for example, (You & Yuan 1994; Lin & Zhao 2004; Costantini & Provetti 2005; Constantini 2005)) and how they affect the existence and number of answer sets. In this work we propose methodology where even loops are replaced by choice rules and odd loops by constraints, so our viewpoint is slightly different. However, the theoretical results of previous work still hold since our programs could be translated back to normal logic programs. In particular, constraints are equivalent to one-rule odd loops.

The most closely related area of odd loop research is Constatini's work on static program analysis (Constantini 2005). She notes that there are two different ways to escape the inconsistency caused by an odd loop: either there has to be one unsatisfied literal in the body of at least one rule of the loop or there has to be a non-circular justification for some atom in the loop. The literals that are present in rule bodies but are not part of the loop are called AND-handles and the extra rules are OR-handles. In every answer set of the program there has to be an applicable handle for every odd loop in it. Since the handles are purely syntactic properties, we can statically analyze the rules to see what conditions have to be met so that all loops are satisfied. This approach seems promising but there is currently the limitation that the definitions demand that the program is in kernel normal form. This is not an essential limitation from theoretical point of view since every normal logic program can be systematically translated to the normal form, but it will cause an extra step in practical debugger since the results have to be translated back to the original program code.

Language

In this paper we construct a debugger for a subset of SMODELS language.[2] We will consider only finite ground programs that do not have cardinality constraint literals but that may have choice rules.

[2]The actual debugger implementation handles the complete language.

The basic building block of a program is an *atom* that encodes a single proposition that may be either true or false. A *literal* is either an atom a or its negation $\mathrm{not}\ a$.

A *basic rule* is of the form:

$$h \leftarrow l_1, \ldots, l_n$$

where the *head* h is an atom and $l_1, \ldots, l_n$ in the *body* are literals. The intuition is that if all literals $l_1, \ldots, l_n$ are true, then h has to be also true. If the body is empty ($n = 0$), then the rule is a *fact*. A *choice rule* has the form:

$$\{h\} \leftarrow l_1, \ldots, l_n$$

where h and l_i are defined as above. The intuition of a choice rule is that if the body is true, then the head may be true but it may also be false. If an atom does not occur in the head of any rule that has a satisfied body, it has to be false.

Basic and choice rules are together called *generating* rules. The other possibility is a *constraint* that is a rule without a head. If the body of a constraint becomes true, then the model candidate is rejected. A *logic program* $P = \langle \mathcal{G}, \mathcal{C} \rangle$ is a pair where $\mathcal{G}$ is a finite set of generating rules and $\mathcal{C}$ a finite set of constraints.

Before we can define the formal ASP semantics for these programs, we need to define notation that allows us to refer to the parts of a rule. Let $r = h \leftarrow a_1, \ldots, a_n, \mathrm{not}\ b_1, \ldots, \mathrm{not}\ b_m$ be a basic rule where a_i and b_i are atoms. Then,

$$\begin{aligned} \mathrm{head}(r) &= h \\ \mathrm{body}^+(r) &= \{a_1, \ldots, a_n\} \\ \mathrm{body}^-(r) &= \{b_1, \ldots, b_m\}\ . \end{aligned}$$

The same notation is used for choice rules. We use $\mathrm{Atoms}(P)$ to denote the set of atoms that occur in a program P.

A set of atoms S *satisfies* an atom a (denoted by $S \models a$) iff $a \in S$ and a negative literal $\mathrm{not}\ a$ iff $a \notin S$. A set S satisfies a set of literals L iff $\forall l \in L : S \models l$. A set S satisfies constraint $\leftarrow l_1, \ldots, l_n$ iff $S \not\models l_i$ for some $1 \leq i \leq n$.

The ASP semantics is defined using the concept of a *reduct* (Gelfond & Lifschitz 1988). The reduct P^S of a program $P = \langle \mathcal{G}, \mathcal{C} \rangle$ with respect of a set of atoms S is:

$$\begin{aligned} P^S &= \langle \mathcal{G}^S, \mathcal{C} \rangle, \text{ where} \\ \mathcal{G}^S &= \{\mathrm{head}(r) \leftarrow \mathrm{body}^+(r) \mid r \in \mathcal{G}, S \models \mathrm{body}^-(r), \\ &\qquad \text{and } r \text{ is either a basic rule or a} \\ &\qquad \text{choice rule and } \mathrm{head}(r) \in S\}\ . \end{aligned}$$

Note that all rules that belong to the generator part of a reduct P are basic rules and all literals that occur in them are positive. Such rules are monotonic so $\mathcal{G}^S$ has a unique least model (Gelfond & Lifschitz 1988) that we denote with $\mathbf{MM}(\mathcal{G}^S)$. If this least model happens to coincide with S and it also satisfies all constraints, then S is an answer set of P.

Definition 1 *Let $P = \langle \mathcal{G}, \mathcal{C} \rangle$ be a program. A set of ground atoms S is an* answer set *of P if and only if:*

1. $\mathbf{MM}(\mathcal{G}^S) = S$*; and*
2. $\forall r \in \mathcal{C} : S \models r$*.*

A program P is *consistent* if it has at least one answer set and *inconsistent* if it has none.

Theory for Debugging

Odd Loops

Definition 2 *The* dependency graph $DG_P = \langle V, E^+, E^- \rangle$ *of a program* $P = \langle \mathcal{G}, \mathcal{C} \rangle$ *is a triple where* $V = Atoms(P)$ *and* $E^+, E^- \subseteq V \times V$ *are sets of* positive *and* negative *edges such that:*

$$E^+ = \{\langle h, a\rangle \mid \exists r \in \mathcal{G} : head(r) = h \text{ and } a \in body^+(r)\}$$
$$E^- = \{\langle h, b\rangle \mid \exists r \in \mathcal{G} : head(r) = h \text{ and } b \in body^-(r)\} \ .$$

Definition 3 *Let* $DG_P = \langle V, E^+, E^- \rangle$ *be a dependency graph. Then the two* dependency relations Odd_P *and* Even_P *are the smallest relations on* V *such that:*

1. for all $\langle a_1, a_2\rangle \in E^-$ *it holds that* $\langle a_1, a_2\rangle \in \mathrm{Odd}_P$*;*
2. for all $\langle a_1, a_2\rangle \in E^+$ *it holds that* $\langle a_1, a_2\rangle \in \mathrm{Even}_P$*;*
3. if $\langle a_1, a_2\rangle \in E^-$ *and* $\langle a_2, a_3\rangle \in \mathrm{Even}_P$*, then* $\langle a_1, a_3\rangle \in \mathrm{Odd}_P$*;*
4. if $\langle a_1, a_2\rangle \in E^-$ *and* $\langle a_2, a_3\rangle \in \mathrm{Odd}_P$*, then* $\langle a_1, a_3\rangle \in \mathrm{Even}_P$*;*
5. if $\langle a_1, a_2\rangle \in E^+$ *and* $\langle a_2, a_3\rangle \in \mathrm{Even}_P$*, then* $\langle a_1, a_3\rangle \in \mathrm{Even}_P$*; and*
6. if $\langle a_1, a_2\rangle \in E^+$ *and* $\langle a_2, a_3\rangle \in \mathrm{Odd}_P$*, then* $\langle a_1, a_3\rangle \in \mathrm{Odd}_P$*.*

The reason for the interleaved definition is that the relations Odd and Even are then easy to compute: we start by initializing them with the edges of the dependency graph, and then compute the transitive closure of the graph where every negative edge changes the parity of the dependency: if b depends on c evenly and there is a negative edge from a to b, then a depends oddly on c.

Definition 4 *Let* P *be a program. Then, an* odd loop *is a set* $L = \{a_1, \ldots, a_n\}$ *of atoms such that* $\langle a_i, a_j\rangle \in \mathrm{Odd}_P$ *for all* $1 \leq i, j \leq n$*. An atom* $a \in Atoms(P)$ occurs in an odd loop *iff* $\langle a, a\rangle \in \mathrm{Odd}_P$*. The program* P *is* odd loop free *if* $\forall a \in Atoms(P) : \langle a, a\rangle \notin \mathrm{Odd}_P$*.*

Diagnoses and Conflict Sets

Definition 5 *Let* $P = \langle \mathcal{G}, \mathcal{C} \rangle$ *be an odd loop free program. Then, a* diagnosis *of* P *is a set* $D \subseteq \mathcal{C}$ *such that the program* $\langle \mathcal{G}, \mathcal{C} \setminus D\rangle$ *is consistent. A diagnosis is* minimal *iff for all diagnoses* D' *of* P *it holds that* $|D'| \geq |D|$*. The set of all minimal diagnoses of* P *is denoted by* $\mathfrak{D}(P)$*.*

Example 1 *Consider the program:*

$$\begin{aligned} &\{a\}\ . \\ &\quad \leftarrow a. && (1) \\ &\quad \leftarrow \text{not } a. && (2) \\ &\quad \leftarrow \text{not } b. && (3) \end{aligned}$$

This program has two minimal diagnoses: $D_1 = \{1, 3\}$ *and* $D_2 = \{2, 3\}$*. To see that* D_1 *is really a diagnosis, note that when its rules are removed, we are left with:*

$$\begin{gathered} \{a\}. \\ \leftarrow \text{not } a. \end{gathered}$$

that has the answer set $\{a\}$*.*

We can observe two properties of diagnoses from Definition 5. First, if P is consistent, then it has a unique minimal diagnosis that is the empty set. The second observation is that every inconsistent program has at least one minimal diagnosis.

Theorem 1 *Let* $P = \langle \mathcal{G}, \mathcal{C} \rangle$ *be an inconsistent odd loop free program. Then there exists at least one minimal diagnosis* D *for it.*

Proof 1 *The rules in* $\mathcal{G}$ *can be systematically translated into an equivalent normal logic program* $\mathcal{G}'$ *where every choice rule is replaced by an even loop (see (Niemelä & Simons 2000) for details). Since* $\mathcal{G}'$ *is odd loop free, it is consistent (You & Yuan 1994). Thus, the set* $D' = \mathcal{C}$ *is a diagnosis. Since* $\mathcal{C}$ *is finite, there has to exist at least one minimal diagnosis* $D \subseteq D'$*.*

Definition 6 *Let* $P = \langle \mathcal{G}, \mathcal{C} \rangle$ *be a program and* $\mathfrak{D}(P)$ *the set of its minimal diagnoses. Then, a* conflict set $C \subseteq \mathcal{C}$ *is a set of constraints such that:*

1. for all diagnoses $D \in \mathfrak{D}(P)$ *it holds that* $|D \cap C| = 1$*; and*
2. for all constraints $r \in C$ *there exists a diagnosis* $D \in \mathfrak{D}(P)$ *such that* $r \in D$*.*

The set of all conflict sets of P *is denoted by* $\mathfrak{C}(P)$*.*

Intuitively, constraints that belong in a conflict set are mutually exclusive in the sense that it is impossible to have all of them satisfied at the same time. Note that with this definition it is possible that a program does not have any conflict sets at all.

Example 2 *In Example 1 we had two diagnoses* $D_1 = \{1, 3\}$ *and* $D_2 = \{2, 3\}$*. We can partition the constraints that occur in them into two conflict sets:*

$$\begin{aligned} C_1 &= \{1, 2\} \\ C_2 &= \{3\}\ . \end{aligned}$$

Example 3 *The program:*

$$\begin{array}{lll} \{a\}. & \leftarrow \text{not } a.\ (1) & \leftarrow a, b.\ (4) \\ \{b\}. & \leftarrow \text{not } b.\ (2) & \leftarrow b, c.\ (5) \\ \{c\}. & \leftarrow \text{not } c.\ (3) & \leftarrow a, c.\ (6) \end{array}$$

has six minimal diagnoses: $\{1, 2\}$*,* $\{1, 3\}$*,* $\{1, 5\}$*,* $\{2, 3\}$*,* $\{2, 6\}$*, and* $\{3, 4\}$*. We see that there is no way to partition the constraints so that every diagnosis contains exactly one rule for each set.*

The ASP Programs

In this section we create three different ASP programs that can be used to debug contradictory programs. We express these programs using the full SMODELS syntax so we need to introduce a few new constructs. We do not give here the full formal semantics but the interested reader may consult (Syrjänen 2004) for details.

A *cardinality constraint literal* is of the form $L\ \{l_1, \ldots, l_n\}\ U$ where L and U are integral *lower* and *upper bounds* and l_i are literals. A cardinality constraint literal is true if the number of satisfied literals l_i is between U and L, inclusive. Next, a *conditional literal* has the form $a(X)\ :\ d(X)$ This construct denotes the set of literals $\{a(t) \mid d(t) \text{ is true}\}$. Finally, a fact may have a *numeric range* in it and $a(1..n)$ denotes the set of n facts $\{a(1), \ldots, a(n)\}$.

Odd Loop Detection

When we do the odd loop detection, we will use the standard meta-programming encoding of logic programs (Sterling & Shapiro 1994). A rule:

$$r = h \leftarrow a, \text{not } b$$

is encoded using the facts:

$$\begin{array}{ll} \textit{rule}(r). & \textit{pos-body}(r, a). \\ \textit{head}(r, h). & \textit{neg-body}(r, b). \end{array}$$

We start the odd loop program by extracting the atoms from the program representation:

$$\begin{aligned} \textit{atom}(H) &\leftarrow \textit{head}(R, H). \\ \textit{atom}(A) &\leftarrow \textit{pos-body}(R, A). \\ \textit{atom}(B) &\leftarrow \textit{neg-body}(R, B). \end{aligned}$$

Next, we construct the dependency graph for the program:

$$\begin{aligned} \textit{pos-edge}(H, A) &\leftarrow \textit{head}(R, H), \\ &\qquad \textit{pos-body}(R, A). \\ \textit{neg-edge}(H, B) &\leftarrow \textit{head}(R, H), \\ &\qquad \textit{neg-body}(R, B). \end{aligned}$$

One step positive dependency is even, negative odd:

$$\begin{aligned} \textit{even}(X, Y) &\leftarrow \textit{pos-edge}(X, Y). \\ \textit{odd}(X, Y) &\leftarrow \textit{neg-edge}(X, Y). \end{aligned}$$

Adding a new positive edge preserves parity:

$$\begin{aligned} \textit{even}(X, Z) &\leftarrow \textit{pos-edge}(X, Y), \textit{even}(Y, Z), \textit{atom}(Z). \\ \textit{odd}(X, Z) &\leftarrow \textit{pos-edge}(X, Y), \textit{odd}(Y, Z), \textit{atom}(Z). \end{aligned}$$

Adding a negative edge flips parity:

$$\begin{aligned} \textit{odd}(X, Z) &\leftarrow \textit{neg-edge}(X, Y), \textit{even}(Y, Z), \textit{atom}(Z). \\ \textit{even}(X, Z) &\leftarrow \textit{neg-edge}(X, Y), \textit{odd}(Y, Z), \textit{atom}(Z). \end{aligned}$$

There is an odd loop if a predicate depends oddly on itself:

$$\textit{odd-loop}(X) \leftarrow \textit{odd}(X, X).$$

Two atoms X and Y are in same odd loop if X depends oddly on Y and Y depends evenly on X:

$$\textit{in-odd-loop}(X, Y) \leftarrow \textit{odd}(X, Y), \textit{even}(Y, X).$$

The above rules correspond directly to the Definitions 1–4. We could stop here, but we can make debugging a bit easier if we also identify which rules belong to which loops. We start by choosing one of the atoms that occur in a loop to act as an identifier for the loop. We take the atom that is lexicographically the first one:

$$\begin{aligned} \textit{first-in-loop}(A) &\leftarrow \textit{odd-loop}(A), \text{not } \textit{has-predecessor}(A). \\ \textit{has-predecessor}(A) &\leftarrow \textit{in-odd-loop}(B, A), B < A. \end{aligned}$$

The final part of the odd loop detection is to compute which rules belong to the loop. The idea is that if X and Y are in the same loop, then a rule that has X in the head and Y in the body participates in the loop. We also have to extract the identifier of the particular loop.

$$\begin{aligned} \textit{rule-in-loop}(R, Z) \leftarrow\ & \textit{in-odd-loop}(X, Y), \\ & \textit{in-odd-loop}(X, Z), \\ & \textit{first-in-loop}(Z), \\ & \textit{head}(R, X), \\ & \textit{pos-body}(R, Y). \\ \textit{rule-in-loop}(R, Z) \leftarrow\ & \textit{in-odd-loop}(X, Y), \\ & \textit{in-odd-loop}(X, Z), \\ & \textit{first-in-loop}(Z), \\ & \textit{head}(R, X), \\ & \textit{neg-body}(R, Y). \end{aligned}$$

Example 4 *Consider the program:*

$$a \leftarrow \text{not } b. \qquad (1)$$

$$b \leftarrow a. \qquad (2)$$

This program is expressed with facts:

$$\begin{array}{ll} \textit{head}(1, a). & \textit{neg-body}(1, b). \\ \textit{head}(2, b). & \textit{pos-body}(2, a). \end{array}$$

When these facts are given as an input for the odd loop detection program, we have a unique answer set. The relevant atoms from it are:

$$\begin{aligned} S = \{&\textit{odd-loop}(a), \textit{odd-loop}(b), \textit{first-in-loop}(a), \\ &\textit{rule-in-loop}(2, a), \textit{rule-in-loop}(1, a)\} \ . \end{aligned}$$

This answer set tells that the rules (1) and (2) form an odd loop whose identifier is a.

Finding Diagnoses

We could use the meta-representation of the previous section for also diagnosis computation but it is more efficient in practice to use a more direct translation. The basic idea is that we add a new literal to the bodies of constraint to control whether it is applied or not. For example, a constraint:

$$r = \leftarrow a, \text{ not } b.$$

is translated into two rules:

$$\leftarrow \text{not } \mathit{removed}(r), a, \text{not } b.$$
$$\mathit{constraint}(r).$$

All generating rules are kept as they were. Next, we add the rule:

$$\{\mathit{removed}(R) : \mathit{constraint}(R)\}\ n.$$

This rule asserts that at most n of the constraints may be removed. Here n is a numeric constant whose value is set from the command line.

The actual diagnoses are then computed by first setting the n to zero and then increasing the value until the translated program has an answer set. The diagnosis can then be extracted by noting the *removed*/1 atoms that are true in the answer.

Example 5 *The program from program from Example 1 is translated into:*

$$\{a\}.$$
$$\leftarrow \text{not } \mathit{removed}(1), a.$$
$$\leftarrow \text{not } \mathit{removed}(2), \text{not } a.$$
$$\leftarrow \text{not } \mathit{removed}(3), \text{not } b.$$
$$\mathit{constraint}(1..3).$$
$$\{\mathit{removed}(R)\ :\!\mathit{constraint}(R)\}\ n.$$

When we start computing the answer sets for the transformed program we note that there are no answer sets when $n = 0$ *and* $n = 1$. *With* $n = 2$ *there are two answer sets:*

$$S_1 = \{\mathit{removed}(1), \mathit{removed}(3), a\}$$
$$S_2 = \{\mathit{removed}(2), \mathit{removed}(3)\}$$

The two diagnoses can then be read directly from S_1 *and* S_3.

Finding Conflict Sets

Once we have computed all diagnoses, we can check whether the program has conflict sets. We use a fact

$$\mathit{in\text{-}diagnosis}(d, r).$$

to denote that the constraint r is in the dth diagnosis.

First, we initialize several type predicates:

$$\mathit{conflict\text{-}set}(1..s).$$
$$\mathit{diagnosis}(S) \leftarrow \mathit{in\text{-}diagnosis}(S, R).$$
$$\mathit{rule}(R) \leftarrow \mathit{in\text{-}diagnosis}(S, R).$$

Here s is again a constant that is set during the instantiation of the program.

We need two rules to compute the sets. The first one states that each rule belongs to exactly one conflict set, and the second states that every diagnosis should have exactly one rule in each conflict set:

$$1\ \{\mathit{in\text{-}set}(S, R) : \mathit{conflict\text{-}set}(S)\}\ 1 \leftarrow \mathit{rule}(R).$$
$$1\ \{\mathit{in\text{-}set}(S, R) : \mathit{in\text{-}diagnosis}(X, R)\}\ 1 \leftarrow \mathit{conflict\text{-}set}(S), \mathit{diagnosis}(X).$$

The conflict sets are computed in a same way as the diagnoses: we start with only one conflict set, and increase their number until we either find a partition or we know that none exists.

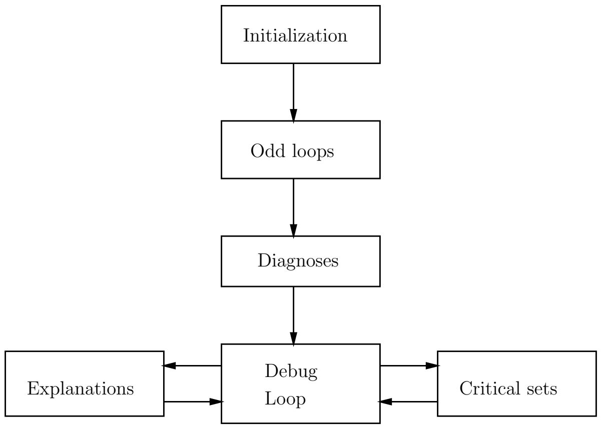

Figure 1: The debugger workflow

Example 6 *From Example 5 we get the facts:*

$\mathit{in\text{-}diagnosis}(1, 1).$	$\mathit{in\text{-}diagnosis}(1, 3).$
$\mathit{in\text{-}diagnosis}(2, 2).$	$\mathit{in\text{-}diagnosis}(2, 3).$

With these facts we find an answer set[3] *when* $s = 2$. *This answer set is:*

$$S = \{\mathit{in\text{-}set}(1, 1), \mathit{in\text{-}set}(1, 2), \mathit{in\text{-}set}(2, 3)\}$$

corresponding to $\mathfrak{C}(P) = \{\{1, 2\}, \{3\}\}$.

Debugger Implementation

We have created a prototype implementation for the ASP debugger, `smdebug`, by combining the SMODELS programs with a driver program that is written with Perl. The debugger implementation is included within the *lparse* instantiator that is available for download at `http://www.tcs.tkk.fi/Software/smodels`.

The general system architecture of `smdebug` is shown in Figure 1. The debugger has four main components:

1. Odd loop detection. If the input program has an odd loop, `smdebug` issues an error message and terminates;
2. Diagnosis computation where `smdebug` calls SMODELS to compute all minimal diagnoses of the program;
3. Conflict set computation where `smdebug` tries to find conflict sets of the program; and
4. Explanation computation where `smdebug` computes derivation trees for constraints that occur in diagnoses.

We did not examine the fourth phase in this work but its idea is to give the programmer more detailed knowledge about the reasons of the contradictions. The user selects one diagnosis, and the debugger computes which set of choices leads to this particular contradiction and presents the information in the form of a derivation tree.

[3]More precisely, we have two isomorphic answer sets.

Conclusions and Further Work

In this work we applied the techniques from the symbolic diagnosis (Reiter 1987) field to ASP debugging. The main concepts have a natural mapping into ASP programs where a diagnosis is a set of constraints whose removal returns consistency to the program. We restrict these diagnoses to programs that are created in such a way that they do not have odd loops. We use another ASP program to find the odd loops that occur in a program and to warn about them. Finally, we defined the concept of the conflict set that can be used to check which constraints are mutually exclusive.

We have created a prototype implementation, `smdebug`, that implements the three debugging techniques of this paper for the full SMODELS input language. Additionally, `smdebug` also can compute derivation trees to act as explanations for the contradictions.

The main limitation for the current version of `smdebug` is that it can be used to find only contradictions. However, some of the techniques can be adapted to also explain why a given atom is or is not in an answer set. In particular, the method of computing explanations should generalize to this direction quite easily.

The next step in continuing with the `smdebug` development is to add support for handling non-contradictory programs. This means that we have to add support for computing and analyzing answer sets of the program.

There are several avenues of further research for improving the current system. The algorithm that `smdebug` uses for finding the minimal diagnoses and conflict sets is rather naive: iteratively increasing the size of the parameter until we get a program that has an answer set. It is possible that some other approach could get us equally useful results faster. Also, using some other minimality condition, like subset minimality, might give better results in some cases.

This debugger has not been used to debug large answer set programs, yet. The largest debugged program thus far has been the part of the debugger itself. One of its early versions of the explanation generation program contained a bug that caused it to be contradictory. The debugger not only identified the place of the error immediately, but it also uncovered two bugs in the *lparse* instantiator.

It may be that the current debugging support is not strong enough to handle really large programs. In those cases probably the best way to proceed is to try to manually find the smallest input program where the error happens and to debug that one.

In conclusion, this approach seems promising for ASP development but only time will tell if it will fulfill those promises.

References

Aho, A. V.; Sethi, R.; and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company.

Anger, C.; Konczak, K.; and Linke, T. 2001. Nomore : A system for non-monotonic reasoning under answer set semantics. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, 406–410.

Babovich, Y. 2002. Cmodels, a system computing answer sets for tight logic programs.

Balduccini, M., and Gelfond, M. 2003. Logic programs with consistency-restoring rules. In *AAAI Spring 2003 Symposium*, 9–18.

Brain, M.; Watson, R.; and De Vos, M. 2005. An interactive approach to answer set programming. In *Answer Set Programming: Advances in Theory and Implementation ASP-05*, 190 – 202.

Constantini, S. 2005. Towards static analysis of answer set programs. Computer Science Group Technical Reports CS-2005-03, Dipartimento di Ingegneria, Universita‘ di Ferrara.

Costantini, S., and Provetti, A. 2005. Normal forms for answer sets programming. *TPLP* 5(6):747–760.

Dell'Armi, T.; Faber, W.; Ielpa, G.; Koch, C.; Leone, N.; Perri, S.; and Pfeifer, G. 2001. System description: Dlv. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*. Vienna, Austria: Springer-Verlag.

East, D., and Truszczyński, M. 2001. Propositional satisfiability in answer-set programming. In *Proceedings of KI 2001: Advances in Artificial Intelligence*, 138–153.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, 1070–1080. The MIT Press.

Lin, F., and Zhao, Y. 2002. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proceedings of the 18th National Conference on Artificial Intelligence*, 112–118. Edmonton, Alberta, Canada: The AAAI Press.

Lin, F., and Zhao, Y. 2004. On odd and even cycles in normal logic programs. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)*, 80–85. The AAAI Press.

Niemelä, I., and Simons, P. 2000. Extending the smodels system with cardinality and weight constraints. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer. 491–521.

Niemelä, I.; Simons, P.; and Syrjänen, T. 2000. Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*.

Reiter, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32:57–95.

Sterling, L., and Shapiro, E. 1994. *The Art of Prolog*. MIT press.

Syrjänen, T. 2004. Cardinality constraint logic programs. In *The Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA'04)*, 187–200. Lisbon, Portugal: Springer-Verlag.

You, J.-H., and Yuan, L. Y. 1994. A three-valued semantics for deductive database and logic programs t. *Journal of Computer and System Science* 49:334–361.