Künstliche Intelligenz manuscript No.
(will be inserted by the editor)

Answer Set Planning in Single- and Multi-Agent Environments

Tran Cao Son · Marcello Balduccini

Received: date / Accepted: date

Abstract We present the main ideas of answer set planning in both single- and multi-agent environments. Specifically, we describe a systematic translation of a dynamic domain—given as set of statements in an action language such as $\mathcal{B}$—into a logic program which can be used for planning and other reasoning tasks (e.g., diagnosis) given the dynamic domain. We illustrate the issues of answer set planning in different settings and their solutions using a well-known problem domain, the Kiva robot system.

Keywords Answer set planning · Multi-agent planning · Action languages

1 Introduction

Answer set planning [12] is one of the earliest and most discussed applications of Answer Set Programming (ASP) [13,15]. To solve a planning problem $\mathcal{P}$ using ASP, we translate it to a logic program $\pi(\mathcal{P}, n)$ where n is an integer representing the maximum length of solutions to $\mathcal{P}$ that we are interested in.[1] In essence, $\pi(\mathcal{P}, n)$ has two parts. The first part encodes the transition system described by the domain specified by $\mathcal{P}$ and the second part encodes the initial and goal state and is responsible for the generation of possible plans and validation of solutions.

Planning domains can be specified by planning domain description language (PDDL) [8] or action languages [7]. In this paper, we will use the language $\mathcal{B}$ defined by [7]. In this language, a domain is characterized by a set $\mathcal{F}$ of *fluents* and a set $\mathcal{A}$ of *actions*. For simplicity, in this paper we only consider discrete domains, Boolean fluents (i.e. fluents whose value is either true or false) and instantaneous actions (i.e., actions that have no duration and whose effects occur immediately after the execution of the action). More specifically, a domain $\mathcal{P}$ over $(\mathcal{F}, \mathcal{A})$ is a collection of statements of the form

$$\textbf{impossible_if}(a, \psi) \quad (1)$$

$$\textbf{causes}(a, l, \psi) \quad (2)$$

$$\textbf{if}(l, \psi) \quad (3)$$

where $a \in \mathcal{A}$, l is either a fluent $f \in \mathcal{F}$ or its negation $\neg f$ (hereafter called *literal*), and ψ is a set of literals. Statements of the form (1) describe when actions cannot be executed. A statement of the form (2) states that if a is executed when ψ is true, then l becomes true. A statement of the form (3) represents a *state constraint*

[1] This is a special case of the more general problem of finding a plan of any length. In this paper, we focus on the simpler variant to keep the presentation simple.

Tran Cao Son
New Mexico State University, Las Cruces, NM 88003, USA
Tel.: +1-575-646-1930
Fax: +1-575-646-1002
E-mail: tson@cs.nmsu.edu

Marcello Balduccini
Saint Joseph's University, Philadelphia, PA 19131, USA
Tel.: +1-610-660-3457
Fax: +1-610-660-1192
E-mail: marcello.balduccini@gmail.com

and says that if ψ is true then l must be true; l can be $\perp$ in (3), which indicates a state constraint of the domain, i.e., the conjunction of the literals in ψ must not hold.

Formally, a dynamic domain, i.e. a domain whose state may change over time as the result of occurrences of actions, can be represented as a directed graph, whose nodes correspond to states of the domain and whose edges represent state transitions. Edges are labeled by sets of actions, which describe the actions whose concurrent occurrence causes that transition. This graph can be described by $\Phi_{\mathcal{P}}$, the *transition function*. The precise formalization of $\Phi_{\mathcal{P}}$ can be found in [7].

The rest of the paper is organized as follows. Section 2 describes the Kiva robot environment that will be used as the running example of the paper. Section 3 describes the translation of a domain into a logic program. Section 4 focuses on answer set planning in multi-agent environments. Conclusions are found in Section 5.

2 A Motivating Example: Kiva Robots

As a motivating example, throughout this paper we refer to the Kiva robot scenario, consisting of an autonomous warehouse system [20] (illustrated by Figure 1) where robots (in orange) navigate around a warehouse to pick up inventory pods from their storage locations (in green) and drop them off at designated inventory stations (in purple) in the warehouse. Relevant properties of the domain are the robot's location, described by a fluent $at(l)$, where l is a location, and whether or not it is carrying a pod p, described by a fluent $carrying(p)$. The location of pods can be described by fluents of the form $pod_at(p, l)$. The actions available to the robot are $move(l)$, which moves the robot to location l, $pick_up(p, l)$, which picks up the pod p situated at the location l (as long as the robot is there as well), and $drop_off$, which drops off the pod currently carried by the robot.

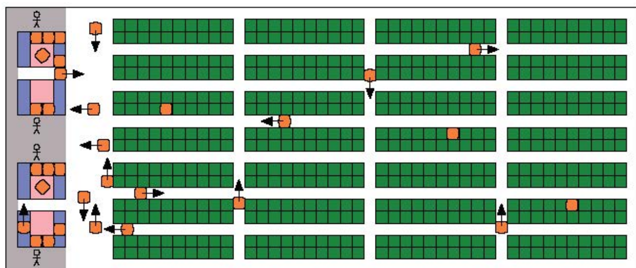

Fig. 1 Layout of an Autonomous Warehouse System [Wurman *et al.*, 2008]

As we mentioned earlier, the domain is formalized by means of statements of the form (1)-(3). For instance, the effect of action $pick_up(p, l)$ is described by:

$$\textbf{causes}(pick_up(p, l), carrying(p), \{\}). \tag{4}$$

Because a Kiva robot can only carry one pod at a time, $pick_up(p, l)$ cannot be executed if $carrying(p')$ holds. This is formalized by a set of statements of the form:

$$\textbf{impossible_if}(pick_up(p, l), \{carrying(p')\}). \tag{5}$$

Furthermore, given a pair l, l' of connected locations, the effect of action $move(l')$ is formalized by:

$$\textbf{causes}(move(l'), at(l'), \{at(l)\}). \tag{6}$$

The statement says that, if the robot is at location l when action $move(l')$ occurs, then the outcome of the action is that the robot is at l'. Note that, intuitively, the robot can only be at a single location at a time. This is formalized, for $l \neq l'$, by an instance of (3):

$$\textbf{if}(\neg at(l'), \{at(l)\}). \tag{7}$$

We will also have similar statements for $pot_at(p, l)$.

3 ASP Planning and ASP-Based Reasoning

When encoding a planning domain in ASP, fluents and actions are represented by ASP terms. An ASP atom of the form $holds(f, i)$ states that fluent f is true at step i in the evolution of the domain. If f is false, then we write $\neg holds(f, i)$. An atom $occurs(a, i)$ indicates the occurrence of action a at step i.

The domain is described by ASP rules. For instance, the ASP counterpart of (4) is

$$holds(carrying(P), I+1) \leftarrow occurs(pick_up(P, L), I). \tag{8}$$

where I is an ASP variable representing a time step in the evolution of the domain. An executability condition such as (5) is mapped to an ASP constraint:

$$\leftarrow occurs(pick_up(P, L), I), holds(carrying(P'), I).$$

Statement (7) is translated as:

$$\neg holds(at(L'), I) \leftarrow holds(at(L), I), L \neq L'.$$

To enable a compact representation, in the following we assume the existence of a relation $connected(l_1, l_2)$,

% $pick_up(p, l)$ requires pod p present at location l
$\leftarrow occurs(pick_up(P, L), I), \neg holds(pod_at(P, L), I).$
$\leftarrow occurs(pick_up(P, L), I), \neg holds(at(L), I).$
% While it is carried, the pod follows the robot
$holds(pod_at(P, L), I) \leftarrow holds(at(L), I), holds(carrying(P), I).$

% Formalization of action $drop_off$
$\neg holds(carrying(P), I + 1) \leftarrow occurs(drop_off, I).$
$holds(pod_at(P, L), I + 1) \leftarrow occurs(drop_off, I),$
$\quad holds(at(L), I), holds(carrying(P), I).$
$\leftarrow occurs(drop_off, I), \neg holds(carrying(P), I).$

Fig. 2 Additional laws for the Kiva robot domain

stating that l_1 and l_2 are connected, i.e. that the robot can move between them. For example, the fact that the domain consists of two locations, l_d, l_r connected to each other, would be represented by

$$\{connected(l_r, l_d). \quad connected(l_d, l_r).\}$$

Thus, the effect of action *move* can be formalized by the rule:

$$holds(at(L'), I + 1) \leftarrow connected(L, L'), holds(at(L), I), occurs(move(L'), I). \tag{9}$$

The remaining rules for the Kiva domain are shown in Figure 2. The ASP encoding of a dynamic domain is completed by rules encoding the *principle of inertia*, due to [9], according to which “things tend to stay as they are.” In the flavor of representation we use in this paper, this principle is captured by the two rules:

$$holds(F, I + 1) \leftarrow holds(F, I), \text{not } \neg holds(F, I + 1).$$
$$\neg holds(F, I + 1) \leftarrow \neg holds(F, I), \text{not } holds(F, I + 1).$$

where variable F ranges over all fluents. The rules intuitively say that a fluent F is to be assumed to maintain its truth value unless there is evidence that it does not.

The above set Π_k of the rules describes a planning domain for the Kiva robot scenario. Next, we turn our attention to the planning task. Generally speaking, given a set of literals that specify a *goal*, the planning task consists in finding a *plan*, i.e. a sequence of actions that, if executed, is expected to bring the domain to a state satisfying the goal. A goal is encoded in ASP by a set Υ_g of expressions of the form $holds(f, n)$ and $\neg holds(f, n)$ (recall that n is the maximum plan length). The requirement for the goal to be satisfied is formalized by the rules:

$$goal \leftarrow \Upsilon_g.$$
$$\leftarrow \text{not } goal.$$

The first rule detects the satisfaction of the goal, while the second requires that the goal be reached by every solution. For a Kiva robot, assuming that only one pod exists, the goal might be for the pod p to be at a location l_d and for the robot to be at a location l_r. This can be represented by the set of rules, Π_g:

$$goal \leftarrow holds(at(l_r), n), holds(pod_at(p, l_d), n).$$
$$\leftarrow \text{not } goal.$$

Additionally to specifying a goal, a planning task typically requires the indication of the initial state of the domain. This is encoded in ASP by a set Π_i of facts of the form $holds(f, 0)$ and $\neg holds(f, 0)$, where the second argument refers to step 0. In our example, Π_i might be

$$\left\{ \begin{array}{c} holds(at(l_d), 0). \quad holds(pod_at(p, l_r), 0). \\ \neg holds(carrying(P), 0) \leftarrow pod(P). \end{array} \right\}$$

In ASP planning, the task of finding a plan is reduced to finding an answer set of the program $\Pi_k \cup \Pi_g \cup \Pi_i \cup \Pi_p$ where Π_p is the *planning module*:

$$occurs(A, I) \ \vee \ \neg occurs(A, I) \leftarrow step(I). \tag{10}$$

Intuitively, Π_p states that any action is allowed to occur at any time step (variable A ranges over all actions). Given an answer set S, the plan described by it can be extracted from the atoms of the form $occurs(a, i)$ of S. In our example, it is not difficult to see that an answer set exists that contains the atoms

$$occurs(move(l_r), 0), occurs(pick_up(p, l_r), 1),$$
$$occurs(move(l_d), 2),$$
$$occurs(drop_off, 3), occurs(move(l_r), 4)$$

corresponding to the plan in which the robot reaches the pod, picks it up, takes it to location l_d, drops it off, and returns to l_r.

It is instructive to reiterate that the ASP program encoding a planning problem above, $\Pi_k \cup \Pi_g \cup \Pi_i \cup \Pi_p$, includes a parameter n which represents the maximum plan length. As such, if the program $\Pi_k \cup \Pi_g \cup \Pi_i \cup \Pi_p$ does not have an answer set, it only means that it does not have a plan of length n. Using this fact, one can compute shortest solutions of the planning problem by iteratively computing the answer sets of $\Pi_k \cup \Pi_g \cup \Pi_i \cup \Pi_p$ for $n = 0, 1, \ldots$. In this sense, ASP-planning is similar to SAT-based planning [10]. In order to use ASP-planning to determine whether a planning problem is

solvable, one might need to (i) determine the maximal bound of the shortest solution of the problem? and (ii) run the above program with this bound. This is an interesting topic but it is outside of the scope of this paper.

ASP-based reasoning is not difficult to extend beyond planning. Next, we discuss how diagnostic reasoning can be accomplished. The presentation is based on previous work [1] aimed at the development of an architecture for a software agent that operates a physical device and is capable of making observations and of testing and repairing the device's components. Due to space considerations, we will limit our attention to the simplest kind of diagnostic task, in which the agent must find explanations for unexpected observations.

We assume that the agent and the domain satisfy the following simplifying conditions: (1) the agent is capable of making correct observations, performing actions, and remembering the domain history; (2) normally the agent is capable of observing all relevant exogenous actions occurring in its environment. By *exogenous action* we mean an action that may spontaneously occur in the domain. These are distinct from the *agent actions* considered so far, through which the agent actively manipulates the domain.

To begin, let us expand our example domain by a fluent *charged*, expressing whether the robot's battery is sufficiently charged to allow the robot to move between locations and operate pods. We also introduce a fluent *stuck*, indicating whether the robot's lift mechanism is stuck, preventing it from picking up and dropping off pods. The rules of Π_k are extended in a straightforward way. For example, rules (8) and (9) become, respectively:

$$\begin{aligned} holds(at(L'), I+1) \leftarrow\ & connected(L, L'), \\ & holds(at(L), I), \\ & occurs(move(L'), I), \\ & holds(charged, I). \end{aligned}$$

$$\begin{aligned} holds(carrying(P), I+1) \leftarrow\ & occurs(pick_up(P, L), I), \\ & holds(charged, I), \\ & \neg holds(stuck, I). \end{aligned}$$

We introduce two exogenous actions, *break*, which causes the lift mechanism to become stuck, and *run_low*, which causes the battery's charge to run low. Their effects are modeled by the rules:

$$\begin{aligned} holds(stuck, I+1) &\leftarrow occurs(break, I). \\ \neg holds(charged, I+1) &\leftarrow occurs(run_low, I). \end{aligned}$$

To enable diagnostic reasoning, the agent's knowledge base is expanded to include a *recorded history*, which contains observations made by the agent together with a record of its own actions. The recorded history defines a collection of paths in the transition diagram, which, from the standpoint of the agent, can be interpreted as the domain's possible pasts. If the agent's knowledge is complete (i.e., it has complete information about the initial state and the occurrences of actions) and the system's actions are deterministic, then there is only one such path. In the more general case, however, there may be multiple paths. The goal of the diagnostic reasoning task is to allow the agent to explain discrepancies between its expectations and the domain's observed behavior. An explanation consists of exogenous actions, whose unobserved occurrence is a possible explanation of the observed behavior.

More precisely, a *recorded history Γ_n up to a current moment n* is a collection of *observations*, i.e. statements of the form: $obs(f, true, i)$, meaning that fluent f was observed to be true at step i; $obs(f, false, i)$, indicating that f was observed to be false[2]; and $hpd(a, t)$, meaning that action a was observed to happen at step t. i is an integer from the interval $[0, n]$ and t is from $[0, n)$.

The first step of the diagnostic task consists in determining whether the current recorded history contains unexpected observations. This is achieved by the following program Π_r:

$$\begin{aligned} holds(F, 0) &\leftarrow obs(F, true, 0). \\ \neg holds(F, 0) &\leftarrow obs(F, false, 0). \\ occurs(A, I) &\leftarrow hpd(F, I). \end{aligned}$$

$$\begin{aligned} &\leftarrow holds(F, I), obs(F, false, I). \\ &\leftarrow \neg holds(F, I), obs(F, true, I). \end{aligned}$$

The first three rules of Π_r establish the relationship between observations and basic relations of Π_k. The last two rules, called the *reality check axioms*, require that observations do not contradict the agent's expectations. Note that these rules establish a strict one-to-one correspondence between observations and expectations on fluents, while the correspondence between observed occurrences of actions and expected occurrences is one-directional. This is a reflection of the assumption that exogenous actions may occur undetected. It is not difficult to see that Π_r, together with the domain encoding, can be used to detect discrepancies between recorded

[2] One may also use $obs(f, i)$ and $\neg obs(f, i)$, but the representation we adopt simplifies the writing of some rules.

history and expectations. That is, discrepancies exist if-and-only-if the program $\Pi_k \cup \Gamma_n \cup \Pi_r$ is inconsistent.

Going back to our example, the description of the initial state from Π_i can be rewritten in the form of observations as

$$\left\{ \begin{array}{l} obs(at(l_d), true, 0). \\ obs(pod_at(p, l_r), true, 0). \\ obs(carrying(p), false, 0). \end{array} \right\}$$

To accommodate the new fluents, the set of observations is expanded by

$$\{obs(charged,\ true, 0).\ obs(stuck, false, 0).\},$$

indicating that the battery is initially charged and the lift mechanism is in working order. Let Γ_1 consist of the above observations, together with

$$\{hpd(move(l_r), 0), obs(at(l_r), true, 1)\}.$$

That is, Γ_1 specifies that the robot moved to l_r and that it observed its final location to be l_r. Clearly, program $\Pi_k \cup \Gamma_1 \cup \Pi_r$ is consistent, which indicates that there are no discrepancies between expectations and observations. In fact, action $move(l_r)$ achieved its intended effect of moving the robot to l_r. Consider now recorded history

$$\Gamma_2 = \Gamma_1 \cup \left\{ \begin{array}{l} hpd(pick_up(p, l_r), 1). \\ obs(carrying(p), false, 2). \end{array} \right\}.$$

Now, program $\Pi_k \cup \Gamma_2 \cup \Pi_r$ is inconsistent. In fact, $pick_up(p, l_r)$ is expected to result in the robot carrying the pod, and thus there is a discrepancy between the agent's expectations and the observation $obs(carrying(p), false, 2)$.

If discrepancies are detected, the second step of the diagnostic task consists in finding explanations for them. This is accomplished by a program Π_d that defines the *explanation space* of the problem – a collection of sequences of exogenous actions that may have occurred, unobserved, in the past, and provides a justification of the unexpected observations. We call such a program *diagnostic module*. While many variants are possible, the simplest diagnostic module is defined by the choice rule:

$$\{occurs(E, I) : exogenous(E)\} \leftarrow 0 \leq I < n.$$

The rule intuitively states that any exogenous action E may have occurred at any past step. Relation *exogenous* is assumed to be defined over all possible exogenous actions. In our example, the relation is defined by the facts $\{exogenous(break).\ exogenous(run_low).\}$. Diagnoses are found by computing the answer sets of the program $\Pi_k \cup \Gamma_n \cup \Pi_r \cup \Pi_d$.

Going back to our example, $\Pi_k \cup \Gamma_2 \cup \Pi_r \cup \Pi_d$ has two answer sets, one containing the atom $occurs(break, 0)$ and the other one containing the atom $occurs(run_low, 0)$, corresponding to the two possible explanations that, while the agent was moving to l_r, either the lift mechanism broke or the battery ran low. More advanced diagnostic techniques can be implemented in a similar fashion. For instance, cardinality-minimal diagnoses can be found by extending the diagnostic module by a weak constraint [5]:

$$:\sim occurs(A, S).$$

which intuitively states that atoms of the form $occurs(A, S)$ should be avoided as much as possible in the solution.

4 ASP Planning in Multi-Agent Environments

The formalization in the previous section can be easily extended to deal with various problems in multi-agent environments (MAE). In these problems, the planning (or reasoning) activity can be carried out either by one system (a.k.a. *centralized planning*) or multiple systems (a.k.a. *distributed planning*). Next, we will discuss the use of ASP in these settings. For simplicity of presentation, we will use a simplified version of the Kiva application from Section 2 as shown in Figure 3. In this example, we have two robots r_1 and r_2, and five pods $p_1, \ldots, p_5$, where each p_i is located in l_i. Initially, r_1 is at l_2, r_2 is at l_4, and neither robot carries anything.

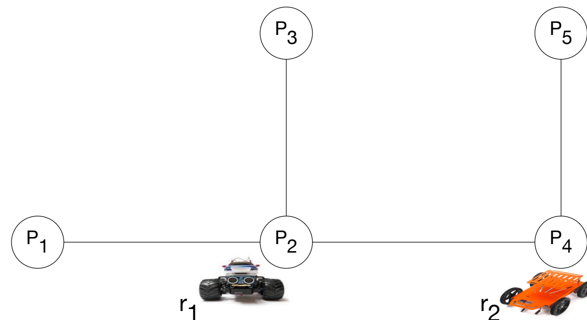

Fig. 3 A Multi-Agent Environment

4.1 Centralized Multi-Agent Planning

Centralized planning for multiple agents has been intensively studied from the beginning of multi-agent planning (see, e.g., [6]). In this subsection, we discuss the use of ASP in centralized planning for multiple agents. Similar to planning in a single-agent environment, we will assume that a language—in the spirit of $\mathcal{B}$ — exists for representing multi-agent domains. Different languages have been proposed (e.g., [4,17,19]) for this purpose. These languages extend the signature of $\mathcal{B}$ with a set of agents, and associate actions and fluents with agents. This association can be explicit, i.e., each action (or fluent) is associated with an agent; or implicit, i.e., the agent can be omitted from an action (or a fluent). In addition, it is also necessary to describe when parallel execution of actions is not feasible. This is achieved by introducing a new type of constraints of the form

$$\mathbf{impossible_if}(sa, \psi). \tag{11}$$

where sa is a set of actions and ψ is a set of literals.

With the above changes, a multi-agent domain appears similar to the single-agent domain and can be encoded using statements of the form (1)-(3) and (11). As an example, an explicit representation of the domain in Figure 3 can be obtained from the representation detailed in the previous section by

- adding the facts $agent(r_1)$ and $agent(r_2)$ for representing r_1 and r_2;
- replacing the fluent $carrying$ with $carrying(r, p)$ where r and p denote a robot and a pod, respectively;
- replacing the fluent $at(l)$ with $at(o, l)$ where o denotes an object (robot or pod) and l denotes a location, respectively;
- replacing the action $pick_up$ with $pick_up(r, p, l)$ where r, p, and l denote a robot, a pod, and a location, respectively;
- replacing the action $move(l)$ with $move(r, l)$ where r and l denote a robot and a location, respectively.

The effect of action $pick_up(r, p, l)$ is described by:

$$\mathbf{causes}(pick_up(r, p, l), carrying(r, p), \{\}). \tag{12}$$

Its executability conditions are captured by

$$\begin{array}{l}\mathbf{impossible_if}(pick_up(r, p, l), \{\neg at(r, l)\}).\\ \mathbf{impossible_if}(pick_up(r, p, l), \{\neg at(p, l)\}).\\ \mathbf{impossible_if}(pick_up(r, p, l), \{carrying(r, p')\}).\end{array} \tag{13}$$

Similarly, given a pair l, l' of connected locations, the effect of action $move(l')$ is formalized by:

$$\mathbf{causes}(move(r, l'), at(r, l'), \{at(l)\}). \tag{14}$$

The static law that formalizes the fact that a robot can only be at a single location at a time is as follows.

$$\mathbf{if}(\neg at(r, l'), \{at(r, l)\}). \tag{15}$$

for every $l \neq l'$.

Besides the above statements, additional constraints are also possible. For example, the state constraint, for $r \neq r'$,

$$\mathbf{if}(\perp, \{at(r, l), at(r', l)\}).$$

indicates that no two robots can be at the same location at the same time. The constraint

$$\begin{array}{l}\mathbf{impossible_if}(\{move(r, l), move(r', l')\},\\ \qquad\qquad\qquad \{at(r, l'), at(r', l)\}).\end{array} \tag{16}$$

for $r \neq r'$ and every pair of connected locations l, l', is an example of constraint of the form (11) preventing two robots from swapping positions in a single move.

A multi-agent planning problem $\mathcal{P}_m$ can be encoded with a multi-agent domain $\mathcal{D}_m$, the initial state of the world, and the set of goals for the agents in $\mathcal{P}_m$. The methodology described in Section 3 can be adapted to solve multi-agent planning problems as follows.

- Statements of the forms (1)-(3) are translated into their ASP encodings as described in Section 3; for example, a statement of the form (2) such as (14) is translated into the rule

$$\begin{array}{rl} holds(at(R, L'), I+1) \leftarrow & connected(L, L'),\\ & holds(at(R, L), I),\\ & occurs(move(R, L'), I).\end{array}$$

- Each statement of the form (11) is mapped to an ASP constraint. For instance, (16) is mapped into

$$\begin{array}{r}\leftarrow occurs(move(R, L), I), occurs(move(R', L'), I),\\ holds(at(R, L'), I), holds(at(R', L), I).\end{array}$$

Let us denote the ASP encoding of a multi-agent domain $\mathcal{D}_m$ by $\Pi_{\mathcal{D}_m}$. Centralized planning for agents in $\mathcal{P}_m$ can be achieved by combining Π_k (the set of ground rules of $\Pi_{\mathcal{D}_m}$ whose time step is at most k), the rules Π_p (the planning module), Π_i (the initial state of the world), and Π_g (goal satisfaction checking). Each answer set of $\Pi_k \cup \Pi_i \cup \Pi_g \cup \Pi_p$ represents a solution for $\mathcal{P}_m$.

One can see that, if robot r_2 needs to carry pod p_3 to l_1 and r_1 needs to carry p_2 to l_5, the program

$\Pi_6 \cup \Pi_i \cup \Pi_g \cup \Pi_p$ can be used to generate the following plans for the two robots

$$\begin{array}{l} occurs(pick_up(r_1, p_2, l_2), 0), \\ occurs(move(r_1, l_1), 1), occurs(move(r_2, l_2), 1), \\ occurs(move(r_1, l_2), 2), occurs(move(r_2, l_3), 2), \\ occurs(move(r_1, l_4), 3), occurs(pick_up(r_2, p_3, l_3), 3), \\ occurs(move(r_1, l_5), 4), occurs(move(r_2, l_2), 4), \\ occurs(drop_off(r_1, p_2, l_5), 5), occurs(move(r_2, l_1), 5), \\ occurs(drop_off(r_2, p_3, l_1), 6) \end{array} \quad (17)$$

4.2 Distributed Planning

A main drawback of centralized planning is that it cannot exploit the structural organization of agents (e.g., hierarchical organization of agents) in the planning process. Distributed planning has been proposed as an alternative to centralized planning that aims at exploiting the independence between agents and/or groups of agents. We discuss distributed planning in two settings: fully collaborative agents and non-/partially-cooperative agents.

4.2.1 Fully Collaborative Agents

When agents are fully collaborative, a possible way to exploit structural relationships between agents is to allow each group of agents to plan for itself (e.g., using the planning system described in Section 3) and then employ a centralized post-planning process (a.k.a. the *controller/scheduler*) to create the joint plan for all agents. The controller takes the output of these planners—individual plans—and merges them into an overall plan. One of the main tasks of the controller is to resolve conflicts between individual plans. This issue arises because individual groups plan without knowledge of other groups (e.g., robot r_1 does not know the location of robot r_2). When the controller is unable to resolve all possible conflicts, the controller will identify plans that need to be changed and request different individual plans from specific individual groups.

Any implementation of distributed planning requires some communication capabilities between the controller and the individual planning systems. For this reason, a client-server architecture is often employed in the implementation of distributed planning. A client plans for an individual group of agents and the server is responsible for merging the individual plans from all groups. Although specialized parallel ASP solvers exist (e.g., [11,16]), there has been no attempt to use parallel ASP solvers in distributed planning. Rather, distributed planning using ASP has been implemented using a combination of Prolog and ASP, where communication between server and clients is achieved through Prolog-based message passing, and planning is done using ASP (e.g., [19]).

Observe that the task of resolving conflicts is not a straightforward one and can require multiple iterations with individual planner(s) before the controller can create a joint plan. Consider again the two robots in Figure 3. If they are to generate their own plans, then the first set of individual solutions can be:

$$\begin{array}{l} occurs(pick_up(r_1, p_2, l_2), 0), \\ occurs(move(r_1, l_4), 1), occurs(move(r_1, l_5), 2), \\ occurs(drop_off(r_1, p_2, l_5), 3) \end{array} \quad (18)$$

and

$$\begin{array}{l} occurs(move(r_2, l_2), 0), occurs(move(r_2, l_3), 1), \\ occurs(pick_up(r_2, p_3, l_3), 2), \\ occurs(move(r_2, l_2), 3), occurs(move(r_2, l_1), 4), \\ occurs(drop_off(r_2, p_3, l_1), 5) \end{array} \quad (19)$$

Obviously, a parallel execution of these two plans will result in a violation of the constraint stating that two robots cannot be at the same location at the same time. One can see that the controller needs to insert a few actions into both plans (e.g., r_1 must move to either l_1 or l_3 before moving to l_4).

Let $P_1, \ldots, P_n$ be the set of plans received by the controller. The feasibility of merging these plans into a single plan for all agents can be checked using ASP. Let π_k be the program that consists of $\Pi_k \cup \Pi_i \cup \Pi_g$ (described in sub-section 4.1), the set of facts of the form $occurs(a, t)$ in $P_1, \ldots, P_k$, and the following rules:

- a planning module, modified to

  $occ(A, I) \vee \neg occ(A, I) \leftarrow step(I).$

- for $1 \leq p \leq n$, a mapping of indices from 0 to k to indices used in $P_1, \ldots, P_n$,

  $1\{map(p, I, J) : step(J)\}1 \leftarrow step(I), I \leq \max_p .$
  $\leftarrow map(p, I, J), map(p, R, S), I < R, J > S.$

  where $\max_p$ is the maximal index in P_p. Intuitively, $map(p, i, j)$ indicates that the ith action in P_p should occur in the jth position in the joint plan. This mapping must conform to the order of action occurrences in P_p. The solid arrows in Figure 4 show a valid mapping of the five step plan P_p into the joint plan which are represented by the atoms $map(p, 1, 2)$,

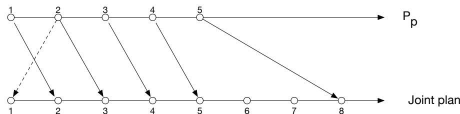

Fig. 4 Mapping of indices of P_p to indices of the joint plan

$map(p, 2, 3)$, $map(p, 3, 4)$, $map(p, 4, 5)$, and $map(p, 5, 8)$. On the other, if we replace the solid arrow from 2 to 3 with the dotted arrow from 2 to 1, the mapping consists of the atoms $map(p, 1, 2)$, $map(p, 2, 1)$, $map(p, 3, 4)$, $map(p, 4, 5)$, and $map(p, 5, 8)$ and is invalid.
– a rule ensuring that an atom $occurs(a, t) \in P_i$ must occur at the specified position:

$$\leftarrow occurs(a, t), map(i, t, j),\ not\ occ(a, j).$$

It can be checked that π_6 would generate an answer set corresponding to (17) if the inputs P_1 and P_2 are given by (18) and (19), respectively.

It is worth mentioning that merging two individual plans, which are computed separately, might be unsuccessful. For example, in the example represented by Figure 3, if robot r_1 needs to take pod p_3 to l_2 and robot r_2 needs to take pod p_5 to p_3 then two possible plans for these robots could be:

$$\begin{array}{l} occurs(move(r_1, l_3), 0), \\ occurs(pick_up(r_1, p_3, l_3), 1), \\ occurs(move(r_1, l_2), 2), \\ occurs(drop_off(r_1, p_3, l_2), 3) \end{array} \quad (20)$$

and

$$\begin{array}{l} occurs(move(r_2, l_5), 0), \\ occurs(pick_up(r_2, p_5, l_5), 1), \\ occurs(move(r_2, l_4), 2), \\ occurs(move(r_2, l_2), 3), \\ occurs(move(r_2, l_3), 4), \\ occurs(drop_off(r_2, p_5, l_3), 5) \end{array} \quad (21)$$

It is easy to check that there is no way to merge the two plans detailed in (20) and (21) into a single plan for the two robots to accomplish their goals. The main reason for this impossibility is that r_1 only moves between l_2 and l_3, which prevents r_2 to get to its destination.

4.2.2 Non/Partially-Collaborative Agents

Centralized planning or distributed planning with an overall controller is most suitable in applications with collaborative (or non-competitive) agents such as the robots in the Kiva application. In many applications, this assumption does not hold, e.g., agents may need to withhold certain private information and thus do not want to share their information freely; or agents may be competitive and have conflicting goals. In these situations, distributed planning as described in the previous sub-section is not applicable and planning will have to rely on a message passing architecture, e.g., via peer-to-peer communications. Furthermore, an online planning approach might be more appropriate. Next, we describe an ASP approach that is implemented centrally in [17] but could also be implemented distributedly.

In this approach, the planning process is interleaved with a negotiation process among agents. As an example, consider the robots in Figure 3 and assume that the robots can communicate with each other, but they cannot reveal their location. The following negotiation between r_2 and r_1 could take place:

– r_2 (to r_1): "can you (r_1) move out of l_2, l_3, and l_4?" (because r_2 needs to make sure that it can move to location l_2 and l_3). This can be translated to the formula $\varphi_1 = \neg at(r_1, l_2) \wedge \neg at(r_1, l_3) \wedge \neg at(r_1, l_4)$ sent from r_2 to r_1.
– r_1 (to r_2): "I can do so after two steps but I would also like for you (r_2) to move out of l_2, l_4, and l_5 after I move out of those places." This means that r_1 agrees to satisfy the formula sent by r_2 but also has some conditions of its own. This can be represented by the formula $\varphi_1 \supset \varphi_2 = \neg at(r_2, l_2) \wedge \neg at(r_2, l_4) \wedge \neg at(r_2, l_5)$.
– r_2 (to r_1): "that is good; however, do not move through l_4 to get out of the area."
– etc.

The negotiation will continue until either the agent accepts (or refutes) the latest proposal from the other agent. A formal ASP based negotiation framework (e.g., [18]) could be used for this purpose.

Observe that during a negotiation, none of the robots changes its location or executes any action. After a successful negotiation, each robot has some additional information to take into consideration in its planning. In this example, if the two robots agree after the second proposal by r_2, robot r_1 agrees to move out of l_2, l_3, and l_4 but should do so without passing by l_4; robot r_2 knows that he can have l_2, l_3, and l_4 for itself after sometime and also knows that it can stand at l_4 until r_1 is out of the requested area; etc. Note, however, that this is not yet sufficient for the two robots to achieve

their goals. To do so, they also need to agree on the timing of their moves. For example, r_1 can tell r_2 that l_2, l_3, and l_4 will be free after two steps; r_2 responds that, if it is the case, then l_2, l_4, and l_5 will be free after 2 steps; etc. This information will help the robots come up with plans for their own goals.

To the best of our knowledge, only a prototype implementation of the approach to interleaving negotiation and planning has been presented in [17]. It is also not implemented distributedly. We will next briefly describe the architecture of a possible implementation of this approach. For simplicity of the presentation, we assume that there are only two agents, named 1 and 2. Agent 1 (resp. 2) needs to solve the planning problem P_1 (resp. P_2) for itself. The two agents communicate through a *negotiation server* that facilitates the communication between them via a protocol.

Let us assume that the ASP encoding of the planning problem P_i $(i = 1, 2)$ is the program Π^i that consists of the modules $\Pi^i_k \cup \Pi^i_g \cup \Pi^i_i \cup \Pi^i_p$ as described in Section 4.1. In order for the agents to facilitate the negotiation between the two agents, we introduce a new type of atoms of the form

$$hyp(\ell, t) \tag{22}$$

which represents the assumption that literal ℓ (a fluent F or its negation $\neg F$) holds at the time step t. For $\ell = F$ or $\ell = \neg F$, we write $\overline{\ell}$ to denote $\neg F$ or F respectively.

Let Π^i_h be the ASP program consisting of the follow rules:

$$\{hyp(F, I), hyp(\neg F, I)\}1. \tag{23}$$

$$\leftarrow hyp(F, I), holds(F, I-1). \tag{24}$$

$$\leftarrow hyp(\neg F, I), \neg holds(F, I-1). \tag{25}$$

$$holds(F, I) \leftarrow hyp(F, I). \tag{26}$$

$$\neg holds(F, I) \leftarrow hyp(\neg F, I). \tag{27}$$

The rule (23) states that the agent can assume that F is true or false at the time step I. Intuitively, when an agent assumes the value v of a fluent F at a time step I, it means that the agent might need to negotiate with the other agent so that F has the value v at time step I.

Although the agent can assume that a fluent F can be true or false at anytime, it is reasonable to assume that the agent should only make the assumption when F has a different value at the time step $I-1$ (rules (24)-(25)). Observe that this conditions can be strengthened by allowing the agent to make an assumption about a fluent F *only if* the agent cannot change the value of F. For example, if the battery of the robot r_1 is insufficient for it to move out of the current location, it might need help (e.g., waiting for a service robot to bring a battery); the robot can make a plan to achieve its goal assuming that the battery is fully changed at time step 3.

The last two rules of Π^i_h indicate that if the agent assumes that F is true (resp. false) then the fluent is true (resp. false).

Intuitively, Π^i_h encodes the fact that agent i can make assumptions about fluent values that can be changed by the other agent. For instance, if the robot 1 in Figure 3 assumes that robot 2 is not at l_4 at step 1, then this fact is represented by the atom $hyp(\neg at(r_2, l_4), 1)$.

Let $\Pi^i_n = \Pi^i \cup \Pi^i_h$ and S be an answer set of Π^i_n. It is easy to see that, because of the rule (23), S can contain zero, one, or many atoms of the form $hyp(\ell, t)$, referred to as assumptions hereafter. Rules (24)—(27) indicate that the values of fluents mentioned in these assumptions change from time step $I-1$ to I but it is possible that the changes might have not been caused by the execution of an actions by the agent i. As such, agent i might need to negotiate with other agents so that the values of the fluents mentioned in the assumptions are exactly as they have been assumed. Under this assumption, it can be shown that every answer set of Π^i_n contains a plan for agent i to achieve its goal.

5 Conclusions

In this paper, we have discussed the main ideas of answer set planning in single- and multi-agent environments. We presented a systematic translation of dynamic domains into logic programs, which can be used for planning or diagnosis. We showed how the encoding of dynamic domains in single-agent environments can be adapted for planning in multi-agent environments. We illustrated the issues of answer set planning in these settings and their solutions using a well-known problem domain, the Kiva robot system. Finally, we would like to note that the proposed planning, diagnosis, and reasoning technologies for single-agent system have been implemented in a real-world application [2,3]; on the other hand, the proposed technologies for multi-agent planning have not been implemented in a real-world application. A preliminary implementation in a Kiva-simulated environment can be found in [14].

Tran Cao Son is a Professor at the Computer Science Department of New Mexico State University, Las Cruces. He is interested in answer set programming, commonsense reasoning, automated planning, reasoning about actions and change, multi-agent systems, and the use of logical representation in practical applications. With his students, he has developed the planning system called CpA(H) that was the best conformant planner in the International Planning Competition in 2008.

Marcello Balduccini is an Assistant Professor at the Department of Decision & System Sciences of Saint Joseph's University. His research includes automated reasoning in environments with non-linear dynamics; planning, scheduling, and optimization; information retrieval; and cyber-analytics, threat modeling and mitigation. He was a 2016 Data Fellow of the National Consortium for Data Science and is currently a Pedro Arrupe Center Research Fellow.

References

1. Balduccini, M., Gelfond, M.: Diagnostic reasoning with A-Prolog. Journal of Theory and Practice of Logic Programming (TPLP) **3**(4–5), 425–461 (2003)
2. Balduccini, M., Gelfond, M.: Logic Programs with Consistency-Restoring Rules. In: International Symposium on Logical Formalization of Commonsense Reasoning, AAAI Spring Symposium Series, pp. 9–18 (2003)
3. Balduccini, M., Gelfond, M., Watson, R., Nogueira, M.: The USA-Advisor: A Case Study in Answer Set Planning. In: Lectures Notes in Artificial Intelligence (Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'01), vol. 2173, pp. 439–442. Springer-Verlag (2001)
4. Baral, C., Son, T.C., Pontelli, E.: Reasoning about Multi-agent Domains Using Action Language $\mathcal{C}$: A Preliminary Study. In: J. Dix, M. Fisher, P. Novák (eds.) Computational Logic in Multi-Agent Systems - 10th International Workshop, CLIMA X, Hamburg, Germany, September 9-10, 2009, Revised Selected and Invited Papers, *Lecture Notes in Computer Science*, vol. 6214, pp. 46–63. Springer (2010)
5. Buccafurri, F., Leone, N., Rullo, P.: Adding Weak Constraints to Disjunctive Datalog. In: Proceedings of the 1997 Joint Conference on Declarative Programming APPIA-GULP-PRODE'97 (1997)
6. Durfee, E.: Distributed Problem Solving and Planning. In: Muliagent Systems (A Modern Approach to Distributed Artificial Intelligence), pp. 121–164. MIT Press (1999)
7. Gelfond, M., Lifschitz, V.: Action Languages. Electronic Transactions on Artificial Intelligence **3**(6) (1998)
8. Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL — the Planning Domain Definition Language. Version 1.2. Tech. Rep. CVC TR98003/DCS TR1165, Yale Center for Comp, Vis and Ctrl (1998)
9. Hayes, P.J., McCarthy, J.: Some Philosophical Problems from the Standpoint of Artificial Intelligence. In: B. Meltzer, D. Michie (eds.) Machine Intelligence 4, pp. 463–502. Edinburgh University Press (1969)
10. Kautz, H., Selman, B.: Planning as satisfiability. In: Proceedings of ECAI-92, pp. 359–363 (1992)
11. Le, H.V., Pontelli, E.: An Investigation of Sharing Strategies for Answer Set Solvers and SAT Solvers. In: J.C. Cunha, P.D. Medeiros (eds.) Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings, *Lecture Notes in Computer Science*, vol. 3648, pp. 750–760. Springer (2005)
12. Lifschitz, V.: Answer Set Programming and Plan Generation. Artificial Intelligence **138**(1–2), 39–54 (2002)
13. Marek, V., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: The Logic Programming Paradigm: a 25-year Perspective, pp. 375–398 (1999)
14. Nguyen, V.D., Obermeier, P., Son, T.C., Schaub, T., Yeoh, W.: Generalized Target Assignment and Path Finding Using Answer Set Programming. In: IJCAI, pp. 1216–1223 (2017)
15. Niemelä, I.: Logic Programming with Stable Model Semantics as a Constraint Programming Paradigm. Annals of Mathematics and Artificial Intelligence **25**(3,4), 241–273 (1999)
16. Schneidenbach, L., Schnor, B., Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Experiences Running a Parallel Answer Set Solver on Blue Gene. In: M. Ropo, J. Westerholm, J. Dongarra (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5759, pp. 64–72. Springer (2009)
17. Son, T., Pontelli, E., Sakama, C.: Logic Programming for Multiagent Planning with Negotiation. In: P.M. Hill, D.S. Warren (eds.) Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5649, pp. 99–114. Springer (2009)
18. Son, T.C., Pontelli, E., Nguyen, N., Sakama, C.: Formalizing Negotiations Using Logic Programming. ACM Trans. Comput. Log. **15**(2), 12 (2014)
19. Son, T.C., Pontelli, E., Nguyen, N.H.: Planning for Multiagent Using ASP-Prolog. In: J. Dix, M. Fisher, P. Novák (eds.) Computational Logic in Multi-Agent Systems - 10th International Workshop, CLIMA X, Hamburg, Germany, September 9-10, 2009, Revised Selected and Invited Papers, *Lecture Notes in Computer Science*, vol. 6214, pp. 1–21. Springer (2010)
20. Wurman, P., D'Andrea, R., Mountz, M.: Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. AI Magazine **29**(1), 9–20 (2008)