A System for Explainable Answer Set Programming

Pedro Cabalar[1], Jorge Fandinno[2] and Brais Muñiz[1]
[1] *University of Corunna, Corunna, Spain.*
(*e-mail:* {cabalar,brais.mcastro}@udc.es)

[2] *University of Potsdam, Germany*
(*e-mail:* fandinno@uni-potsdam.de)

submitted 21 June 2009; revised ; accepted

Abstract

We present `xclingo`, a tool for generating explanations from ASP programs annotated with text and labels. These annotations allow tracing the application of rules or the atoms derived by them. The input of `xclingo` is a markup language written as ASP comment lines, so the programs annotated in this way can still be accepted by a standard ASP solver. `xclingo` translates the annotations into additional predicates and rules and uses the ASP solver `clingo` to obtain the extension of those auxiliary predicates. This information is used afterwards to construct derivation trees containing textual explanations. The language allows selecting which atoms to explain and, in its turn, which atoms or rules to include in those explanations. We illustrate the basic features through a diagnosis problem from the literature.

KEYWORDS: Answer Set Programming, Causal justifications, Non-Monotonic Reasoning, ASP debugging, Diagnosis.

1 Introduction

Answer Set Programming (ASP) (Niemelä 1999; Marek and Truszczyński 1999; Brewka et al. 2011) is a successful paradigm for Knowledge Representation and problem solving. Under this paradigm, the programmer represents a problem as a logic program formed by a set of rules and obtains solutions to that problem in terms of models of the program called answer sets. Thanks to the availability of efficient solvers, ASP is nowadays applied in a wide variety of areas including robotic intra-logistics, routing, bioinformatics, medicine, music composition, and many more.

An ASP program does not contain information about the method to obtain the answer sets, something that is completely delegated to the ASP solver. This, of course, has the advantage of making ASP a fully declarative language, where the programmer must concentrate on specification rather than on design of search algorithms. However, when it comes to *explainability* of the obtained results, the information provided by answer sets themselves is usually scarce. Unlike in procedural languages, where we can always follow the execution trace to obtain a given result, in ASP we cannot simply generate an explanation for some fact derived in one of the answer sets of the program, unless we introduce new atoms or rules in the program. There exist several approaches for obtaining

justifications for answer sets: for a recent review, see (Fandinno and Schulz 2019). Some of them are more oriented to *debugging of ASP programs* while others are interested in the causal nature of justifications themselves. What these approaches generally do is to offer some kind of enlightening about the derivation process of the rules that led to finally include (or not) some literal in an answer set.

Justifying the result of an ASP program is not only interesting for the programmer but has also other implications, especially in the context of *explainable Artificial Intelligence*. For instance, since the approval of the General Data Protection Regulation (GDPR) by the European Union (EU) every system that makes automatic decisions that affect to persons must offer some kind of *explanation* on the logic involved in the decision making process, obviously in a human-readable way.

In this paper, we present `xclingo`, a tool for generating explanations of annotated programs for the ASP solver `clingo` (Gebser et al. 2016). The input accepted by `xclingo` is a markup language that introduces annotations through program comments, specifying which rules or atoms will be traced. Using these annotations, `xclingo` generates derivation trees following the theoretical framework of *causal graph justifications* introduced in (Cabalar et al. 2014). The tool uses the `clingo` python API to translate the annotations into additional rules with auxiliary predicates and computes the explanations from the information obtained from these predicates.

Since causal graphs show possible derivations for the conclusions, it is difficult to avoid that, as the size of a rule system increases, the readability and comprehension becomes more difficult. A large explanation may be tractable by a computer but not too useful for a human reader, who is normally more concerned about *relevant* pieces of information. Because of that, `xclingo` puts an extra effort on creating a flexible way to format these explanations at a detailed level, style and size. It is the programmer who chooses the verbosity of the explanations, adapting them to the context (information detail, user expertise, language, and so on).

Although its main purpose is the explanation of ASP program conclusions, we also find `xclingo` helpful for program debugging, again, because of the flexibility of its explanation configuration system.

In order to show its features and to demonstrate its usefulness, we will use a diagnosis problem example from the literature as a guide. We have chosen this example because we find the nature of `xclingo`'s explanations very helpful in diagnosis.

The rest of the paper is organised as follows. First we introduce our diagnosis running example. Then, the input language and the features of `xclingo` are described. Then, we describe how the translation of the input into a standard ASP program work. Then, we describe how the explanations are computed from the solution of the translated program. Then we talk about some related work. Finally, we close the paper with some conclusions about the tool and some future work.

2 Motivating Example

We will use an example from (Balduccini and Gelfond 2003) as a guide (Figure 2). In the example, an analog circuit AC is presented. In it, an agent can close a switch that should ultimately cause a bulb to turn on. However, there are some exogenous actions that can modify the environment and make the bulb not to turn on by closing the switch. Our

goal is to develop a diagnostic system that can identify the reasons why the light does not turn on and present them to the user in the form of readable explanations.

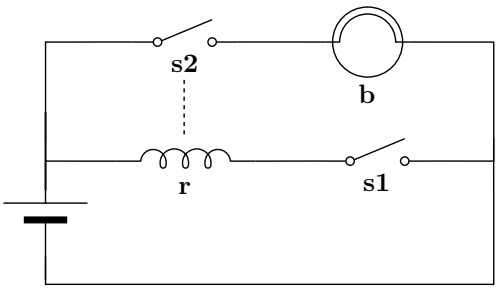

Fig. 1. A circuit with a bulb b, a relay r and two switches, $s1$ and $s2$.

Example 1. (From (Balduccini and Gelfond 2003)) *Consider a system S consisting of an agent operating an analog circuit AC from Figure 1. We assume that switches s_1 and s_2 are mechanical components which cannot become damaged. Relay r is a magnetic coil. If not damaged, it is activated when s_1 is closed, causing s_2 to close. Undamaged bulb b emits light if s_2 is closed. For simplicity of presentation we consider the agent capable of performing only one action, close(s_1). The environment can be represented by two damaging exogenous*[1] *actions: brk, which causes b to become faulty, and srg (power surge), which damages r and also b assuming that b is not protected. Suppose that the agent operating this device is given a goal of lighting the bulb. He realizes that this can be achieved by closing the first switch, performs the operation, and discovers that the bulb is not lit.* □

Our ASP implementation of this example (we call program P_1 in Listings 1 and 2) loosely follows the one presented in (Balduccini and Gelfond 2003) with some minor modifications for improving the explanation results. Listing 1 contains the basic type definitions. The predicate names are self-explanatory, except perhaps lines 15-24. This is because we allow arbitrary fluent domains that can be specified explicitly through predicate `value(F,V)`, meaning that fluent `F` may have value `V`. When no value has been specified in that way, fluents are assumed Boolean by default. Finally, predicate `domain(F,V)` collects all domain values for `V`, regardless of whether they are defined explicitly or by default. In our example, fluents `relay, light, s1` and `s2` can take values `on` and `off` (to make them more readable) whereas the rest of fluents are Boolean.

Listing 2 contains the description of the problem. Given any action `A`, fluent `F`, value `V` and time point `I` we use the following predicates:

`h(F,V,I)`	= `F` holds value `V` at `I`
`obs_h(F,V,I)`	= `F` was observed to hold value `V` at `I`
`c(F,V,I)`	= `F`'s value was caused to be `V` at `I`
`c(F,I)`	= `F`'s value was caused at `I`
`o(A,I)`	= `A` occurred at `I`
`obs_o(A,I)`	= `A` was observed to occur at `I`

If we execute `clingo` on this code, we obtain the three answer sets that correspond

```
plength(1).
time(0..L) :- plength(L).
step(1..L) :- plength(L).

switch(s1). switch(s2).
component(relay). component(bulb).

fluent(relay).
fluent(light).
fluent(b_prot).
fluent(S):-switch(S).
abfluent(ab(C)) :- component(C).
fluent(F) :- abfluent(F).

value(relay,on). value(relay,off).
value(light,on). value(light,off).
value(S,open) :- switch(S).
value(S,closed) :- switch(S).
hasvalue(F) :- value(F,V).
% Fluents are boolean by default
domain(F,true) :- fluent(F), not hasvalue(F).
domain(F,false) :- fluent(F), not hasvalue(F).
% otherwise, they take the specified values
domain(F,V) :- value(F,V).


agent(close(s1)).
exog(break).
exog(surge).
action(Y):-exog(Y).
action(Y):-agent(Y).
```

Listing 1. Some type predicates for program P_1.

to the possible diagnosis: one including an exogenous action `o(break,1)`; a second one with the exogenous action `o(surge,1)`; and, finally, a third, non-minimal diagnosis where both exogenous actions occur. Of course, in the original work by Balduccini and Gelfond, diagnoses were additionally minimised to avoid unnecessary addition of exogenous actions, but for the purpose of this paper, we will consider the three answer sets of program P_1 as equally interesting for generating explanations. At the moment, `xclingo` cannot properly deal with minimisation clauses in `clingo` yet.

In the rest of the paper, we will be using this code as a running example. We will complete it using different `xclingo` features in order to get the diagnoses in a fully readable and understandable way.

3 The `xclingo` system

To understand the purpose of the different types of annotations in `xclingo` it is perhaps better to start illustrating the kind of output we expect to achieve. For instance, Figure 2 shows the result we obtain for program P_1 encoding Example 1 once it is annotated – we will see how these annotations look like later on. As we can see, the programmer has requested explanations for fluents `light` and `relay`. Each explanation must be understood as follows. Below each explained (true) atom (preceded by `>>`) we get a list of trees that correspond to alternative (and equally effective) causes for the atom. Then, in each tree, two explanations at the same level must be understood as a joint cause (the two effects

```
% Inertia
h(F,V,I) :- h(F,V,I-1), not c(F,I), step(I).

% Axioms for caused
h(F,V,J) :- c(F,V,J).
c(F,J)   :- c(F,V,J).

% Direct effects
c(s1,closed,I) :- o(close(s1),I), step(I).

% Indirect effects
c(relay,on,J)   :- h(s1,closed,J), h(ab(relay),false,J), time(J).
c(relay,off,J)  :- h(s1,open,J), time(J).
c(relay,off,J)  :- h(ab(relay),true,J), time(J).
c(s2,closed,J)  :- h(relay,on,J), time(J).
c(light,on,J)    :- h(s2,closed,J), h(ab(bulb),false,J), time(J).
c(light,off,J)   :- h(s2,open,J), time(J).
c(light,off,J)   :- h(ab(bulb),true,J), time(J).

% Malfunctioning
c(ab(bulb),true,I) :- o(break,I), step(I).
c(ab(relay),true,I) :- o(surge,I), step(I).
c(ab(bulb),true,I) :- o(surge,I), not h(b_prot,true,I-1), step(I).


% Executability
:- o(close(S),I), h(S,closed,I-1), step(I).

% Something happening actually occurs
o(A,I) :- obs_o(A,I), step(I).

% Check that observations hold
:- obs_h(F,V,J), not h(F,V,J).

% Completing the initial state
h(F,V,0) :- domain(F,V), not -h(F,V,0).
-h(F,V,0) :- h(F,W,0), domain(F,V), W!=V.


% A history
obs_h(s1,open,0).
obs_h(s2,open,0).
obs_h(b_prot,true,0).
obs_h(ab(bulb),false,0).
obs_h(ab(relay),false,0).
obs_o(close(s1),1).

% Something went wrong
obs_h(light,off,1).

% Diagnostic module: generate exogenous actions
o(Z,I) :- step(I), exog(Z), not no(Z,I).
no(Z,I) :- step(I), exog(Z), not o(Z,I).
```

Listing 2. Program P_1 describing Example 1.

act together) and each time we jump into a lower level, we can read this as a "because" relation.

Figure 2 shows the same three answer sets we obtain with plain `clingo`. Answer set 1 corresponds to the case in which both a power surge occurred and something broke the bulb. The explanation for the relay being off (lines 2-6) can be read as follows: "the relay

is not working at 1 *because* it has been damaged *because* there has been a power surge." We have started all explanations for exogenous actions with the word `Hypothesis` to clarify which these are assumptions added to explain the observations. As we can see, in this first answer set, there are two alternative valid causes for the light being off. The first one (lines 9-12) is that the bulb was damaged because something broke it. The second cause (lines 14-16) is that the light was already off in the initial state because switch `s2` was initially open. Note that `xclingo` only collects *effective causes*, but not counterfactual explanations: had the relay been activated, `s2` would be closed, something that the power surge has avoided. Instead, `xclingo` only reflects causes that *produce* an effect, normally breaking a default.

```
1  Answer: 1
2  >> h(relay,off,1)       [1]
3    *
4    |__"The relay is not working at 1"
5    |  |__"The relay has been damaged at 1"
6    |  |  |__"Hypothesis: there has been a power surge at 1"
7
8  >> h(light,off,1)       [2]
9    *
10   |__"The light is off at 1"
11   |  |__"The bulb has been damaged at 1"
12   |  |  |__"Hypothesis: something has broken the bulb at 1"
13
14   *
15   |__"The light is off at 1"
16   |  |__"s2 was initially open"
17
18
19 Answer: 2
20 >> h(relay,off,1)       [1]
21   *
22   |__"The relay is not working at 1"
23   |  |__"The relay has been damaged at 1"
24   |  |  |__"Hypothesis: there has been a power surge at 1"
25
26 >> h(light,off,1)       [1]
27   *
28   |__"The light is off at 1"
29   |  |__"s2 was initially open"
30
31
32 Answer: 3
33 >> h(relay,on,1)        [1]
34   *
35   |__"The relay is working at 1"
36   |  |__"The agent has closed switch s1 at 1"
37   |  |__"Initially, the relay was not damaged"
38
39 >> h(light,off,1)       [1]
40   *
41   |__"The light is off at 1"
42   |  |__"The bulb has been damaged at 1"
43   |  |  |__"Hypothesis: something has broken the bulb at 1"
```

Fig. 2. Explanations obtained for the annotated version of P_1.

Answer set 2 (lines 19-29) corresponds to the case in which we just had a power surge.

When this happens, the relay is not working (as in the answer set 1) and the light simply remains off, since `s2` was initially open. In this case, we do not get the additional reason for having the light off, since the bulb is not broken.

Finally, answer set 3 shows the case where something breaks the bulb but there is no power surge. In this case, we can see that the relay eventually worked because the agent closed switch 1 and the relay was not initially damaged. As nothing else happens, the relay is still undamaged in state 1. Notice that these two things (the agent closing the switch and the relay being initially undamaged) constitute a *joint cause* altogether: lines 36 and 37 share the same parent at the explanation tree.

As we can see, the output of `xclingo` can be properly tuned to display enough information to provide a reasonable justification of the outcome, but not so much information that the explanation becomes too large or contains irrelevant derivations. For instance, we have carefully avoided to trace the application of inertia, since this is a default and one would expect to be warned only when the default is violated.

Let us proceed now to describe how these explanations are generated, including the markup language accepted by `xclingo`. Each explanation is a tree-based graph that shows the derivation proof for an atom. In these graphs, each node is a *trace label* that corresponds to a `string` associated to some fired rule or to some derived atom. Trace labels can be created manually through *annotations* or can be automatically generated by `xclingo`. For instance, the explanation for `h(light,off,1)` in answer set 3 is displayed in Figure 3. As it can be seen, in this case the explanation is fine grained and contains the complete derivation tree obtained from the (positive part of) the program. Explanations do not contain information about default negation: as explained before, `xclingo` focuses on exceptions to defaults. When we use manually defined trace labels, as in Figure 2, the tool skips in the derivation tree any intermediate node that we have not specified to be traced.

In this way we give the programmers the control of the detail to be shown, allowing them to choose which information is relevant but, at the same time, the option to automatically trace everything possible application of rules. Our feeling is that having all the detail can be useful in specific moments or for debugging the behaviour of the rules.

```
1  >> h(light,off,1)        [1]
2    *
3    |__h(light,off,1)
4    |   |__c(light,off,1)
5    |   |   |__h(ab(bulb),true,1)
6    |   |   |   |__c(ab(bulb),true,1)
7    |   |   |   |   |__o(break,1)
8    |   |   |   |   |   |__step(1)
9    |   |   |   |   |   |   |__plength(1)
10   |   |   |   |   |   |__exog(break)
11   |   |   |   |   |__step(1)
12   |   |   |   |   |   |__plength(1)
13   |   |   |__time(1)
14   |   |   |   |__plength(1)
```

Fig. 3. Explanation using automatically generated trace labels.

For adding custom trace labels to a program, the programmer has to make use of the `xclingo`'s markup language. It works by adding annotations to the program that

start with `%!`, so they are just treated as comments by a plain ASP solver like `clingo`. There exist two different types of annotations for writing custom trace labels. The first, `trace_rule`, allows the user to write a custom trace label and to associate it with a specific rule in the program. Listing 3 shows how we have modified lines 11-23 from Listing 2 to associate some custom trace labels with those rules.

```
1  %%%%%% Indirect effects
2  %!trace_rule {"The relay is working at %",J}
3    c(relay,on,J)   :- h(s1,closed,J), h(ab(relay),false,J), time(J).
4
5  %!trace_rule {"The relay is not working at %",J}
6    c(relay,off,J)  :- h(s1,open,J), time(J).
7
8  %!trace_rule {"The relay is not working at %",J}
9    c(relay,off,J)  :- h(ab(relay),true,J), time(J).
10
11 c(s2,closed,J)  :- h(relay,on,J), time(J).
12
13 %!trace_rule {"The light is on at %",J}
14   c(light,on,J)    :- h(s2,closed,J), h(ab(bulb),false,J), time(J).
15
16 %!trace_rule {"The light is off at %",J}
17   c(light,off,J)   :- h(s2,open,J), time(J).
18
19 %!trace_rule {"The light is off at %",J}
20   c(light,off,J)   :- h(ab(bulb),true,J), time(J).
21
22 %%%%%% Malfunctioning
23 %!trace_rule {"The bulb has been damaged at %",I}
24   c(ab(bulb),true,I) :- o(break,I), step(I).
25
26 %!trace_rule {"The relay has been damaged at %",I}
27   c(ab(relay),true,I) :- o(surge,I), step(I).
28
29 %!trace_rule {"The relay has been damaged at %",I}
30   c(ab(bulb),true,I) :- o(surge,I), not h(b_prot,true,I-1), step(I).
```

Listing 3. Fragment F_1. Adding trace labels to specific rules with `trace_rule`

`trace_rule` annotations are associated to the rule they precede. Inside the braces, the first argument is mandatory and it must be a string enclosed by quotes. The rest of the arguments are optional and must be variable names used either in the head or in the body of the rule. The `%` placeholders are special characters that will be replaced by the values of the variables after solving, according to the order that variables are listed after the first argument.

Trying to write trace labels only using `trace_rule` annotations soon make the code bigger, redundant and harder to maintain. For those cases, `trace` annotations are more suitable instead. They allow to associate one label to multiple atoms at once. Listing 4, shows the `trace` annotations added for obtaining the output from Figure 2. For the part between braces, the syntax works the same that in `trace_rule` annotations, but instead of having a rule after that, `trace` annotations have a *conditional atom* that is used to define the set of atoms that the label should be associated with. Also, the annotation must end with a dot. `trace` annotations are associated with a defined set of atoms, regardless of which rules have triggered them. They are less specific, but allow other

general interesting features. For instance, they can be stored separately from the base code, multiple versions of the same trace labels could be written in different languages, for different users, with different detail level, etc.

```
1 %!trace {"Hypothesis: there has been a power surge at %",J} o(surge,J).
2 %!trace {"Hypothesis: something has broken the bulb at %",J} o(break,J).
3 %!trace {"The agent has closed switch s1 at %",J} o(close(s1),J).
4
5 %!trace {"The % was initially damaged",C} h(ab(C),true,0).
6 %!trace {"Initially, the % was not damaged",C} h(ab(C),false,0).
7
8 %!trace {"% was initially %",F,V} h(F,V,0) : not abfluent(F).
```

Listing 4. Fragment F_2. `trace` annotations added to Program P_1.

With the additions in Listings 3 and 4, the explanations we obtain look like those in Figure 2. But, as a `clingo` program without `#show` annotations, the output would include the explanation of every atom in every answer set. `xclingo` also includes its own version of `#show` for choosing which atoms should be explained. Listing 5 contains the `show_trace` annotations we added for obtaining the output seen in Figure 2. Again, in general, `show_trace` annotations allow conditional atoms, as happened with `trace`.

```
1 %!show_trace h(light,V,1).
2 %!show_trace h(relay,V,1).
```

Listing 5. Fragment F_3. `show_trace` annotations added to Program P_1.

4 Intermediate translation

Before solving, `xclingo` translates the rules of the program into a format that enables to record which rules have been triggered after solving, but without altering the results of the original program. Also, annotations have to be translated into something that `clingo` can handle. Since these two needs are very different, the translation is explicitly divided into two phases: phase 1 translates annotations and phase 2 translates the rest of the rules.

After phase 1 both `trace_rule` and `trace` annotations are translated into theory atoms whose behaviour will be handled by a python program (using `clingo` python API) after solving. `show_trace` annotations are translated into traditional rules to which a prefix `show_all_` is added in the head of the rule. Listings 6 and 7 show a phase 1 translation example for each type or annotation. As it can be seen, `trace_rule` annotations are translated into a `theory atom` in the body of the associated rule, in order to keep this association during the rest of the process. Otherwise, `trace` annotations are translated into independent rules with a `&trace_all` theory atom in the head. The body of the rule contains the condition of the `trace` annotation and also the atom itself, for ensuring safety. The same is done for the body of `show_trace` annotations.

After phase 2, each rule in the program (except `show_all_` and `&trace_all` rules) is splitted into two different rules: one with a `fired_` prefix and another with a `holds_` prefix.

```
1 %!trace_rule {"The relay is working at %",J}
2   c(relay,on,J)   :- h(s1,closed,J), h(ab(relay),false,J), time(J).
3
4 %!trace {"Hypothesis: there has been a power surge at %",J} o(surge,J).
5
6 %!show_trace h(light,V,J) : o(close(s1),T), J=T+1.
```

Listing 6. Fragment f4.

```
1 c(relay,on,J) :- h(s1,closed,J), h(ab(relay),false,J), time(J), &trace{"The
      relay is working at %",J}.
2
3 &trace_all{o(surge,J),"Hypothesis: there has been a power surge at %",J : }
      :- o(surge,J).
4
5 show_all_h(light,V,J):-h(light,V,J),o(close(s1),T), J=T+1.
```

Listing 7. Fragment f4 after phase 1 translation.

Fired rules keep the original body of the rule, and are used for having a record of which specific rules have been activated. Holds rules are used for keeping the behaviour that functions would have in the original program. Listings 8, 9 and 10 show the translation process of Program P_2 phase by phase. We will use them for explaining some details about the phase 2 translation.

```
1  step(0..4).
2
3  obs_o(close(s1),1).
4  o(A,I) :- obs_o(A,I), step(I).
5
6  %!trace_rule {"The agent has closed switch s1 at %",I}
7     c(s1,closed,I) :- o(close(s1),I), step(I).
8
9  %!trace_rule {"Switch s1 was opened automatically at %",I+3}
10    c(s1, open, I+3) :- o(close(s1), I), not o(open(s1), J), J < I+3, step(I
          ), step(J), step(I+3).
```

Listing 8. Program P_2

Each rule from the original program receives a unique numerical identifier which is used with the prefix `fired_` (Listing 10, ls. 2,6,10,14,19) in its fired rule. Fired rules also carry all the variables used in the body as argument in the head (even if the original rule does not do it) (Listing 10, line 19). When retrieving the `fired_` atoms after solving, the fired values of that extra variables will be used to know which specific atoms triggered the rule body. The original heads are kept in a different rule with a `holds_` prefix. These rules are responsible for maintaining the behaviour of the original program (Listing 10, ls. 3,7,11,15,17). Since the original heads have this `holds_` prefix in the translation, all the literals in the fired rule bodies also use that prefix. The body of the holds rules just contains a `fired_` literal, in order to avoid redundancy while getting the same results (Listing 10 ls. 3,7,11,15,20).

The trace labels used (Listing 8, ls. 6, 9) have been first translated into theory atoms in the body of its rules after phase 1 (Listing 9, l. 4,5) and finally into independent rules

```
1  step(0..4).
2  obs_o(close(s1),1).
3  o(A,I) :- obs_o(A,I), step(I).
4  c(s1,closed,I) :- o(close(s1),I), step(I), &trace{"The agent has closed
       switch s1 at %",I}.
5  c(s1, open, I+3) :- o(close(s1), I), not o(open(s1), J), J < I+3, step(I),
       step(J), step(I+3), &trace{"Switch s1 was opened automatically at %",I
       +3}.
```

Listing 9. Program P_2 after phase 1

```
1  %step((0..4)).
2  fired_0((0..4)).
3  holds_step(Aux0) :- fired_0(Aux0).
4
5  %obs_o(close(s1),1).
6  fired_1(close(s1),1).
7  holds_obs_o(Aux0,Aux1) :- fired_1(Aux0,Aux1).
8
9  %o(A,I) :- obs_o(A,I); step(I).
10 fired_2(A,I) :- holds_obs_o(A,I),holds_step(I).
11 holds_o(Aux0,Aux1) :- fired_2(Aux0,Aux1).
12
13 %c(s1,closed,I) :- o(close(s1),I); step(I); &trace { "The agent has closed
       switch s1 at %",I :  }.
14 fired_3(s1,closed,I) :- holds_o(close(s1),I),holds_step(I).
15 holds_c(Aux0,Aux1,Aux2) :- fired_3(Aux0,Aux1,Aux2).
16 &trace{3,c(s1,closed,I),"The agent has closed switch s1 at %",I : } :-
       fired_3(s1,closed,I).
17
18 %c(s1,open,(I+3)) :- o(close(s1),I); not o(open(s1),J); J<(I+3); step(I);
       step(J); step((I+3)); &trace { "Switch s1 was opened automatically at
       %",(I + 3) :  }.
19 fired_4(s1,open,(I+3),I,J) :- holds_o(close(s1),I),not holds_o(open(s1),J),
       J<(I+3),holds_step(I),holds_step(J),holds_step((I+3)).
20 holds_c(Aux0,Aux1,Aux2) :- fired_4(Aux0,Aux1,Aux2,Aux3,Aux4).
21 &trace{4,c(s1,open,(I+3)),"Switch s1 was opened automatically at %",(I + 3)
       : } :- fired_4(s1,open,(I+3),I,J).
```

Listing 10. Program P_2 after phase 2

after phase 2 (Listing 10, l. 16,21). The body of the `&trace` rules is also formed by a `fired_` literal, for ensuring it only be triggered only when that specific rule does. Also, the theory atom carries the associated rule id as its first argument.

During this translation process, some information is stored for further processing for each translated rule: (1) the original head of the rule; (2) the original body of the rule; and (3) a list with all the variable names used in the `fired_` rule. This information will be used to reconstruct the derivation proof of the atoms after solving.

5 Construction of explanations

In order to generate the explanations, some pre-processing is needed. First, the theory atoms `&trace` and `&trace_all` are retrieved from the models of the program, and the `%` placeholders are replaced by its actual values. Once processed, the trace labels are stored into a dictionary, where they can be retrieved either by their associated `fired_id` or by

their atom. Then, we identify which rules have been fired for each model by finding the `fired_` atoms in it. For each fired rule, we save the different sets of values the rule was fired with in a dictionary indexed by `fired_id`. With this information, together with the information about the original rules that was stored during translation (the original head, the original body, and all the variable names) we build the derivation proof and, as a consequence, the explanations. To this aim, the first step made by `xclingo` is to merge all this information into a *causes table*, that will be used later to build and print the explanation trees. Of course, this is done for each different answer set. As an example, we can check the causes table of Program P_3 (Listings 11, 12 and 13).

```
step(0). step(1).
time(0). time(1).

o(close(s1),0).
h(ab(relay),false,0).
h(ab(bulb),false,0).

h(F,V,J) :- c(F,V,J).

%!trace_rule {"The agent has closed switch s1 at %",I}
    c(s1,closed,I) :- o(close(s1),I), step(I).

%!trace_rule {"The relay is working at %",J}
    c(relay,on,J)   :- h(s1,closed,J), h(ab(relay),false,J), time(J).

%!trace_rule {"Switch s2 is closed at %",J}
    c(s2,closed,J)  :- h(relay,on,J), time(J).

%!trace_rule {"The light is on at %",J}
    c(light,on,J)    :- h(s2,closed,J), h(ab(bulb),false,J), time(J).
```

Listing 11. Program p3

```
id  fired_head            trace labels                              fired_body
-------------------------------------------------------------------------------------------------
0   step(0)               []                                        []
1   step(1)               []                                        []
2   time(0)               []                                        []
3   time(1)               []                                        []
4   o(close(s1),0)        []                                        []
5   h(ab(relay),false,0)  []                                        []
6   h(ab(bulb),false,0)   []                                        []
7   h(s1,closed,0)        []                                        [c(s1,closed,0)]
7   h(relay,on,0)         []                                        [c(relay,on,0)]
7   h(s2,closed,0)        []                                        [c(s2,closed,0)]
7   h(light,on,0)         []                                        [c(light,on,0)]
8   c(s1,closed,0)        ["The agent has closed switch s1 at 0"]   [o(close(s1),0), step(0)]
9   c(relay,on,0)         ["The relay is working at 0"]             [h(s1,closed,0),h(ab(relay),
                                                                     false,0), time(0)]
10  c(s2,closed,0)        ["Switch s2 is closed at 0"]              [h(relay,on,0), time(0)]
11  c(light,on,0)         ["The light is on at 0"]                  [h(s2,closed,0), h(ab(bulb),
                                                                     false,0), time(0)]
```

Listing 12. Cause table for p3.

The *causes table* of a model will have a row for each different set of values fired for each individual rule in the program. Each row needs two steps to be built: (1) to replace the variables used in the original head and body by its fired values; and (2) to find the trace labels that are associated with the row by its id or its `fired_` head. Once the `causes`

```
>> c(light,on,0)        [1]
  *
  |__"The light is on at 0"
  |  |__"Switch s2 is closed at 0"
  |  |  |__"The relay is working at 0"
  |  |  |  |__"The agent has closed switch s1 at 0"
```

Listing 13. Explanation of `c(light,on,0)` from the cause table for `p3`.

`table` is obtained, the next step is to figure out which atoms are being requested to be explained, that is, the atoms affected by `show_trace` annotations. All the atoms in the model that start with the `show_all_` prefix are retrieved and stored in a list after removing the prefix. If that list is empty, then all the atoms in the model will be explained. Finally, the explanation of each atom in that list has to be built and printed. The Listing 14 shows the pseudocode for the recursive function that builds the explanation graph for a given atom and a given *causes table*. Since they are tree graphs, each explanation is a python dictionary where keys are trace labels and values are another dictionary (a subtree). Since an atom can have multiple explanations, the function returns a list of dictionaries. For a leaf of the tree graph (a fact), the result would be a list with an empty dictionary (Line 12).

```
1  def build_explanations(atom, causes_table, stack):
2      for row in causes_table.find_by_fired_head(atom):
3          if not_empty(row['f_body']):
4              entry_expls = []  # Explanations of the current row.
5              for atom in row['f_body']:
6                  if atom not in stack:
7                      stack.push(a)
8                      atom_expls = build_explanations(a, causes, stack)
9                      stack.pop(a)
10                     entry_expls = _combine(entry_expls, a_expls)
11         else:
12             entry_expls = [{}]
13
14         if not_empty(row['traces']):
15             explanations = []
16             for t in row['traces']:
17                 for e in entry_expls:
18                     explanations.append({t: e})
19         else:  # skip nodes without trace labels
20             explanations = entry_expls
21
22     return explanations
```

Listing 14. `build_explanations` pseudocode

While exploring the explanations of an atom A, we can find that one atom a in its fired body has multiple explanation. In that case, A will have a different explanation for each a explanation. Function `_combine` in line 10, performs this combination. Lines 14 to 21 manage the trace labels, which are the root of the tree graphs. The atom will have one

explanation for each different trace label it is associated with. If an atom has no trace labels, nothing is added as root of the explanations (lines 20-21). As a consequence, one level will be skipped in that subtree.

6 Related Work

The justifications of `xclingo` are directly based on the *causal justifications* from (Cabalar et al. 2014). The `xclingo` explanations constitute a partial implementation (still under development) of that causal extension of ASP. In particular, although `xclingo` performs some simplifications, it does not guarantee that the explanations obtained are minimal as the ones obtained by subsumption properties in (Cabalar et al. 2014).

In the following survey (Fandinno and Schulz 2019), different approaches to the problem of "answering the why" in ASP are reviewed and their similarities and differences are discussed. In addition to causal justifications, also *off-line justifications* (Pontelli et al. 2009) and *argumentative explanations* (Schulz and Toni 2016) are reviewed. An important difference of the causal orientation followed by `xclingo` from these approaches is that, in the former, no information is derived from negative literals. This is because default negation is understood, under this semantics, as the absence of cause. As a result, explanations only contain information that break the default, becoming shorter but, usually, more relevant. Another difference is that, thanks to the labelling system inherited from causal justifications, `xclingo` allows selecting in a fine grained way which information can be used in the justification trees.

7 Conclusions and Future Work

We presented `xclingo`, an ASP extension interpreter that allows obtaining justifications of the literals in the answer sets of logic programs. At the moment, the tool is a partial implementation of the Logic Programming extension described by *causal justifications*. The source code of xclingo is available in github[1] as a python script. It requires python 3.* and the following python modules to be installed: clingo[2], pandas[3] and more_itertools[4]. Its usage is described in the previously mentioned github page.

We feel `xclingo`'s explanation capabilities are very useful for obtaining readable explanations, that can even be expressed in natural language. Even if explanations are not needed, `xclingo` can be very handy if used as a complement with `clingo` for debugging purposes, since the additions of the syntax did not make the code incompatible. We have also illustrated how to use `xclingo` explanations in combination with a diagnostic reasoning example.

The first version of `xclingo` is still a prototype that will be augmented with new features. Immediate future work includes the treatment of minimization directives (something essential, for instance, to obtain minimal solutions for diagnosis problems), and other usual `clingo` constructs like *choice rules* or *pooling*, which are not currently accepted yet.

[1] `https://github.com/bramucas/xclingo`
[2] `https://potassco.org/clingo/python-api/5.4/`
[3] `https://pandas.pydata.org/`
[4] `https://github.com/more-itertools/more-itertools`

Another extension we plan to include in the future is the addition of trace labels for constraints so that, when specified by the user, these constraints may be transformed into their weak version. In that way, we will be able to obtain traces for those constraints that are relevantly involved in the absence of answer sets for a given program. Other extensions for the long term include the use of causal literals as in (Fandinno 2016) or the use of other types of annotations in addition to trace and show annotations. One idea is to add a new annotation for asserting or verifying conditions over rules and set of rules. The verification would be done in a similar way than in (Lifschitz et al. 2019), where the condition to assert is first translated into a finite set of first-order sentences that are then verified using an external theorem prover.

References

Balduccini, M. and Gelfond, M. 2003. Diagnostic reasoning with a-prolog. *CoRR cs.AI/0312040*.

Brewka, G., Eiter, T., and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

Cabalar, P., Fandinno, J., and Fink, M. 2014. Causal graph justifications of logic programs. *Theory and Practice of Logic Programming TPLP 14,* 4-5, 603–618.

Fandinno, J. 2016. Deriving conclusions from non-monotonic cause-effect relations. *Theory Pract. Log. Program. 16,* 5-6, 670–687.

Fandinno, J. and Schulz, C. 2019. Answering the "why" in answer set programming - A survey of explanation approaches. *Theory Pract. Log. Program. 19,* 2.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Wanko, P. 2016. Theory solving made easy with clingo 5. In *Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16)*, M. Carro and A. King, Eds. OpenAccess Series in Informatics (OASIcs), vol. 52. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2:1–2:15.

Lifschitz, V., Lühne, P., and Schaub, T. 2019. Verifying strong equivalence of programs in the input language of gringo. 270–283.

Marek, V. W. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm*, K. R. Apt, V. W. Marek, M. Truszczyński, and D. Warren, Eds. Artificial Intelligence. Springer Berlin Heidelberg, 375–398.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25,* 3-4, 241–273.

Pontelli, E., Son, T. C., and El-Khatib, O. 2009. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming TPLP 9,* 1.

Schulz, C. and Toni, F. 2016. Justifying answer sets using argumentation. *Theory and Practice of Logic Programming TPLP 16,* 1, 59–110.