An Experience of Using ASP for Toy Examples

Yves Moinard

INRIA, IRISA
Campus de Beaulieu, 35042 Rennes cedex, France
moinard@irisa.fr

Abstract. We present a causal formalism, and its translation in DLV, one of the best-known working ASP implementation. We consider that this precise formalism has its own interest, however, our purpose here is not to propose a causal formalism. We describe a few problems we have encountered while trying to provide a program for such a formalism, and we make a few reflexions about this task. Since there are now serious ASP solvers, one of the interesting things to do is to use it in our daily research. This has a real interest since it is important for a new theory to be tested with a lot of toy or near-toy examples. Our concern is to see whether ASP is now well-fitted for this task, and what could be done in order to make this task easier. Some of our reflexions could appear naive for experienced ASP users, but we think that if ASP is to be used seriously, it should at least be easily usable for toy examples.

1 Introduction

We introduce a "causal formalism" which we are presently designing in collaboration with Philippe Besnard and Marie-Odile Cordier (see [1] for a related formalism). Our motivations are diagnostic, explanations and predictions, from a set of "causal formulas". The aim is to provide truth-preserving conditions between causal statements. In particular, the logic will distinguish between a pair of statements essentially related as cause and effect and a pair of statements which are merely effects of a common cause (correlations being not confused with instances of causation).

Given a few "causal formulas" expressed in this logic, we want to compute "explanation atoms" that state whether a explains b or not. This can be considered as a way of examining a situation described in terms of ordinary and causal formulas. Indeed, explanation atoms are not intended to provide new conclusions, they provide a point of view about a given situation. These explanation atoms can be used directly for diagnostic or prevision tasks.

We translate our formalism into DLV [6]. We evoke various options before providing a solution. Also, we make some comments about a few features that we would like to find in DLV, but which are not present, since we think that these improvements could help any programmer. We consider that toy examples may be, after all, a good evaluator of the possibilities of ASP.

2 The causal formalism: causal and ontological atoms

The formalism contains causal atoms such as (δ *causes* β).

The present framework (taking its inspiration in [1]) is minimal in that only a few properties (all of which seemingly uncontroversial) are imposed upon it. Additional technicalities may later enrich the framework.

As it is generally the case when dealing with situations from the real world, a minimum amount of ontological information is mandatory. This is also important here since in formulas such as (α *causes* β), α and β must be propositional symbols. This restriction allows to keep things simple and, thanks to ontological atoms, it is not too limiting. Abbreviating *"temperature in °C"* by "*t*", let us consider the atom (flu *causes* t_{38-41}): flu and t_{38-41} are propositional symbols, and we add the following four ontological atoms: $(t_{38} \rightarrow_{IS_A} t_{38-41}), \cdots, (t_{41} \rightarrow_{IS_A} t_{38-41})$.

An ontological atom is written $(\alpha \rightarrow_{IS_A} \beta)$, where again α and β are propositional formulas.

From such formulas, some consequences are to be derived, which state what can be "explained", and from what. The ontological information consists in a set of ontological atoms, while boolean combinations of propositional atoms and causal atoms are allowed:

1. *Propositional atoms*: α, $\beta, \ldots$
2. *Causal atoms*: (α *causes* β), where β is called an *effect* of α, which is a *cause*.
3. *Propositional formulas*: Boolean combinations of propositional atoms.
4. *Causal formulas*: Boolean combinations of propositional and causal atoms.
5. *Ontological atoms*: $(\alpha \rightarrow_{IS_A} \beta)$. It reads: α *is a* β.
6. *Causal theory*: Set of causal formulas and ontological atoms.
7. *Explanation atoms* A third kind of *atoms* exist, in two kinds:
   (a) (α *explains* β *if_possible* Φ), where α and β are propositional atoms, and Φ is a set of propositional atoms. It reads: α *is an explanation for* β *provided* Φ *is possible*
   (b) (α *explains* β) reads: α *is an explanation for* β.

Explanation atoms cannot appear as data of a *causal theory*. The next section describes how explanation atoms are obtained from a causal theory.

The causal and ontological operators possess only a few properties:

1. *The causal operator entails the implication*: If α *causes* β, then $\alpha \rightarrow \beta$.
2. *Properties of the ontological operator*
   (a) *Entailing the implication*: If $(\alpha \rightarrow_{IS_A} \beta)$, then $\alpha \rightarrow \beta$.
   (b) *Transitivity*: If $(\alpha \rightarrow_{IS_A} \beta)$ and $(\beta \rightarrow_{IS_A} \gamma)$, then $(\alpha \rightarrow_{IS_A} \gamma)$.
   (c) *Reflexivity*: $(\alpha \rightarrow_{IS_A} \alpha)$.

Reflexivity is unconventional for an IS_A hierarchy. It is included because it helps keeping the number of inference schemes low (in the ASP translations, since the main rules, given below, are split into different cases, reflexivity of ontology is not necessary). As for the transitivity of the causal operator [if (α *causes* β)

and (β *causes* γ), then (α *causes* γ)], it can be added or not, this will not affect the explanation atoms. We do not require reflexivity of the causal operator [(α *causes* α) for each α], but we allow an atom such as (α *causes* α) for some given α as part of a causal theory. Notice that $\alpha \rightarrow \beta$ does not entail (δ *causes* α) $\rightarrow$ (δ *causes* β) or (β *causes* δ) $\rightarrow$ (α *causes* δ).

3 The causal formalism: deriving explanation atoms

The entailment relation $\models$ in a causal theory CT is classical entailment, each atom (propositional, causal or ontological) being considered as a separate propositional atom. Remind that each causal theory contains the formulas given in the properties of Points 1 and 2a–2c § 2. Here is how explanation atoms are generated (where "$\models_{CT}$ stands for $CT \models$ and $\bigwedge \Phi$ stands for $\bigwedge_{\varphi \in \Phi} \varphi$):

1. *Explanation through an "inverted bump" ontology (descending atom, then ascending one)* If $\models_{CT}$ (α *causes* β), $\models_{CT}$ ($\beta_1 \rightarrow_{IS_A} \beta$), and $\models_{CT}$ ($\beta_1 \rightarrow_{IS_A} \gamma$), then ($\alpha$ *explains* γ *if_possible* $\{\alpha, \beta_1\}$).
2. *Transitivity of explanation* If $\models_{CT}$ (α *explains* β *if_possible* Φ_1) and $\models_{CT}$ (β *explains* γ *if_possible* Φ_2), then (α *explains* γ *if_possible* $\Phi_1 \cup \Phi_2$).
3. *Simplifying the condition-set* If $\models_{CT}$ (α *explains* β *if_possible* $\Phi \cup \Phi_i$) for $i \in \{1, \ldots, n\}$ and $\models_{CT} \bigwedge \Phi \rightarrow \bigvee_{i=1}^{n} \bigwedge \Phi_i$, then ($\alpha$ *explains* β *if_possible* Φ).
4. *Effective explanation (after checking the condition)* If $\not\models_{CT} \neg \bigwedge \Phi$ and $\models_{CT}$ (α *explains* β *if_possible* Φ), then (α *explains* β).

Due to the reflexivity of ontology the number of the rules is very low. Notice however that in the DLV translation, more "rules" are provided, in order to speed up the computation. Since it is easier to understand the behavior of these rules with a few more details, here are three particular cases of the first rule:

1. (a) *Direct explanation* The basic case is the following one: If $\models_{CT}$ (α *causes* β), then (α *explains* β *if_possible* $\{\alpha\}$).
   (b) *Explanation via a descending ontology* If $\models_{CT}$ (α *causes* β) and $\models_{CT}$ ($\beta_1 \rightarrow_{IS_A} \beta$), then ($\alpha$ *explains* β_1 *if_possible* $\{\alpha, \beta_1\}$).
   (c) *Explanation via an ascending ontology* If $\models_{CT}$ (α *causes* β) and $\models_{CT}$ ($\beta \rightarrow_{IS_A} \beta_1$), then ($\alpha$ *explains* β_1 *if_possible* $\{\alpha\}$).

Rules (1a) and (2) look uncontroversial. Here is a small illustration for rule (1b). Let consider the "flu example" again. flu causes a temperature between 38 and 41 °C and each temperature between 38 and 41 is a "temperature between 38 and 41": (flu *causes* t_{38-41}), $(t_{38} \rightarrow_{IS_A} t_{38-41}), \cdots, (t_{41} \rightarrow_{IS_A} t_{38-41})$. Then, it is natural to "explain" a temperature of 39 by flu (flu is a possible explanation for a temperature of 39). Thus, the explanation atom (flu *explains* t_{39} *if_possible* $\{flu, t_{39}\}$) is produced. In order to get this possible explanation from the preceding formulas, we need that flu and t_{39} (taken together) are compatible with the available information.

Let us illustrate the interest of the "inverted bump" rule (1) in full generality: (*serious_problem causes alarm*), (*strong_ring* $\rightarrow_{IS_A}$ *alarm*), (*strong_ring* $\rightarrow_{IS_A}$ *loud_noise*). Then we get (*serious_problem explains loud_noise if_possible* {*serious_problem*, *strong_ring*}).

Since this is not the main object of the paper, we stop these short justifications now. This is a reasonably looking set of rules, and, if we want to check whether or not it is a solution for the problems we wanted to solve, it is good to consider a lot of toy examples. These examples should be small enough in order to be relatively easy to grasp, but they should be significantly more complex than the previous ones, and much more numerous. This is where ASP could help, since it should be easy to encode such a formalism into an ASP formulation. For practical reasons, we have only considered DLV, but this should have only a small impact on our conclusions.

4 Towards an ASP translation of this causal formalism

What makes ASP a particularly good candidate for this kind of formalism is:

1. The formalism uses logical formulation (and with non-monotonicity in it).
2. Our problem is sort of a way to progress into a graph, which is a domain in which ASP is known to be good enough.
3. Small changes in an evolving theory should be easy to make in the ASP translation, since this translation is close to the original rules.

As with any kind of programming technique, there are many ways to encode a given problem. So, we describe a central solution (partial by the way, for reasons to be given below) together with the evocation of a few different approaches. Let us first consider a way to encode the classical logical part of the theory.

4.1 Reconstructing a small part of classical logic

Even if this may look strange, it seems that the best way to deal with the classical logic part is to partially rebuild classical logic. We introduce predicates `Symbol` for "propositional symbol"), `T` (for *True*), `Imp` (for "implies"), `And`, `Or` and `Neg`.

```
(Imp1)    Imp(X,X) :- Symbol(X).       Imp(X,Y) :- Imp(X,Z), Imp(Z,Y).
(Imp2)    T(Y) :- Imp(X,Y), T(X).       -T(X) :- Imp(X,Y), -T(Y).
(AndOp1)  T(Z) :- And(X,Y,Z), T(X), T(Y).
(AndOp2)  -T(Z) :- And(X,Y,Z), -T(X).    -T(Z) :- And(X,Y,Z), -T(Y).
(AndRev1) T(X) :- And(X,Y,Z), T(Z).      T(Y) :- And(X,Y,Z), T(Z).
(AndRev2) -T(X) v -T(Y) :- And(X,Y,Z), -T(Z).
(OrOp1)   T(Z) :- Or(X,Y,Z), T(X).      T(Z) :- Or(X,Y,Z), T(Y).
(OrOp2)   -T(Z) :- Or(X,Y,Z), -T(X), -T(Y).
(OrRev1)  T(X) v T(Y) :- Or(X,Y,Z), T(Z).
(OrRev2)  -T(X) :- Or(X,Y,Z), -T(Z).      -T(Y) :- Or(X,Y,Z), -T(Z).
(Neg1)    Neg(X,Y) :- T(X), -T(Y).        Neg(X,Y) :- Neg(Y,X).
(Neg2)    -T(Y) :- Neg(X,Y), T(X).        T(Y) :- Neg(X,Y), -T(X).
```

The "Rev" parts are required if we want ternary predicates, and not only "operational" ones [(Neg2) is a kind of (NegRev)]. Their necessity depends on the way formulas are introduced, and on the way we need to use the combinators.

(Gener) `T(X) v - T(X) :- Symbol(X).`

If we need all the logical models, we generate all the possibilities for each propositional symbol, thanks to this generating rule. Sometimes this can be avoided, since it provides too many answer sets for nothing. This important and frequent point when using ASP is not easy to settle for a precise situation.

If CT contains as given facts that a must be true and b false, the following four fact-rules are enough: `Symbol(a).` `Symbol(b).` `T(a).` `-T(b).`

Here we do not need the artillery with `And` and so on (we need `Imp` anyway). For formulas, if we do not want that the user makes almost all the job before the automatic computation, we need these predicates. And even then, the solution given here is not very elegant (probably a better one exists). Let us suppose that we have as data the formula $\phi_1 = (a \wedge b) \vee \neg(b \wedge c)$. We can write the following rules: `And(a,b, aandb).` `And(b,c,bandc).` `Neg(bandc,negbandc).` `Or(aandb, negbandc, phi1).`, together with the constraint `:- not T(phi1).` or with the "direct affirmation" `T(phi1).` [(Gener) being mandatory].

We need also to check that a ***set*** of propositional symbols is consistent within a theory. Here is a solution with "implicit" sets: let $SetEl(I, E)$ being true for all the elements E of the set indexed by the parameter I. Then, the rules would read as follows (the last term `SetEl(I,E)` ensures "safety" in DLV meaning):

```
-TSet(I) :- -T(E), SetEl(I,E).    Cons(I) :- not -TSet(I), SetEl(I,E).
```

This part of the program works well if there is one answer set, which roughly means that no disjunctive information (thus no "formulas" and no (Gener)) and no different explaining paths for some couple (I, J) of propositional symbols, should be allowed. Dealing with any situation would need real "brave reasoning" (one answer set ensures consistency) which is not yet available with DLV. Notice that here we need "brave reasoning" for $Cons$, together with standard reasoning otherwise. In DLV, only one "brave query" can be answered, which would need some tricky method, such as various programs working together, with each program using the outputs of the other one. A more complicated program could be designed which simulates brave and cautious reasoning together, which is what we need here. It suffices (in the case of the present program this is relatively easy) to provide a parameter which works as an index for each answer set of the present program and then gives as its answer one "new answer set" only. This looks unnatural and slightly "against the spirit of ASP", moreover some care should be taken in order to keep the program reasonable in time and space, even for toy examples. Designing this kind of meta-program is not an immediate task, and such complications should be avoided as far as possible.

The conclusion is that brave reasoning should be a very good point (cautious also, since the `-det` option of DLV does not give easy to predict results).

Before providing the heart of our program, let us make a second digression, about sets.

4.2 Dealing with explicit sets

Even if this should be avoided as far as possible for efficiency reasons (space being even more crucial than time here), it may be convenient to manipulate variables representing sets. We present a solution here, where sets are represented by an integer, thanks to the classical binary encoding. Firstly, all the objects potentially elements of the sets considered must be enumerated. Aggregate predicates cannot be used, since DLV insists on not allowing recursions on aggregate predicates: this produces the disappointing message:

```
% Error on rule [...] : the predicates appearing in the aggregate atom
cannot be recursive with the head of the rule.
```

This is a real theoretical problem with ASP in general. Various solutions have been proposed (see e.g. [3], which introduces a small change in the basic definitions of answer sets). However, the present version of DLV remains very strict, even when we are clearly not in a tricky case. We were unable to use aggregates in the full program [it would work in the following "isolated" rules].

Let us suppose that $Element(E)$ is true exactly when E is a potential element of a set. The first element (by alphabetic order) receives Number "1", then among the remaining elements, the first one receives the next number. MaxEl provides the cardinality of the set of the potential elements.

```
-NumberEl(X,1) :- Element(X), Element(Y), Y < X.
NumberEl(X,1) :- Element(X), not -NumberEl(X,1).

-NumberEl(X,I) :- Element(X), I=I1+1, NumberEl(Y,I1), Element(Z), Y<Z, Z < X.
 NumberEl(X,I) :- Element(X), I=I1+1, NumberEl(Y,I1), Y < X, not -NumberEl(X,I).

-MaxEl(I) :- NumberEl(X,I), I1 = I + 1, NumberEl(Y,I1).
MaxEl(I) :- NumberEl(X,I), not -MaxEl(I).
```

We need "arithmetic", thus "`-N=` " must appear in the DLV call, and, since DLV does not give any warning when the number is too low (the results are unpredictable) we need a warning:

```
Numbered(X) :- NumberEl(X,I).   ErrorNEnum :- Element(X), not Numbered(X).
```

Now, we are ready to define a "set type" of variables.

$Singl(SE, IE)$ is true iff the integer $SE = 2^{IE-1}$ encodes the singleton $\{E\}$ where E is the "element" represented by the integer IE. $SumSingl$ is an auxiliary predicate [$SumSingl(SSE, IE)$ is true iff $SSE = \Sigma_{I=0}^{IE-1} 2^I$] which speeds up some computations, including for the in predicate.

```
Singl(1,1). SumSingl(1,1).      % 1 = 2^(1-1)

Singl(SE,IE) :- IE=IF+1, NumberEl(E,IE), Singl(SF,IF), SE = 2 * SF.
SumSingl(SSE,IE) :- SumSingl(SSF,IF), IE=IF+1, NumberEl(E,IE),
                    Singl(SE,IE), SSE = SSF + SE.

MaxSet(MS) :- SumSingl(MS,Max), MaxEl(Max).
Set(S) :- #int(S), S<=MS, MaxSet(MS).   % S encodes a ``set of elements''
highestin(I,S) :- Set(S), Singl(SI,I), SumSingl(SSI,I), SSI >= S, S >= SI.
in(I,S) :- highestin(I,S).
in(I,S) :- highestin(J,S), Singl(SJ,J), S = SJ + SR, in(I,SR).
% in(I,S) is true iff I is the number of an ``element'' E and
%                    S encodes a set containing E
% highestin is an auxiliary predicate, which speeds up the computation.
```

Again, since "arithmetic" is involved, a warning is mandatory. This is important here since we need at least $2^{card(Element)}$ as our value for $N = \cdots$. The following rule does the job: `errorNSet :- not Set(0).`

Now that we have a "type Set" and the predicate "in" for "$\in$", it is easy to define predicates for intersection, union and inclusion. However, it happens that a real ternary predicate such as $union(S1, S2, SU)$ true iff $SU = S1 \cup S2$ when SU, $S1$ and $S2$ are of "type set", is intractable. With about 20 "elements" (in our full program), we got the unwelcome message `"abandon after throwing an instance of 'std::bad_alloc' "`. This huge requirement in memory is apparently due to the [almost] systematic instantiation made by DLV (it does not seem that SMODELS/LPARSE is better). We would need "built-in predicates", dealing with sets exactly as the built-in arithmetic predicates deal with integers and the operations $+$, $-$ and $*$. These "built-in arithmetic predicates" allow to manipulate big integers without the need for full instantiation (when some care is taken for writing the rules). We could not hope to manipulate too big sets, but sets with less than a hundred elements could likely be considered. Notice that for our purpose, we can avoid general union, since all we need is union of a set with a singleton, which can be done as follows:

```
union1(Set1,E,Set1) :- NumberEl(E,I), in(I,Set1).
union1(Set1,E,SetU1) :- NumberEl(E,I), Set(Set1), Singl(S1,I),
                        not in(I,Set1), SetU1 = Set1 + S1.
```

Then, with $MaxEl(n)$, we get only $n * 2^n$ instantiations for $union1$, instead of 2^{2*n} for "union". Even this reduced union with explicit sets is costly, and it is better to avoid explicit set variables.

5 DLV translation of the causal formalism

Here is where ASP is really nice: it is almost immediate to translate our inference rules into an ASP formalism. This has a great advantage: it will be easy (once the previous "preliminaries" settled) to test no too small toy examples, and similarly, it will be easy, if we modify our formalism, to make the same modifications in the ASP program. Our first programs used explicit sets, but due to their severe limitations (about 20 symbols were too many), we came up with the following translation which avoids explicit sets. The main concern is that the results are not very easy to read, which is why we have given these preliminaries about explicit sets.

We define a predicate $EClist$ (Explanation with Conditions): $EClist(\alpha, \beta, E)$ is true when there is an atom α *explains* β *if_possible* Φ where Φ is the set $\{E : EClist(\alpha, \beta, E)\}$. In this way, we avoid the costly need for explicit sets, and we fully use the power of the present ASP implementations (the difference with the explicit sets versions was indeed impressive!).

The given elements of a causal theory are given as follows:

```
C(alpha,beta).      O(gamma,delta).      T(alpha).
```

respectively for (α *causes* β) ("C" for "causes"), ($\gamma \rightarrow_{IS_A} \delta$) ("O" for "ontology") and α ("T" for "true"). For complex formulas see subsection 4.1: this method also works for causal formulas, with `T(t)` replaced by some `C(t1,t2)`, with "`C(t1,t2) v -C(t1,t2).`" for generating rule for each causal atom $C(t1, t2)$ involved in a complex formula. Here comes the translation of our formalism.

```
O(X,Z) :- O(X,Y), O(Y,Z).    % Transitivity of ontology
Imp(I,J) :- C(I,J).    Imp(I,J) :- O(I,J). % Entailing the implication
Rules (Imp1) and (Imp2) given in subsection 4.1.
```

There is no rule for the reflexivity of the ontology, since the generating rule for *EClist* will be split into particular cases. Indeed, translating the formal rules is easy, except for the rule simplifying the set of conditions. Thus, for efficiency reasons (and also for reasons given below), this rule is not fully translated. Instead, the generating rules are slightly modified in order to take into account some obvious simplifications till the generation of the set of conditions. Comments are provided in order to explain this ("coherence", cf `ECpath`, which makes things slightly more complex, can be ignored at first reading).

```
ECinit1(I,J) :- C(I,J).    % Point 1a page § 3
ECinit2(I,J) :- C(I,X), O(J,X), not ECinit1(I,J), J != X, I != J.
%    Point 1b § 3, only if no better initial explanation exists
ECinit1(I,I) :- C(I,X), O(I,X).    % Particular case of Point 1b
ECinit1(I,J) :- C(I,X), O(X,J).    % Point 1c § 3
ECinit1(I,J) :- C(I,X), O(I,X), O(I,J). %Particular case of Point 1 § 3
ECinit3p(I,J,E) :- ECinit2(I,E), C(I,X), O(E,X), O(E,J), not O(J,X),
                   not ECinit1(I,J), not ECinit2(I,J).
% Point 1 § 3, ``p'' for ``potential'' since various paths are
% possible here, with non comparable conditions.
```

Among the three kinds of "initial explanations" (generated without the use of transitivity of explanation), $ECinit1(I,J)$ is optimal, since the condition is only $\{I\}$, and no better path is possible. Then, $ECinit2(I,J)$ is optimal among the other ones, since the condition-set is $\{I,J\}$. Thirdly, $ECinit3(I,J,E)$, issued from Point 1 in full generality and in its (1b) version, is more complicated, since the condition set is $\{I,E\}$, where E implies J. Thus, it is possible that other initial explanations $ECinit3(I,J,E')$ exist, with an E' incomparable with E. For the sake of completeness, from all the potential $ECinit3p(I,J,Ei)$'s existing for any given (I,J), we will now choose exactly one in each answer set:

```
ECinit3(I,J,E) v -ECinit3(I,J,E) :- ECinit3p(I,J,E).
:- ECinit3(I,J,E1), ECinit3(I,J,E2), E1 != E2. % at most one
ECinit3red(I,J) :- ECinit3(I,J,E).
:- not ECinit3red(I,J), ECinit3p(I,J,E). % one if it occurs
```

Now comes the part dealing with transitivity of explanation. For efficiency reasons, and without loss of generality, we use a transitivity restricted to "explanation followed by initial explanation". We also consider the "explanation path" used for a given set of conditions. Here is the list of the auxiliary predicates:

1. $ECaux1(I,J)$: There exists an explanation from I towards J with $\{I\}$ for set of conditions (optimal case, no other case needs to be investigated).
2. $ECaux2(I,J)$: There exists no ECaux1(I,J), but there exists an explanation from I towards J with $\{I,J\}$ for set of conditions (optimal if not $ECaux1$).
3. $ECaux3(I,J,E)$: There exists an explanation from I towards J with $\{E : ECaux3(I,J,E)\}$ for its set of conditions.
4. $ECpath(I,J,C)$: There exists an explanation from I towards J, with $\{C : ECpath(I,J,C)\}$ as its set of "choice points".

5. $ECaux4p(I, J, E, C)$: There exists a *p*otential explanation from I towards J, obtained by a transitivity between some $ECaux3(I, K, E1)$ and some $ECinit3(K, J, E2)$. E is a condition-elements, and C a choice point.
6. $ECaux4c(I, J, E, C)$: In each answer set, only one of all the possible $ECaux4p(I, J, E, _)$'s is *c*hosen. This is the result of transitivity.

Initialisations:

```
ECaux1(I,J) :- ECinit1(I,J).    ECaux1(I,J) :- ECaux1(I,K), ECinit1(K,J).
ECpath(I,J,C) :- ECaux1(I,K), ECpath(I,K,C), ECinit1(K,J).
ECpath(I,J,K) :- ECaux1(I,K), ECinit1(K,J). % and not ", ECpath(I,K,C)"
ECaux2(I,J) :- ECinit2(I,J), not ECaux1(I,J).
ECaux2(I,J) :- ECaux1(I,K), ECinit2(K,J), not ECaux1(I,J).
ECpath(I,J,C) :- ECpath(I,K,C), ECinit2(K,J), not ECaux1(I,J).
ECpath(I,J,K) :- ECaux1(I,K), ECinit2(K,J), not ECaux1(I,J).
ECaux3(I,J,I) :- ECaux3(I,J,E).    % I is always in the set of conditions
ECaux3(I,J,I) :- ECaux1(I,J).           ECaux3(I,J,J) :- ECaux2(I,J).
```

As for the third kind of "initial explanation" $ECinit3$, it must be considered as a non initial case, through a special choice point called "forbiddenname" (no symbol should receive this name). Then it will be compared with the other possible paths, in order to keep only "optimal" sets of conditions:

```
ECaux4p(I,J,E,forbiddenname) :- ECinit3(I,J,E), not ECaux1(I,J),
                                not ECaux2(I,J).
```

The transitivity will result in some $ECaux4c(I, J, E, K)$, able to be recycled into another transitivity:

```
ECaux3(I,J,E) :- ECaux4c(I,J,E,K).
ECpath(I,J,C) :- ECaux4c(I,J,E,K), ECpath(I,K,C).
ECpath(I,J,K) :- ECaux4c(I,J,E,K), ECinit1(K,J).
ECpath(I,J,K) :- ECaux4c(I,J,E,K), ECinit2(K,J).
ECpath(I,J,K) :- ECaux4c(I,J,E,K), ECinit3(K,J,E).
```

Here are the rules dealing with the general case of transitivity (Point 2 § 3 except for the "optimal cases" $ECaux1$ and $ECaux2$):

```
ECaux4p(I,J,E1,K) :- not ECaux1(I,J), not ECaux2(I,J), ECaux3(I,K,E1),
                     ECinit1(K,J), K != J, I != J.
ECaux4p(I,J,E1,K) :- not ECaux1(I,J), not ECaux2(I,J), ECaux3(I,K,E1),
                     ECinit2(K,J), not ECinit1(K,J), K != J, I != J.
ECaux4p(I,J,J,K) :- not ECaux1(I,J), not ECaux2(I,J), ECaux3(I,K,E1),
                    ECinit2(K,J), not ECinit1(K,J), K != J, I != J.
ECaux4p(I,J,E1,K) :- not ECaux1(I,J), not ECaux2(I,J), ECaux3(I,K,E1),
                     ECinit3(K,J,E2), K != J, I != J.
ECaux4p(I,J,E2,K) :- not ECaux1(I,J), not ECaux2(I,J), ECaux3(I,K,E1),
                     ECinit3(K,J,E2), not ECinit1(K,J), not ECinit2(K,J),
                     K != J, I != J, E2 != K, E2 != I.
```

We choose exactly one $ECaux4c(I, J, E2, K)$ for each couple (I, J), when there exists some $ECaux4p(I, J, E1, K)$. Firstly, if there are several potential $ECaux4p(I, J, Ei, Ki)$ for some (I, J), to each choice point Ki is associated the condition set $\{E : ECaux4p(I, J, E, Ki)\}$. We exclude the Ki's which are clearly not optimal since they do not correspond to smallest sets (for set inclusion): $incl(I, J, K1, K2)$ is true iff the condition-set associated to $K1$ is included in the condition-set associated to $K2$ (the first two following rules give the standard method defining inclusion between "implicit sets" in logic programming).

```
-incl(I,J,K1,K2) :- ECaux4p(I,J,E,K1), not ECaux4p(I,J,E,K2),
                    ECaux4p(I,J,E2,K2).
incl(I,J,K1,K2) :- ECaux4p(I,J,E1,K1), ECaux4p(I,J,E2,K2),
                   not -incl(I,J,K1,K2).
-ECaux4c(I,J,E,K2) :- incl(I,J,K1,K2), not incl(I,J,K2,K1),
                      ECaux4p(I,J,E,K2).
ECaux4c(I,J,E,K) v -ECaux4c(I,J,E,K) :- ECaux4p(I,J,E,K).
```

The last two rules generate all the possible cases $ECaux4c(I, J, E, K)$, with non optimal cases excluded. In each answer set, a unique "$ECaux4c$" is chosen, which must contain all the condition elements of the chosen path:

```
:- ECaux4c(I,J,E1,K), -ECaux4c(I,J,E2,K), E1 != E2. % All condition-el.'s
:- ECaux4c(I,J,E1,K1), ECaux4c(I,J,E2,K2), K1 != K2. % at most one path
ECaux4red(I,J) :- ECaux4c(I,J,K,E).
:- not ECaux4red(I,J), ECaux4p(I,J,K,E). %one path, if some path exists
```

This program is not satisfying from a practical point of view. Indeed, choice points are not used, thus we get answer sets with "incoherent paths": some path can be chosen for explaining J from I, and some path for explaining K from I, and, if each path from I to K must contain J, a given answer set may choose, for explaining K from I, a different path for the part going from I to J than when explaining J from I. This is not false, but it does not help in reading the results. Indeed, this generates useless answer sets, while the total number of explanation atoms (I *explains* J *if_possible* Φ), which is the interesting part of the list of the answer sets, is the same. This is why checking "coherence" is useful:

```
coherent(I,J,C,E1) :- ECaux3(I,J,E), not ECaux1(I,J), not ECaux2(I,J),
                          ECpath(I,J,C), ECaux3(I,C,E1), Imp(E,E1).
coherent(I,J,C,E1) :- ECaux3(I,J,E), not ECaux1(I,J), not ECaux2(I,J),
                          ECpath(I,J,C), ECaux3(C,J,E1), Imp(E,E1).
incoherent(I,J,C) :- ECaux3(I,J,E), not ECaux1(I,J), not ECaux2(I,J),
             ECpath(I,J,C), ECaux3(I,C,E1), not coherent(I,J,C,E1).
incoherent(I,J,C) :- ECaux3(I,J,E), not ECaux1(I,J), not ECaux2(I,J),
             ECpath(I,J,C), ECaux3(C,J,E1), not coherent(I,J,C,E1).
:- incoherent(I,J,C). % excludes ``incoherent'' answer sets
```

Explanation atoms (7a): $EClist(I, J, E)$ means that there exists an explanation atom (I *explains* J *if_possible* $Setcond$) with $Setcond = \{E : EClist(I, J, E)\}$ (set taken in the answer set considered).

```
EClist(I,J,I) :- ECaux1(I,J).       EClist(I,J,I) :- ECaux2(I,J).
EClist(I,J,J) :- ECaux2(I,J).       EClist(I,J,E) :- ECaux3(I,J,E).
```

We could now simplify the sets of conditions (remind however that the rules given have already made some simplification, and also that dealing with the rule given in Point (3) § 3 in full generality would be more complex).

```
ECless(I,J,El) :- EClist(I,J,El), EClist(I,J,El1), Imp(El1,El), not Imp(El,El1).
ECless(I,J,El) :- EClist(I,J,El), EClist(I,J,El1), Imp(El1,El),
Imp(El,El1), El > El1. % El and El1 equivalent, El1 first in alphabetic order
ECsimpl(I,J,El) :- EClist(I,J,El), not ECless(I,J,El).
```

$EC(I, J)$ denotes an ***explanation atom*** (7b) (I *explains* J) where the consistency of the set of conditions has been tested:

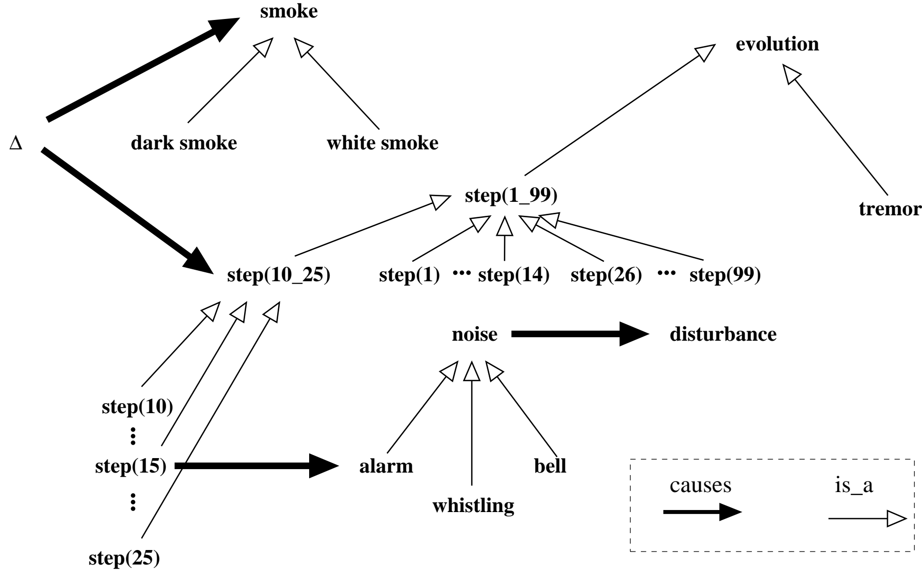
```
-TSetCond(I,J) :- ECsimpl(I,J,El), -T(El).
ECchecked(I,J,El) :- ECsimpl(I,J,El), not -TSetCond(I,J).
EC(I,J) :- ECchecked(I,J,El).
```

Ideally, this last set of rules should work using "brave reasoning", which cannot be done in DLV for now. In practice, it generally works in the absence of disjunctive data, since only the predicate T is used, and not full deductive logic. So, for now, it is better not to use these last rules: $EClist$ is not affected by this restriction, $ECsimpl$ should be almost unaffected (but anyway all the possible simplifications are not always made) while $ECchecked$ and EC are less sure.

6 Two examples, and some desiderata

Here are two examples, in order to illustrate the program given above. Remind that we can add additional information such as a propositional formula α written `T(alpha)` (see subsection 4.1).

Example 1



(C1) Δ *causes* $step(10_25)\}$ — *(C2)* Δ *causes smoke*

(C3) $step(15)$ *causes alarm* — *(C4) noise causes disturbance*

(O1) $whitesmoke \rightarrow_{IS_A} smoke$ — *(O2)* $darksmoke \rightarrow_{IS_A} smoke$

(O3) $step(i) \rightarrow_{IS_A} step(10_25)$ *[for each* $i \in \{10, \cdots, 25\}$*]*

(O3a) $step(i) \rightarrow_{IS_A} step(1_99)$ *[for each* $i \in \{1, \cdots, 99\}$*]*

(O4) $alarm \rightarrow_{IS_A} noise$

(O5) $step(10_25) \rightarrow_{IS_A} step(1_99)$

(O6) $bell \rightarrow_{IS_A} noise$ — *(O7)* $whistling \rightarrow_{IS_A} noise$

(O8) $step(1_99) \rightarrow_{IS_A} evolution$ — *(O9)* $tremor \rightarrow_{IS_A} evolution$

Symbol is omitted here, but notice that all terms are propositional symbols, including the "set" $step(10_25)$*. We get two resulting ontological atoms:*

$(O5.8)$ $step(10_25) \rightarrow_{IS_A} evolution$

$(O3a.8)$ $step(i) \rightarrow_{IS_A} evolution$ [*for each* $i \in \{1, \cdots, 99\}$]

Here are the explanation atoms given by the formalism and by above program (in less than one second). Our first version, with explicit sets, could not consider such an example (too many symbols).

	Explanation atoms with condition-set	Point	from
E1	Δ *explains* $step(10_25)$ *if_possible* $\{\Delta\}$	1a	C1
E2	Δ *explains* smoke *if_possible* $\{\Delta\}$	1a	C2
E3	$step(15)$ *explains* alarm *if_possible* $\{step(15)\}$	1a	C3
E4	noise *explains* disturbance *if_possible* {noise}	1a	C4
E5	Δ *explains* $step(i)$ *if_possible* $\{\Delta, step(i)\}$ [for each $i \in \{10, \cdots, 25\}$]	1b	C1O3
E6	Δ *explains* $step(1_99)$ *if_possible* $\{\Delta\}$	1c	C1O5
E7	Δ *explains* evolution *if_possible* $\{\Delta\}$	1c	C1O5.8
E8	Δ *explains* whitesmoke *if_possible* $\{\Delta, \text{whitesmoke}\}$	1b	C2O1
E9	Δ *explains* darksmoke *if_possible* $\{\Delta, \text{darksmoke}\}$	1b	C2O2
E10	$step(15)$ *explains* noise *if_possible* $\{step(15)\}$	1c	C3O4
E11	Δ *explains* alarm *if_possible* $\{\Delta, step(15)\}$	2	E3E5
E12	Δ *explains* noise *if_possible* $\{\Delta, step(15)\}$	2	E5E10
E13	$step(15)$ *explains* disturbance *if_possible* $\{step(15)\}$	2 + 3	E4E10
E14	Δ *explains* disturbance *if_possible* $\{\Delta, step(15)\}$	2 + 3	E4E12

Notice that $step(1)$ is not explained, while $step(15)$ is explained. Similarly, *tremor* is not explained. These are features of the formalism.

Example 2 *This example exhibits all the kinds of explanations, and various explaining paths. We write* `C(I,J)` *instead of* (I *causes* J) *and* `O(I,J)` *instead of* ($I \rightarrow_{IS_A} J$). *Again, we omit* `Symbol`.

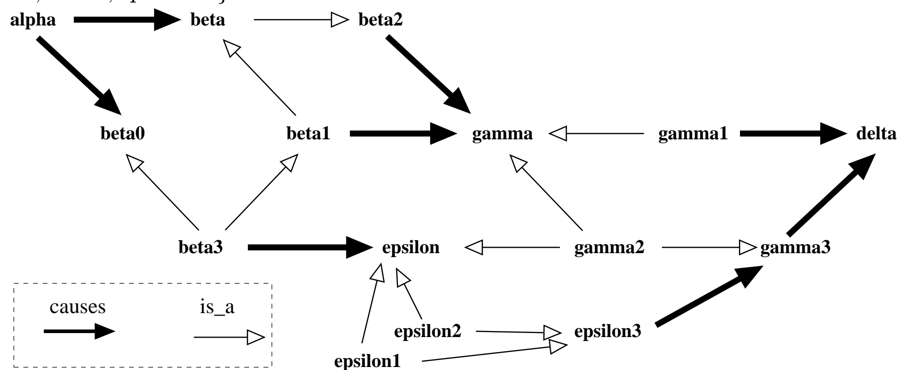
```
C(alpha,beta).            C(beta1,gamma).         C(beta2,gamma).
C(gamma1,delta).          C(gamma3,delta).        C(alpha,beta0).
C(beta3,epsilon).         C(epsilon3,gamma3).
-----------------------------------------------------------------------
O(beta1,beta).            O(beta,beta2).          O(gamma2,gamma).
O(gamma1,gamma).          O(gamma2,gamma3).       O(beta3,beta0).
O(beta3,beta1).           O(gamma2,epsilon).      O(epsilon1,epsilon).
O(epsilon1,epsilon3).     O(epsilon2,epsilon).    O(epsilon2,epsilon3).
```

The program gives 56 answer sets in less than two seconds (56 AS seems to be optimal). This is much better than the 1024 AS given by the same program without the $ECpath + incoherent$ part (some implicit coherence remains, since otherwise we would get $2^7 \times 3^3 = 3456$ AS), but it is still not an easy task to read the answers. Since this is a serious problem with this kind of solution, let us give a few comments. Here are a few answers common to all the answer sets:

```
EClist(alpha,beta0,alpha),
EClist(epsilon3,epsilon,gamma2), EClist(epsilon3,epsilon,epsilon3),
EClist(beta3,gamma,beta3), EClist(beta3,gamma,gamma2)
```

There are 30 couples (I, J) corresponding to an answer common to all AS (not all given by the "`-det`" option of DLV), which thus constitute the "real cautious" answers. We mean that this exhausts the predicate *EClist*. The "`-cautious`" option cannot be used: with query "`EClist(A,B,E)?`", the fact that

$EClist(alpha, gamma3, alpha)$ is true in all answer sets shows that $gamma3$ can be explained by $alpha$ in each answer set (since in this case, $alpha$ is clearly in all the possible condition-sets), but it does not give a real indication about the condition set. We find, in the various answer sets, besides (1) `EClist(alpha,gamma3,alpha)`, either (2a) `...,EClist(alpha,gamma3,gamma2)`, or (2b) `..., EClist(alpha,gamma3,beta3), EClist(alpha,gamma3,epsilon1)` or (2c) `..., EClist(alpha,gamma3,beta3), EClist(alpha,gamma3,epsilon2)`, which constitutes the three incomparable solutions, corresponding respectively to the three sets of conditions $(a)\{alpha, gamma2\}$, (b) $\{alpha, beta3, epsilon1\}$, and (c) $\{alpha, beta3, epsilon2\}$.



With "`-brave`" query', we get the five possibilities for the third parameter, while a "`-cautious`" query provides `EClist(alpha,gamma3,alpha)` only: these two answers do not allow to guess the real answer. For this, we must examine all the answer sets, which is not so easy, and which is why coherence, which minimizes the number of answer sets, is important. Notice that, as expected by "coherence", if we have (b) in an answer set, then we have (a1) `ECenum(beta3,gamma3,beta3), ECenum(beta3,gamma3,epsilon1)`, and that if we have (c), then we have (a2) `ECenum(beta3,gamma3,beta3), ECenum(beta3,gamma3,epsilon2)`. Without "coherence", we could have an answer set with choice $(beta3, epsilon1)$ for $(alpha, gamma3)$ and choice $(epsilon2)$ for $(beta3, gamma3)$.

A ***cautious query with different roles given to the different parameters*** of a predicate would be very useful: A (cautious) query such as "`EClist(I,J,_)?`" would give the expected answer: exactly the $EClist(I, J, E)$'s which have the same extension for a given (I, J) in each answer set.

We could complicate the program, getting a single answer set by choosing an index for each old answer set. This is not immediate, and it seems to be against the spirit of ASP. We came close to a solution, but we gave up because it was becoming hard to follow. This would be wasting time and clearness.

More generally, it would be nice to allow re-usability of the result of a program as data for another program. Indeed, what we made was to use Xemacs in a not immediate way (replacing ", " by ". " is easy, but other manipulations are necessary). It is frequent that such a thing is necessary, and going through

"SQL" is not very convenient. As an example here, we could try some cautious queries after a transformation of `EClist(I,J,E)` by `EClistIJ(E)`, then a cautious query"`EClistIJ(E)?`" would work. But this necessitates as many successive isolated queries as there are possible couples `(I,J)`!

Another example is for designing Example 1 here: instead of typing the hundred *stepi*'s, what we have done is writing a generator in DLV. Then the result needs some modifications with a text editor. It would be fine if these things could be fully automatized.

It does not seem to us that the various ways for representing the results of an ASP program and the reflexions about "cautious" and "brave" reasoning examined and mentioned in [2], can provide an answer to this kind of concern.

7 Conclusion and future work

It would be nice if ASP could be used easily during research in computer science (or other research by the way). This may not be a crucial issue, but this would be useful, and could help advertising about the beauty of ASP. The present implementations are very close to this goal, however, a few cosmetic modifications would help a lot. It seems that the main concern is that we should easily be allowed to "play with the list of answer sets". In fact, this would mean that the fact that the ***answer*** of program in ASP is the set of its answer sets would be taken seriously. In this way, real cautious (and brave) reasoning would be possible, which is not really the case now. We have examined only DLV, because it has disjunction and is convenient to use, but most of our comments could be made with "lparse + Smodels" and other implementations. Playing with the answer sets would also facilitate the task of re-using the results of a program as data for another program. It would avoid in many cases the necessity of using explicit sets, which is time and space consuming. It is possible that this is unrealistic for huge "serious" applications, but this would be a great help for many "hard small" examples.

Another useful addition would be, even if it seems in opposition with some of the previous statements, to incorporate *built-in predicates for dealing with [real] sets*. This would include the enumeration/encoding given in § 4.2, in a much more efficient way. Again, this could not be really practical for huge examples, but there are many cases where it would be helpful.

It seems that there is a real difference between "small hard problems" (to which the present formalism could probably be assimilated) and "huge easy problems" (see [5]). Each kind deserves some attention, and even if, a priori, huge easy problems should not be concerned by these cosmetic improvements, we think that this kind of request has its importance.

While we are asking, we could also generalize this request to various kinds of "auxiliary predicates" which could be useful, but in practice cannot be used due to the problem of the "full instantiation" which makes an explosion in memory size. Is there some hope that real auxiliary "sub programs" become possible some day, which would, as far as possible, avoid this explosion?

The power of the implementations is still progressing regularly (see e.g. the recent [4]). Are there recent extensions which could help for our concerns?

Thus, the purpose of this text is to get from gurus in ASP answers to these questions:

Are the present requests stupid, since they only show that we have not designed our program properly? There is probably some truth here, but we think that a program without these requirements would be more difficult to read, while the programming time would be longer. Using ASP is interesting if we gain time, in designing a program, and in each subsequent modification.

Can we hope to get some positive answers in a near future?

As future work, it could be interesting to try alternative implementations of ASP, and also to make a serious comparison with other kinds of logical programming (including Prolog) and with related programming techniques.

Acknowledgements

I would like to thank the referees, who provided various pertinent hints.
Moreover, this work would not have been possible without previous theoretical work by Philippe Besnard and Marie-Odile Cordier.

References

1. Ph. Besnard, M.-O. Cordier, and Y. Moinard. Configurations for Inference between Causal Statements. In J. Lang, F. Lin, and Ju Wang, eds, *Knowledge Science, Engineering and management, in LNAI 4092*, pages 292–304, 2006. Springer.
2. M. Brain, W. Faber, M. Maratea, A. Polleres, T. Schaub, and R. Schindlauer. What should an ASP Solver output? [Position Paper]. In *First Int. Workshop on Software Engineering for Answer Set Programming*, pages 26–38, Tempe, AZ, USA, May 2007.
3. W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In J. J. Alferes and J. A. Leite, eds, *LNAI 3229*, pages 200–212. 2004.
4. M. Gebser, T. Schaub, and S. Thiele. Gringo : A new grounder for answer set programming. In C. Baral, G. Brewka, and J.S. Schlipf, eds, *LPNMR 2007*, *LNCS 4483*, pages 266–271, 2007.
5. E. Giunchiglia, Y. Lierler, and M. Maratea. Answer Set Programming based on Propositional Satisfiability. *Journal of Automated reasoning*, 36(4):345–377, 2006.
6. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. on Computational Logic (TOCL)*, 7(3):499–562, 2006.