

Using ASP with recent extensions for causal explanations

Yves Moinard

► **To cite this version:**

Yves Moinard. Using ASP with recent extensions for causal explanations. Marcello Balduccini and Stefan Woltran. ASPOCP10, Answer Set Programming and Other Computing Paradigms Workshop, associated with ICLP, Jul 2010, Edinburgh, United Kingdom. 2010, Answer Set Programming and Other Computing Paradigms. <inria-00542880>

HAL Id: inria-00542880

<https://hal.inria.fr/inria-00542880>

Submitted on 3 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Using ASP with recent extensions for causal explanations*

Yves Moinard

INRIA Bretagne Atlantique, IRISA, Campus de Beaulieu,
35042 Rennes cedex, France email: moinard@irisa.fr

Abstract

We examine the practicality for a user of using Answer Set Programming (ASP) for representing logical formalisms. We choose as an example a formalism aiming at capturing causal explanations from causal information. We provide an implementation, showing the naturalness and relative efficiency of this translation job. We are interested in the ease for writing an ASP program, in accordance with the claimed “declarative” aspect of ASP. Limitations of the earlier systems (poor data structure and difficulty in reusing pieces of programs) made that in practice, the “declarative aspect” was more theoretical than practical. We show how recent improvements in working ASP systems facilitate a lot the translation, even if a few improvements could still be useful.

1 Introduction

We examine a few difficulties encountered when trying to translate a logical formalism into a running answer set programming (ASP) program, showing how recent developments in ASP systems are of great help. We are not concerned here by complexity matters (also an important matter, for which exists a large literature). Rather, we deal with the ease for writing, and

*Paper presented at ASPOCP10, Answer Set Programming and Other Computing Paradigms Workshop, associated with ICLP, Edinburgh, July 20, 2010.

more importantly reading (thus modifying easily) programs in ASP for a given problem. Indeed, ASP is presented as (and is) a declarative formalism where it should be immediate to write a program from a logical formalization of a given problem. In practice, this is rarely as easy as claimed. We choose as an example a formalism that we have designed in collaboration with Philippe Besnard and Marie-Odile Cordier. This formalism aims at a logical formalization of explanations from causal and “is-a” statements. Given some information such as “fire causes smoke” and “grey smoke is a smoke”, if “grey smoke” is established, we want to infer that “fire” is a (tentative) explanation for this fact. The formalization [3, 4] is expressed in terms of *rules* such as

“if α causes β and δ isa β , then α explains δ because $\{\alpha, \delta\}$ is possible”.

Here, α, \dots may be first order atoms (without function symbols). Thus, we can express these rules in a “Datalog” formulation. When various explanations are possible, some of them can be subsumed by other ones, and the set of the solutions should be pruned. This concerns looking for paths in some graph and ASP is good for these tasks. There currently exist systems which are rather efficient, such as DLV¹ [9] or gringo/clasp or claspD². Transforming formal rules into an ASP program is easy. ASP should then be an interesting tool for researchers when designing a new theoretical formalization as the one examined here. An ASP program should help examining a great number of middle sized examples, and check whether the results are in accordance with our expectations. Then if middle sized examples could work, a few more optimization techniques could make real sized examples work. Since ASP rules are close to a natural language, a final user could easily adapt a general framework to his precise needs, without requiring a complex specific system.

Even if ASP allows such direct and efficient translation, a few problems complicate the task.

Firstly, the poor data types available in pure ASP systems is a real drawback. Our rules involve sets. There are many ways to represent sets in existing ASP systems. We had designed programs working in this way [11] (for systems not allowing functional terms), by representing a set as follows: $Expl(I, J, N, E)$ meaning there exists an “explanation” from I to J with a set of conditions which is the set $\{E/Expl(I, J, N, E)\}$, where N is an index,

¹<http://www.dbai.tuwien.ac.at/proj/dlv/>

²<http://potassco.sourceforge.net/>

necessary when M sets of conditions exist ($N \in \{1, \dots, M\}$). However, the program is rather hard to read, and thus to be adapted. Part of this difficulty comes from the second drawback given now.

Secondly, in pure ASP, it is hard to reuse part of a program. Similar rules should be written again, in a slightly different way. Thus, e.g. the rules necessary to deal with sets should be rewritten in many parts of the full program.

Thirdly, there are restrictions concerning “brave” or “cautious” reasoning. In ASP, “the problem is the program” and the solution consists in one or several sets of atoms satisfying the problem. Each such set is an *answer set*. These answer sets are the “ASP models”. *Brave* (respectively *cautious*) solutions mean to look for atoms true in some answer set (respectively all the answer sets). In the existing ASP systems, this is generally possible, but in a restricted way only.

All these difficulties make that in practice, the claimed advantage in favour of the use of ASP is not so clear when the final program is written. The rules in the program are encumbered by various tricks necessary to overcome these limitations, and any subsequent modification in the program becomes complex.

However, things are evolving. To take the example of DLV, points 1 and 2 above are partly solved: DLV-Complex³ deals with the data structure problem, since it admits sets and lists. DLT⁴ allows the use of “templates”, which solves to a great extent the problem for reusing part of a program. It is not yet possible to use these two improvements together. Since here the most problematic case was due to the use of sets, we have used DLV-Complex, without DLT. When DLT will be able to work with DLV-Complex, the task will be yet easier.

In the next section, we present what should be known about our explanation formalism in order to understand its ASP translation. Then, in section 3, we describe its ASP translation, explaining the interest, and the drawback, of using ASP for this kind of job. Finally, we conclude by a few reasonable expectations about the future ASP systems which could help a final user, and make ASP a really interesting programming paradigm for such problems.

³<http://www.mat.unical.it/dlv-complex>

⁴<https://www.mat.unical.it/ianni/wiki/dlt>

2 The causal explanation formalism

2.1 Preliminaries

For simplicity, we fully present the propositional version only. The full formalism, with predicates (without functions) and with an elementary “ontology” has been described in [3, 4]. The extension from the propositional case to the general case is not difficult, as will be explained in §3.5. We distinguish various types of statements in our formal system:

C: A theory expressing causal statements.

E.g. *On_alarm causes Heard_bell* or *Flu causes Fever_Temperature*.

O: An ontology in the form of a set of *IS-A* links between two items which can appear in a causal statement.

E.g., *Temperature_39* \rightarrow_{IS-A} *Fever_Temperature*,

Temperature_41 \rightarrow_{IS-A} *Fever_Temperature*,

Heard_loud_bell \rightarrow_{IS-A} *Heard_bell*,

Heard_soft_bell \rightarrow_{IS-A} *Heard_bell*.

W: A classical propositional theory expressing truths (incompatible facts, co-occurring facts, ...). E.g., *Heard_soft_bell* $\rightarrow \neg$ *Heard_loud_bell*.

Intuitively, propositional symbols denote elementary properties describing states of affairs, which can be *facts* or *events* such as *Fever_Temperature*, *On_alarm*, *Heard_bell*. The causal statements express causal relations between facts or events expressed by these propositional symbols.

Some care is necessary when providing these causal and ontological atoms. If “*Flu causes Fever_Temperature*”, we will conclude *Flu explains Temperature_39* from *Temperature_39* \rightarrow_{IS-A} *Fever_Temperature*, but we cannot state *Flu causes Temperature_39*: we require that the causal information is provided “on the right level” and in this case, *Temperature_39* is not on the right level, since “*Temperature_39*” is too narrow with respect to our knowledge about flu and temperatures.

The formal system is meant to infer, from such premises $C \cup O \cup W$, formulas denoting explanations. This inference will be denoted \vdash_C . The ontological atoms express some common sense knowledge which is necessary to infer these “explanations”. Notice that a feature of our formalism is that standard implication alone cannot help to infer explanations [3, 4].

In the following, α, β, \dots denote the propositional atoms and Φ, Ψ, \dots denote sets thereof.

Atoms

1. *Propositional atoms*: α, β, \dots
2. *Causal atoms*: α causes β .
3. *Ontological atoms*: $\alpha \rightarrow_{IS-A} \beta$.
4. *Explanation atoms*: α explains β because_possible Φ .

An ontological atom reads: α is a β .

An explanation atom reads: α is an explanation for β because Φ is possible.

Notation: In order to help reading long formulas, explanation atoms are sometimes abbreviated as α expl β bec_poss Φ .

Formulas

1. *Propositional formulas*: Boolean combinations of propositional atoms.
2. *Causal formulas*: Boolean combinations of causal or propositional atoms.

The premises of the inference \vdash_C , namely $C \cup O \cup W$, consist of propositional and causal formulas, and ontological *atoms* (no ontological formulas). Notice that explanation atoms cannot occur in the premises.

The properties of causal and ontological formulas are as follows.

1. Properties of the causal operator

- (a) *Entailing [standard] implication*: If α causes β , then $\alpha \rightarrow \beta$.

2. Properties of the ontological operator

- (a) *Entailing implication*: If $\alpha \rightarrow_{IS-A} \beta$, then $\alpha \rightarrow \beta$.
- (b) *Transitivity*: If $a \rightarrow_{IS-A} b$ and $b \rightarrow_{IS-A} c$, then $a \rightarrow_{IS-A} c$.
- (c) *Reflexivity*: $c \rightarrow_{IS-A} c$.

Reflexivity is an unconventional property for an *IS-A* hierarchy. It is included here because it helps keeping the number of inference schemes low (note that in the ASP translation we do not need reflexivity).

W is supposed to include (whether explicitly or via inference) all the implications induced by the ontological atoms. For example, if $Heard_loud_bell \rightarrow_{IS-A} Heard_bell$ is in O then $Heard_loud_bell \rightarrow Heard_bell$ is in W .

Similarly, W is supposed to include all conditionals induced by the causal statements in C . For example, if $Flu\ causes\ Fever_Temperature$ is in C , then $Flu \rightarrow Fever_Temperature$ is in W .

2.2 The formal system

The above ideas are embedded in a short proof system extending classical logic:

1. **Causal atoms entail implication:** $(\alpha\ causes\ \beta) \rightarrow (\alpha \rightarrow \beta)$.
2. **Ontological atoms**
 - (a) entail implication: If $\beta \rightarrow_{IS-A} \gamma$ then $\beta \rightarrow \gamma$.
 - (b) transitivity: If $\alpha \rightarrow_{IS-A} \beta$ and $\beta \rightarrow_{IS-A} \gamma$ then $\alpha \rightarrow_{IS-A} \gamma$.
 - (c) reflexivity: $\alpha \rightarrow_{IS-A} \alpha$
3. **Generating the explanation atoms**
 - (a) *Initial case*
If $\delta \rightarrow_{IS-A} \beta$, $\delta \rightarrow_{IS-A} \gamma$, and $W \not\models \neg(\alpha \wedge \delta)$,
then $(\alpha\ causes\ \beta) \rightarrow \alpha\ expl\ \gamma\ bec_poss\ \{\alpha, \delta\}$
 - (b) *Transitivity (gathering the conditions)* If $W \not\models \neg \wedge(\Phi \cup \Psi)$,
then $(\alpha\ expl\ \beta\ bec_poss\ \Phi \wedge \beta\ expl\ \gamma\ bec_poss\ \Psi)$
 $\rightarrow \alpha\ expl\ \gamma\ bec_poss\ (\Phi \cup \Psi)$.
 - (c) *Simplification of the set of conditions* If $W \models \wedge \Phi \rightarrow \bigvee_{i=1}^n \wedge \Phi_i$,
then $\wedge_{i \in \{1, \dots, n\}} \alpha\ expl\ \beta\ bec_poss\ (\Phi_i \cup \Phi) \rightarrow \alpha\ expl\ \beta\ bec_poss\ \Phi$.

These schemes allow us to obtain the inference patterns evoked in the previous section:

The most elementary “initial case” applies (2c) upon (3a) where $\beta = \gamma = \delta$, together with an obvious simplification (3c) since $\alpha \rightarrow \beta$ here, getting:

If α *causes* β and $W \not\models \neg\alpha$ then α *expl* β *bec_poss* $\{\alpha\}$.

Two other particular cases read as follows (respectively $\gamma = \delta$ and $\beta = \delta$):
 If α *causes* β , $\delta \rightarrow_{IS-A} \beta$ and $W \not\models \neg(\alpha \wedge \delta)$ then α *expl* δ *bec_poss* $\{\alpha, \delta\}$.
 If α *causes* β , $\beta \rightarrow_{IS-A} \gamma$ and $W \not\models \neg\alpha$ then α *expl* γ *bec_poss* $\{\alpha\}$.

Notice that we do not allow explanation through the opposite ontological sequence $\beta \rightarrow_{IS-A} \gamma$ and $\delta \rightarrow_{IS-A} \gamma$ [cf the general formulation 3a]. This is from experiences with situations of the kind of those evoked in the preliminaries, and it is in accordance with the notion of “right level”. Suppose we would admit these sequences, then if e.g.

α *causes* *Heard_loud_bell*, we would necessarily get the conclusion α *expl* *Heard_soft_bell* *bec_poss* $\{\alpha, \text{Heard_soft_bell}\}$, from the natural data $\text{Heard_loud_bell} \rightarrow_{IS-A} \text{Heard_bell}$ and $\text{Heard_soft_bell} \rightarrow_{IS-A} \text{Heard_bell}$. We consider such a conclusion as unwanted in this situation.

It is possible that some domains of application would need other rules. Our rules are intended as a compromise between expressive power, naturalness of description and relatively efficient computability.

The system is independent of any opinion about the controversial discussion about transitivity of *causation*. However, we provide some transitivity for *explanations* (the conditions are gathered then, cf 3b).

The simplification rule (3c) is powerful and can be considered as of little interest with respect to the extra computational cost associated with it. In practice, simpler rules could suffice, and our ASP translation implements a weaker rule defined as follows:

[3c'] If $W \models \wedge \Phi - \{\varphi\} \rightarrow \wedge \Phi$, and α *explains* β *because_possible* Φ
 then α *explains* β *because_possible* $\Phi - \{\varphi\}$.

Notice a minor point: we consider as important to keep α is such a case, thus the ASP translation does not apply this simplification (removing a redundant φ from the set of conditions) when $\varphi = \alpha$.

It is important to introduce the notion of *optimal explanation atoms*: An atom α *explains* β *because_possible* Φ is *optimal* if there is no explanation atom α *explains* β *because_possible* Ψ where $W \models \wedge \Phi \rightarrow \wedge \Psi$ while $W \not\models \wedge \Psi \rightarrow \wedge \Phi$. Keeping only these weakest sets of conditions is particularly

useful when the derivation is made only thanks to the part of W coming from Points 1 and 2a above. Indeed, this makes the result easier to read in case where many possible explanation atoms exist from α to β . In this case, we are certain to keep all the relevant explanation atoms, even if some additional W is introduced afterwards.

2.3 A generic diagram

Below an abstract diagram is depicted that summarizes many patterns of inferred explanations from various cases of causal statements and \rightarrow_{IS-A} links. The theory is described as follows (see Figure 1, with greek letters in their latin names):

α causes β , α causes β_0 , β_2 causes γ , β_1 causes γ ,
 β_3 causes ϵ , γ_1 causes δ , γ_3 causes δ , ϵ_3 causes γ_3 ;
 $\beta \rightarrow_{IS-A} \beta_2$, $\beta_1 \rightarrow_{IS-A} \beta$, $\beta_3 \rightarrow_{IS-A} \beta_0$, $\beta_3 \rightarrow_{IS-A} \beta_1$,
 $\gamma_1 \rightarrow_{IS-A} \gamma$, $\gamma_2 \rightarrow_{IS-A} \gamma$, $\gamma_2 \rightarrow_{IS-A} \gamma_3$, $\gamma_2 \rightarrow_{IS-A} \epsilon$,
 $\epsilon_1 \rightarrow_{IS-A} \epsilon$, $\epsilon_2 \rightarrow_{IS-A} \epsilon$, $\epsilon_1 \rightarrow_{IS-A} \epsilon_3$, $\epsilon_2 \rightarrow_{IS-A} \epsilon_3$.

This example shows various different “explaining paths” from a few given causal and ontological atoms. Here there is a first “explaining path” from α to δ (Figure 1, see also path (1a) on Figure 2). We get successively:

$\alpha \text{ expl } \beta_2 \text{ bec_poss } \{\alpha\}$, $\alpha \text{ expl } \gamma_1 \text{ bec_poss } \{\alpha, \gamma_1\}$, and

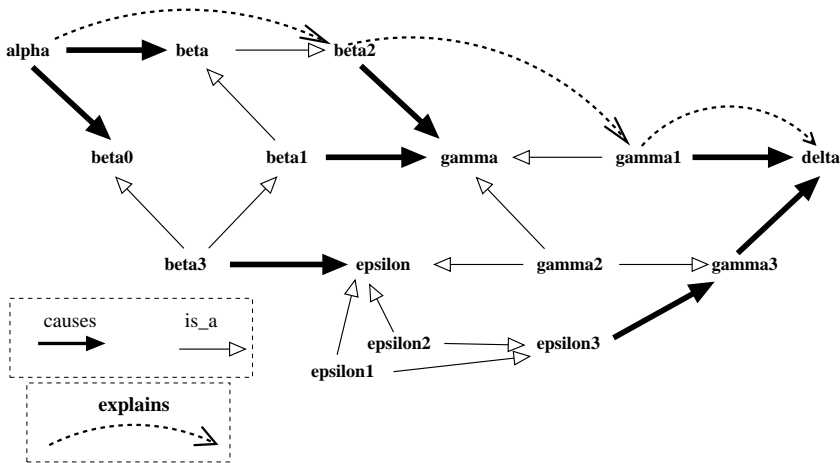


Figure 1: A generic diagram, the theory with a first explaining path

$\alpha \text{ expl } \delta \text{ bec_poss } \{\alpha, \gamma_1\}$.

As another “explaining path”, we get: $\alpha \text{ expl } \delta \text{ bec_poss } \{\alpha, \beta_1, \gamma_1\}$.

This second path is not optimal since $\{\alpha, \gamma_1\} \subset \{\alpha, \beta_1, \gamma_1\}$. The simplifying rule produces $\alpha \text{ expl } \delta \text{ bec_poss } \{\alpha, \gamma_1\}$ from $\alpha \text{ expl } \delta \text{ bec_poss } \{\alpha, \beta_1, \gamma_1\}$ but, from a computational point of view, it can be better not to generate the second path at all.

Here are the four optimal explanation atoms from α to δ (Fig. 2):

- | | | | |
|------|--|------|--|
| (1a) | $\alpha \text{ expl } \delta \text{ bec_poss } \{\alpha, \gamma_1\}$ | (1b) | $\alpha \text{ expl } \delta \text{ bec_poss } \{\alpha, \gamma_2\}$ |
| (2a) | $\alpha \text{ expl } \delta \text{ bec_poss } \{\alpha, \beta_3, \epsilon_1\}$ | (2b) | $\alpha \text{ expl } \delta \text{ bec_poss } \{\alpha, \beta_3, \epsilon_2\}$. |

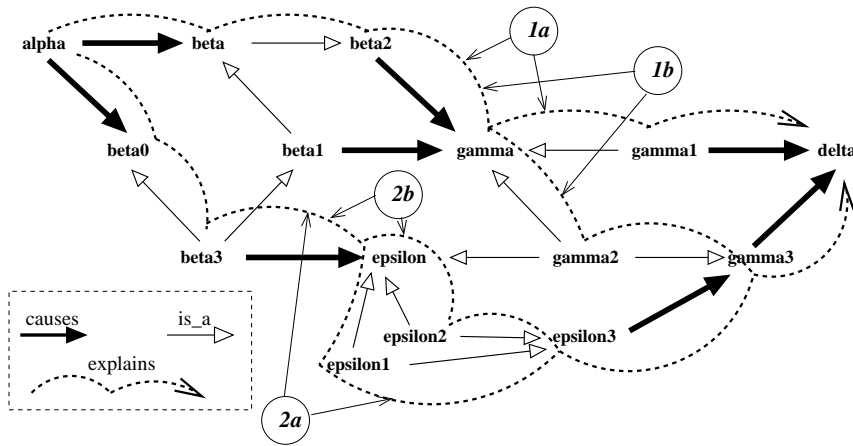


Figure 2: Four optimal explanation paths from “alpha” to “delta”

3 An ASP translation of the formalism

3.1 Presentation

We have implemented a program in DLV [9], an implementation of the Answer Set Programming (ASP) formalism [1], that takes only a few seconds to give all the results $\sigma_1 \text{ expl } \sigma_2 \text{ bec_poss } \Phi$, for all examples of the kind depicted in Fig. 2. The first version [11] used pure DLV, and was encumbered with tricks allowing to deal with sets. Now that DLV-Complex exists, the new version is simpler. As already written, it does not make full “simplification”, however it makes the most important ones. For greater examples, it still works, if we omit the simplification/optimization step. We have tried a

much greater example, involving two different copies of the example of the diagram, linked through an additional small set of data. This ends up with an example with more than a hundred symbols and more than 10 different explanation atoms for some (I, J) . This is not a “real world example” yet, but it is not too small, and it shows that we are close to realistic examples. Hopefully, the progress in ASP systems will make that in a near future real world examples could work.

As explained below, we have encountered a fourth problem, not listed in the “three problems” evoked above. Indeed, for this “big example”, the full program, including simplification and testing the set of conditions, did not work on our computer (crash after more than one hour...). However, since the simplification/optimization step is clearly separated from the first generating step, and since the verification is separated also, we have separated the program into three successive programs:

The first one generates explanation atoms (not all of them, but sufficiently many to be able to retrieve all the optimal ones from those produced here).

Then, starting from the results of this first program, a second program makes all the relevant simplifications and optimizations, in order to help reading the set of the solutions. The simplification does not take disjunction into account as in the powerful rule in Point 3c in §2.2, rather it corresponds to Point 3c'. When two possible sets Φ and Ψ of conditions exist for some (I, J) , if each $\psi \in \Psi$ is entailed (defined as above) by some $\varphi \in \Phi$, and if the contrary is not true, then the explanation atom $I \text{ expl } J \text{ bec_poss } \Phi$ is disregarded, only the explanation atom with the weaker condition set $I \text{ expl } J \text{ bec_poss } \Psi$ (the most likely to be satisfied) is kept as a result. This tends to keep only optimal explanation atoms defined at the end of §2.2 above, while remaining efficient enough (similarly to the simplification part, only a reasonably efficient part of the formal definition is implemented). This facilitates the human reading of the result, and in fact it is almost mandatory with some sets of data.

A third program takes the result of the second one (or directly of the first one if the second seems useless) and checks whether the set of conditions is satisfied in the answer set considered. In fact, in the formalism, we should check if it is true in some answer set. This needs full brave reasoning, which

for now is not possible with the running systems. However, in our separated programs (not detailed here), it is easier to regroup all the answer sets into a unique big one, where the original answer sets are distinguished by an index (from 1 to the number of original answer sets). In this way, we can do brave and cautious reasoning, and check the set of conditions as described in the formalism.

This exhibits a fourth problem with the existing systems. This problem has been addressed in various papers about ASP, but apparently it has not yet given rise to a real running system. Indeed, [13] describes a system which allows to enumerate the answer sets, which is what we need here. However, no running system is referenced in this very interesting paper, which describes small, natural and very useful improvements for ASP systems. A more recent paper [2] describes a more ambitious system which also deals with this point. Hopefully, these very interesting and natural improvements will be introduced in available systems in a near future.

3.2 The generating part: getting the relevant explanation atoms

The “answer” of an ASP program is a set of *answer sets*, that is a set of concrete literals satisfying the rules (see e. g. [1] for exact definitions and [9] for the precise syntax of DLV).

The user provides the following data:

`symbol(alpha)`. for each propositional symbol `alpha` [mandatory only if the symbol does not appear in a causal, ontological or classical atom].
`cause(alpha,beta)`. for each causal atom α *causes* β ,
`ont(alpha,beta)`. for each ontological atom $\alpha \rightarrow_{IS-A} \beta$,
`true(alpha)`. or `-true(alpha)`. for each propositional atom α true or false.

Causal and propositional formulas must be put in conjunctive normal form, in order to be entered as sets of clauses such as

`{-true(epsilon1) v -true(gamma1) v -true(gamma2).
-code>true(epsilon2) v -true(gamma1) v -true(gamma2).}`

for the formula $(\neg\epsilon1 \wedge \neg\epsilon2) \vee \neg\gamma1 \vee \neg\gamma2$;

or `{cause(beta2,gamma) v cause(epsilon3,gamma3).}`

for $(\beta2 \text{ causes } \gamma) \vee (\epsilon3 \text{ causes } \gamma3)$.

Notice that if we really need all the logical models, at least with respect to

some propositional or causal atoms, we must “complete” each propositional or causal atom concerned as follows:

$\text{true}(\alpha) \vee \neg \text{true}(\alpha)$. or $\text{cause}(\alpha, \beta) \vee \neg \text{cause}(\alpha, \beta)$.

This is generally not necessary, and should be avoided as far as possible since it is computationally demanding.

The interesting result consists in the explanation predicates:
 $\text{ecSet}(\alpha, \beta, \{\alpha, \delta, \gamma\})$ represents the explanation atom
 $\alpha \text{ expl } \beta \text{ bec_poss } \{\alpha, \delta, \gamma\}$.

Here come the first rules:

$\text{ontt}(I, J) :- \text{ont}(I, J)$. $\text{ontt}(I, K) :- \text{ontt}(I, J), \text{ont}(J, K)$. (cf 2b §2.2).

For the sake of “safety” of some rules, and for defining impCO introduced below we may need to define all the symbols, and the ones which can appear in an explanation set (suffix “E”).

$\text{symbolE}(X) :- \text{cause}(X, Y)$. $\text{symbolE}(Y) :- \text{cause}(X, Y)$.

$\text{symbolE}(X) :- \text{ont}(X, Y)$. $\text{symbolE}(Y) :- \text{ont}(X, Y)$.

$\text{symbol}(X) :- \text{symbolE}(X)$.

Implication derived from causal and ontological atoms (“s” for “strict”):

$\text{impCO}(I, J) :- \text{cause}(I, J)$. $\text{impCO}(I, J) :- \text{ont}(I, J)$.

$\text{impCO}(I, K) :- \text{impCO}(I, J), \text{impCO}(J, K)$. $\text{impCO}(I, I) :- \text{symbolE}(I)$.

$\text{impCOs}(I, J) :- \text{impCO}(I, J), \text{not impCO}(J, I)$.

We split the general basic generation rule 3a §2.2, in order to improve the computational performances. Indeed, in the first three particular cases, only one optimal initial explanation atom in (I, J) exists, thus the computation can be simplified.

$\text{ecinit}(I, J, E)$ represents $I \text{ expl } J \text{ bec_poss } \{I, E\}$ where this explanation atom is obtained without using the transitivity rule:

$\text{ecinit}(I, J, I) :- \text{cause}(I, J)$. $\text{ecinit}(I, J, I) :- \text{cause}(I, X), \text{ontt}(J, X), \text{impCO}(I, J)$.

$\text{ecinit}(I, J, I) :- \text{cause}(I, X), \text{ontt}(X, J)$.

$\text{ecinit}(I, J, I) :- \text{cause}(I, X), \text{ontt}(E, X), \text{ontt}(E, J), \text{impCO}(I, E)$.

The most complicated case (with the two ontological axioms) may lead to several explanation atoms from I to J , which requires some complications:

$\text{ecinit3p}(I, J, E) :- \text{ecinit}(I, E, E), \text{cause}(I, X), \text{ontt}(E, X), \text{ontt}(E, J), \text{not ecinit}(I, J, I),$
 $\text{not ecinit}(I, J, J)$.

$\text{nonecinit}(I, J, E) :- \text{ecinit3p}(I, J, E1), \text{ecinit3p}(I, J, E), \text{impCOs}(E, E1)$.

$\text{ecinit}(I, J, E) :- \text{ecinit3p}(I, J, E), \text{not nonecinit}(I, J, E)$.

[Avoids keeping clearly non optimal ones.]

Since the set of conditions are singletons or pairs in the initial explanation atoms [ecinit], set representation was not required. Real explanation atoms

[ecSet] are introduced now, together with explicit sets which allow a serious simplification of the ASP rules.

Firstly, we initialize ecSet with ecinit [#insert(Set1,E,Set) means in DLV-Complex: $Set = Set1 \cup \{E\}$]:

ecSet(I,J,{I}) :- ecinit(I,J,I).

ecSet(I,J,{I,J}) :- ecinit(I,J,J), not ecSet(I,J,{I}).

ecSet(I,J,{I,E}) :- ecinit(I,J,E), not ecSet(I,J,{I}), not ecSet(I,J,{I,J}).

Then comes the translation of Point 3b §2.2.

ecSet(I,J,Set) :- ecSet(I,K,Set1), not ecSet(I,J,Set1), ecinit(K,J,E2), E2 != K, #insert(Set1,E2,Set).

ecSet(I,J,Set) :- ecSet(I,K,Set), ecinit(K,J,K).

[Nothing to add in this case, since we know that $Set \vdash_C K$.]

And this is enough for getting all the relevant explanation atoms. Only a few computational optimization tricks complicate a little bit the writing, however, the ASP rules remain very close to the formal rules given in §2.2.

3.3 Optimizing the explanation atoms

This is the most computationally demanding part. Rigorously, it is not mandatory. However, it is important in order to avoid providing too many unnecessary explanation atoms which complicate the interpretation of the result.

As a short example, suppose we have following data:

α causes β , α causes β_0 , β_2 causes γ , β_1 causes γ ,
 $\beta_2 \rightarrow_{IS-A} \beta_0$, $\beta_1 \rightarrow_{IS-A} \beta$, $\beta_2 \rightarrow_{IS-A} \beta_1$.

Then, we get the following explanation atoms concerning (α, γ) :

Expl1: $\alpha \text{ expl } \gamma \text{ bec_poss } \{\alpha, \beta_1\}$ and Expl2: $\alpha \text{ expl } \gamma \text{ bec_poss } \{\alpha, \beta_2\}$.

Since we have $\beta_2 \rightarrow_{IS-A} \beta_1$, we get $\beta_2 \rightarrow \beta_1$ from 2a §2.2 in W , even if the user does not provide any explicit W . Thus, each element in $\{\alpha, \beta_1\}$ is entailed by some element in $\{\alpha, \beta_2\}$ and not conversely. Thus, the weaker set $\{\alpha, \beta_1\}$ is more likely to be satisfied, whatever are the other data, and in particular whatever may be an explicit W given as additional data. Thus, it is useless, and disturbing, to provide Expl2: Expl1 is enough. It happens here that Expl1 is *optimal* with the given data. Obviously, in more complex cases involving disjunction, the “element wise” test would not be enough to discard all the sets of conditions which are too strong for entailment. However, the

test described here is a good compromise between efficient computation and readability of the result.

We have tried various other programs providing only the [quasi] optimal sets, in order to avoid this “optimizing” part. All of them were much slower: it is better to provide first the explanation atoms with a program such as the one given in §3.2, which does not take great care for avoiding superfluous answers, and then to prune the set of solutions.

Here comes the “optimizing part”. Remind that not all the simplifications or optimizations possible are made. As explained above, only those which are easy to compute are made. Indeed, except in artificially complicated data, most of the simplifications are made in this way, while dealing with the tricky cases would make the program too slow for a marginal advantage. These simplification/optimizations are mandatory in order to facilitate human reading, and could be omitted in a purely formal perspective since anyway the produced explanation atoms cover all the possible situations. The rules are very simple:

`imp(I,J) :- impCO(I,J)`. (additional predicate `imp` useless in the version presented here).

Propagating the truth values:

`true(J) :- true(I), imp(I,J)`. `-true(I) :- -true(J), imp(I,J)`.

Eliminating sets of conditions which contain another set

[`#subSet(Set1,Set)` means $Set1 \subseteq Set$ and `#member(E,Set)` means $E \in Set$]:
`toolargeSet(I,J,Set) :- ecSet(I,J,Set1), ecSet(I,J,Set), Set1 != Set,`
`#subSet(Set1,Set).`

`ecSetsmall(I,J,Set) :- ecSet(I,J,Set), not toolargeSet(I,J,Set).`

`impCOSetEl(Set1,E2) :- ecSetsmall(I,J,Set1), ecSetsmall(I,J,Set2),`
`Set1 != Set2, impCO(E1,E2), #member(E1,Set1), #member(E2,Set2),`
`not #member(E1,Set2), not #member(E2,Set1).`

`nonImpCOSet(Set1,Set2) :- ecSetsmall(I,J,Set1), ecSetsmall(I,J,Set2),`
`Set1 != Set2, symbolE(E2), #member(E2,Set2), not #member(E2,Set1),`
`not impCOSetEl(Set1,E2). % (“not #member(E2,Set1),” is optional)`

`toostrongSet(I,J,Set) :- ecSetsmall(I,J,Set), ecSetsmall(I,J,Set1),`
`Set != Set1, nonImpCOSet(Set1,Set), not nonImpCOSet(Set,Set1).`

`ecSetRes(I,J,Set) :- ecSetsmall(I,J,Set), not toostrongSet(I,J,Set).`

As a result, we only keep the explanation atoms `ecSetRes(I,J,Set)` where no element can be removed from `Set` and where no set `StrongSet` is such that `ecSet(I,J,StrongSet)` and each element of `Set` is implied (in the meaning of `impCO`) by some element of `StrongSet`, and not conversely.

This program is rather slow: the two programs (§3.2 and the present 3.3) can be launched together for examples such as in the diagram of §2.3 and for slightly larger example, but it is impossible for our “big example”. However, it is possible to launch the first program (instantaneous on our examples), and then the second one starting with the results of the first one. Then, our “big example” is solved in far less than one minute on our computer.

3.4 Checking the set of conditions

Finally, the following program starts from the result of the preceding programs and checks, in each answer set, whether the set of conditions is satisfied or not. This program could also be launched starting from the result of the first program (replacing `ecSetRes` with `ecSet`), simply superfluous explanation atoms would be checked also, and almost no “optimization” would be done.

The result is given by `explVer(I,J,Set)`: *I expl J bec_poss Set* where *Set* is satisfiable in the answer set considered (“Ver” stands for “verified”).

`explSuppr(I,J,Set) :- ecSetRes(I,J,Set), -true(E), #member(E,Set).`

`explVer(I,J,Set) :- ecSetRes(I,J,Set), not explSuppr(I,J,Set).`

Notice that only “individual” checking is made here, in accordance with the requirement that the computational properties remain manageable. In each answer set, this check is enough.

Notice that, as already evoked, in our running split program (not provided in full here for lack of space), all the answer sets are put into a single one, with an index parameter added for each old answer set. This allows to make real cautious reasoning when checking the consistency of the explanation sets (and only for this point), in accordance with the formalism. Then, a few more rules could be added in order to get results in full accordance with our formalism (since e.g. answer sets do not provide all the classical models). However, even if these rules are written with care, it seems unlikely that the resulting program can run in practice for great examples. As already evoked above, if we want to get all the classical models, a possibility is to require the *completion* of all the atoms (propositional or causal) which can provoke the existence of various answer sets, by adding the disjunctive rules

`true(alpha) v -true(alpha).` and `cause(gamma,delta) v -cause(gamma,delta).` for each of these atoms involved in a non atomic formula. This solution is easy to write in the program, but becomes clearly unmanageable from a computational point of view since the number of answer sets may explode. It is the standard, but not practical, way for getting all the classical models

in ASP. Then, together with real cautious reasoning, checking for possible consistency of the sets could be made in full accordance with the formalism. Notice that it is not always clear whether the intended meaning is better rendered by classical models than by answer sets.

We have taken care to describe all the practical limitations of our solution. However, apart from these (generally minor in practice) points, we hope to have convinced that ASP is very interesting in order to deal with this kind of problem.

One of the advantages is that if we want to modify a rule of the formalism, this can be done easily, since there is a close relation between the formal rules and the ASP rules. The main difference between formal rules and the ASP rules described above concern the “initial rule” in the first program §3.2. This difference comes from the fact that we have done our best to provide a program running in a reasonable time. This is another interest of using ASP: such computational optimizations can be introduced in a relatively natural way. Even with these complications, modifying the rules remains easy.

Also, for the example described here, the gain of using DLV-Complex instead of pure DLV (or gringo/claspD) is significant and worth mentioning.

3.5 Dealing with predicate logic

For the sake of conciseness we have given the translation of the propositional version of our formalism. In fact, this is not a serious concern. Indeed, the computational efficiency is similar, since the real tricky points all appear in the propositional version. A standard way to deal with predicates in ASP, starting from a propositional version, is to add a few parameters representing the predicate symbols. As an example, let us consider atoms such as *heard(bell)* or *like(bell)* or *own(student,book)* in place of α or β . In the formalism, all we need to do is to replace e. g. facts such as *heard_bell causes on_alarm.* by *heard(bell) causes on(alarm).*, and to replace all the rules accordingly.

In this way, we can also translate the slightly extended ontology described in [4]. There, two kinds of parameters are considered for each predicate: the ones for which the predicate is *essentially universal* and the ones for which the predicate is *essentially existential*. Let us suppose that we intend that *heard* is essentially existential with respect to its parameter and *like* essentially universal. Intuitively, this means that *heard* is intended as meaning

heard some while *like* means *like all*. The ontological information would be provided by the user as follows: $loud_bell \rightarrow_{IS-A(object)} bell$ and $white_car \rightarrow_{IS-A(object)} car$.

This would provide the following \rightarrow_{IS-A} relation between atoms
 $heard(bell) \rightarrow_{IS-A} heard(loud_bell)$. and
 $like(car) \rightarrow_{IS-A} like(white_car)$..

This means that the ASP formulation would contain the following facts and rules: $ont_object(loud_bell, bell)$. $ont_object(white_car, car)$.
 $ont([P,X],[P,Y]) :- onekind(P), ont_object(X,Y)$. and
 $ont([P,Y],[P,X]) :- allkind(P), ont_object(X,Y)$.

Dealing with lists (denoted inside brackets) of DLV-Complex is convenient: a small set of ASP rules can deal with predicates of any arity, even if we do not describe the full formalism here for the sake of conciseness.

Propositional parameters would be dealt with the following rule:
 $ont([A],[B]) :- ont_object(A,B), propkind(A), propkind(B)$.

Let us consider a binary predicate *own*, with $own(student, book)$ meaning that the individual (or object) *student* (whatever it denotes in our formalism) owns the object called *book*. Here, $own(X, Y)$ is intended to mean: “*all X own some Y*”. The following rule (0-0) can deal with this predicate:
 $ont([P,X,Y],[P,X1,Y1]) :- all_onekind(P), ont_object(X1,X), ont_object(Y,Y1)$.

The user should state the kind of parameters for each predicate, as follows:
 $onekind(heard)$. $allkind(like)$. $all_onekind(own)$. $propkind(alpha)$.

(A more extensive use of the list notation would allow to write rules dealing with any arity in a yet more natural way.)

Writing $onekind(heard)$ means that the classes or objects which can appear as parameters of the unary predicate *heard* of the formal system have adapted meaning. In this example, *bell* may denote the class of some bells considered in the situation at hand, *loud_bell* the class of those bells which are loud and *some_bell* could denote a precise *loud_bell*. Notice that the objects are intended to denote either individuals or classes of individuals.

In this situation the user could add the following atom to the one given above about bells: $ont_object(some_bell, loud_bell)$.

As an example, let us consider our binary predicate *own*, with the known informations provided by the user as follows in ASP:
 $ont_object(tom, student)$. $ont_object(book, document)$.

Then, the following ontological atoms would be deduced by the system (again in ASP notation):

```
ont([own,tom,book], [own,tom,document])
ont([own,student,book], [own,tom,book])
ont([own,student,book], [own,tom,document]).
```

A predicate for which no “all/one” information is given could be used, but it would never give rise to ontological atoms.

This kind of formulation is in accordance with our requirements for the formalism: allowing serious expressiveness while keeping computation manageable. To this respect, for large examples it can be useful to reduce the instantiation size by restricting the parameters which can be used for some predicates, by adding `not unrestr(P)`, `kindPar(P,X,Y)` to the body of the rule (0-0). Rules such as the following ones should be added

```
unrestr(P) :- not restr(P), pred(P). pred(P) :- onekind(P). , [...allkind(P),...].
```

The user would provide the relevant information about `restr` and `kindPar`.

3.6 Conclusion and future work

We have shown how a recent version of running ASP systems allow easy translation of logical formalisms. The example given here is a formalism allowing to infer “explanations” from “causal and ontological” information, with two requirements: 1 It must be easy and natural to formalize a given situation. 2 Computation should remain as manageable as possible.

These requirements justify some restrictions of our formalism, in particular the fact that we accept only classical atoms inside the causal and ontological atoms, without operators such as negation or disjunction. The formalism is not restricted to the propositional case, and, as explained in the last subsection, it involves an ontology slightly more general than the rudimentary one developed in the full ASP description given in the preceding subsections.

From the perspective of existing (and predictable in a near future) ASP systems, here are some conclusions that can be drawn:

The existing recent extensions of ASP systems are a real bonus. It is much easier to *write* the program now than it was a few years ago. More importantly perhaps, the difference is even greater when *reading* such a program. Indeed the claimed assertions about ASP, namely that it makes programs easy to write and to read, are true only when such extensions are used. Oth-

erwise, due to the fact that in practice it is very hard to use some portion of a program in different places in a greater program, the real ASP programs are either very restricted in their application, or hard to read (and thus to evolve). In our opinion, DLV with templates (DLT) solves a serious part of this problem. However, on our formalism, it happens that it is DLV-Complex, since it accepts sets and lists in a natural and efficient way, which has made the difference. Indeed, our formalism involves sets in a non trivial way, thus the present program is far more readable than the previous ones, in pure DLV.

From a formal perspective, neither DLV-Complex nor DLT are revolutionary. To a great extent, all they do is add way for writing natural programs where pure ASP systems need cumbersome rewritings of parts of the programs. Generally, using them does not improve computational efficiency. The fact is that they always allow more convenient writing, and particularly much easier reading, which changes the life of the programmer. Expert ASP programmers were able to use previous ASP systems together with small parts of more standard programming, but this was not very attracting for most of the potential users of ASP systems.

So, what can we expect now?

For now, DLT cannot work together with DLV-Complex, and this is the first thing we can expect, hopefully in a very near future, together with similar facilities for gringo/clasp and other systems.

Also, “brave” and “cautious” reasoning do not exist in full generality (to our knowledge) in available systems. May be our formalism is an extreme example of this point (a mix of cautious and standard reasoning would be useful), but in fact this happens to be a serious concern in many circumstances. Again, programmers can redirect the result (the set of all answer sets) to a very simple other ASP program, for playing with the answer sets in a more or less complex way, but this is not very convenient. In the literature, we can encounter various texts about this problem, describing systems which solve it in a natural manner, but, to our knowledge, such systems are not yet available. Some [meta?]predicates, as described in e.g. [13], which present a relatively simple and natural way to write programs doing this, would suffice in many cases. In this way real brave and cautious reasoning could be envisioned, and much more. We hope that this will be the case in available systems in a near future. Clearly, dealing with too many answer sets cannot be done with huge domains, but there are many cases where it would be interesting to have full access to the set of all the answer sets.

Here is a last feature that could be interesting. The full program did not work for what is called our “great example”. However, when split in two or three parts, it worked relatively well. In this case the split was very easy to do. Could it be that future systems detect such possible split, and thus extend their range of application? In our example, computing `ecSet` first, then `ecSetRes` and finally `ecSetVer` should be automatized. It seems easy to detect such one way dependencies, without retro-action. The great difference in practice between launching the three programs together, and launching them one after the other, shows that such improvement could have spectacular consequences.

For what concerns our own work, the important things to do are to apply the formalism to real situations, and, to this respect, firstly to significantly extend our notion of “ontology” towards a real one.

Acknowledgement

The author thanks the reviewers for their helpful and constructive comments, and Marie-Odile Cordier and Philippe Besnard who made this work possible.

References

- [1] Chitta Baral, Knowledge representation, reasoning and declarative problem solving, Cambridge University Press, 2003.
- [2] Chitta Baral, Juraj Dzifcak and Hiro Takahashi, Macros, Macro Calls and Use of Ensembles in Modular Answer Set Programming, ICLP, in LNCS 4079, 2006.
- [3] Besnard, Philippe and Cordier, Marie-Odile and Moinard, Yves, Ontology-based inference for causal explanation, KSEM 2007, in LNAI 4798, Zili Zhang and Jrg Siekmann (eds), Springer, pp.153–164, 2007.
- [4] Besnard, Philippe and Cordier, Marie-Odile and Moinard, Yves, Ontology-based inference for causal explanation, Integrated Computer-Aided Engineering Journal, IOS Press, 15(4): 351–367, 2008.

- [5] Francesco Calimeri and Giovambattista Ianni, Template programs for Disjunctive Logic Programming: An operational semantics, *AI Communications*, 19(3): 193–206, 2006, IOS Press
- [6] M. Gebser and B. Kaufmann and T. Schaub, The Conflict-Driven Answer Set Solver *clasp*: Progress Report, In E. Erdem and F. Lin and T. Schaub (eds), *LPNMR'09* In *LNAI* 5753, pp. 509-514. Springer-Verlag, 2009.
- [7] Giunchiglia E., Lee J., Lifschitz V., McCain N., Turner H. Nonmonotonic Causal Theories. *Artificial Intelligence* 153(1–2):49–104, 2004.
- [8] Halpern J. and Pearl J. Causes and Explanations: A Structural-Model Approach - Part II: Explanations. In *IJCAI-01*, pp. 27-34. Morgan Kaufmann, 2001.
- [9] 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.
- [10] Mellor D. H. *The Facts of Causation*. Routledge, 1995.
- [11] Yves Moinard An Experience of Using ASP for Toy Examples, *ASP'07*, in *Advances in Theory and Implementation*, publ. Faculdade de Ciencias, Universidade do Porto, pp. 133–147, 2007.
- [12] Shafer G. Causal Logic. In H. Prade (ed), *ECAI-98*, pp. 711-720, 1998.
- [13] Luis Tari and Chitta Baral and Saadat Anwar, A Language for Modular Answer Set Programming: Application to ACC Tournament Scheduling, *ASP 05*, M. De Vos and A Proveti (eds), *Answer Set Programming, Advances in Theory and Implementation*, CEUR-WS.org publ., CEUR Workshop Proc., vol. 142, 2005.