TPLP: Page 1–8. © The Author(s), 2021. Published by Cambridge University Press 2021
doi:10.1017/xxxxx

Commonsense Explanations for the Blocks World[*]

PEDRO CABALAR
University of Corunna

BRAIS MUÑIZ
University of Corunna

submitted xx xx xxxx; revised xx xx xxxx; accepted xx xx xxxx

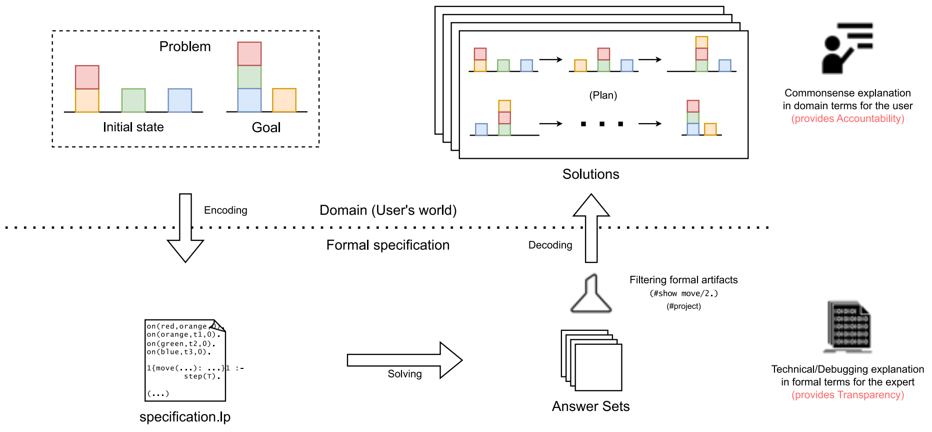
1 Introduction

Recently, the scale and complexity of Artificial Intelligence (AI) systems have increased, often at the cost of losing comprehensibility. This is true even for symbolic approaches like Answer Set Programming (ASP), where an interest in explainability research has grown and several tools have been recently proposed, including `s(CASP)` (by Arias et al. (2020)), `xASP` (by Alviano et al. (2023)) or *Justifying Answer Sets using Argumentation* (by SCHULZ and TONI (2015)), among many others. Consequently, ASP users have now access from the more technical, detailed explanations to the simplest, natural language-based ones.

In this work, we focus on a particular tool: `xclingo`. This tool, which has been developed by the authors, provides explanations for ASP programs that can be seen as a derivation proof (displayed as a tree) for an atom from a particular Answer Set. While the obtainment of technical explanations is very straightforward, displaying the real proof in plain text, it also expands the language with an annotation system that allows the user to customise the explanation. The user can decide which parts of the proof are shown and they can be replaced by meaningful natural language text instead of showing the raw atoms. Previous works on this tool have been already published, describing the theory underpinning the explanations obtained Cabalar and Muñiz (2023), and how the new version of the tools works and how is implemented Cabalar et al. (2020)

Figure 1 depicts the typical KR pipeline and emphasises how the different steps belong either to the domain's or user's world in the upper half or to the formal specification scope. Some of the previously mentioned tools focus on providing highly detailed explanations, debugging-oriented explanations that fail to meet the common user's expectations, often being too large and revealing formal underlying artefacts, to which users are oblivious. Moreover, we have found that just filtering these artefacts to only show the domain concepts of which users are aware is often not enough but how that information is displayed is also fundamental to actually meet their expectations, which could also be particular for specific users. Even with the substantial flexibility provided by `xclingo` for designing the final explanations, it is important to emphasise that, not any desirable user's world explanation is in fact achievable from an arbitrary specification. In fact, this extended

* Research partially funded by . . .



```
on(red,orange,0).
on(orange,t1,0).
on(green,t2,0).
on(blue,t3,0).

1{move(...): ...}1 :-
          step(T).

(...)
```

Fig. 1. Typical KR diagram. A problem is specified as a program written in some KR language and then solved. The relevant information within the solutions to the program is finally shown to the final user.

abstract's main aim is to discuss how, under this demanding and variable setting, a considerable effort to modify the original encoding is ultimately needed to obtain a particular explanation.

To argue this we will use `xclingo` to obtain explanations for the classical planning scenario block's world which will act as a running example. In the block's world problem, we consider several blocks which can be placed on the table or on top of other blocks. The goal is to transform a given initial arrangement of blocks into a desired configuration or order (Figure 2 shows a typical setup). A solution consists of a sequence of actions (or

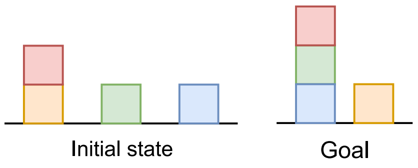


Fig. 2. Typical set up of a blocks world problem.

plan), each one associated with a time step, where one block is moved from one stack to another following two main constraints: (1) only blocks on the top of each stack can be moved; and (2) they only can be placed on the top of another stack.

In Section 2, we propose up to 4 different specifications for that domain, its differences and how they lead to different explanations for the user. Finally, Section 3 draws the conclusions for the discussion and underlines some takeaway messages on the topic.

2 Commonsense explanations versus debugging explanations

Let us briefly introduce how `xclingo` obtains explanations for an arbitrary ASP encoding. First, the original program is reified into a program that treats the head-to-body dependencies as cause-effect relations.

This way, any literal from the body of an *enabled* rule will act as causes of any atom derived from the head (the consequence). Using this reification, `xclingo` computes (one or more) acyclic graphs for each answer set of the original program. In this graph, each node represents any true atom within that answer set and the edges resemble the cause-effect relations between them. The obtainment of such graphs and their properties are detailed in Cabalar and Muñiz (2023). Through the use of the `xclingo` annotations, the user can customise which information is visible to the final user and the natural language text shown. For brevity, familiarity with the usage of each particular type of annotation is assumed from now on.

Our specification for the blocks world follows the conventions used for diagnosis in Balduccini and Gelfond (2003), where the predicate $h(F, V, T)$ indicates the value V of fluent F at time step T, and the predicate $o(A, T)$ indicates that the action A occurs at time step T.

```
1 #const last=3.
2 #const n=4.
3 h(on(2),table,0). h(on(3),table,0).
4 h(on(1),3,0).     h(on(4),1,0).
5 g(on(4),table).   g(on(1),4).
6 g(on(2),1).       g(on(3),table).
```

Listing 1. *Example of problem instance*

Listing 1 shows the representation of an example's problem instance. Lines 1 and 2 define constants `last` and `n`, which respectively represent the number of time steps and blocks. The initial state is provided (lines 2 and 3) as well as the goal state (lines 5 and 6).

```
1  time(0..last). step(1..last). block(1..n).
2  location(table). location(B):-block(B).
3
4  unclear(C,T) :- h(on(B),C,T),C!=table,time(T).
5  :- o(move(B,_),T), unclear(B,T-1).                  % executability
6  :- o(move(_,L),T), unclear(L,T-1).                  % executability
7  :- o(move(B,table),T), h(on(B),table,T-1), step(T).  % executability
8  :- g(on(B),L), not h(on(B),L,last).
9
10 h(F,V,T) :- h(F,V,T-1), not c(F,T), step(T).        % inertia
11 c(F,T)   :- h(F,V,T-1),h(F,W,T),V!=W, step(T).      % changes
12
13 % Text labels
14 timetext(0,initially). timetext(last,finally).
15 changed(on(B),T) :- h(on(B),L,T), h(on(B),L',T-1),L!=L'.
16 %!trace {h(on(B),L,T), "Block % is % on %",B,Txt,L} :- h(on(B),L,T), timetext(T,Txt
      ).
17 %!trace {h(on(B),L,T), "Block % is now on % at %",B,L,T} :- changed(on(B),T),h(on(B
      ),L,T), T!=last,T!=0.
18
19 %!show_trace {h(F,V,last)}.
20 #show o/2.
```

Listing 2. *Common blocks world code*

The specification used for obtaining the plans is broken into two parts. A static part, shown in Listing 2, that includes: (1) some domain predicates (lines 1 and 2); (2) the executability and goal constraints (lines 4-8); (3) an inertia rule (line 10); (4) some additional code to customise the explanations (lines 13-17); and finally (5), the `show` sentences for `xclingo` and clingo respectively (lines 19-20).

Secondly, a generative code for modelling the plan's actions. We propose 4 different encodings for the generative part and we discuss how they lead to different explanations, showing that the design of the code for obtaining a particular kind of explanation is, in general, not trivial. Every encoding will be fed with the same problem instance, to compare its results fairly. For brevity, we will only explain the state of one block.

```
1 action(move(B,table)):- block(B).
2 action(move(B,C)):- block(B),block(C),B!=C.
3
4 1 {o(A,T): action(A) } 1 :- step(T).
5 h(on(B),L,T) :- o(move(B,L),T).
6
7 %!trace {o(move(B,L),T), "We moved block % to % at step %",B,L,T} :- o(move(B,L),T)
      .
```

Listing 3. *Encoding 1*

Encoding 1 (Listing 3) corresponds to the straightforward planning implementation, where any action can be selected for each time step (line 4) and the state of moved blocks is captured at line 5. The annotation at line 7 labels each movement with a descriptive text. The explanations obtained by clingo for Encoding 1 can be seen in Listing 4

```
*
|__"Block 14 is finally on 13"
|  |__"Block 14 is now on 13 at 22"
|  |  |__"We moved block 14 to 13 at step 22"
```

Listing 4. *Explanations obtained for Encoding 1*

Contrary to what a common user would expect, the explanation for the final position of block 14 does not include the entire history of movements concerning the block, but rather only talks about the final step 22. To understand why this happens, let us recall that `xclingo` extracts the cause-effect relations from the rules in the program. Since there is no cause-effect connection between one state and its previous ones (neither directly nor transitively), any explanation about a state will never allude to any other time step. Note how the states just depend on the decided action (line 9), which, at last, only depends on domain predicates (lines 1-5).

While this result feels incorrect, it still corresponds to the most straightforward application of explainability to a planning problem, where we just run the tool with the well-known standard specification, without any further modification. To eventually obtain an explanation that includes the complete history of actions several modifications to the original program may be done. Let us now introduce encodings 2, 3 and 4 and discuss how they fulfil this goal in different ways.

Encoding 2 only introduces one change with respect to the original.

```
h(on(B),L,T) :- o(move(B,L),T), h(on(B),L',T-1).
```

Listing 5. *Encoding 2*

In Listing 5, we can see how the effect axiom (originally at line 5, Listing 3) now includes a new literal recalling the position of block B at the previous state $T-1$. Note how that previous state is not used to obtain the current step's position, but just introduces the transitive cause-effect relation needed to generate the desired explanations (as we can see in Listing 6.

```
*
|__"Block 14 is finally on 13"
|  |__"Block 14 is now on 13 at 22"
|  |  |__"We moved block 14 to 13 at step 22"
|  |  |__"Block 14 is now on table at 16"
|  |  |  |__"We moved block 14 to table at step 16"
|  |  |  |__"Block 14 is initially on 11"
```

Listing 6. *Explanations obtained for Encoding 2*

Note how *movement* and *state* causes in the explanations are grouped in the same indentation level, showing how they are joint causes for the next state of the block. However, if we take lines 3 to 5 as an example we can see how a movement at step 22 and the state achieved at step 16 act together as a joint cause despite the fact that the rule from Listing 5 explicitly recalls time step $T-1$ This effect in the explanations can be confusing and occurs because of the way `xclingo` works. The code only labels the states when the fluent value changes and non-labelled states are omitted by `xclingo` to obtain clearer explanations. However, the causes are still propagated through the steps transitively via inertia.

On the other hand, Encoding 3 fixes this confusing effect by introducing the same dependence but this time in the choice rule instead (see Listing 7), leaving the rest of the code untouched with respect to Encoding 1. Similarly to Encoding 2, the previous position is introduced as a cause in the body of the choice rule but not used for obtaining the new movement.

```
{o(move(B,L),T): location(L),L!=B} :- step(T), h(on(B),L',T-1).

:- o(move(B,_),T),o(move(C,_),T),B!=C.
```

Listing 7. *Encoding 3*

Therefore, in the obtained explanations at Listing 8, we can observe something similar to what was achieved with Encoding 2, but this time every consequent cause is in a new level of indentation.

```
  *
  |__"Block 14 is finally on 13"
  |   |__"Block 14 is now on 13 at 22"
  |   |   |__"We moved block 14 to 13 at step 22"
  |   |   |   |__"Block 14 is now on table at 13"
  |   |   |   |   |__"We moved block 14 to table at step 13"
  |   |   |   |   |   |__"Block 14 is initially on 11"
```

Listing 8. *Explanations obtained for Encoding 3*

Explanations obtained from Encoding 3 are finally closer to what a common user would find intuitive and expected. However, certain literals have been added to the program only to model the causal relationships and not to solve the problem, which feels unclean.

In final Encoding 4 (Listing 9), we use the literals introduced in Encoding 3 to actually make the action itself tell from where the block was picked.

```
1 % The action actually reflects From and To
2 {o(move(B,From,To),T): location(To),To!=B} :- step(T),h(on(B),From,T-1).
3
4 % Again, From is not used at all but is "recorded"
5 o(move(B,To),T) :- o(move(B,From,To),T).
6 :- o(move(B,_),T),o(move(C,_),T),B!=C.
7
8 %!trace {o(move(B,From,To),T), "We moved block % from % to % at step %",B,From,To,T
      } :- o(move(B,From,To),T).
```

Listing 9. *Encoding 4*

This introduces the literals into the code in a more organic way but can be also used to enrich the explanations. We also modified the annotation in line 8 (originally in line 7 in Listing 3 accordingly to tell the user from where the block was moved. This leads to the final explanations shown in Listing 10.

```
*
|__"Block 14 is finally on 13"
|  |__"Block 14 is now on 13 at 22"
|  |  |__"We moved block 14 from table to 13 at step 22"
|  |  |  |__"Block 14 is now on table at 12"
|  |  |  |  |__"We moved block 14 from 11 to table at step 12"
|  |  |  |  |  |__"Block 14 is initially on 11"
```

Listing 10. *Explanations obtained for Encoding 4*

3 Conclusions

Some of the explainability approaches offer large, fully detailed explanations that expose formal artefacts which, while can be right for debugging purposes, are far away from what feels right for explainable AI systems in the long term. Even when some of them offer tools to use natural language and to select which information to show, this can be insufficient to accomplish the user's needs. In this extended abstract, we have discussed how to obtain explanations that meet specific user criteria, a modification of the original specification can be needed. We have started with a naive specification which solves the block's world problem, but whose explanations do not include the complete plan. Then, we propose up to three different modifications that actually accomplish the goal but in different ways, showing that there's no trivial solution to this.

In our case, this happens because `xclingo` interprets the rules as causal-effect relations, and so, they lastly determine the aspect of the final explanations. In any case, though, this is an unavoidable problem in general for any tool whose explanations are based on the original ASP program. This leads us to two general conclusions.

On one hand, if the specification affects the structure of the explanations, we conclude that is important to at least have an idea of the user's expectations before starting working on the specification. On the other hand, introducing modifications may feel inelegant and could even conflict with the declarative nature of the language, but more importantly in practice, it can affect the efficiency, which is something that wasn't studied in this work for brevity.

References

Mario Alviano, Ly Ly Trieu, Tran Cao Son, and Marcello Balduccini. Explanations for answer set programming. *Electronic Proceedings in Theoretical Computer Science*, 385:27–40, sep 2023. doi: 10.4204/eptcs.385.4. URL `https://doi.org/10.4204%2Feptcs.385.4`.

Joaquín Arias, Manuel Carro, Zhuo Chen, and Gopal Gupta. Justifications for goal-directed constraint answer set programming. In *Intl. Conf. on Logic Programming, ICLP*, 2020.

Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3(4-5):425–461, 2003.

Pedro Cabalar and Brais Muñiz. Model explanation via support graphs, 2023.

Pedro Cabalar, Jorge Fandinno, and Brais Muñiz. A system for explainable answer set programming. In Francesco Ricca et al, editor, *Proc. 36th Intl. Conf. on Logic Programming (Technical Communications), UNICAL, Rende, Italy, 2020*, volume 325 of *EPTCS*, 2020.

CLAUDIA SCHULZ and FRANCESCA TONI. Justifying answer sets using argumentation. *Theory and Practice of Logic Programming*, 16(1):59–110, feb 2015. doi: 10.1017/s1471068414000702. URL `https://doi.org/10.1017%2Fs1471068414000702`.