Online appendix for the paper

Hybrid Conditional Planning using Answer Set Programming

published in Theory and Practice of Logic Programming

Ibrahim Faruk Yalciner, Ahmed Nouman, Volkan Patoglu, and Esra Erdem
Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey
{*fyalciner,ahmednouman,vpatoglu,esraerdem*}*@sabanciuniv.edu*

submitted 27 April 2017; revised 18 July 2017; accepted 31 August 2017

Appendix A ASP Programs

We consider ASP programs (i.e., nondisjunctive HEX programs (Eiter et al. 2005)) that consist of rules of the form

$$\mathit{Head} \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$$

where $n \geq m \geq 0$, *Head* is an atom or $\bot$, and each A_i is an atom or an external atom. HEX programs can be extended by allowing classical negation $\neg$ in front of atoms. A rule is called a *fact* if $m = n = 0$ and a *constraint* if *Head* is $\bot$.

An external atom is an expression of the form $\&g[y_1, \ldots, y_k](x_1, \ldots, x_l)$ where $y_1, \ldots, y_k$ and $x_1, \ldots, x_l$ are two lists of terms (called input and output lists, respectively), and $\&g$ is an external predicate name. Intuitively, an external atom provides a way for deciding the truth value of an output tuple depending on the extension of a set of input predicates. External atoms allow us to embed results of external computations into ASP programs. They are usually implemented in a programming language of the user's choice, like C++. For instance, the following rule

$$\begin{aligned}\bot \leftarrow\ & \mathit{at}(r, x_1, y_1, t), \mathit{goto}(r, x_2, y_2, t),\\ & \mathit{not}\ \&\mathit{path\_exists}[x_1, y_1, x_2, y_2]()\end{aligned}$$

is used to express that, at any step t of the plan, a robot r cannot move from (x_1, y_1) to (x_2, y_2) if there is no collision-free trajectory between them. Here collision check is done by the external predicate $\&\mathit{path\_exists}$ implemented in C++, utilizing the bidirectional RRT (Rapidly Exploring Random Trees) (Kuffner Jr and LaValle 2000) as in the OMPL (Sucan et al. 2012) library.

In addition to the classical negation, ASP considers another sort of negation: "negation as failure" (denoted *not*, and often called "default negation"). Intuitively, $\neg p$ expresses that we know that p does not hold, whereas *not* p expresses that we do not know that p holds. This second sort of negation empowers ASP to express our assumptions (called "defaults") when we do not have complete knowledge. For instance, we can express that "every object o in a kitchen is assumed to be on the counter unless they are known to be on the table" by

the following ASP rule

$$\mathit{at}(o, \mathit{Counter}, t) \leftarrow \mathit{not}\ \mathit{at}(o, \mathit{Table}, t).$$

In ASP, we use special constructs to express nondeterministic choices, cardinality constraints, and optimization statements. For instance, the following ASP rule (called a 'choice rule")

$$\{\mathit{sense}(\mathit{at}(o), t)\}$$

contains the choice expression $\mathit{sense}(\mathit{at}(o), t)$ in the head. For every object o and time step t, this choice expression describes a subset of the set $\{\mathit{sense}(\mathit{at}(o), t)\}$. Therefore, this rule expresses that, for every object o and time step t, the action of sensing that the location of o may occur at t.

The following ASP constraint (called a "cardinality constraint")

$$\leftarrow 2\{\mathit{atRob}(l, t) : \mathit{robloc}(l)\}$$

contains the cardinality expression $2\{\mathit{atRob}(l, t) : \mathit{robloc}(l)\}$ in the body. For every time step t, this expression describes subsets of the set $\{\mathit{atRob}(l, t) : \mathit{robloc}(l)\}$ whose cardinality is at least 2. Therefore, this constraint is used to ensure that, for every time step t, the robot cannot be at two different locations at t.

The following ASP expression (called an "optimization statement")

$$\#\mathit{minimize}\ [\ \mathit{cost}(r, c, t) : \mathit{robot}(r) = c\ ] \tag{A1}$$

is used to minimize the sum of all costs c of robotic actions performed in a plan, where costs of actions performed by robot r at time step t are defined by atoms of the form $\mathit{cost}(r, c, t)$.

A version of external atoms (where predicate names are not passed as input arguments), and all the constructs described above are supported by the ASP solver CLINGO used as part of HCP-ASP. For more information about the input language of CLINGO, we refer the reader to CLINGO's manual: `https://sourceforge.net/projects/potassco/files/guide/2.0/guide-2.0.pdf` (June 18, 2017).

Appendix B Hybrid Classical Planning in ASP

Classical planning considers complete knowledge (under full observability) over a dynamic domain. A dynamic domain is defined by means of fluent constants and (actuation) action constants: A world state can be characterized by an interpretation of fluent constants, whereas an action is characterized by an interpretation of action constants. Then, dynamic domains under full observability can be modeled as transition systems — directed graphs where nodes denote the world states of the domain, and edges denote the transitions between these states caused by occurrences or nonoccurrences of actions in that domain. Note that such transition systems respect the Markov principle (i.e., actions do not have delayed effects).

Given an initial state s_0, goal conditions G, and a nonnegative integer k, classical planning asks for a sequence $P = \langle a_0, a_1, ..., a_{k-1}\rangle$ of actions, which characterizes a path $X = \langle s_0, s_1, ..., s_{k-1}, s_k\rangle$ from s_0 to a goal state s_k in this transition system such that every

edge (s_i, s_{i+1}) in the path characterizes an occurrence of action a_i. This sequence P of actions is called a plan, with makespan k. The history $H = \langle s_0, a_0, s_1, a_1, ..., s_{k-1}, a_{k-1}, s_k \rangle$ of a plan describes the path X by depicting also the relevant actions. Classical planning is NP-complete for polynomially bounded plans (Erol et al. 1995).

For robotic domains, to ensure executability of classical plans, e.g., to ensure continuous movements along collision-free trajectories, further checks need to be performed. This requires combining classical planning with feasibility checks. We call this problem hybrid classical planning. As discussed in Section 2 of the main paper, there are different methods of integrating planning with feasibility checks. We consider solving hybrid classical planning problems in ASP, using HEX programs, with respect to appropriate methods of integration (Erdem et al. 2016a).

Representing hybrid action domains in ASP We formalize hybrid dynamic domains in ASP, under full observability and as a transition system, in the spirit of (Erdem et al. 2016). Such a description of a hybrid dynamic domain in ASP relies on three important forms of rules.

For a formula H and an index i (for time step), let us denote by $H(i)$ the formula obtained from H by replacing every atom q by $q(i)$. Intuitively, $H(i)$ expresses that the formula H holds at time step i.

Effect rules: Direct outcomes of actions are expressed with *effect rules* of the form

$$E(i+1) \leftarrow A(i), F(i) \tag{B1}$$

where A is a conjunction of action atoms, E is a fluent literal, and F is conjunction of fluent literals. This rule indicates that if the actions in A are executed at time step i where F holds then at the next state E holds. For instance, the following effect rule describes an effect of a "move" action of a mobile robot r navigating to a location l at time step i:

$$at(r, l, i+1) \leftarrow move(r, l, i).$$

It expresses that, as a direct effect of this action, the location of robot r changes to l at the next time step $i+1$.

Precondition rules: Preconditions of actions are expressed with *precondition rules* of the form

$$\leftarrow A(i), F(i), \mathit{not}\ G(i). \tag{B2}$$

where A is a conjunction of action atoms, and F and G are conjunctions of fluent literals. The precondition rule above expresses that, to execute an action A at time step i at a state where F holds, the action's preconditions G must hold. For instance, according to the following precondition rule

$$\leftarrow move(r, l, i), at(r, l, i)$$

action $move(r, l)$ is possible if the robot is not already at the destination location l.

Hybrid rules: A *hybrid rule* is a rule where the right hand side of $\leftarrow$ includes external atoms. External atoms (Eiter et al. 2005) are not fluent or action constants; their truth values are computed externally (out of ASP).

These rules are important for robotics applications since low-level feasibility checks

for each action can be computed externally and then integrated into transition system description by means of external atoms. For instance, the following hybrid precondition rule ensures that, at time step i, a robot r can move from its current location x to its destination location l if there is a collision-free trajectory between them:

$$\leftarrow at(r, x, i), move(r, l, i), \textit{not } \&move\_is\_feasible[r, l, x]().$$

The external atom $\&move\_is\_feasible[r, l, x]()$ passes r, l, x as inputs to the external computation (e.g., a Python program) that calls a motion planner to check the existence of a collision free trajectory for r from x to l, and then returns the result of the computation to the precondition rule.

Defining planning problems in ASP Once the domain is described in ASP, we can specify the initial state of the world in different ways, e.g., by a set of facts, like

$$\textit{atRob}(\textit{Table}, 0),$$

by choice rules accompanied with constraints, like

$$\{\textit{atRob}(l, 0)\}$$
$$\leftarrow \textit{not atRob}(\textit{Table}, 0),$$

or by assumptions, like

$$\textit{atRob}(\textit{Table}, 0) \leftarrow \neg\, \textit{atRob}(\textit{Table}, 0).$$

We can describe the goal conditions by a set of rules, and add constraints to ensure their reachability, as in the examples below:

$$\textit{goal} \leftarrow \textit{atRob}(\textit{Table}, t)$$
$$\leftarrow \textit{not goal}.$$

Then, with a domain description and planning problem description, plans of actuation actions can be computed using the ASP solver CLINGO.

Appendix C An Example for Hybrid Conditional Planning: Kitchen Domain

As a benchmark for hybrid conditional planning, we consider a dynamic service robotics scenario, where a bimanual mobile manipulator is responsible for setting up a kitchen table, as depicted in Figure C 1. This domain is introduced in (Nouman et al. 2016): "The mobile manipulator can navigate around the kitchen to pick up and place objects as long as collision free trajectories exist. Kitchenware, such as mugs, spoons, knives, plates may be found in cabinets or may be left on other flat surfaces, such as counter tops or shelves. In the kitchen, there also exists a faucet to clean kitchenware as required. Finally, there is a kitchen table, where the proper kitchenware must be placed on to comply with table setting etiquette."

In this domain, there are four actuation actions: $goto$, $pickup$, $placeon$ and $clean$. For the feasibility of these actions, existence of a collision-free trajectory is implemented based on OMPL (Sucan et al. 2012) (to be used as a precondition of $goto$ action), while reachability, graspability and inverse kinematics checks are implemented based on OPENRAVE (Diankov 2010) (to be used as preconditions of $pickup$ and $placeon$ actions).

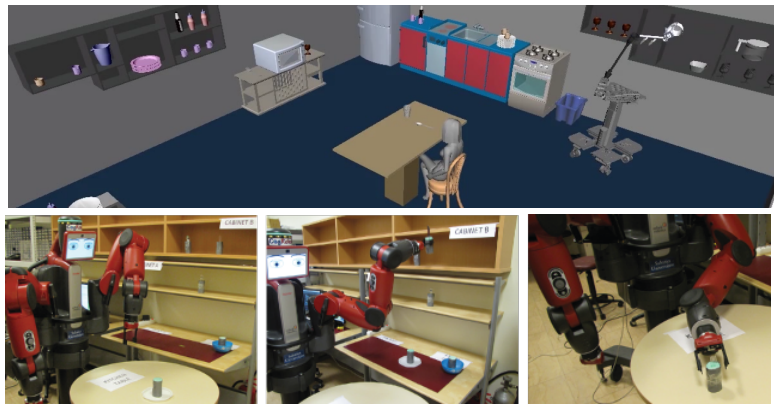

Fig. C 1: The robot is manipulating a bowl from Cabinet B in dynamic simulation (top) and a fork during physical implementation (bottom).

This domain contains three types of uncertainties. First, the person might have different food preferences (e.g., soup, pizza, salad), which can only be revealed when directly communicated with the user during plan execution. Second, the locations of some kitchenware may not be known by the robot during the planning phase. These locations can be reliably gathered only if the robot actively searches for these objects when it needs to use them. Third, the cleanliness/dirtiness of the objects may not be known in advance for sure. Along these lines, three sensing actions are considered: $checkFoodType$, $checkLoc$ and $checkisClean$.

An ASP description of this domain presented to HCP-ASP, in the input language of CLINGO, is provided in Figures C 2–C 8. The description consists of three parts, appropriate for incremental grounding (and thus incremental computation of plans), and preceded by the expressions `#program base`, `#program step(t)`, and `#program check(t)`. Intuitively, the first part describes the domain predicates and the general knowledge about the world at time step 0; so it is instantiated once. The second part describes the states at time step `t` and transitions of the world for time steps `t-1` and `t`. Here, the value of `t` increases one by one starting from `1` until a plan is found; so this part is instantiated incrementally. This incremental grounding guarantees finding a plan with a minimum length. The direct/indirect effects of actions, inertia, and action occurrences are defined in the second part. The third part describes all the constraints to be checked at every time step `t`. Here, the value of `t` also increases one by one starting from `0` until a plan is found; so this part is instantiated incrementally as well. The preconditions of actions, state constraints, transition constraints, concurrency constraints, and the goal are defined in the third part.

The actuation actions are defined as described in Appendix B, whereas the sensing actions are defined as described in Section 5 of the main paper.

Appendix D Experimental comparison of HCP-ASP with ASCP

As discussed in Section 2 of the main paper, although not compilation-based, the offline non-hybrid conditional planner ASCP also uses ASP to compute conditional plans. Therefore, we have compared these two ASP-based conditional planners, over one of the benchmarks of ASCP: Bomb in the Toilet with Sensing Actions (BTS) (Weld et al. 1998). In this domain, it has been alarmed that there is a bomb in the toilet. There are m suspicious

packages, and one of them contains the bomb. The bomb can be defused by dunking the package with bomb into the toilet; dunking a package clogs the toilet and flushing the toilet unclogs it. The existence of a bomb in a package can be sensed by a metal detector, by a dog to sniff the bomb, or by an x-ray machine. Initially, the bomb is armed and the toilet is not clogged; the goal is that the bomb is disarmed and the toilet is not clogged.

We have experimented with ASCP using the ASP encoding of BTS (`bt_3sa.smo`)[1] transformed from $\mathcal{A}^c_K$ with the ASP solver CLINGO. The results of experiments for $m = 10, 11, .., 17$ are shown in Table D 1; the computation time for ASCP does not contain the time for transformation.

According to these results, finding a tree with one call of CLINGO (using ASCP) takes more time, compared to computing and combining the branches of the tree in parallel (using HCP-ASP). For instance, for $m = 17$, it takes more than an hour to compute a tree with ASCP whereas it takes about a second for HCP-ASP.

Table D 1: *Comparison of ASCP with HCP-ASP.*

No of Package	Max Depth	Tree Size	ASCP Time [sec]	HCP-ASP Time (parallel with 20 threads) [sec]
10	10	19	21	0.4
11	11	21	21	0.5
12	12	23	46	0.6
13	13	25	433	0.5
14	14	27	406	0.7
15	15	29	1953	0.9
16	16	31	2896	0.8
17	17	33	5807	1.0

References

DIANKOV, R. 2010. Automated construction of robotic manipulation programs. Ph.D. thesis, Carnegie Mellon University, Robotics Institute.

EITER, T., IANNI, G., SCHINDLAUER, R., AND TOMPITS, H. 2005. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In *Proc. of IJCAI*. 90–96.

ERDEM, E., GELFOND, M., AND LEONE, N. 2016. Applications of answer set programming. *AI Magazine 37,* 3, 53–68.

ERDEM, E., PATOGLU, V., AND SCHÜLLER, P. 2016a. A systematic analysis of levels of integration between high-level task planning and low-level feasibility checks. *AI Commun. 29,* 2, 319–349.

EROL, K., NAU, D. S., AND SUBRAHMANIAN, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artif. Intell. 76,* 1–2, 75–88.

KUFFNER JR, J. AND LAVALLE, S. 2000. RRT-connect: An efficient approach to single-query path planning. In *Proc. of ICRA*. 995–1001.

NOUMAN, A., YALCINER, I. F., ERDEM, E., AND PATOGLU, V. 2016. Experimental evaluation of hybrid conditional planning for service robotics. In *Proc. of ISER*.

SUCAN, I. A., MOLL, M., AND KAVRAKI, L. E. 2012. The open motion planning library. *Robotics & Automation Magazine, IEEE 19,* 4, 72–82.

WELD, D. S., ANDERSON, C. R., AND SMITH, D. E. 1998. Extending graphplan to handle uncertainty & sensing actions. In *Proc. of AAAI*. 897–904.

[1] `https://www.cs.nmsu.edu/~tson/ASPlan/Sensing/test/bt_3sa.smo`.

```
#include <incmode>.

% maximum plan length
#const step_limit=40.

#program base.

% robotic manipulators
manip(manip_R;manip_L).

% objects and their types
object(bowl_0). object(bowl_1). object(spoon_0).  object(spoon_1).
object(fork_0). object(fork_1).  object(knife_0). object(knife_1).
object(plate_0).  object(plate_1).  object(wineGlass_0).
object(wineGlass_1). object(waterGlass_0). object(waterGlass_1).

type(bowl,bowl_0). type(bowl,bowl_1). type(spoon,spoon_0).
type(spoon,spoon_1). type(fork,fork_0). type(fork,fork_1).
type(wineGlass,wineGlass_0). type(wineGlass,wineGlass_1).
type(knife,knife_0). type(knife,knife_1).
type(plate,plate_0). type(plate,plate_1).
type(waterGlass,waterGlass_0). type(waterGlass,waterGlass_1).

% number of objects in each type
type_T(T) :- type(T,O).
type_no(T,N) :- #count{O: type(T,O)}=N, type_T(T).

% possible locations of objects
objlocnotonhold(extratable; cabinetA; cabinetB; faucet; table).
objloc(L) :- objlocnotonhold(L).
objloc(M) :- manip(M).
#const objloc_size= 7.

% possible locations of the robot
robloc(L):- objlocnotonhold(L).

% food types
food(soup; pizza; chicken).

% which utensils are expected to be on the table for which food type
expected_T(soup,bowl). expected_T(soup,spoon).
expected_T(soup,waterGlass).
expected_T(pizza,fork). expected_T(pizza,knife).
expected_T(pizza,plate). expected_T(pizza,wineGlass).
expected_T(chicken,fork). expected_T(chicken,knife).
expected_T(chicken,plate). expected_T(chicken,bowl).
expected_T(chicken,wineGlass).
unexpected(F,O) :- type(T,O), not expected_T(F,T), food(F).
```

Fig. C 2: Kitchen table setting domain presented to HCP-ASP, in the input language of CLINGO: Part 1 – Domain predicates.

```
% time_min = 0

% ramifications for t=0

% if an object is located at L1 then it is not anywhere else
-atObj(O,L,time_min) :- atObj(O,L1,time_min), object(O), objloc(L),
  objloc(L1), L!=L1.
% if a food type is requested, no other food type can be requested
-requested(F,time_min) :- requested(F1,time_min), food(F),
  food(F1), F!=F1.

% if an object's location is unknown then it cannot be on
%   the robot's hand either.
-atObj(O,M,time_min) :- {atObj(O,L,time_min):objloc(L)}0,
  manip(M), object(O).

% if an object O is not at any of the object locations (except L),
%   then it should be at L
atObj(O,L,time_min) :-
  objloc_size-1{-atObj(O,L1,time_min):objloc(L1),L1!=L}objloc_size-1,
  objloc(L), object(O).

% by default the objects are not on the table
-atObj(O,table,time_min) :- not atObj(O,table,time_min), object(O).

% actions are initially exogenous
{move(L,time_min)} :- robloc(L).
{pickUp(M,O,time_min)} :- manip(M), object(O).
{place(M,time_min)} :- manip(M).
{clean(M,time_min)} :- manip(M).
{sense(cleanObj(O),time_min)} :- object(O).
{sense(locObj(O),time_min)} :- object(O).
{sense(food_request,time_min)}.
```

Fig. C 3: Kitchen table setting domain presented to HCP-ASP, in the input language of CLINGO: Part 2 – State constraints and possible action occurrences initially.

```
#program step(t).

% inertia for fully observed fluents
%   (with uniqueness and existence constraints)
{atRob(L,t+time_min)} :- atRob(L,t+time_min-1), robloc(L).

% inertia for partially observed fluents
atObj(O,L,t+time_min) :- not -atObj(O,L,t+time_min),
  atObj(O,L,t+time_min-1), object(O), objloc(L).
-atObj(O,L,t+time_min) :- not atObj(O,L,t+time_min),
  -atObj(O,L,t+time_min-1), object(O), objloc(L).

isclean(O,t+time_min) :- not -isclean(O,t+time_min),
  isclean(O,t+time_min-1), object(O).
-isclean(O,t+time_min) :- not isclean(O,t+time_min),
  -isclean(O,t+time_min-1), object(O).

requested(F,t+time_min) :- not -requested(F,t+time_min),
  requested(F,t+time_min-1), food(F).

% ramifications for t>0

% if an object is located at L1 then it is not anywhere else
-atObj(O,L,t+time_min) :- atObj(O,L1,t+time_min), object(O),
  objloc(L), objloc(L1), L!=L1.

% if a food type is requested, no other food type can be requested
-requested(F,t+time_min) :- requested(F1,t+time_min), food(F),
  food(F1), F!=F1.

% if an object's location is unknown then it cannot be on
%   the robot's hand either.
-atObj(O,M,t+time_min) :- {atObj(O,L,t+time_min):objloc(L)}0,
  manip(M), object(O).

% if an object O is not at any of the object locations (except L),
%   then it should be at L
atObj(O,L,t+time_min) :-
  objloc_size-1{-atObj(O,L1,t+time_min):objloc(L1),L1!=L}objloc_size-1,
  objloc(L), object(O).
```

Fig. C 4: Kitchen table setting domain presented to HCP-ASP, in the input language of CLINGO: Part 3 – Inertia and ramifications.

```
% action occurrences
{move(L,t+time_min)} :- robloc(L).
{pickUp(M,O,t+time_min)} :- manip(M), object(O).
{place(M,t+time_min)} :- manip(M).
{clean(M,t+time_min)} :- manip(M).
{sense(cleanObj(O),t+time_min)} :- object(O).
{sense(locObj(O),t+time_min)} :- object(O).
{sense(food_request,t+time_min)}.

% direct effects of actions
% move(R,L)
atRob(L,t+time_min) :- move(L,t+time_min-1), robloc(L).

% pickUp(R,M,O)
atObj(O,M,t+time_min) :- pickUp(M,O,t+time_min-1),
  object(O), manip(M).

% place(robot,man,loc)
atObj(O,L,t+time_min) :- place(M,t+time_min-1),
  atObj(O,M,t+time_min-1), atRob(L,t+time_min-1),
  manip(M), object(O), objlocnotonhold(L).

% clean(R,M)
isclean(O,t+time_min) :- clean(M,t+time_min-1),
  atObj(O,M,t+time_min-1), manip(M), object(O).

% sensing actions
1{isclean(O,t+time_min);-isclean(O,t+time_min)}1 :-
  sense(cleanObj(O),t+time_min-1), object(O).
1{atObj(O,L,t+time_min):objlocnotonhold(L)}1 :-
  sense(locObj(O),t+time_min-1), object(O).
1{requested(F,t+time_min):food(F)}1 :-
  sense(food_request,t+time_min-1).
```

Fig. C 5: Kitchen table setting domain presented to HCP-ASP, in the input language of CLINGO: Part 4 – Action occurrences and their direct effects.

```
#program check(t).

% uniqueness and existence constraints
:- 2{atRob(L,t+time_min): robloc(L)}.
:- {atRob(L,t+time_min): robloc(L)}0.

:- 2{atObj(O,M,t+time_min):object(O)}, manip(M).

% preconditions of actions
% the robot cannot move to L if it is already there
:- move(L,t+time_min), atRob(L,t+time_min), robloc(L).

% the robot cannot pick up an object
%  if it is already holding one
pickUpRM(M,t+time_min) :-  pickUp(M,O,t+time_min),
  object(O), manip(M).
:- pickUpRM(M,t+time_min), 1{atObj(O,M,t+time_min):object(O)},
  manip(M).

% the robot cannot pick up an object
%  if the object is not at the same place as robot.
pickUpRO(O,t+time_min) :-  pickUp(M,O,t+time_min),
  manip(M), object(O).
:- pickUpRO(O,t+time_min), not atRob(L,t+time_min),
  atObj(O,L,t+time_min), object(O), robloc(L).
:- pickUpRO(O,t+time_min), atRob(L,t+time_min),
  not atObj(O,L,t+time_min), object(O), robloc(L).

% the robot cannot place an object if it is not holding any
:- place(M,t+time_min), {atObj(O,M,t+time_min):object(O)}0,
  manip(M).

% the robot cannot clean if it is not at the faucet
cleanR(t+time_min) :- clean(M,t+time_min), manip(M).
:- cleanR(t+time_min), not atRob(faucet,t+time_min).
:- clean(M,t+time_min), {atObj(O,M,t+time_min):object(O)}0,
  manip(M).
:- clean(M,t+time_min), atObj(O,M,t+time_min),
  object(O), manip(M),
{isclean(O,t+time_min); -isclean(O,t+time_min)}0.

% sensing is not possible if the values of relevant
%  fluents are known
:- sense(cleanObj(O),t+time_min),
  1{isclean(O,t+time_min); -isclean(O,t+time_min)},
  object(O).
:- sense(cleanObj(O),t+time_min),
  {atObj(O,M,t+time_min): manip(M)}0, object(O).
:- sense(locObj(O),t+time_min),
  1{atObj(O,L,t+time_min): objloc(L)}.
:- sense(food_request,t+time_min),
  1{requested(F,t+time_min): food(F)}.
```

Fig. C 6: Kitchen table setting domain presented to HCP-ASP, in the input language of CLINGO: Part 5 – State constraints and preconditions of actions.

```
% feasibility checks

:- move(L1,t+time_min), atRob(L,t+time_min),
  robloc(L), robloc(L1), @move_feasible(L,L1)!=1.

:- pickUp(M,O,t+time_min), manip(M), object(O),
  atRob(L,t+time_min), atObj(O,L,t+time_min),
  robloc(L), @pickUp_feasible(M,O,L)!=1.

:- place(M,t+time_min), atObj(O,M,t+time_min),
  manip(M), object(O), atRob(L,t+time_min),
  robloc(L), @place_feasible(M,O,L)!=1.

% concurrency constraints

moveR(t+time_min) :- move(L,t+time_min), robloc(L).
pickUpR(t+time_min) :- pickUp(M,O,t+time_min), manip(M), object(O).
placeR(t+time_min) :- place(M,t+time_min), manip(M).

% the robot cannot pick/place/clean an object while moving
:- moveR(t+time_min), pickUpR(t+time_min).
:- moveR(t+time_min), placeR(t+time_min).
:- moveR(t+time_min), cleanR(t+time_min).

% the robot cannot pick/place an object while cleaning it
:- pickUpRM(M,t+time_min), clean(M,t+time_min), manip(M).
:- place(M,t+time_min), clean(M,t+time_min), manip(M).
:- place(M,t+time_min), pickUpRM(M,t+time_min), manip(M).

actAction(t+time_min):- moveR(t+time_min).
actAction(t+time_min):- pickUpR(t+time_min).
actAction(t+time_min):- placeR(t+time_min).
actAction(t+time_min):- cleanR(t+time_min).

sensAction(t+time_min):- sense(cleanObj(O), t+time_min), object(O).
sensAction(t+time_min):- sense(locObj(O), t+time_min), object(O).
sensAction(t+time_min):- sense(food_request, t+time_min).

% no sensing action and actuation action can occur
%  at the same time
:- actAction(t+time_min), sensAction(t+time_min).

% no two sensing actions are allowed at the same time
:- 2{sense(cleanObj(O),t+time_min):object(O);
     sense(locObj(O1),t+time_min):object(O1);
     sense(food_request,t+time_min)}.
```

Fig. C 7: Kitchen table setting domain presented to HCP-ASP, in the input language of CLINGO: Part 6 – Feasibility checks and concurrency constraints.

```
% goal conditions

% what is expected on the table wrt food type
expected(F,T,t+time_min) :-
  1{atObj(O,table,t+time_min):type(T,O),isclean(O,t+time_min)}1,
  N-1{-atObj(O,table,t+time_min):type(T,O)}N-1,
  food(F), expected_T(F,T), type_no(T,N).

% the goal is not satisfied
%  if the expected objects are not on the table or
%  if there is an unexpected object on the table
notgoal(t+time_min) :- not expected(F,T,t+time_min),
  expected_T(F,T), requested(F,t+time_min), food(F).
notgoal(t+time_min) :- not -atObj(O,table,t+time_min),
  unexpected(F,O), object(O), requested(F,t+time_min), food(F).

% the goal is satisfied otherwise
goal(t+time_min) :- not notgoal(t+time_min),
  expected(F,_,t+time_min), requested(F,t+time_min), food(F).

% ensure that goal is reached some time before step_time
:- query(t), not goal(t+time_min).
```

Fig. C 8: Kitchen table setting domain presented to HCP-ASP, in the input language of CLINGO: Part 7 – Goal conditions.