“Add Another Blue Stack of the Same Height!”: ASP Based Planning and Plan Failure Analysis

Chitta Baral[1] and Tran Cao Son[2]

[1]Department of Computer Science and Engineering, Arizona State University, Tempe, AZ
[2]Department of Computer Science, New Mexico State University, Las Cruces, NM

Abstract. We discuss a challenge in developing intelligent agents (robots) that can collaborate with human in problem solving. Specifically, we consider situations in which a robot must use natural language in communicating with human and responding to the human's communication appropriately. In the process, we identify three main tasks. The first task requires the development of planners capable of dealing with *descriptive goals*. The second task, called *plan failure analysis*, demands the ability to analyze and determine the reason(s) why the planning system does not success. The third task focuses on the ability to understand communications via natural language. We show how the first two tasks can be accomplished in answer set programming.

1 Introduction

Human-Robot interaction is an important field where humans and robots collaborate to achieve tasks. Such interaction is needed, for example, in search and rescue scenarios where the human may direct the robot to do certain tasks and at times the robot may have to make its own plan. Although there has been many works on this topic, there has not been much research on interactive planning where the human and the robot collaborate in making plans. For such interactive planning, the human may communicate to the robot about some goals and the robot may make the plan, or when it is unable it may explain why it is unable and the human may make further suggestions to overcome the robot's problem and this interaction may continue until a plan is made and the robot executes it. The communication between the robot and the human happens in natural language as ordinary Search and Rescue officials may not be able to master the Robot's formal language to instruct it in situations of duress. Following is an abstract example of such an interactive planning using natural language communication.

Consider the block world domain in Figure 1 (see, [1]). The robot has its own blocks and the human has some blocks as well. The two share a table and some other blocks as well. Suppose that the human communicates to the robot the sentence “*Add another blue stack of the same height!*.” Even if we assume that the robot is able to recognize the color of the blocks, create, and execute plans for constructions of stack of blocks, such a communication presents several challenges to a robot. Specifically, it requires that the robot is capable of understanding natural language, i.e., it requires that the robot is able to identify that

- the human refers to stacks of only blue blocks (*blue stack*);
- the human refers to the height of a stack as the number of blocks on the stack;

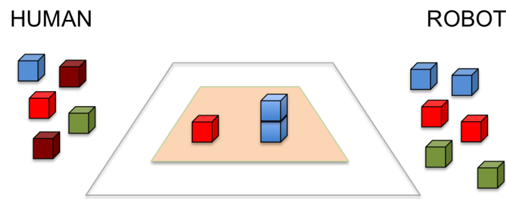

Fig. 1: *A Simplified Version of The Blocks World Example from the BAA*

– there is a blue stack of the height 2 on the table; and
– it should use its two blue blocks to build a new stack of two blue blocks.

It is easy to see that among the above points, the first three are clearly related to the research in natural language processing (NLP) and the last one to planning, i.e., an integration of NLP and planning is needed for solving this problem. Ideally, given the configuration in Figure 1 and the reasoning above, the robot should devise a plan to create a new stack using its two blue blocks and execute the plan. On the other hand, if the robot has only one blue block, the robot should realize that it cannot satisfy the goal indicated by the human and it needs to respond to the human differently, for example, by informing the human that it does not have enough blue blocks or by asking the human for permission to use his blue blocks.

As planning in various settings has been considered in answer set programming and there have been attempts to translate NLP into ASP, we would like to explore the use of ASP and related tools in solving this type of problems. *Our focus in this paper is the interactive planning problem that the robot needs to solve*.

2 ASP Planning for "Add Another Blue Stack of the Same Height!"

What are the possible responses to the communication from the human "*Add another blue stack of the same height!*" for the robot? Assume that the robot can recognize that it needs to create and execute a plan in responding to the communication. We next describe an ASP implementation of an ASP planning module for the robot. Following the literature (e.g., [2]), this module should consist of the planning domain and the planning problem.

2.1 The Planning Domain

The block world domain D_b can be typically encoded by an ASP-program as follows:

– blocks are specified by the predicate `blk(.)`;
– actions such as `putontable(X)` or `put(X,Y)`, where `X` and `Y` denote blocks, are defined using ASP-rules;
– properties of the world are described by fluents and encoded using predicates—similar to the use of predicates in Situation Calculus—whose last parameter represents the situation term or an time step in ASP. For simplicity, we use two fluents `on(X,Y,T)` and `ontable(X,T)` which denote `Y` is on `X` or `X` is on the table, respectively, at the time point `T`;

– Rules encoding executability conditions and effects of actions are also included, for example, to encode the executable condition and effects of the action `putontable(X)`, we use the the rules

```
executable(putontable(X),T):-time(T),action(putontable(X)),clear(X,T).
ontable(X,T+1):-time(T),occ(putontable(X),T).
ontable(X,T+1):-time(T),ontable(X,T),#count{Z:occ(put(Z,X),T)}==0.
```

D_b can be used for planning [2]. To use D_b in planning, we need the following components: (*i*) rules for generating action occurrences; (*ii*) rules encoding the initial state; and (*iii*) rules checking for goal satisfaction.

2.2 What Are We Planning For? and How Do We Plan For It?

Having defined D_b, we now need to specify the planning problem. The initial state can be easily seen from the figure. The question is then what is the goal of the planning problem. Instead of specifying a formula that should be satisfied in the goal state, the sentence describes a desirable state. This desirable state has two properties: (*i*) it needs to have the second (or a new) blue stack; and (*ii*) the new blue stack has the same height as the existing blue stack. In other words, the goal is *descriptive*. We will call a planning problem whose goal is descriptive as a *descriptive planning problem*.

In the rest of this subsection, we will present an encoding of the planning problem for "*Add another blue stack of the same height!*." First, we need to represent the goal. Considering that a stack can be represented by the block at the top, the goal for the problem can be represented by a set of goal conditions in ASP as follows.

```
g_cond(S,is,stack):-blk(S).    g_cond(S,type,another):-blk(S).
g_cond(S,color,blue):-blk(S). g_cond(S,height,same):-blk(S).
```

These rules state that the goal is to build a stack represented by `S`, the stack is blue, has the same height, and it is a new stack. Note that these rules are still vague in the sense that they contain unspecified concepts, e.g., "*what is a stack?*," "*what does it mean to be the same height?*," etc.

The problem can be solved by adding rules to the block program D_b to define when a goal condition is satisfied. Afterwards, we need to make sure that all goal conditions are satisfied. As we only need to obtain one new stack, the following rules do that:

```
not_sat_goal(S,T):-blk(S),g_cond(X,Y,Z), not satisfied(X,Y,Z,T).
sat_goal(S, T)    :-not not_sat_goal(T).
:- X = #count {S  : sat_goal(S, n)}, X ==0.
```

The first two rules define blocks that are considered to satisfy a goal condition. The last rule ensures that at least one of the blocks satisfies all four goal conditions.

We now need rules defining when different kinds of goal conditions are satisfied. For simplicity, let us consider stacks to have at least one block. In that case we have the following rule defining when a block is a stack.

```
satisfied(S,is,stack,T):-blk(S),time(T),clear(S,T).
```

We next define that the color of a stack at time point `T` is blue if all blocks in that stack at that time point are blue. The next rule simply says that a stack is blue if its top is blue and all blocks below the top are also blue.

```
satisfied(S,color,blue,T):-blk(S),time(T),color(S,blue),clear(S,T),
                   #count{U:above(U,S,T), not color(U,blue)}==0.
```

Defining `satisfied(S,height,same,T)` is less straightforward. First, we define a predicate `same_height(S,U,T)` meaning that in the context of time step `T`, `S` and `U` have the same height (or two stacks represented by `S` and `U` have the same height). We then use this predicate to define `satisfied(S,height,same,T)`. Note that the definition of the predicate `satisfied(S,height,same,T)` also enforces the constraint that the blue stack that is compared to the stack represented by `S` does not change.

```
same_height(S,U,T):-blk(S),blk(U),S!=U,ontable(S,T),ontable(U,T).
same_height(S,U,T):-blk(S),blk(U),S!=U,on(S1,S,T),on(U1,U,T),
                    same_height(S1, U1, T).
satisfied(S,height,same,T):- satisfied(S,is,stack,T), S!=U,
             satisfied(U,is,stack,T),satisfied(U,color,blue,T),
             unchanged(U,T),clear(S,T),clear(U,T),same_height(S,U,T).
```

Similarly, defining `satisfied(S, type, another,T)` is also not straightforward. Intuitively, it means that there is another stack `U` different from `S` and `U` has not changed over time. We define it using a new predicate `unchanged(U,T)` which means that the stack with `U` at the top has not changed over time (from step 0 to step `T`).

```
satisfied(S,type,another,T):-blk(U),unchanged(U,T),same_height(S,U,T).
unchanged(U,T):- time(T),blk(U),not changed(U,T).
changed(U,T)  :- time(T),T>0,blk(U),above(V,U,0), not above(V,U,T).
changed(U,T)  :- time(T), T > 0, blk(U),ontable(U,0), not ontable(U,T).
changed(U,T)  :- time(T), T > 0, blk(U),not ontable(U,0), ontable(U,T).
changed(U,T)  :- time(T), T > 0, blk(U),not ontable(U,0), on(X,U,T).
```

Let us denote with G_b the collection of rules developed in this section. To compute a plan, we will only need to add the initial state and the rules described above for enforcing that all goal conditions must be satisfied. Initializing the initial situation as 0 (Figure 1), the description of who has which blocks and what is on the table can be expressed through the following facts, denoted with I_b:

```
has(human,b1,0). color(b1,red).  has(human,b2,0). color(b2,brown).
has(human,b3,0). color(b3,brown).has(human,b4,0). color(b4,green).
has(human,b5,0). color(b5,blue). has(robot,b6,0). color(b6,red).
has(robot,b7,0). color(b7,red).  has(robot,b8,0). color(b8,blue).
has(robot,b9,0). color(b9,blue). has(robot,b10,0).color(b10,green).
has(robot,b11,0).color(b11,green).ontable(b12,0). color(b12,red).
ontable(b13,0).  color(b13,blue).on(b13,b14,0).  color(b14,blue).
```

It is easy to see that the program $D_b \cup I_b \cup G_b$ with `n=2` has two answer sets, one contains the atoms `occ(putontable(b8),0), occ(put(b8,b9),1)` that corresponds to a possible plan `putontable(b8),put(b8,b9)` for the robot; the other one corresponds to the plan `putontable(b9),put(b9,b8)`.

2.3 What If Planning Fails?

The previous subsection presents an initial configuration in which the robot can generate a plan satisfying the request from the human. Obviously, there are several configurations

of the block world in which the planning generation phase can fail. What should be the robot's response? What is the ground for such response? Let us consider one of such situations.

Assuming a different initial configuration in which the robot has only one blue block, say b8 (or b9). Let I_{b_f} be the new initial situation. Furthermore, we require that the robot cannot use the block belonging to the human if it does not get the human's permission. In this case, the robot will not be able to add another blue stack of the same height to the table because there is no plan that can satisfy this goal. More specifically, it is because the robot does not have enough blue blocks for the task at hand. This is what the robot should respond back to the human user. We next discuss a possible way for the robot to arrive at this conclusion.

First, the encoding in the previous subsection must be extended to cover the requirement that the robot can only use its own blocks or the blocks on the table in the construction of the new stack. This can be achieved with the following rules:

```
available(X, T) :- time(T),blk(X),has(robot,X,T).
available(X, T) :- time(T),blk(X),ontable(X, T),clear(X,T).
available(X, T) :- time(T),blk(X),above(Y,X,T),clear(X,T).
:-occ(putontable(X),T),not available(X,T).
:-occ(put(Y,X),T),not available(X,T).
```

Let D_{b_f} be D_b unions with the above rules. It is easy to see that, for any constant n, the program $D_{b_f} \cup I_{b_f} \cup G_b$ does not have an answer set. When the planning fails, the robot should analyze the situation and come up with an appropriate response. A first reasonable step for the robot is to identify why the planning fails. As it turns out, the program described in the previous subsection only needs only minor changes to accomplish this task. First, we need to modify the goals as follows.

```
g_cond(S,is,stack):-blk(S),ok(1). g_cond(S,height,same):-blk(S),ok(3).
g_cond(S,color,blue):- blk(S),ok(2). g_cond(S,type,another):-blk(S),ok(4).
{ok(1..4)}4.  ngoals(X):-X=#count{I:ok(I)}.  #maximize {X:ngoals(X)}.
```

Let G_{b_f} be the new goal, obtained from G_b by replacing the rules defining goal_cond(.) with the above rules. It is easy to check that every answer set of the program $D_{b_f} \cup I_{b_f} \cup G_{b_f}$ does not contain the atom ok(2) which indicates that the goal condition goal_cond(S,color,blue) cannot be satisfied. As such, the robot should use the missing goal condition as a ground for its response. However, the robot could use this information in different ways. For example, it can tell the human that it does not have enough blue blocks or it could ask the human for permission to use the human's blue blocks to complete the goal. It means that the robot needs the ability to make assumptions and reason with them. This can be defined formally as follows.

Definition 1. *A* planning problem with assumptions *is a tuple $\mathcal{P} = (D, I, G, AP, AF)$ where (D, I, G) is a planning problem, AP is a set of actions, and AF is a set of fluents. We say that $\mathcal{P}$ needs a* plan failure analysis *if (D, I, G) has no solution.*

Intuitively, AP is the set of actions that the robot could execute and AF is a set of assumptions that the robot could assume. For example, for our running example, AP

could contain the logic program encoding of an action `ask(blue)` whose effect is that the robot can use the blue block from the human; AF could be {`has(robot,blx)`, `color(blx,blue)`}. So, a planning problem with assumption for the robot is $\mathcal{P}_{b_f} = (D_{b_f}, I_{b_f}, G_b, \{ask(blue)\}, \{has(robot, blx), color(blx, blue)\})$.

Definition 2. *A* plan failure analysis *for a planning problem with assumptions* $\mathcal{P} = (D, I, G, AP, AF)$ *is a pair* (A, F) *such that* $A \subseteq AP$*,* $F \subseteq AF$*, and the planning problem* $(D \cup A, I \cup F, G)$ *is solvable.* (A, F) *is a* preferred plan failure analysis *if there exists no analysis* (A', F') *such that* $A' \subsetneq A$ *or* $F' \subsetneq F$*.*

It is easy to see that $\mathcal{P}_{b_f} = (D_{b_f}, I_{b_f}, G_b, \{ask(blue)\}, \{has(robot, blx), color(blx, blue)\})$ has two preferred plan failure analyses; one, ($\emptyset$, {`has(robot,blx)`,`color(blx,blue)`}), tells the robot that it does not have enough blue blocks; another one $(\{ask(blue)\}, \emptyset)$ tells the robot to ask for permission to use the human's blue blocks.

To compute a preferred plan failure analysis, we can apply the same method used in identifying a minimal set of satisfying goal conditions. We assume that for each action `act` in AP, AP consists of the rules defines its effects and a declaration of the form `is_ap(act)`. For each fluent l in AF, we assume that AF contains a declaration `is_af(l)` as well as the rules for describing how l changes when actions are executed. Let D_a be the following set of rules:

```
{action(A) : is_ap(A)}.      a_assume(A):- action(A),is_ap(A).
{assume(L) : is_af(L)}.      L@0:-assume(L).
nl_assume(F):-F=#count{L:assume(L)}.
na_assume(Y):-Y=#coun {A:a_assume(A)}.
#minimize {1@1,F:nl_assume(F)}.  #minimize {1@1,A:na_assume(A)}.
```

The first rule says that the robot can assume any action in AP. Any action that is assumed will be characterized by the predicate `a_assume(.)` (the second rule). The third rule says that the robot can assume any fluent in AF. `L@0` represents the fact that `L` is true at the time 0. The rest of the rules minimizes the number of actions and the number of assumed fluents, independent from each other. We can show that the new program returns possible preferred plan failure analyses of the problem.

3 Conclusions and Future Work

We describe two problems related to planning that need to be addressed for an effective collaboration between an intelligent system and human when communication via natural language is necessary. We show that ASP based approaches can be employed to deal with these two problems. Our discussion shows that ASP can play an important role in the development of intelligent systems that can interact with human via natural language. Our future goal is to develop further applications that integrate various aspects of AI including NLP, Ontologies and Reasoning by using KPARSER that has been used to address the Winograd Schema Challenge [3].

References

[1] DARPA. Communicating with Computers (CwC), 2015.
[2] V. Lifschitz. Answer set programming and plan generation. *AIJ*, 138(1–2):39–54, 2002.
[3] A. Sharma, S. Aditya, V. Nguyen, and C. Baral. Towards Addressing the Winograd Schema Challenge - Building and Using Needed Tools. In *Proceedings of IJCAI*, 2015.