"Add Another Blue Stack of the Same Height!": Plan Failure Analysis and Interactive Planning Through Natural Language Communication

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 and discuss how the third task could be solved using the available tools and answer set programming and identify the challenges that need to be addressed.

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 are a few abstract examples of such an interactive planning using natural language communication.

Imagine a robot that can pick up cubical named blocks (each block is labeled with one letter on all sides) and stack them atop of each other to create towers of blocks. The robot can recognize the letters written on the blocks. Furthermore, a user can communicate with the robot via a screen. When the user types "LPNMR" on the screen, the robot will pick up the blocks with letters 'L', 'P', 'N', 'M', and 'R', and creates a stack LPNMR[1] (with L on P, P on N, etc.).

[1] One of such robot was on display at the Tech Museum of Innovation, San Jose, CA in 2000.

It is easy to see that when the string LPNMR is communicated to the robot and the robot understands that it needs to create a tower using letters specified in the string, i.e., it needs to create a plan for the following properties[2] `on(P,L)`, `on(N,P)`, `on(M,N)`, `on(R,M)`, and `ontable(R)`. In other words, the robot understands that it needs to (*a*) find a solution of a planning problem; and then (*b*) execute the plan. Observe that beside the ability of creating and executing a plan, the key for the success of the robot is the ability to understand the goal of the planning problem specified by the string. In this case, this task is simple as the translation of a string to the planning goal is rather straightforward.

Imagine now that the robot can recognize also colors and the blocks come with color. The following scenario, discussed in [7], is significantly more challenging:

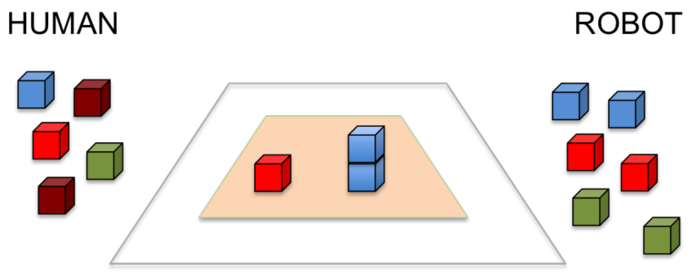

Fig. 1: *A Simplified Version of The Blocks World Example from the BAA*

Consider the block world domain in Figure 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;
– 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

[2] `on(X,Y)` states that `Y` is on `X`.

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 (e.g., [8, 12, 18, 19]) and there have been attempts to translate NLP into ASP (e.g., [1, 2, 15]), we would like to explore the use of ASP and related tools in solving this type of problems. We demonstrate that ASP can be a viable option for the building of such a system. Specifically, we discuss components and methodologies that are already available and can be used. We identify challenges that lie ahead and discuss possible solutions.

The main contributions of the paper is combining natural language processing (semantic parsing), and use of linguistic knowledge with ASP based reasoning to address human-computer interactive planning. Such combination is needed in building integrated AI systems[3]. In addition we give an initial formulation of plan failure analysis and show its role in human-computer interactive planning.

2 Background: Answer Set Programming (ASP)

A logic program Π is a set of rules of the form

$$c_1 \mid \ldots \mid c_k \leftarrow a_1, \ldots, a_m,\ not\ a_{m+1}, \ldots,\ not\ a_n \tag{1}$$

where $0 \leq m \leq n,\ 0 \leq k$, each a_i or c_j is a literal of a first order language and not represents *negation-as-failure (naf).* For a literal a, $not\ a$ is called a naf-literal. For a rule of the form (1), the left and right hand sides of the rule are called the *head* and the *body*, respectively. Both the head and the body can be empty. When the head is empty, the rule is called a *constraint*. When the body is empty, the rule is called a *fact*.

For a ground instance r of a rule of the form (1), $head(r)$ denotes the set $\{c_1, \ldots, c_k\}$; and $pos(r)$ and $neg(r)$ denote $\{a_1, \ldots, a_m\}$ and $\{a_{m+1}, \ldots, a_n\}$, respectively.

Consider a set of ground literals X. X is consistent if there exists no atom a such that both a and $\neg a$ belong to X. The body of a ground rule r of the form (1) is *satisfied* by X if $neg(r) \cap X = \emptyset$ and $pos(r) \subseteq X$. A ground rule of the form (1) with nonempty head is satisfied by X if either its body is not satisfied by X or $head(r) \cap X \neq \emptyset$. In particular, a constraint is *satisfied* by X if its body is not satisfied by X.

For a consistent set of ground literals S and a ground program Π, the *reduct* of Π w.r.t. S, denoted by Π^S, is the program obtained from Π by deleting (**i**) each rule that has a naf-literal *not* a in its body with $a \in S$, and (**ii**) all naf-literals in the bodies of the remaining rules.

S is an *answer set* of a ground program[4] Π [11] if it satisfies the following conditions: **(i)** If Π does not contain any naf-literal (i.e., $m = n$ in every rule of Π) then S is a minimal consistent set of literals that satisfies all the rules in Π; and **(ii)** If Π contains some naf-literal ($m < n$ in some rule of Π), then S is an answer set of Π^S. Note that Π^S does not contain naf-literals, and thus its answer set is defined in case (**i**).

[3] Similar example of such combination is the work [15] where NLP is combined with ASP based reasoning to address a class of Winograd Schema Challenge problems.

[4] A program with variables is viewed as a collection of all ground instances of its rules.

A program Π is said to be *consistent* if it has a consistent answer set. Otherwise, it is inconsistent.

To increase the expressiveness of logic programming and simplify its use in applications, logic programming has been extended with several constructs such as

- Weight constraint atom (e.g., [13]): atoms of the form
$$l\,[a_1 = w_1, \ldots, a_n = w_n,\ not\, b_{n+1} = w_{n+1}, \ldots, b_{n+k} = w_{n+k}]\, u \qquad (2)$$
where a_i and b_j are literals and l, u, and w_j's are integers, $l \leq u$.
- Aggregates atoms (e.g, [9, 14, 16]): Atoms of the form
$$f(S)\ op\ v \qquad (3)$$
where S is a set-literal defined in , $f \in \{\textsc{Sum}, \textsc{Count}, \textsc{Max}, \textsc{Min}\}$, $op \in \{>, <, \geq, \leq, =\}$, and v is a number; and, a set-literal is of the form (*i*) $\{\boldsymbol{X} \mid p(\boldsymbol{W})\}$ where $\boldsymbol{X}$ is a vector of variables, $\boldsymbol{W}$ is vector of parameters and constants such that each variable in $\boldsymbol{X}$ also occurs in $\boldsymbol{W}$; or (*ii*) $\{\!\{\boldsymbol{X} \mid \boldsymbol{Y}.p(\boldsymbol{W})\}\!\}$ where $\boldsymbol{X}$ and $\boldsymbol{Y}$ are vectors of variables, $\boldsymbol{W}$ is vector of parameters and constants such that each variable in $\boldsymbol{X}$ or $\boldsymbol{Y}$ also occurs in $\boldsymbol{W}$.

The semantics of logic programs with such atoms has been defined (e.g., [9, 13, 14, 16]. Standard syntax for these types of atoms has been proposed and adopted in most state-of-the-art ASP-solvers such as CLASP [10] and DLV [5].

3 Planning in The Blocks World Domain

In this section, we present an ASP encoding of the block world domain that will be used throughout the paper. This section can also be seen as a review of answer set planning. We assume that blocks are specified by the predicate `block(.)`. We start with writing the basic actions in the blocks word domain. For simplicity, we only consider two actions, one says that the robot can put down a block onto the table and the other on top of a block.

```
action(putontable(X))    :- block(X).
action(put(X,Y))         :- block(X), block(Y).
```

We describe properties of the world 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. In particular, we use two fluents `on(X,Y)` and `ontable(X)` which denote `Y` is on `X` or `X` is on the table, respectively. The `time(.)` atom represents a time step whose value ranges from 0 to n where n is a constant denoting the maximal number of steps that we are interested in. We now define some derived fluents such as when a block is clear and when it is not; and when a block is above another block.

```
occupied(X,T) :- time(T),block(X),block(Y),X!=Y,on(X, Y, T).
clear(X,T)    :- time(T),block(X),not occupied(X, T).
above(X,Y,T)  :- time(T),on(X,Y,T).
above(X,Y,T)  :- time(T),on(Z,Y,T),above(X,Z,T).
```

We define the executability conditions of actions. The next lines below specify when the action `putontable(X)` and the action `put(X,Y)` are executable, respectively. More specifically, the first two lines say that `putontable(X)` is executable when it is an action and the block is clear; the last two lines state that `put(X,Y)` is executable when both `X` and `Y` are clear.

```
executable(putontable(X),T):-time(T),
                              action(putontable(X)),clear(X,T).
executable(put(X,Y),T):-time(T),
                         action(put(X,Y)),clear(X,T),clear(Y,T).
```

We now specify the effect of the actions on the two fluents `on` and `ontable`. The first line below says that putting `Y` on `X` causes `on(X,Y)` to be true in the next time step. The next two lines account for the inertial effects of actions, i.e, if `Y` is on `X` and `Y` was not moving somewhere or on to the table then `Y` will still be on `X`; likewise, `X` will be on the table if it is put on the table (line 4) or it was on the table and it is not moved on top of another block.

```
on(X,Y,T+1):-time(T),occ(put(X,Y),T).
on(X,Y,T+1):-time(T),on(X,Y,T),#count{Z:occ(put(Z,Y),T)}==0.
on(X,Y,T+1):-time(T),on(X,Y,T),not occ(putontable(Y),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.
```

The above code, denoted with D_b, encodes a set of actions of a domain, in this case the actions `putontable` and `put` in the block world domain. D_b can be used for planning, hypothetical reasoning, and diagnosis. Formally, a planning problem is specified by a triple (D, I, O) where D is the set of actions, I specifies the initial state of the world, and O is a fluent formula. Observe that each these components can be given by a set of logic programming rules as shown above, i.e., D_b encodes the set of actions. To use D_b in planning, we need the following components:

– *Action generation rules*: To find a linear plan, we need to specify that at each time, exactly one action occurs. Furthermore, we also need to make sure that an action can occur when it is executable. This can be encoded as follows.

  ```
  1 {occ(A, T) : action(A)} 1:- time(T), T < n.
  :- occ(A, T), not executable(A, T).
  ```

– *Initial state specification*: The initial state will be specified by a set of ground atoms of the form `block(x)`, `ontable(z)`, and `on(x,y)` that describes the initial configuration of the block world. An encoding of the configuration for the block world domain in Figure 1 is shown in the next section.

– *Goal state specification*: The goal state is specified by a set of rules that defines when a configuration is achieved at a certain step, say n. For example, to create the tower 'LPNMR', a goal is specified by the following rules:

  ```
  satisfied_goal(n) :- on('P','L',n),on('N','P',n),
                       on('M','N',n),on('R','M',n),
                       ontable('R',n).
  :- not satisfied_goal(n).
  ```

Proof of correctness of logic programming encoding of planning has been discussed extensive in the literature (see, e.g., [8, 12, 18, 19]).

4 Add Another Blue Stack of the Same Height!: How To Respond?

In this section, we discuss possible responses to the communication from the human "*Add another blue stack of the same height!*" to the robot. We start with the planning problem implied by this communication.

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

Assume that the robot can recognize that it needs to create and execute a plan in responding to the communication. 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 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*. Intuitively, a descriptive planning problem is given by a set of actions D, an initial state I, and a formula φ about the trajectory of the plan. For example, the formula describing the desirable state communicated to the robot relates the goal state and the initial state. Such a formula can be represented in the goal language proposed in [17]. How to translate the given sentence to a goal is a challenging problem that we will attempt to address in the next section.

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.

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

The first rule states that the goal is to build a stack represented by `S`. The second rule indicates the color of the stack. The third rule states that the stack needs to have the same height. The fourth rule requires that this is another 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 described in the last section 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_satisfied_goal(S,T)   :-  block(S),goal_cond(X,Y,Z),
                              not satisfied(X,Y,Z,T).
satisfied_goal(S, T)      :-  not not_satisfied_goal(T).
:- X = #count {S  : satisfied_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 at the time step `n`.

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):-block(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):-block(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)` as follows. 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) :- block(S), block(U), S!= U,
                        ontable(S, T), ontable(U, T).
same_height(S, U, T) :- block(S), block(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),
                             satisfied(U,is,stack,T),
                             satisfied(U,color,blue,T),
                             S != U, 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):-block(U),unchanged(U,T),
                             same_height(S,U,T).
unchanged(U,T):- time(T),block(U),not changed(U,T).
changed(U,T)  :- time(T),T>0,block(U),
                 above(V,U,0), not above(V,U,T).
changed(U,T)  :- time(T), T > 0, block(U),
                 ontable(U,0), not ontable(U,T).
changed(U,T)  :- time(T), T > 0, block(U),
                 not ontable(U,0), ontable(U,T).
changed(U,T)  :- time(T), T > 0, block(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,bl1,0). color(bl1,red).   has(human,bl2,0). color(bl2,brown).
has(human,bl3,0). color(bl3,brown).has(human,bl4,0). color(bl4,green).
has(human,bl5,0). color(bl5,blue). has(robot,bl6,0). color(bl6,red).
has(robot,bl7,0). color(bl7,red).   has(robot,bl8,0). color(bl8,blue).
has(robot,bl9,0). color(bl9,blue). has(robot,bl10,0).color(bl10,green).
has(robot,bl11,0).color(bl11,green).ontable(bl12,0). color(bl12,red).
ontable(bl13,0).  color(bl13,blue).on(bl13,bl14,0).  color(bl14,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(block8),0), occ(put(block8,block9),1)` that corresponds to a possible plan `putontable(block8),put(block8,block9)` for the robot; the other one corresponds to the plan `putontable(block9),put(block9,block8)`.

4.2 What If Planning Fails?

The previous subsection presents an initial configuration in which the robot can generate and execute 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 `block8` (or `block9`). 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 take into consideration 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 done by adding the following rules:

```
available(X, T) :- time(T),block(X),has(robot,X,T).
available(X, T) :- time(T),block(X),ontable(X, T),clear(X,T).
available(X, T) :- time(T),block(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).
```

The first three rules define the predicate `availale(.,.)`, i.e., when a block is available for the robot to use. The next two rules enforce the constraint that the robot can only use blocks at its disposal in its actions. 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.

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

The above rules are self-explanatory. 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 = #count {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 the following:

Proposition 1. *For* planning problem with assumptions *is a tuple* $\mathcal{P} = (D, I, G, AP, AF)$, *the program* $D \cup I \cup G \cup AP \cup AF \cup D_a$ *returns the possible preferred plan failure analyses of* $\mathcal{P}$.

5 How To Get The Goal Conditions?

We now discuss a possible approach to extracting the goal conditions from the communications and to creating natural language response for the robot. Obviously, to convert the communication "Add another blue stack of the same height!" to the goal conditions `goal_cond(S,_,_)`, we will need to use a system with NLP capability. In this paper, we propose to use KPARSER [15] which is a graph based semantic parser[5] that has the following salient features: (i) a well established and rich vocabulary to represent event-entity and inter-event relations from the KM component library [4, 6] which has been proven useful in building many knowledge bases such as AURA [3]; (ii) relations such as *prototype_of* and *instance_of*, from [6], that allows expression of limited but useful quantification; (iii) conceptual classes of nodes based on word sense disambiguation on WordNet senses of the words represented by those nodes; (iv) semantic roles of the entities present in the text; and (v) accumulation of correct relations between nodes through the use of Named Entity Tagging and Syntactic Dependency Parsing.

KPARSER takes text as input and parses it as an acyclic semantic graph $G = (V_L, E_L)$, such that V_L consists of nodes from the union of four sets of nodes, namely entities (V_{en}), events (V_{ev}), properties (V_{prop}) and classes (V_{class}). We used KM library for labeling the relationship edges between the nodes in graph G. There are 118 total relations available in KM and currently we used some of the important relations such

[5] The system is available online at `http://kparser.org`.

as *causes, next-event, agent*, *recipient* etc. We also defined a few new relations to represent some of the edge labels that were not captured in KM. These relations include *instance_of*, *superclass*, *participant* and *related_to*.

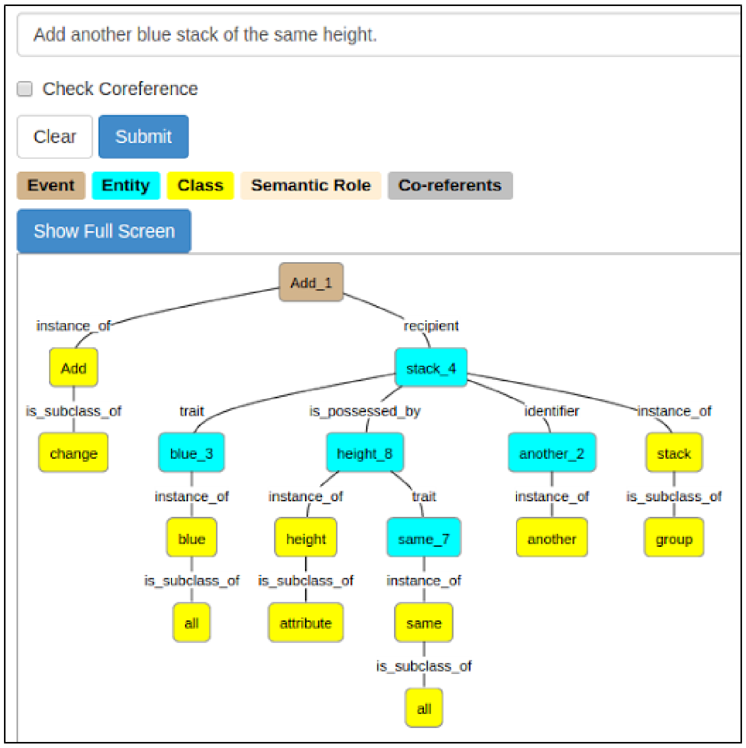
Add another blue stack of the same height.

Fig. 2: Semantic Parsing by KPARSER

Figure 2 shows the output of KPARSER on the sentence "Add another blue stack of the same height." For illustration, let us write the event-entity, entity-entity, event-class and entity-class facts of the KPARSER output.

```
edge(add_1,recipient,stack_4).   edge(add_1,instance_of,add).
edge(add,is_subclass_of,change). edge(stack_4,trait,blue_3).
edge(stack_4,is_possessed_by,height_8).edge(blue,is_subclass_of,all).
edge(stack_4,identifier,another_2).edge(stack_4,instance_of,stack).
edge(stack,is_subclass_of,group). edge(height_8,trait,same_7).
edge(blue_3,instance_of,blue).  edge(another_2,instance_of,another).
edge(same_7,instance_of,same).  edge(same,is_subclass_of,all).
edge(height_8,instance_of,height).edge(height,is_subclass_of,attribute).
```

In addition to the edges, KPARSER also produces the classification of the nodes such as `event(add_1)`, `entity`, etc. To be able to convert such facts into the set of goal conditions, the robot will need a knowledge base about the block world domain that provides a context for the sentence. From this knowledge base, given the communication, the robot should be able to infer the following:

– the event `add_1`, an instance of `add`, indicates the action that the robot should do;
– `stack_4`, an instance of `stack`, is the objective of the action of the robot;
– `blue_3`, an instance of `blue`, is an attribute of the objective of the robot's action;
– `height_8`, an instance of `height`, is an attribute of the objective of robot's action which has again an attribute `same`; and
– `another_2`, an instance of `another`, is an attribute of the objective of the robot's action.

Ideally, the robot should be able to use the above information, together with its knowledge base, to come up with the goal conditions or an equivalent formula that will trigger its planning module to generate a plan for responding to the human. To the best of our knowledge, there exists no such system that can complete this task. We observe that an ad-hoc solution could be developed by making use of the nodes classification and the graph. For example, by observing that attribute nodes are *instance_of* some entities, the facts such as `add(stack)`, `attribute(blue)`, `attribute(same, height)`, and `attribute(another)` could be derived using the graph and the classification of the nodes. Such facts can then be translated into the aforementioned goal conditions. We observe that several reasonable instructions within the block world domain seem to have this structure. Although this works, this ad-hoc solution is certainly not the best solution as it does not account for all possible scenarios. Since a system that can understand all possible communications is yet to be developed, this ad-hoc solution might still be a viable option for restricted applications.

A formal solution to the above problem is to develop a system that translates the output of KPARSER to ASP code. One of the systems that could potentially be used for this purpose is the system NL2KR (available at `http://bioai.lab.asu.edu/nl2kr/`) that has been used in translating puzzles into ASP programs [1]. For NL2KR to be useable in this type of applications, we need an initial lexicon of word meanings in terms of λ-expressions and a set of training sentences and their translations in a target language. The NL2KR-Learning component learns the meaning (λ-expressions) of new words from these examples and the NL2KR-Testing component uses the enhanced lexicon to translate new sentences. The λ notation can be used to express ambiguities than can be resolved with contextual information and is useful in composition. It can also be used in generating a response such as "*I do not have enough blue blocks*" for the robot. Improving NL2KR to obtain a system that can address this need is a challenge that we would like to investigate in the near future.

6 Conclusions and Future Work

In this paper, we describe three challenging problems 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 two problems, the descriptive planning problem and plan failure analysis problem. We discuss the difficulties and possible options for addressing the third problem that is related to the bidirectional translation between natural language and ASP.

Our discussion shows that ASP can play an important role in the development of intelligent systems that can interact with human via natural language. It highlights the need of a tight integration between ASP and NLP processing tools with advanced features for translating natural language sentences into ASP. Our future goal is to develop further applications that integrate various aspects of AI including NLP, Ontologies and Reasoning. We believe that our experience in using KPARSER to address the Winograd Schema Challenge will be important for us to achieve this goal [15].

References

[1] C. Baral and J. Dzifcak. Solving puzzles described in english by automated translation to answer set programming and learning how to do that translation. *KRR*. AAAI Press, 2012.

[2] C. Baral, J. Dzifcak, and T. C. Son. Using Answer Set Programming and Lambda Calculus to Characterize Natural Language Sentences with Normatives and Exceptions. *AAAI*, 2008.

[3] K. Barker, V. K. Chaudhri, S. Y. Chaw, P. Clark, D. Hansch, B. E. John, S. Mishra, J. Pacheco, B. W. Porter, A. Spaulding, and M. Weiten. AURA: enabling subject matter experts to construct declarative knowledge bases from science textbooks. In *AAAI*, 2007.

[4] K. Barker, B. W. Porter, and P. Clark. A library of generic concepts for composing knowledge bases. In *Proceedings of the 1st K-CAP*, pages 14–21. ACM, 2001.

[5] S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dlv system: Model generator and application frontends. In F. Bry, B. Freitag, and S. D., editors, *Proceedings of the 12th WLP*, pages 128–137, 1997.

[6] P. Clark and B. Porter. *KM (v2.0 and later): Users Manual*, 2011.

[7] DARPA. Communicating with Computers (CwC), 2015.

[8] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge State Planning, II: The DLV$^{\mathcal{K}}$ System. *AIJ*, 144(1-2):157–211, 2003.

[9] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *JELIA*, LNCS 3229, pages 200–212. Springer, 2004.

[10] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. CLASP: A conflict-driven answer set solver. In *Proceedings of the 9th LPNMR*, LNAI 4483, pages 260–265. 2007.

[11] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 579–597, 1990.

[12] V. Lifschitz. Answer set programming and plan generation. *AIJ*, 138(1–2):39–54, 2002.

[13] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics for weight constraint rules. In *Proceedings of the 5th LPNMR*, pages 315–332, 1999.

[14] N. Pelov, M. Denecker, and M. Bruynooghe. Partial stable models for logic programs with aggregates. In *Proceedings of the 7th LPNMR* , LNCS 2923, 207–219. Springer, 2004.

[15] A. Sharma, S. Aditya, V. Nguyen, and C. Baral. Towards Addressing the Winograd Schema Challenge - Building and Using a Semantic Parser and a Knowledge Hunting Module. In *Proceedings of IJCAI*, 2015.

[16] T. C. Son and E. Pontelli. A Constructive Semantic Characterization of Aggregates in Answer Set Programming. *TPLP*, 7(03):355–375, 2007.

[17] T. C. Son, E. Pontelli, and C. Baral. A non-monotonic goal specification language for planning with preferences. In *Essays Dedicated to Gerhard Brewka on the Occasion of His 60th Birthday*, pages 202–217, 2014.

[18] P. H. Tu, T. C. Son, and C. Baral. Reasoning and planning with sensing actions, incomplete information, and static causal laws using logic programming. *TPLP*, 7:1–74, 2006.

[19] P. H. Tu, T. C. Son, M. Gelfond, and R. Morales. Approximation of action theories and its application to conformant planning. *AIJ*, 175(1):79–119, January 2011.