

# A System for Explainable Answer Set Programming\*

Pedro Cabalar

University of Corunna, Spain

`cabalar@udc.es`

Jorge Fandinno

University of Potsdam, Germany

`fandinno@uni-potsdam.de`

Brais Muñiz\*

University of Corunna, Spain

`brais.mcastro@udc.es`

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) [13, 12, 4] 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 robotics, bioinformatics, music composition [7, 5, 3], 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. There exist several approaches for obtaining justifications for answer sets: for a recent review, see [9]. 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 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` [10]. The input accepted by `xclingo` is a markup language that introduces

---

\*Partially supported by MINECO, Spain, grant TIN2017-84453-P and CITIC Research Center, Xunta de Galicia, Spain and ERDF (ED431G 2019/01). The second author is funded by the Alexander von Humboldt Foundation, Germany.

annotations through program comments, specifying which rules or atoms must be traced. Using these annotations, `xclingo` generates derivation trees following the framework of *causal graph justifications* introduced in [6]. Under that framework, answer sets are multi-valued interpretations where the value for each true atom is an algebraic expression constructed with labels associated to program rules. Each expression is an alternative of multiple derivation trees that have been proved to correspond to the set of minimal proofs for the atom built with the Horn clauses in the program reduct. By now, `xclingo` just guarantees correctness of the obtained derivations, but not their minimality, which is planned to become a future optional feature. 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 of the explanations 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.

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 use a diagnosis problem example from the literature as a guide. We choose 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. Next, we describe how the translation of the input into a standard ASP program works. Afterwards, we describe how the explanations are computed from the solution of the translated program. Finally, we comment about related work and we conclude the paper.

## 2 Motivating Example

We consider an example from [2] (Fig. 1). In the example, an analog AC circuit is presented. In it, an agent can close a switch that should ultimately cause a bulb to turn on. However, there are 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.

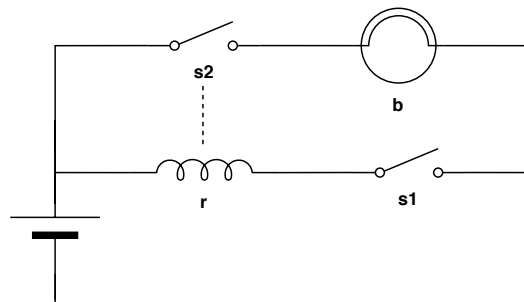


Figure 1: A circuit with a bulb  $b$ , a relay  $r$  and two switches,  $s1$  and  $s2$ .

**Example 1 (From Balduccini and Geldfond 2003)** Consider a system  $S$  consisting of an agent operating an analog circuit AC from Fig. 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<sup>1</sup> 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) follows the one presented in [2] with the addition of  $c/3$  and few other predicates for improving the explanation results. At a first sight, the encoding may seem too involved for our small example, but this is because the representation is general enough to cover a whole family of similar diagnosis problems. 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  $relay$ ,  $light$ ,  $s_1$  and  $s_2$  can take values  $on$  and  $off$  (to make them more readable) whereas the rest of fluents are Boolean.

```

1  plength(1).
2  time(0..L) :- plength(L).
3  step(1..L) :- plength(L).
4
5  switch(s1). switch(s2).
6  component(relay). component(bulb).
7
8  fluent(relay).
9  fluent(light).
10 fluent(b_prot).
11 fluent(S):-switch(S).
12 abfluent(ab(C)) :- component(C).
13 fluent(F) :- abfluent(F).
14
15 value(relay,on). value(relay,off).
16 value(light,on). value(light,off).
17 value(S,open) :- switch(S).
18 value(S,closed) :- switch(S).
19 hasvalue(F) :- value(F,V).
20 % Fluents are boolean by default
21 domain(F,true) :- fluent(F), not hasvalue(F).
22 domain(F,false) :- fluent(F), not hasvalue(F).
23 % otherwise, they take the specified values
24 domain(F,V) :- value(F,V).
25
26
27 agent(close(s1)).
28 exog(break).
29 exog(surge).
30 action(Y):-exog(Y).
31 action(Y):-agent(Y).

```

Listing 1: Type predicates for program  $P_1$ .

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

```

1 % Inertia
2 h(F,V,I) :- h(F,V,I-1), not c(F,I), step(I).
3
4 % Axioms for caused
5 h(F,V,J) :- c(F,V,J).
6 c(F,J) :- c(F,V,J).
7
8 % Direct effects
9 c(s1,closed,I) :- o(close(s1),I), step(I).
10
11 % Indirect effects
12 c(relay,on,J) :- h(s1,closed,J), h(ab(relay),false,J), time(J).
13 c(relay,off,J) :- h(s1,open,J), time(J).
14 c(relay,off,J) :- h(ab(relay),true,J), time(J).
15 c(s2,closed,J) :- h(relay,on,J), time(J).
16 c(light,on,J) :- h(s2,closed,J), h(ab(bulb),false,J), time(J).
17 c(light,off,J) :- h(s2,open,J), time(J).
18 c(light,off,J) :- h(ab(bulb),true,J), time(J).
19
20 % Malfunctioning
21 c(ab(bulb),true,I) :- o(break,I), step(I).
22 c(ab(relay),true,I) :- o(surge,I), step(I).
23 c(ab(bulb),true,I) :- o(surge,I), not h(b_prot,true,I-1), step(I).
24
25
26 % Executability
27 :- o(close(S),I), h(S,closed,I-1), step(I).
28
29 % Something happening actually occurs
30 o(A,I) :- obs_o(A,I), step(I).
31
32 % Check that observations hold
33 :- obs_h(F,V,J), not h(F,V,J).
34
35 % Completing the initial state
36 h(F,V,0) :- domain(F,V), not -h(F,V,0).
37 -h(F,V,0) :- h(F,W,0), domain(F,V), W!=V.
38
39
40 % A history
41 obs_h(s1,open,0).
42 obs_h(s2,open,0).
43 obs_h(b_prot,true,0).
44 obs_h(ab(bulb),false,0).
45 obs_h(ab(relay),false,0).
46 obs_o(close(s1),1).
47
48 % Something went wrong
49 obs_h(light,off,1).
50
51 % Diagnostic module: generate exogenous actions
52 o(Z,I) :- step(I), exog(Z), not no(Z,I).
53 no(Z,I) :- step(I), exog(Z), not o(Z,I).
  
```

Listing 2: Program  $P_1$  describing Example 1.

As usual in diagnosis problems, we differentiate between what happens in the real world, with predicates  $h/3$  and  $o/2$ , and the partial observations we have about that world, with predicates  $obs\_h/3$  and  $obs\_o/2$ , respectively. If we execute `clingo` on this code, we obtain the three answer sets that correspond to the possible diagnosis: one including an exogenous action  $o(break, 1)$ ; a second one with

the exogenous action  $o(\text{surge}, 1)$ ; 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 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 use 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, Fig. 2 shows the result we obtain for program  $P_1$  encoding from 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 act together) and each time we jump into a lower level, we can read this as a “because” relation.

Fig. 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 that 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 (Fig. 2, lines 9–12) is that the bulb was damaged because something broke it. The three lines in the explanation respectively come from the annotations (we will see later) for lines 18, 21 and 52 in Listing 2. The second cause (lines 14–16) is that the light was already off in the initial state because switch `s2` was initially open: in this case, because of the activation of rules in lines 17 and 5 in Listing 2.

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.

Let us proceed now to describe how these explanations are generated, including the markup language accepted by `xclingo`. Each explanation is a tree that shows the derivation proof for an atom. In these trees, 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, Fig. 3 displays the explanation for  $h(\text{light}, \text{off}, 1)$  under the “auto-tracing” mode, where every rule is automatically traced using the rule head as a label. As we can see, we obtain a complete derivation tree for  $h(\text{light}, \text{off}, 1)$  following the positive part of the program. Note the difference with respect to Fig. 2, where we used manually defined (textual) trace labels. In that case, the tool

```

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"

```

Figure 2: Explanations obtained for the annotated version of  $P_1$ .

```

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)

```

Figure 3: Explanation using automatically generated trace labels.

skips any intermediate node in the derivation tree that has no explicit trace label. In this way, knowledge engineers may decide the detail to be shown: either automatically tracing all possible rule applications, which may be helpful for debugging, or selecting the relevant information, something more interesting for explanation. In the latter, the explanation design, that is, selecting the right amount of information, may become a non-trivial Knowledge Representation effort in itself, but would easily become a problem instead if the tool did not offer such a possibility. For instance, one important decision is to avoid tracing the inertia rule (line 2 of Listing 2). In that way, if we ask for the explanation of `h(light,off,20)` in answer set 2 of Fig. 2 we still get the same derivation tree (replacing time stamp 1 by 20) because the switch was initially off and *nothing else changed that* in the whole interval. Another important feature that helps avoiding irrelevant information is that a negative literal `not p` is never used in derivations, since it is understood as “there is no cause for `p`.” If this was not done in this way, then the explanation for `h(light,off,20)` in an answer set where no real action occurred would include the negation of *all combinations of actions* that could have changed the light value along the way from 1 to 20 (and there are too many, even in this simple example). This information may be relevant for answering why was not the light turned on<sup>1</sup>, but is irrelevant for explaining why it has simply remained off, which is the purpose of `xclingo`.

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 `%!trace_rule`, 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 bulb 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: Adding trace labels to specific rules with `trace_rule`.

<sup>1</sup>Including “why not” queries is an interesting topic for future study.

The `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 makes the code larger, redundant and harder to maintain. For those cases, `trace` annotations are more suitable instead: they allow a permanent association of a label to an atom, regardless of which rules have triggered it. Thus, their information is less specific but allows other general interesting features. For instance, `trace` annotations can be stored separately from the base code, multiple versions of the same trace labels could be written depending on the context: different languages, different users or different detail level. Lines 1–8 in Listing 4 show the `trace` annotations added for obtaining the output from Fig. 2. For the part between braces, the syntax works the same as in `trace_rule` annotations, but instead of being followed by a rule, they are followed by a *conditional atom* defining the set of atoms affected by the trace label. Finally, the `show_trace` annotations (Lines 10–11 in Listing 4) work in a similar way to the `#show` directives in `clingo`, choosing which atoms are displayed in each answer set, but in this case, asking for their explanation. Again, `show_trace` annotations allow conditional atoms, as happened with `trace`.

```

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).
9
10 %!show_trace h(light,V,1).
11 %!show_trace h(relay,V,1).

```

Listing 4: Tracing atoms through `trace` annotations for Program  $P_1$ .

## 4 Implementation

The tool `xclingo` performs two main tasks: (1) a *translation* of the annotated program  $P$  into a logic program  $P'$ ; and (2) a *construction* of derivation trees by decoding the answer sets of  $P'$ . Program  $P'$  built in the translation phase is equivalent to the non-annotated version of  $P$  but includes auxiliary predicates and `clingo` theory atoms to keep track of the rules that have been fired. The translation is further divided into two steps. In a first step, `trace_rule` and `trace` annotations become `clingo` theory atoms without much transformation. These atoms are accepted by the grounder and can be handled after solving through the `clingo` Python API. The `show_trace` annotations are just transformed into traditional rules for an auxiliary head predicate `show_all_p` for each predicate  $p$  to be shown.

Listing 5 shows the result of this first step when applied to some of the annotations presented before.

In a second step, each predicate of the original program  $P$  is prefixed with `holds_` and it is assigned a numeric identifier  $N$ . After that, each rule of the form  $H :- B, \&trace\{label\}$  is split into



```

1 %% Translation of lines 2,3 in Listing 3
2 c(relay,on,J) :- h(s1,closed,J), h(ab(relay),false,J), time(J),
3                 &trace{"The relay is working at %",J}.
4
5 %% Translation of line 1 in Listing 4
6 &trace_all{o(surge,J),"Hypothesis: there has been a power surge at %",J : } :- o(surge,J)
7
8 %% Translation of line 10 in Listing 4
9 show_all_h(light,V,J):-h(light,V,J).

```

Listing 5: Transforming annotations into theory atoms and auxiliary predicates.

the following rules:

$$\begin{aligned} \text{fired}_N(X_1, \dots, X_n) &:- B \\ H &:- \text{fired}_N(X_1, \dots, X_n) \\ \&\text{trace}\{N, H, \text{label}\} &:- \text{fired}_N(X_1, \dots, X_n) \end{aligned}$$

where the  $X_1, \dots, X_n$  also include the free variables in the body  $B$ . If the original rule body does not contain any trace label, then the last rule is not generated. As an example, suppose that rule in lines 2–3 from Listing 5 is assigned identifier 33. Then, its translation is shown in Listing 6.

```

1 fired_33(relay,on,J) :-
2   holds_h(s1,closed,J), holds_h(ab(relay),false,J), holds_time(J).
3 holds_c(Aux0,Aux1,Aux2) :-
4   fired_33(Aux0,Aux1,Aux2).
5 &trace{33,c(relay,on,J),"The relay is working at %",J} :-
6   fired_33(relay,on,J).

```

Listing 6: Translation of lines 2–3 from Listing 5.

During this translation process, some additional rule information is stored: (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 is used to reconstruct the derivation proof of the atoms after solving.

Once the translated program  $P'$  is generated, `xclingo` makes a call to `clingo`'s `solve` function to retrieve its answer sets and, for each one, proceeds to construct the explanations from the information retrieved in the answer set. To do so, the first step consists in collecting all the theory atoms `&trace` and `&trace_all` and replacing the `%` placeholders in strings by their 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, `xclingo` identifies 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 proofs and print the explanations. The construction of the derivation proof is made with a “*causes table*” that has a row per each rule and includes, the trace labels and the bodies that fired those labels. Once the `causes table` is obtained, the next step is filtering those 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 are explained. Finally, the explanation of each atom in that list has to be built and printed. Listing 7 shows the pseudocode for the recursive function that builds the explanation graph for a given atom and a given

causes table. According to the tree structure, each explanation is a python dictionary where keys are trace labels and values are in 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 is a list with an empty dictionary (Line 12). While exploring the explanations of an atom  $A$ , we can find that one atom  $a$  in its

```

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 a in row['f_body']:
6                 if atom a 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 7: Pseudocode for the build\_explanations function.

fired body has multiple explanations. In that case,  $A$  has 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 has 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 is skipped in that subtree.

Given that atoms may have multiple alternative explanations, the size of the set of explanations, when expressed as a set of trees, can grow exponentially in the worst case. To see why, just consider the program in Listing 8 for some constant value  $n$ . Here, the explanation for atom  $p(n)$  is a chain formed by  $n$  labels "a( $n$ )", ..., "a(1)". If we add a second trace for  $p(X)$  using "b( $\%$ )", then each atom  $p(X)$  has now two alternative labels a( $X$ ) and b( $X$ ), and so, the number of alternative derivations for  $p(n)$  becomes  $2^n$ .

```

1 p(1).
2 p(X+1) :- p(X), X<=n.
3 %!trace {"a(%)" , X} p(X).

```

Listing 8: A sequential chain for predicate  $p$ .

We have tested the performance of `xclingo` on a simple encoding of the well-known *blocks world* domain varying the size of the scenario in terms of the number of blocks, but also the size of the expla-

```

1 >> unclear(9,14)           [1]
2 *
3 |__"Block 9 is finally unclear"
4 |  |__"Block 1 is finally on 9"
5 |  |  |__"Block 1 was moved on top of block 9 at t=11"

```

Figure 4: Explanation answering why block 9 is unclear at final step 14.

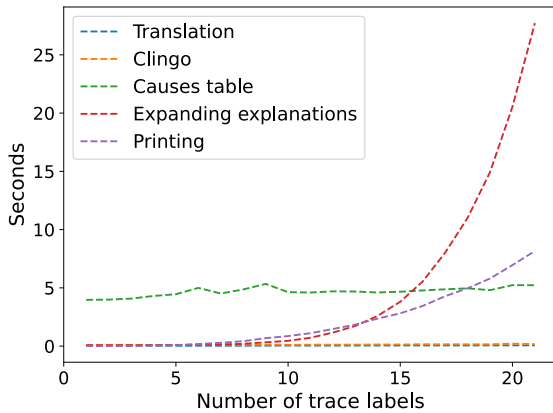


Figure 5: Execution time vs labels per atom.

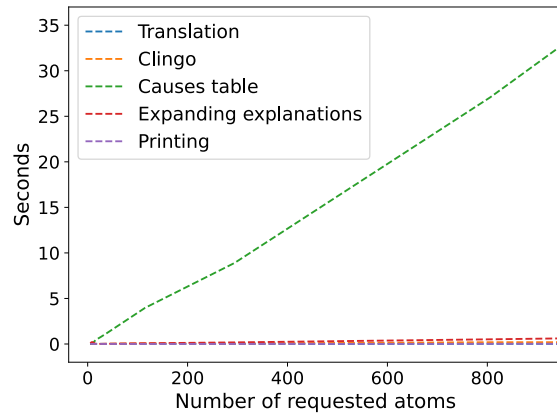


Figure 6: Execution time vs num. of atoms.

nations, by adding more trace labels per atom. We asked `xclingo` to explain the actions performed, the final location of each block and the final value of predicate `unclear(B)` that states whether a block `B` has anything on top (see example in Fig. 4). We measured the time spent in five different steps of the process: (1) the translation; (2) the execution of `clingo` for solving the planning problem; (3) the construction of the causes table and the dependencies among labels; (4) the expansion of all derivation trees; and finally, (5) their printing. Fig. 5 shows how the different times evolve when we fix the number of atoms requested, but we progressively add new trace labels per each atom. As we explained before, increasing the number of labels per atom causes an exponential grow in the number of alternative explanations that is well reflected in the graphic. As future work, we plan to include an execution mode for generating a maximum number of explanations per atom instead of expanding all of them. Fig. 6 shows how, when we fix the number of trace labels per atom, the time spent in the construction of the causes table and its dependencies has a linear increase as we add more atoms. We also plan to explore alternatives to the construction of the explanations like performing a tabled evaluation or even using `clingo` itself to solve this part of the problem.

## 5 Related Work

The explanations of `xclingo` correspond to *causal justifications* from [6] except that `xclingo` does not guarantee that the derivation trees are minimal, although it performs some minor simplifications. In the survey [9], different approaches to the problem of answering the “why” in ASP are reviewed, including the two related ones called *off-line justifications* [14] and *argumentative explanations* [15]. An important difference with respect to these approaches is that, as we explained, `xclingo` does not derive information

from negative literals. This is because default negation is understood, under `xclingo` semantics, as the absence of cause. As a result, explanations only provide information when defaults are broken (like inertia), rather than including all possible events that could have changed the outcome, although they did not happen. The latter can be interesting for answering “why not” queries, but the size of explanations may simply become unmanageable. 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.

Other systems that provide explanations for logic programs are `ErgoAI`<sup>2</sup>, for Datalog programs, and `s(ASP)` [1] for non-ground ASP programs. An advantage of these two systems is that their explanations can be generated without resorting to a full grounding of the program. Besides, `ErgoAI` shows some similarities to `xclingo` like selecting the information in the explanations (although, in this case, we must explicitly define which information must be omitted), or allowing text labels and variables in the justifications. However, both `ErgoAI` and `s(ASP)` include information about negated literals, unlike `xclingo`.

## 6 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* [6]. The python source code of `xclingo` and instructions for its usage are available at `github`<sup>3</sup>. It requires python 3.\* and the following python modules to be installed: `clingo`<sup>4</sup>, `pandas`<sup>5</sup> and `more_itertools`<sup>6</sup>. We illustrated, on a diagnostic reasoning example, how `xclingo`’s features can be used for obtaining readable explanations, that can be even expressed in natural language. Even if explanations are not needed, `xclingo` can also be used as a complement to `clingo` for debugging purposes, since the annotations are included as comments and do not make the code incompatible.

The current version of `xclingo` is still in a preliminary stage and 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 accepted yet. Another extension we plan to include in the future is the addition of trace labels for constraints, so that those that are labelled become weak constraints. Other extensions for the long term include the use of causal literals as in [8] or the use of verification annotations to be combined with a similar theorem proving technique as done in [11].

## References

- [1] Joaquín Arias, Manuel Carro, Elmer Salazar, Kyle Marple & Gopal Gupta (2018): *Constraint Answer Set Programming without Grounding*. *Theory and Practice of Logic Programming* 18(3-4), pp. 337–354, doi:10.1017/S1471068418000285.

<sup>2</sup><http://coherentknowledge.com/product-overview-ergoai-platform/>

<sup>3</sup><https://github.com/bramucas/xclingo>

<sup>4</sup><https://potassco.org/clingo/python-api/5.4/>

<sup>5</sup><https://pandas.pydata.org/>

<sup>6</sup><https://github.com/more-itertools/more-itertools>

- [2] Marcello Balduccini & Michael Gelfond (2003): *Diagnostic reasoning with A-Prolog*. *Theory and Practice of Logic Programming* 3(4-5), pp. 425–461, doi:10.1017/S1471068403001807.
- [3] Georg Boenn, Martin Brain, Marina De Vos & John ffitich (2010): *Automatic Music Composition using Answer Set Programming*. *Theory and Practice of Logic Programming* 11, doi:10.1017/S1471068410000530.
- [4] Gerhard Brewka, Thomas Eiter & Mirosław Truszczyński (2011): *Answer set programming at a glance*. *Commun. ACM* 54(12), pp. 92–103, doi:10.1145/2043174.2043195.
- [5] Daniel Brooks, Esra Erdem, Selim Erdoan, James Minett & Don Ringe (2007): *Inferring Phylogenetic Trees Using Answer Set Programming*. *Journal of Automated Reasoning* 39, pp. 471–511, doi:10.1007/s10817-007-9082-1.
- [6] Pedro Cabalar, Jorge Fandinno & Michael Fink (2014): *Causal Graph Justifications of Logic Programs*. *Theory and Practice of Logic Programming TPLP* 14(4-5), pp. 603–618, doi:10.1017/S1471068414000234.
- [7] Esra Erdem, Erdi Aker & Volkan Patoglu (2012): *Answer set programming for collaborative housekeeping robotics: Representation, reasoning, and execution*. *Intelligent Service Robotics* 5, doi:10.1007/s11370-012-0119-x.
- [8] Jorge Fandinno (2016): *Deriving conclusions from non-monotonic cause-effect relations*. *Theory Pract. Log. Program.* 16(5-6), pp. 670–687, doi:10.1017/S1471068416000466.
- [9] Jorge Fandinno & Claudia Schulz (2019): *Answering the “why” in answer set programming - A survey of explanation approaches*. *Theory Pract. Log. Program.* 19(2), doi:10.1017/S1471068418000534.
- [10] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub & P. Wanko (2016): *Theory Solving Made Easy with Clingo 5*. In M. Carro & A. King, editors: *Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP’16)*, *OpenAccess Series in Informatics (OASICs)* 52, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 2:1–2:15, doi:10.4230/OASICs.ICLP.2016.2.
- [11] Vladimir Lifschitz, Patrick Lhne & Torsten Schaub (2019): *Verifying Strong Equivalence of Programs in the Input Language of gringo*, pp. 270–283. doi:10.1007/978-3-030-20528-7\_20.
- [12] Victor W. Marek & Mirosław Truszczyński (1999): *Stable Models and an Alternative Logic Programming Paradigm*. In Krzysztof R. Apt, Victor W. Marek, Mirosław Truszczyński & DavidS Warren, editors: *The Logic Programming Paradigm*, Artificial Intelligence, Springer Berlin Heidelberg, pp. 375–398, doi:10.1007/978-3-642-60085-2\_17.
- [13] Ilkka Niemelä (1999): *Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm*. *Annals of Mathematics and Artificial Intelligence* 25(3-4), pp. 241–273, doi:10.1023/A:1018930122475.
- [14] Enrico Pontelli, Tran Cao Son & Omar El-Khatib (2009): *Justifications for logic programs under answer set semantics*. *Theory and Practice of Logic Programming TPLP* 9(1), doi:10.1017/S1471068408003633.
- [15] Claudia Schulz & Francesca Toni (2016): *Justifying answer sets using argumentation*. 16, pp. 59–110, doi:10.1017/S1471068414000702.