Junior Engineer Internship Report
Chloe Benz — ISAE-ENSMA

June - September 2020

Autonomous waiter knowledge base

Under the supervision of Dr. Mohan Sridharan

This document encloses the report for my junior engineering internship, conducted remotely from June to September 2020 in collaboration with Arthur Findelair (ISAE-ENSMA) and under the supervision of Dr. Mohan Sridharan (School of Computer Science - Birmingham University, UK). The following pages focus on a basic restaurant knowledge domain coded in Answer Set Programming (ASP) - SPARC for an autonomous waiter to reason on. Arthur Findelair's work, that will not be discussed in details here, consisted in building a simulated environment using PyBullet in which to emulate an autonomous agent, as well as developing part of the interface to enable the agent to reason on the knowledge base I developed.
*Note: There are some clickable links in this report in its PDF format, on bold font "***here***" occurrences. They redirect the reader to the appropriate websites - most of the time, subfolders from the GitHub repository containing all of the code done during this internship.*

Contents

1 Introduction

1.1 Global presentation

The goal of this internship was to develop an autonomous waiter, that is able to reason by itself to serve clients in a restaurant, co-existing with other employees on shift in that same restaurant. Since having access to an actual robot to test on was impossible because of the ongoing pandemic at the time this internship was conducted, the decision was made that the reasoning agent would be tested in a simulated environment.
The project is split in two distinct parts, that have to be able to communicate between each other. On one side, there is the knowledge base and planning module that enable the agent to plan its actions in the restaurant. On the other, there is the simulated world, that mimics a simple restaurant and has to include clients, employees of the restaurant, as well as furniture, objects and so on. In this report, the knowledge base and planning module are described in details, as well as some portions of the communication script that enables the reasoning agent to react to events happening in the simulated world.

1.2 Resources

The resources used can be found in the bibliography at the end of the report. The code and/or direct GitHub links to access the code written during this internship can be found in the appendix. The programming languages used are ASP-SPARC [3] for the knowkedge base and reasoning, and Python for the communication script. The solver used is clingo [1], with the java extension for it to be able to support ASP-SPARC [2]. The simulated environment was created using PyBullet and the GUI using PyQt. An application to run simulations is downloadable and runnable and can be found **here**, along with explanations on how to set everything up.

2 Modelling a simple restaurant domain in ASP-SPARC

Answer Set Programming (ASP) is a form of logic programming optimized for difficult search problems. There are multiple ASP languages, such as AnsProlog, or SPARC (that is used here - mostly referred to as ASP-SPARC in this report). In ASP, a stable model is computed first from the implemented rules and consists in ground terms - every possible instantiation of said rules given the declared variables - that are then used to perform search. For this internship, ASP is used to design a knowledge base. The aim is to write all the rules of the environment an autonomous agent has to evolve in and to give the said autonomous agent a set of actions it can perform or witness for it to search for the best course of action to take to reach its goal. In that case, the environment is a restaurant, and the goal is to give an autonomous agent all it needs to perform the same tasks as a waiter.

2.1 ASP-SPARC file

An ASP-SPARC file has three main sections, which are:

- **sorts** - to declare "types", "variables" and (in this context) fluents (or instantaneous states that are bound to change, like variables that can hold true at a certain point in time and false at other points) and actions that the agent can perform
- **predicates** - to declare the predicates on which the rules will be based, a predicate is always either true, false or unknown (i.e. it is not known if it is true or false)
- **rules** - to declare the rules of the knowledge domain

Those sections must appear in the order given below, and each section starts at its (eponymous) section keyword and ends when another section keyword is written. Integer constants can be defined before the **sorts** section, at the very beginning of the file, using a command of the form:

```
#const n = 4.
```

Another (optional) section is **display**, which can be used to set which subsets of the answer sets are to be displayed (very useful when there are a lot of ground terms at play). Note that the line comment character is %, and that every line has to end with a dot, except the lines containing the section keywords.
A major improvement from ASP is that ASP-SPARC has sorts, which adds some kind of class/type feature that renders the language more structured and legible. With that, each time an instance is declared and evaluated with the wrong sort, the program returns an error, which prevents some logical inconsistencies that could have easily been missed otherwise.

The initial idea was to create a simple restaurant domain incorporating the following instances:

- waiter, agent and customer where employee = {waiter, agent} and person = {waiter, customer}
- table, chairs, bar where furniture = {table, chairs, bar}
- basket, jug, plate, badge where refillable = {basket, jug} and object = {basket, jug, plate}
- node

The items underlined are the ones that made it into the version used in the simulator, but it the others can still be added into the ASP file to reason on, as well as the associated actions and rules - except for the chairs, which would cause a total redesign of the seating rules as they are currently implemented. The first version of the knowledge domain actually featured chairs associated to a table and to a customer, but this structure was abandoned because it was challenging to find an efficient and simple seating process attributing a chair to a unique customer. Instead, (groups of) customers are now associated to the table they are sat at, assuming there is enough chairs to accommodate them. To check if there is enough chairs at a table, a fluent `has_chairs(#num(N), #table(T))` can be added to the actual program without much work. If the number of chairs matches the maximum capacity of the table, then the table is in its "default state" and ok to use. If there is less chairs that the capacity of the table but that the group of customers that is meant to be sat somewhere has a lower head count that the number of chairs, then the table is also ok to use. Otherwise, an action `bring_chairs(#employee(E), #table(T), #num(N))` could be added to the list of actions that the agent can perform. Speaking of actions, the following actions were initially thought of:

- `go_to(#employee(E), #node(N), #node(N))` - to track the agent and waiters' positions on defined nodes in the restaurant
- `pick(#employee(E), #customer(Cu))` - to pick clients arriving in the restaurant
- `seat(#employee(E), #customer(Cu), #table(T))` - to seat clients that have been picked up at a table
- `give_bill(#employee(E), #table(T))` - to give the bill to a table
- `bring(#employee(E), #object(O), #table(T))` - to bring any item (food, bread or water) at a table
- `refill(#employee(E), #refillable(R), #table(T))` - to pick an empty jug or bread basket on a table and to refill it - needs to be followed by the `bring` action
- `take_order(#employee(E), #table(T))` - to take the order from a table

- `send_order(#employee(E), #table(T))` - to send the order from table T to the kitchen

Again, the actions underlined are the ones that made it into the actual version of the knowledge base, but the other ones can still be added on top of it, provided the right fluents are added in the process. In the actual state of the program, only the agent is able to "perform" the actions. Since the knowledge domain is from the agent's point of view, the answer sets will contain grounded terms such as `go_to(agent, n0)` but never `go_to(w1, n0)` (w1 being the waiter labeled as 1 on shift at the same time as the agent). The program file does not have a diagnostic module at this time - which means it cannot generate explanations for - for instance - a waiter's whereabouts in the restaurant. If waiters were to perform actions and were observed to be where the knowledge base does not expect them, the program would be inconsistent.
Out of the fluents, instances and actions that have been implemented in the current version of the program, the `node` sort and the `go_to` actions hold a central place. The nodes are used to simplify the displacements of the agent and waiters in the restaurant. The map of the restaurant was defined arbitrarily and is visible on figure 1.

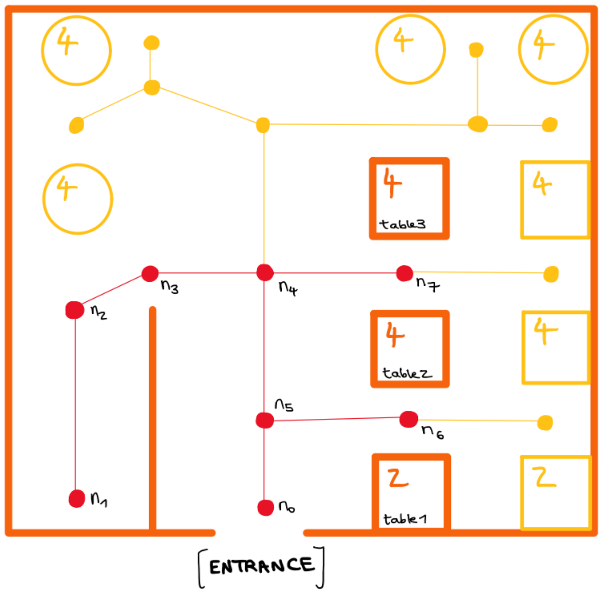

Figure 1: Restaurant layout

The tables in orange (named `table1`, `table2` and `table3`) and paths and nodes (named n_i $\forall i$ in [0;7]) in red are the instances that can be actually reasoned upon. The rest of it (in yellow) can be seen in the simulation, but not interacted with at all. This can be changed by adapting lines 140 to 141, 147 to 152 and 161 to 163 to define the capacity of each table, the paths between nodes and the tables that are reachable from a certain node, without forgetting to increment the constants `tnum`, `nnum` and `maxcap` if a table of more that 4 people capacity is needed.
This restaurant layout can be easily changed and only necessitates a rather easy (but manual, for now) instantiation of the ASP-SPARC file. The initial idea was to include the layout generation in the GUI so that the ASP can be broadly used without much manual changes in the code. What the user has control on is the number of tables, the number of nodes and how the nodes are placed in regard to the tables an the entrance of the restaurant. The exact layout can be set in the simulated world (see presentation video of the PyBullet environment). The simulated world is necessary, in the sense that it puts in perspective the nodes' locations in the chosen restaurant layout. But if the ASP-SPARC program were to be used in the real world, a simple sketch of the restaurant would be sufficient to instantiate everything correctly.

2.2 Problems encountered

While coding the knowledge base, some problems worth mentioning were encountered. Those are listed below, in no particular order. To understand, the reader must have a good understanding of the syntax of ASP-SPARC and of how the ASP file described is structured.

- **Node naming:** the nodes used to model the positions and paths possible in the restaurant were first named alphabetically, as in the early development, very few nodes were necessary. The node named ”a” caused a lot of trouble during the grounding of the rules - possibly because some variables started with an ”a”. The naming was consequently changed to ”n” with an integer as a subscript - which also allows for more flexibility when setting up the restaurant layout.
- **Inertia axiom conflict:** let us consider the following rule:

  ```
      -holds(F(A), I):- not holds(F(A), I).
  ```

  Where F is a fluent with some parameter A, and I a time step between 0 and N. N is the maximum number of steps and specified in the constants in the beginning of the ASP file, so it is fixed. Such a rule conflicts with the inertia axiom, and causes the planning module to behave strangely. In fact, it is impossible to know if the state of F at a certain time step J is due to the inertia axiom or the rule above - the first to be grounded for that time step will determine it. To ”solve” that issue, the rule above has been changed to the following one, for each fluent F that necessitated it:

  ```
      -holds(F(A), 0):- not holds(F(A), 0).
  ```

  This sets the initial situation and lets the inertia axiom take over for all time steps $J \in [1, N]$.
- **Forcing a goal to happen at a specific step:** this can be done properly by adding specific rules to ensure that the planning module will not crash, and is prevented in the basic restaurant knowledge base described here.

3 Communication between the simulation and the knowledge base

3.1 Running the ASP file

3.1.1 Outside of the simulation

Running the ASP file to retrieve the answer sets is straightforward and can be done using very few lines of code. The **display** section keyword can here be very useful, since it can restrict the displayed computed answer sets to whatever fluents are needed. What was done here is a bit different, since the time ordering of the fluents is important for clarity and quick debugging. Usually, answer sets are displayed in no particular order, which is not convenient in that case. A small Python script was used, which can be found **here**. Each fluent or action or observation appear in instances of the form:

```
holds(F(A_f, B_f, ...), I).
occurs(A(A_oc, B_oc, ...), J).
obs(O(A_ob, B_ob, ...), K).
```

Where I, J, K are integers representing the time steps. What the Python script provided with the ASP-SPARC implementation does is it reorders the fluents/actions/observations from the answer sets regarding the time step they occurred at. With this script, a keyword search within the answer sets is possible, and the user can choose the number of answer sets displayed. Pieces from that code are used in the communication script that allows the simulation to reason with and update the knowledge base, with a few tweaks because the script described in this section is not time-efficient.

3.1.2 Within the simulation

Some of the features of the Python script mentioned before have been kept and integrated into the communication script - right **here**. The lines from 226 to 285 feature those, in a way that's a little bit more efficient than the code mentioned in the previous section. The program can actually be run with a single command (see line 217 in the `get_minimal_plan()` function body). The answer set (only one is needed, here - technically, the one that gives the lowest number of actions to be performed, for service efficiency) is then reordered with regard to time and fed to the simulation.

3.2 Updating the ASP file

The ASP-SPARC knowledge base needs to be accessed and updated at each step of the simulation, which means the communication script has to be efficient time-wise.

3.3 Minimal planning

There is no way to directly code a planning module that does minimal planning in ASP-SPARC (though it is feasible in classic ASP, and some other answer set programming languages allow that too). Thus, a Python script was added to the communication program. It starts computing answer sets with N=0 (i.e. it tries to compute single-steps answer sets). If the output is "The SPARC program is inconsistent.", then it increments N and tries running again - and repeats until it finds a non inconsistent answer set. This is only possible if there exists such an answer set, which is not always true. This method can work fairly well though, with a threshold on N so that it does not enter an endless loop. A better way would be searching the minimal N using dichotomy, so that the number of times the ASP has to run is minimized too. At this time, there is no way to be sure that the ASP to be ran has a consistent answer set. An amelioration of the communication program would include this feature if there is an efficient and costless way to do so.

3.4 ASP run and update flowchart

How the ASP is ran and updated based on new observations received from the simulation is represented on figure 2 below.
Note that for now, the observations sent from the simulator must be inputted somehow by the user. This can be seen in the video linked in the appendix, with the "make the customer want the bill" part at the end. For future work, as described in A. Findelair's work, the aim would be to use a gesture recognition unit to update the modelled environment with real-time observations.

3.5 Managing multiple goals

It is possible to manage multiple goals at once if those goals are not conflicting, and do not require to be executed in a specific order. By that, it is meant that a new client arriving, for instance, and a client demanding the bill are treated in the order that gives the minimal plan (i.e. the least number of actions required). If some goals need to be managed in a specific order, then one can set up rules so that a specific observation triggers a fluent that triggers an action, that results in another observation or change of state of fluent and so on.

4 Conclusion

During this internship, the groundwork to develop an autonomous agent capable to work along with human waiters in a restaurant was laid. The goal was to understand how commonsense reasoning can be implemented for real-world applications using a refined model comprised of a certain set of rules about the environment. By using an ASP language to define the knowledge base, integrity in

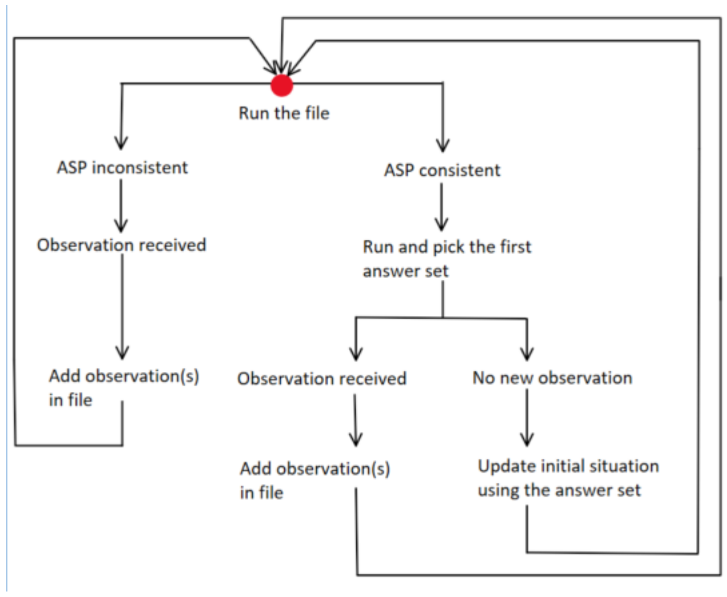
Run the file
ASP inconsistent
ASP consistent
Observation received
Run and pick the first answer set
Add observation(s) in file
Observation received
No new observation
Add observation(s) in file
Update initial situation using the answer set

Figure 2: Diagram

the sense that the course of action to take to perform a series of task to reach a certain goal without breaking any rules is assured. But setting up the rules and making sure they do not conflict with each other is a gruesome task, and would benefit a lot from having dedicated unit tests to ensure that there is no unwanted behavior, which were not performed during this internship. Future work would include refining the modelling of the restaurant domain, as well as designing an UI to parameterize the restaurant model at will (number of tables, restaurant layout, *etc.*) and an API to link a gesture control unit to perform real-world and real-time tests. At at time where gesture-controlled AI applications are becoming widely used and researched, especially in the navigation domain, using a knowledge base such as the one developed here could be a solution to make sure the behavior of the reasoning agents (*e.g.* UAV$_s$ or UGV$_s$) is reliable and does not violate any rule that's necessary for the safety of possible human operators or bystanders.

A Code

A.1 ASP-SPARC code

```
%%%%%%%%%%%%%%%%%%%%%%%%%% SIMPLE RESTAURANT DOMAIN %%%%%%%%%%%%%%%%%%%%%%%%%%
% This file, written in SPARC (a variant of ASP - Answer Set Programming) %
% models a simple restaurant layout, with a defined number of customers and %
% employees.                                                              %
% This domain is built for an agent to reason on, and the goal is to enable %
% said agent to perform simple actions (picking or seating customers, taking %
% orders, bringing bills and diverse items to tables) autonomously in a %
% restaurant.                                                             %
%-----------------------------------------------------------------------------%
%% CONSTANTS - All of the necessary constants %%

% Number of steps to plan on %
#const nstep = 8.

% Total number of waiters %
#const wnum = 1.
% Total number of customers - Must be incremented each time a new customer enters the restaurant %
#const cnum = 6.
% Total number of tables %
#const tnum = 2.
% Maximum capacity within the tables %
#const maxcap =4.
% Total number of nodes %
#const nnum = 7.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sorts
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% SORTS - all the types declaration %%
#step = 0..nstep.

#num = 0..maxcap.

#boolean = {true, false}.

#node = [n][0..nnum].

#table = [table][1..tnum].
#furniture = #table.

#customer = [c][1..cnum].
#waiter = [w][1..wnum].
#agent = {agent}.

#employee = #waiter + #agent.
#person = #customer + #waiter.
%-----------------------------------------------------------------------------%
%% FLUENTS - Declaration of the fluents used in the restaurant domain %%
#inertial_fluent = currentlocation(#employee, #node) % Location of employees

               + iswaiting(#customer)            % Customer management
               + iswith(#agent, #customer)       % Customer management
               + isattable(#customer, #table)    % Customer management
               + haspaid(#customer)              % Customer management

               + wantsbill(#table)               % Payment

               + aretogether(#customer, #customer). % Groups management

#defined_fluent = hasoccupancy(#table, #num).    % Table current occupancy

#defined_fluent_special = isfree(#table).        % Table current state (free/occupied)

#observable = has_entered(#customer)             % Triggers iswaiting inertial fluent
          + bill_wave(#table)                    % Triggers wantsbill inertial fluent
          + group(#customer, #customer).         % Triggers aretogether inertial fluent

#fluent = #inertial_fluent + #defined_fluent + #defined_fluent_special + #observable.
%------------------------------------------------------------------------------%
%% ACTIONS - Performed by the agent %%

#a_go = go_to(#agent(A), #node(N)).             % Go to a given node in the restaurant
#a_pick = pick(#agent(A), #customer(Cu)).       % Pick up a (group of) customer(s)
#a_seat = seat(#agent(A), #customer(Cu), #table(T)). % Seat a (group of) customer(s)
#a_bill = give_bill(#agent(A), #table(T)).      % Give the bill to a table

#action = #a_go + #a_pick + #a_seat + #a_bill.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
predicates
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
edge(#node, #node).
areassociated(#node, #furniture).

hascapacity(#table, #num).

holds(#fluent, #step).
occurs(#action, #step).

obs(#observable, #boolean, #step).

goal(#step).
success().
something_happened(#step).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
rules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% FLUENT RULES - Inertia axiom for fluents %%
holds(F, I+1):- #inertial_fluent(F), holds(F,I), not -holds(F, I+1).
-holds(F, I+1):- #inertial_fluent(F), -holds(F, I), not holds(F, I+1).
%------------------------------------------------------------------------------%
%% ACTIONS RULES - Closed World Assumption (CWA) for actions %%
-occurs(A,I):- not occurs(A,I).
%------------------------------------------------------------------------------%
%% ACTIONS RULES - Non-simultaneity of actions %%
:- occurs(A1,I), occurs(A2,I), A1!=A2.
%------------------------------------------------------------------------------%
%% OBSERVATIONS RULES - Reality-check %
```

```
:- obs(F,true,I), -holds(F,I).
:- obs(F,false,I), holds(F,I).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% SIMPLE PLANNING MODULE %%

success:- goal(I), I <= nstep.
:- not success.
occurs(A,I) | -occurs(A,I) :- not goal(I).
something_happened(I):- occurs(A,I).
:- not goal(I), not something_happened(I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%% GOALS TO BE SET BELOW: %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%b_goal
goal(I):- holds(isattable(c1, T),I).
%e_goal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%% OBSERVATIONS TO BE SET BELOW: %%%%%%%%%%%%%%%%%%%%%%%%%
%b_obs
obs(has_entered(c1),true,0).
%e_obs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%% INITIAL SITUATION TO BE SET BELOW: %%%%%%%%%%%%%%%%%%%%%%%
% Set the "fluent" initial situation here: %
%b_init
holds(currentlocation(agent, n1), 0).
holds(currentlocation(w1, n7), 0).
%e_init
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% DOMAIN RULES - Restaurant layout %%

% Capacity of tables in the restaurant %
hascapacity(table1, 2).
hascapacity(table2, 4).
%------------------------------------------------------------------------------%
% A table can only have one fixed capacity %
-hascapacity(T, N):- not hascapacity(T, N).
%------------------------------------------------------------------------------%
% Pathways in the restaurant %
edge(n1, n2).
edge(n2, n3).
edge(n3, n4).
edge(n4, n5).
edge(n4, n7).
edge(n5, n6).
%------------------------------------------------------------------------------%
% The edge relation is symmetric %
edge(N2, N1):- edge(N1, N2).
%------------------------------------------------------------------------------%
% If it is not known that there is an edge between nodes then there is not %
-edge(N1, N2):- not edge(N1, N2).
%------------------------------------------------------------------------------%
% Node/furniture association %
areassociated(n6, table1).
areassociated(n6, table2).
areassociated(n7, table2).
%------------------------------------------------------------------------------%
```

```
% Two nodes that do not seem to be associated with a piece of furniture are not %
-areassociated(N, F):- not areassociated(N, F).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% DOMAIN RULES - Locations of employees in the restaurant %%

% Two employees cannot share the same location %
-holds(currentlocation(E2, N), I):- holds(currentlocation(E1, N), I), E2 != E1.
%------------------------------------------------------------------------------%
% An employee can only have one current location at a time %
-holds(currentlocation(E, N2), I):- holds(currentlocation(E, N1), I), N2 != N1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% DOMAIN RULES - Waiting to be seated %%

% A customer that is not initially known to be with the agent is not %
-holds(iswith(agent, Cu), 0):- not holds(iswith(agent, Cu), 0).
%------------------------------------------------------------------------------%
% A customer cannot be with the agent and waiting at the same time %
-holds(iswaiting(Cu), I):- holds(iswith(agent, Cu), I).
%------------------------------------------------------------------------------%
% A customer cannot be at at table and waiting at the same time %
-holds(iswaiting(Cu), I):- holds(isattable(Cu, T), I).
%------------------------------------------------------------------------------%
% The agent cannot manage two groups at once %
-holds(iswith(agent, Cu2), I):- holds(iswith(agent, Cu1), I), -holds(aretogether(Cu1, Cu2), I).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% DOMAIN RULES - Customer placement %

% The "aretogether" group relation is symmetric, transitive and reflexive %
holds(aretogether(Cu2, Cu1), I):- holds(aretogether(Cu1, Cu2), I).
holds(aretogether(Cu1, Cu3), I):- holds(aretogether(Cu1, Cu2), I), holds(aretogether(Cu2, Cu3), I).
holds(aretogether(Cu, Cu), I).
%------------------------------------------------------------------------------%
% If two customers are not initially known to be together, then they are not %
-holds(aretogether(Cu1, Cu2), 0):- not holds(aretogether(Cu1, Cu2), 0), Cu2 != Cu1.
%------------------------------------------------------------------------------%
% If a customer is not initially known to be at a table then he is not %
-holds(isattable(Cu, T), 0):- not holds(isattable(Cu, T), 0).
%------------------------------------------------------------------------------%
% A customer cannot be at two tables at the same time %
-holds(isattable(Cu, T2), I):- holds(isattable(Cu, T1), I), T2 != T1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% DOMAIN RULES - Tables %%

% A table's default state is free %
holds(isfree(T), I):- not -holds(isfree(T), I).
%------------------------------------------------------------------------------%
% Current occupancy is determined by counting the # of clients at a table %
holds(hasoccupancy(T, X), I):- X = #count{Cu : holds(isattable(Cu, T), I)}.
%------------------------------------------------------------------------------%
% A table cannot have multiple occupancies %
-holds(hasoccupancy(T, X2), I):- holds(hasoccupancy(T, X1), I), X2 != X1.
%------------------------------------------------------------------------------%
% The occupancy of a table cannot exceed its maximum capacity %
-holds(hasoccupancy(T, X2), I):- hascapacity(T, X1), X2 > X1.
%------------------------------------------------------------------------------%
% A table is not free if its occupancy is not 0 %
```

```
-holds(isfree(T), I):- holds(hasoccupancy(T, X), I), X != 0.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% DOMAIN RULES - Payment %%

% A table does not initially wantsbill unless specified %
-holds(wantsbill(T), 0):- not holds(wantsbill(T), 0).
%------------------------------------------------------------------------------%
% If a customer is not initially known to have paid then he has not unless specified %
-holds(haspaid(Cu), 0):- not holds(haspaid(Cu), 0).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% DOMAIN RULES - Observables %%

% Observing a customer in the entrance triggers the iswaiting fluent %
holds(iswaiting(Cu), I):- obs(has_entered(Cu), true, I).
%------------------------------------------------------------------------------%
% Observing a bill_wave triggers the wantsbill fluent %
holds(wantsbill(T), I):- obs(bill_wave(T), true, I).
%------------------------------------------------------------------------------%
% Observing a group triggers the aretogether fluent %
holds(aretogether(Cu1, Cu2), I):- obs(group(Cu1, Cu2), true, I).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ACTIONS - Causal laws %%

% go_to(#agent, #node) %
holds(currentlocation(agent, N), I+1):- occurs(go_to(agent, N), I).
-holds(currentlocation(agent, M), I+1):- occurs(go_to(agent, N), I), M !=N.

% pick(#agent, #customer) %
-holds(iswaiting(Cu2), I+1):- occurs(pick(agent, Cu1), I), holds(aretogether(Cu1, Cu2), I).
holds(iswith(agent, Cu2), I+1):- occurs(pick(agent, Cu1), I), holds(aretogether(Cu1, Cu2), I).

% seat(#agent, #customer, #table) %
holds(isattable(Cu2, T), I+1):- occurs(seat(agent, Cu1, T), I), holds(aretogether(Cu1, Cu2), I).
-holds(iswith(agent, Cu2), I+1):- occurs(seat(agent, Cu1, T), I), holds(aretogether(Cu1, Cu2), I).

% give_bill(#table) %
holds(haspaid(Cu), I+1):- occurs(give_bill(agent, T), I), holds(isattable(Cu, T), I).
-holds(wantsbill(T), I+1):- occurs(give_bill(agent, T), I).
%------------------------------------------------------------------------------%
%% ACTIONS - Executability conditions %%

% go_to(#agent, #node) %
-occurs(go_to(agent, N), I):- holds(currentlocation(agent, M), I), -edge(M, N).
-occurs(go_to(agent, N), I):- holds(currentlocation(E, N), I), #waiter(E).
-occurs(go_to(agent, N), I):- holds(currentlocation(agent, N), I).

% pick(#agent, #customer) %
-occurs(pick(agent, Cu), I):- holds(currentlocation(agent, N), I), N != n5.
-occurs(pick(agent, Cu), I):- -holds(iswaiting(Cu), I).
-occurs(pick(agent, Cu), I):- #count{T: holds(isfree(T), I)} = 0.

% seat(#agent, #customer, #table) %
-occurs(seat(agent, Cu, T), I):- -holds(iswith(agent, Cu), I).
-occurs(seat(agent, Cu, T), I):- holds(currentlocation(agent, N), I), -areassociated(N, T).
```

```
-occurs(seat(agent, Cu, T), I):- -holds(isfree(T), I).
-occurs(seat(agent, Cu1, T), I):- #count{Cu2: holds(aretogether(Cu1, Cu2), I)} > X, hascapacity(T, X).

% give_bill(#table) %
-occurs(give_bill(agent, T), I):- -holds(wantsbill(T), I).
-occurs(give_bill(agent, T), I):- holds(isfree(T), I).
-occurs(give_bill(agent, T), I):- holds(currentlocation(agent, N), I), -areassociated(N, T).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The ASP-SPARC code code can also be found **here**.

A.2 Python code

My contribution to the Python code can be found **in this file**. I co-wrote the functions with Arthur Findelair, which I quoted earlier in this report.

B Recorded simulation - commentary

The recorded simulation can be found **here**.

B.1 1-sized customer group - seating

From 00:00 to 00:22 in the video.
In this part, the button labeled "New customer(s)" is clicked, and a customer spawns in the entrance of the restaurant. The ASP file is also provided with a new observation (`obs(has_entered(c1), I)`, I being the discrete time-step at which the observation is made), and then ran.
Note that:

- The agent goes to pick the customer up - once it has done so, the customer disappears from the simulation to be respawned sitting at its table. Picking the customer up "triggers" the `is_with(agent, c1)` fluent in the ASP, which is translated by the agent "putting the customer in its inventory", and getting it out at its table. When the customer is seated, he is no longer with the agent.
- The customer is seated at the 2-seats table, which is convenient for the remainder of the simulation, but he could have been seated at the adjacent 4-seats table indifferently. In theory, there is some kind of minimization of the number of actions performed by the agent to achieve a goal in the ASP program, so the customer could not have been seated at the farthest away 4-seats table.

B.2 4-sized customer group - seating

From 00:22 to 00:36 in the video.
In this part, the button labeled "New customer(s)" is clicked, and a group of 4 customers spawns in the entrance of the restaurant. The ASP file is also provided with a set of four new observations (`obs(has_entered(cj), I)` $\forall j$ in [2, 5], (since c1 is already seated) I being the discrete time-step at which the observations are made), and then ran.

B.3 1-sized customer group - bringing the bill

From 00:36 to 00:47 in the video.
In this section, a right click on the table of capacity 2 simulates the solo customer requesting the bill. From this point on, the behavior of the agent becomes somehow erratic. In theory, it is assumed that

the agent has the means to print the bill directly from where it is (say he has an embedded printer), so it does not need to reach to the counter to pick the bill up and then go back at the table asking for the bill. In practise, the agent is right by the table asking for the bill (the 2-seats table by the entrance) but it goes to the counter nonetheless and then comes back by the table. This ”feature” is due to how the communication algorithm was initialized (see `get_minimal_plan()` function in the `CommunicationASP.py` file in the ASP section of the code, that can be found **here**). Line 212 of the `CommunicationASP.py` file, n is initialized to 2 which means the agent will not look for plans of length strictly inferior to 2. That means it will have to move in order to have an answer set of length 3, because it can’t compute the one of length one it could have come up with.

B.4 4-sized customer group - bringing the bill

From 00:47 to 01:01 in the video.
In this last section of the video, the same as before happens - a right click on the table of capacity 4 simulates the group of 4 customers requesting the bill - except the agent does way more back and forth moving before dropping the bill at the right table. This behavior can to some extent be explained by the same ”feature” as in the previous subsection, but this only does apply to the first back and forth move. Either the n has been updated to a higher value at some point, or the minimization of the number of actions is somehow blocked. I did not look into it in details.

References

[1] URL: `https://potassco.org/clingo/`.

[2] Evgenii Balai. URL: `https://github.com/iensen/sparc`.

[3] Evgenii Balai, Michael Gelfond, and Yuanlin Zhang. “SPARC – Sorted ASP with Consistency Restoring Rules”. In: (2013). URL: `https://ui.adsabs.harvard.edu/abs/2013arXiv1301.1386B/abstract`.