

Case Studies using the Epistemic-Jason extension

Ethan Le Trung¹ and Babak Esfandiari²

¹ Department of Computer Science, Polytech Lyon, Lyon 1 University, Lyon, France
`ethan.le-trung@etu.univ-lyon1.fr`

² Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada
`babak@sce.carleton.ca`

1 Introduction

Jason is a popular agent-oriented programming language and platform based on the Belief-Desire-Intention (BDI) architecture, which models the mental attitudes of agents in terms of what they believe, desire, and intend. This architecture allows agents to reason explicitly about their goals and the actions needed to achieve them, offering a natural framework for implementing intelligent, autonomous behavior.

While Jason’s classical BDI model captures the agent’s internal state, it does not natively support explicit representation of uncertainty or reasoning about the knowledge or beliefs of other agents. To address this, an epistemic extension of Jason has been proposed by M. Vézina, incorporating the semantics of possible worlds—a foundational concept in epistemic logic. This semantic framework enables agents to represent and reason about what is known or believed by themselves and others in a multi-agent system.

In this paper, we present a concrete example demonstrating how this epistemic extension can be used within Jason. Through this case study, we illustrate how the integration of possible-world semantics enhances the expressiveness of BDI agents, allowing them to perform higher-order epistemic reasoning such as modeling uncertainty, nested beliefs, and knowledge-based decision-making in dynamic environments.

Consider reading **Michael Vezina’s thesis** to better understand the underlying principles of the *epistemic-jason* extension.

2 Case Study : A weather example

In this case, the PAL (Public Announcement Logic) mode is used for the reasoner. To run this project, you have to run the **weather.mas2j** file in the **ProjectRunner** file. In this example we consider a single-agent system. Two worlds are possible here :

- raining.

- not raining.

As you can see in Figure 1, many ways are possible to declare the set of possible worlds.

```

1)
2  range(raining(X)) :- .range(X, 0, 1).
3
4  range(none) :- true.
5  none :- .findall(not(raining(X)), range(raining(X)), List) & .big_and(X, List) & X.
6  ~none.
7
8  ~raining(X1) :- raining(X2) & (X1 \== X2).
9
10 2)
11 range(raining) :- true.
12
13 3)
14 range(raining) :- true.
15 range(notRaining) :- true.
16
17 range(none) :- true.
18 none :- not raining & not notRaining.
19 ~none.
20
21 ~raining :- notRaining.
22 ~notRaining :- raining.

```

Fig. 1. Navigation under uncertainty

2.1 Worlds constraints generation

First Method The first method uses the Internal Action `.range(X,Y,Z)` (line 2). This internal action unifies the variable `X` with all values between `Y` and `Z` (inclusive), one by one, using backtracking. This will generate constraints for the two values : `raining(0)` (not raining) and `raining(1)` (raining).

Then, lines 4 to 6 ensure that the world cannot be empty: either it is raining or it is not raining — there is no other possibility. The Internal Action `findall(Var, Condition, List)` collects all values that `Var` can take such that `Condition` holds true, and returns these values as a list unified with `List`. The Internal Action `big_and(Res, List)` takes a list of logical formulas and constructs their conjunction as a single formula, which is unified with the variable `Res`.

This will gives :

```
none :- not(raining(0)) & not(raining(1))
```

The last constraints are introduced by line 8, enforcing the mutual exclusion of worlds. This ensures that both `raining(0)` and `raining(1)` cannot be true simultaneously within the same world. After all, how could it possibly be both raining and not raining at the same time?

This method is the most commonly used. It allows the generation of a wide range of possible worlds with just a few lines of code.

Figure 2 shows what the model generated looks like on reasoner's side.

```
2  CURRENT MODEL :  
3  0 : raining(1)  
4  1 : raining(0)
```

Fig. 2. Semantic model generated with first method

Second Method This is the easiest way in this case. Line 11 will generate the following constraint :

```
raining or not(raining)
```

This will be enough to create the set of possible worlds. Figure 3 shows what the model generated looks like on reasoner's side.

```
2  CURRENT MODEL :  
3  0 :  
4  1 : raining
```

Fig. 3. Semantic model generated with second method

Thus, as you can see one of the world is empty. It is recommended to not do this in more complex cases : specifying rules for each case will help avoid any misconceptions.

Third Method Finally, third method is kind of a mix of the two previous ways to declare constraints for worlds generation.

In lines 22 and 23, only one of those two lines would be enough to declare exclusivity in this case as only two propositions could potentially coexist.

Figure 4 shows what the model generated looks like on reasoner's side.

First and third methods are the most suitable for this problem. The first one will be used for the rest of the case study.

```

2  CURRENT MODEL :
3  0 : notRaining
4  1 : raining

```

Fig. 4. Semantic model generated with third method

2.2 Code Execution & Epistemic Evaluation

Figure 5 presents the code of the agent.

He has the goal to go out and the belief that he his home. If you have a look back to the worlds generated in the previous steps, none of the worlds generated had the proposition `home`. This results from using the reasoner in PAL mode.

The plan corresponding to the `!goOut` triggering_event will triggers the `!prepareForGoingOut` event (line 47). At this point two plans are relevant for this event. The first one has the context `poss(raining(1))` (line 29) and the second one has an empty context (thus, a `true` context) (line 32). At this point the epistemic-extension is going to parse each context in epistemic classes/formulas. It is possible to see how this works in the `AgentEpistemic` class, in the `applicablePlans` function.

Thus, the formula `poss(raining(1))` will be evaluated by the reasonner and returns `true`. So both plans are considered applicable for the `!prepareForGoingOut` event.

As the one with the `poss(raining(1))` is first-written in code order, he is the one that will be selected.

Then, the code will keep going on, As the agent considers it is possible that it is raining outside, he takes his umbrella. After that, the agent goes out. And here is an interesting point on line 54 : the *belief deletion* event changes depending on the mode in which the reasoner operates. As explained previously, in **PAL** mode, the belief `home` (and all beliefs that are not in the `range(..)` Literal in general) is not added to the reasoner semantic model. In the **DEL**, all beliefs events are added to the reasoner. So in the PAL case, we just delete the belief `home` using the standard belief deletion event. In the DEL case, we use the DEL event deletion. Then, when the agent goes out he discovers that it is not raining. As a result, a deletion event on the fact that it is raining is communicated to the reasoner (line 58). In other way, the agent says to himself "oh it is not raining".

At that point, the agent wants to go back home (line 59) and then prepares again himself to go out (line 60).

As the deletion event `+~raining(1)` appeared, the world containing `raining(1)` was removed from the possible worlds Set. At this point the evaluation of `poss(raining(1))` will returns false. Finally, the agent will go out in tee.

Figure 6 shows the logs of the agent on reasoner's side.

```

26     home.
27     !goOut.
28
29     +!prepareForGoingOut: poss(raining(1))
30         <- !prepareForRain.
31
32     +!prepareForGoingOut
33         <- !putATee.
34
35     +!goBackHome: not home
36         <- .print("going back home");
37         |
38         +home.
39
40     +!putATee: home
41         <- .print("putting a tee shirt").
42
43     +!prepareForRain: home
44         <- .print("putting a jacket");
45         |
46         .print("taking my umbrella").
47
48     +!goOut: home
49         <- !prepareForGoingOut;
50         |
51         !leaveTheHouse;
52         .wait(2000);
53         !discoverItIsSunny.
54
55     +!leaveTheHouse : home
56         <- .print("going out");
57         |
58         +~home. // -home; for the PAL +~home; for the DEL
59
60     +!discoverItIsSunny: not home
61         <- .print("Oh it is not raining..");
62         |
63         +~raining(1);
64         !goBackHome;
65         !prepareForGoingOut;
66         !leaveTheHouse.

```

Fig. 5. Weather agent's code

```

16
17 *****SINGLE-EVALUATE*****
18
19 FORMULA : (Kpos a raining(1)) --> true
20
21
22 ***APPLY-EVENT***
23
24 EVENT : ~raining(1)
25
26 RESULT : weather
27 1_~raining(1) : raining(0)
28
29 Event Metrics:
30 -----
31 Event Creation (ms): 1
32 Event Application (ms): 0
33 Total Time (ms): 1
34 -----
35 Events: 1
36 Previous Worlds: 2
37 Resulting Worlds: 1
38 Edges/Simulated: 0 / 1
39 =====
40
41
42
43 *****SINGLE-EVALUATE*****
44
45 FORMULA : (Kpos a raining(1)) --> false
46

```

Fig. 6. Weather agent's sematinc model on reasoner's side

3 Case Study : Navigation with uncertainty over own localization & goal localization. Changing the model

In this case, the DEL (Dynamic Epistemic Logic) mode is used for the reasoner. To run this project, you have to run the **navigation.mas2j** file in the **ProjectRunner** file. In this example, we consider a single-agent system. The agent is placed in a 5x5 grid world, but it does not initially know its own position. Its objective is to reach a target located on one of the grid's cells. The scenario is based on the following assumptions:

- The world is a 5x5 grid (25 cells), contains two obstacles, and is open—when the agent moves off one edge of the grid, it reappears on the opposite side.
- The agent necessarily occupies exactly one position on the grid at any given time.
- The agent cannot occupy a cell that contains an obstacle.

Furthermore, the agent has full knowledge of the structure of the world and encodes belief rules accordingly. For instance, it may declare rules such as: *“If I am at position X,Y, then I should move in direction D to reach the target.”*

3.1 Directions and obstacle rules

The first thing the code is going to do when he is running is to read the map and generate an .asl file base on it that contains :

- The range line defining the range of possible worlds.
- Lines excluding impossible worlds due to obstacles.
- Rules about obstacle perceptions.
- Direction rules based on the agent's position.

3.2 Semantic model generation

In possible-world semantics, the semantic model includes a set of possible worlds, each representing a different state the agent considers possible.

In Figure 7, the red squares indicate the possible positions the agent believes it could be in. Each of these possible location represent a different world. The yellow square is the goal the agent wants to go to. The blue circle is the agent. Here, is located in loc(3,0) but doesn't know it. So the first thing we want to do when we develop such epistemic Jason code, is to declare the Set of possible worlds of the agent.

In order to generate all possible worlds, we use a SAT solver (TouIST server). It receives a set of constraints and tries to find models that satisfy them.

The definition of those constraints are made in the Jason agent code. Figure 8 shows a snippet of this code.

An important point to note here is the use of the **range** literal on line 1. This should not be confused with the internal action **.range**, which is also used. The

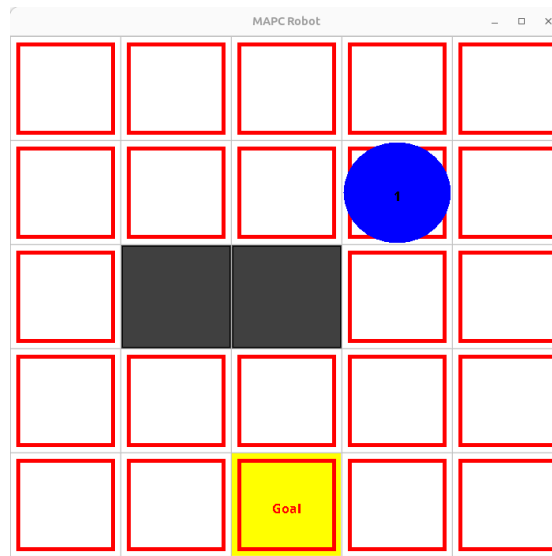


Fig. 7. Navigation under uncertainty

```

1  range(loc(X,Y)) :- (.range(X,0,4) & .range(Y,0,4)).
2  range(none) :- true.
3  none :- .findall(not(loc(X, Y)), range(loc(X, Y)), List) & .big_and(Y, List) & Y.
4  ~none.
5
6  ~loc(X1, Y1) :- loc(X2, Y2) & (X1 \== X2 | Y1 \== Y2).

```

Fig. 8. agent code snippet, constraints declarations

range literals are interpreted by the Jason interpreter and translated into logical constraints.

To define the *range* values here, we use the Internal Action `.range(X,Y,Z)`. This internal action unifies the variable `X` with all values between `Y` and `Z` (inclusive), one by one, using backtracking.

In this case, it will declare 25 range values.

```
loc(0,0),loc(0,1),... loc(0,4),loc(1,0),...,loc(4,3),loc(4,4).
```

Then, lines 2 to 4 ensure that the agent must be in at least one position; that is, it cannot be in no position at all. The Internal Action `findall(Var, Condition, List)` collects all values that `Var` can take such that `Condition` holds true, and returns these values as a list unified with `List`. The Internal Action `big_and(Res, List)` takes a list of logical formulas and constructs their conjunction as a single formula, which is unified with the variable `Res`.

Finally we will have :

```
none :- not(loc(0,0)) & not(loc(0,1)) & not(loc(0,2)) & ... & not(loc(4,4))
```

Then, line 4 indicates that it is not possible for the agent to not have a location.

Finally, line 6 enforces the mutual exclusivity of positions: if the agent is in one location, it cannot be in any other.

The constraints are then passed to the reasoner, which forwards them to the TouIST server. The server executes the corresponding command on the SAT solver. TouIST also stores these constraints in files located in the "cache_touist" folder, making them accessible for inspection.

On the reasoner's side, logs are also saved in the "logs/date_time/" directory. These logs allow observation of the agent's behavior, including the results of its evaluations and/or modifications to the semantic model.

Figure 9 shows the semantic model generated with the 23 possible worlds, each one corresponding to an agent position.

3.3 Events

At this point the agent will perceive if there is an obstacle in every of the four directions around him. The perceptions are handled by the Environment coded in Java. Here, the agent is not surrounded by anything, so he will have the following percepts : `~obs(right)`, `~obs(down)`, `~obs(left)`, `~obs(up)`.

Another key point to understand is the implementation of the `onEvents`. In the case of the DEL reasoner, the semantic model can evolve over time. The `onEvent` mechanism, introduced by M. Vezina, allows events to be applied to the model.

When using DEL, all belief events are transformed by prefixing them with `on`. Then, by writing specific-designed rules it is possible to configure how the model will evolve according to those events. In this case, the rules being referred to are those discussed earlier, automatically generated by the code during map reading. We can see such rules in Figure 10.

```

1 Agent mapc started
2 CURRENT MODEL :
3 0 : loc(4,4)
4 1 : loc(4,3)
5 2 : loc(4,2)
6 3 : loc(4,1)
7 4 : loc(4,0)
8 5 : loc(3,4)
9 6 : loc(3,3)
10 7 : loc(3,2)
11 8 : loc(3,1)
12 9 : loc(3,0)
13 10 : loc(2,4)
14 11 : loc(2,1)
15 12 : loc(1,4)
16 13 : loc(1,3)
17 14 : loc(1,1)
18 15 : loc(1,0)
19 16 : loc(0,3)
20 17 : loc(0,2)
21 18 : loc(0,1)
22 19 : loc(0,0)
23 20 : loc(0,4)
24 21 : loc(2,0)
25 22 : loc(2,3)

```

Fig. 9. model generated

```

64 +on(obs(D)) : obs(D).
65 +on(~obs(D)) : not obs(D).

```

Fig. 10. agent code snippet, on event for observations

For example, when a perception like $\sim\text{obs}(D)$ is added to the agent's belief base, it triggers the corresponding event (line 65): **not** $\text{obs}(D)$.

These onEvent rules share the same structure as standard Jason rules, but they behave differently. They follow a format of the form:

pre : **post**

Which means: if pre holds, then post. In this example, $\sim\text{obs}(D)$ leads to **not** $\text{obs}(D)$.

By unifying this event with previously declared rules (which also follow the “if pre then post” logic), the final event to be applied is determined. We can see those rules in Figure 11.

For example, the agent perceives no obstacle on his right. So the percept $\sim\text{obs}(\text{right})$ is added. The event $\text{on}(\sim\text{obs}(\text{right}))$ is generated such as : $\text{on}(\sim\text{obs}(\text{right})) :- \text{not } \text{loc}(0,2)$ due to rule line 19 in Figure 11.

When an onEvent is generated he is then sent to the reasonner. Reasonner apply the events received. In this case, all worlds containing the proposition $\text{loc}(0,2)$ will be removed.

At the end of this step, the list of possible worlds can be seen in Figure 12.

3.4 Evaluating formulas

The agent has the goal to navigate. three plans are existing for this goal as we can see in Figure 13. Since the epistemic evaluation occurs when determining the set

```
19  obs(right) :- loc(0,2).
20  obs(down) :- loc(1,1).
21  obs(down) :- loc(2,1).
22  obs(up) :- loc(1,3).
23  obs(up) :- loc(2,3).
24  obs(left) :- loc(3,2).
```

Fig. 11. agent code snippet, observations rules

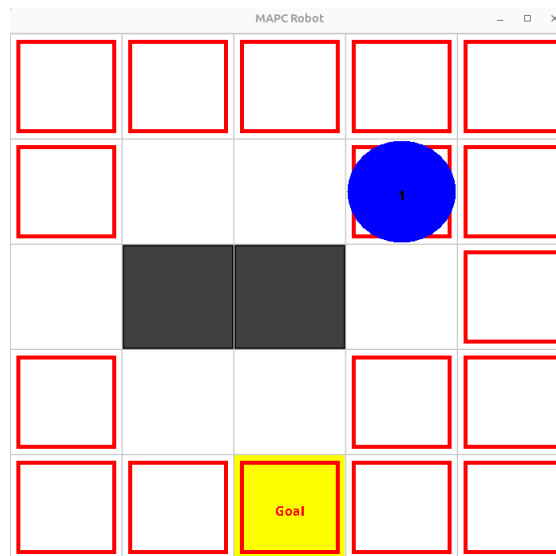


Fig. 12. agent mental state after first observation

of applicable plans, both certainty and uncertainty are taken into account. This makes it possible to later select the desired plan in the `selectOption` function. As Jason, by default, returns the first applicable plan based on the order of declaration, this approach prioritizes certainty (1.54) over uncertainty (1.58).

```

51   +!nav : at(goal)
52       <- .print("At goal.").
53
54   +!nav : dir(D, goal)
55       <- move(D);
56       !nav.
57
58   +!nav : poss(dir(D, goal))
59       <- move(D);
60       !nav.

```

Fig. 13. agent plans for the `!nav` triggering event

Here is how the evaluation works in this case. Foreach location, one to four rules indicating the direction to take are written. Here too, we are dealing with rules automatically generated by the code. Some of these rules can be seen in Figure 15.

```

17   // Direction mappings
18   dir(right,goal) :- loc(0,0).
19   dir(up,goal) :- loc(0,0).
20   dir(right,goal) :- loc(0,1).
21   dir(up,goal) :- loc(0,1).
22   dir(down,goal) :- loc(0,2).
23   dir(right,goal) :- loc(0,3).
24   dir(down,goal) :- loc(0,3).

```

Fig. 14. rules indicating direction to goal from a given location

In Figure 13 the context `dir(X,goal)` on line 54 will unify with all rules. For example considering the rule in Figure 15, line 18 :

`dir(right,goal) :- loc(0,0).`

it will generate the following formula to evaluate : `know(loc(0,0))`. In the epistemic extension, if no modality functor (i.e. `know` or `poss`) is present on a

literal, this one is automatically turned into a *KNOW* formula. This is not true for nested formulas.

For example :

`poss(A & B)` will stay the same and won't become `poss(know(A) & know(B))` as a modality is already present in front of this formula.

`A & poss(B)` will become `know(A) & poss(B)` as no modality is present in front of A, thus evaluating A means nothing.

So, for all values that can be unified with the first context we will evaluate the corresponding formula as shown previously. As none of these proposition are true in all worlds of the semantic model, the plan with the certainty context will not be considered applicable.

Then, as the position $\langle 0,0 \rangle$ (top left corner) is considered possible, the plan will be considered applicable with the Unifier corresponding to the `loc(0,0)` rule explained before : $X = \text{right}$. Following the plan, the agent will move to the right.

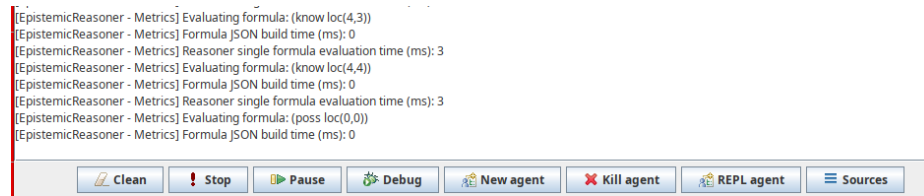


Fig. 15. Evaluation

3.5 Moving events

Still using the `onEvents`, this move event will update the worlds in the semantic model. `OnEvents`' rules can be seen in Figure 16.

```

14  +on(moved(right)) : loc(X, Y) & X < 4
15      <- -loc(X, Y);
16      +loc(X + 1, Y).
17
18  +on(moved(right)) : loc(X, Y) & X >= 4
19      <- -loc(X, Y);
20      +loc(0, Y).

```

Fig. 16. moved `onEvents` rules

For example, the world containing the proposition $\text{loc}(0,0)$ will become $\text{loc}(1,0)$. And so the possible worlds will now look like in the Figure 17.

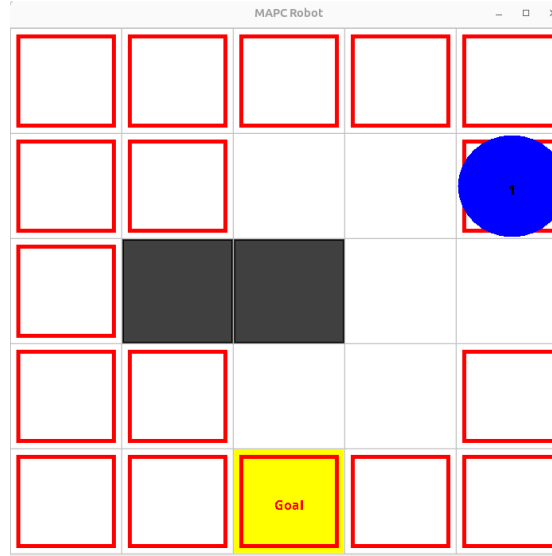


Fig. 17. Possibles worlds evolution after the first move

And then, it will just repeat this cycle. It will updates the worlds he considers possible according to his obstacle perceptions (Figure 18).

Then, will be moving right again as the agent still evaluates possible the fact that he is in position $\langle 0,0 \rangle$ (Figure 19).

The agent is then gonna arrive in position $\langle 1,1 \rangle$. He is gonna perceives an obstacle under him and thus determine with the corresponding `onEvent` that he might be in the possible positions :

```
on(obs(down)) :- loc(1,1).
on(obs(down)) :- loc(2,1).
```

As the agent just moved right, he can't be in the $\text{loc}(2,1)$. He now knows where he is (Figure 20) and will just have to follow his was defined by his rules.

To conclude, this example is likely the best starting point for exploring the epistemic extension, as it provides a clear and practical way for developers to understand how the extension works.

However, the rules generated during map loading introduce a subtle issue. The agent's movements and perceptions are managed by two distinct classes: one representing the environment and the other the grid in which the agent operates. The class that represents the grid is currently responsible for generating the agent's behavioral rules (such as `obs` and `dir`). This design conflicts with

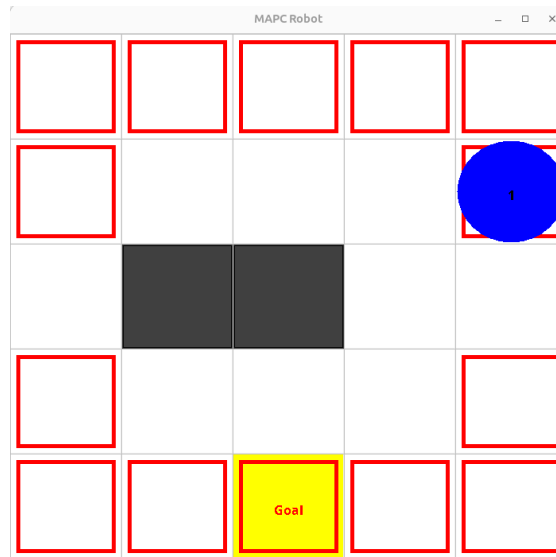


Fig. 18. second obstacle perceptions evolution

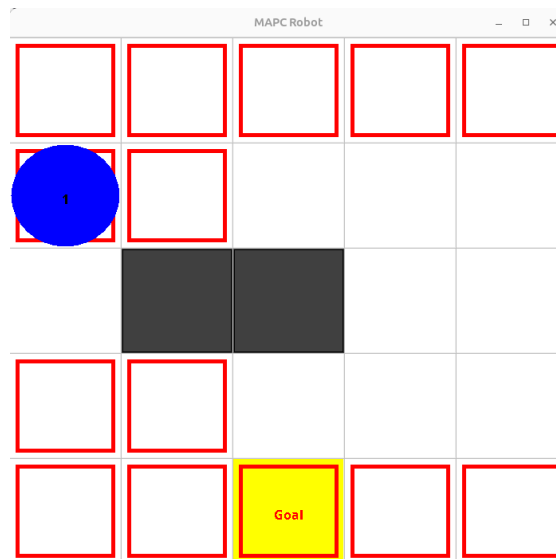


Fig. 19. Possibles worlds evolution after the second move

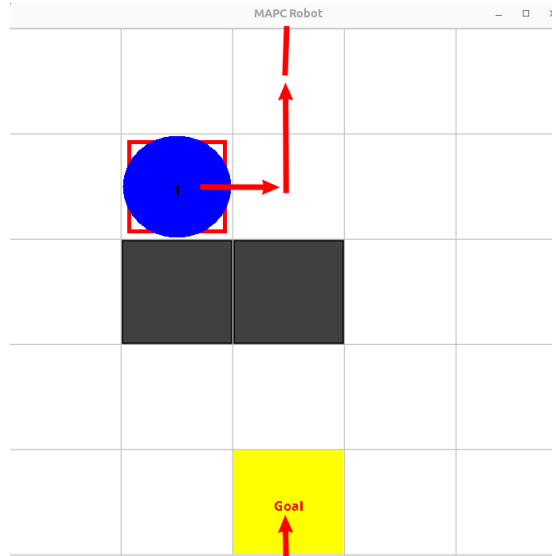


Fig. 20. Agent certainty about his own location

both the BDI paradigm and the possible worlds semantics. It is conceptually problematic for an agent to receive knowledge-related information from a component that has an omniscient view of the world. The only acceptable case for external rule generation should be for perceptions — for example, a class checking whether there is an obstacle to the agent's right. This aligns with the idea that perception is based on limited, local information, not global knowledge.

4 Case Study : Navigation with uncertainty over own localization & goal localization. Changing the model

In this case, the DEL (Dynamic Epistemic Logic) mode is used for the reasoner. To run this project, you have to run the **navigation-changing.mas2j** file in the **ProjectRunner** file. In this example, the scenario is the same as the previous example with an additional source of uncertainty for the agent :

- the localization of his goal.

Here, each pair $\langle \text{agent position}, \text{goal position} \rangle$ corresponds to a possible world. Thus, if the agent considers three possible goal positions on the same map as before, the number of possible worlds will triple. As a result, it is no longer sufficient to evaluate only the agent's position — each pair of $\langle \text{agent position}, \text{goal position} \rangle$ must now be considered. For instance, if the agent is at $\langle 1,1 \rangle$, it should move left if the goal is at $\langle 0,1 \rangle$, but it should move right if the goal is at $\langle 2,1 \rangle$.

This approach scales very poorly and might lead to an exponential evaluation problem in more complex cases. Therefore, a solution was needed to first evaluate something more general, and only later refine the evaluation to more specific formulas as the set of possible worlds narrows down.

As a result, it was decided to evaluate the directions the agent should take instead.

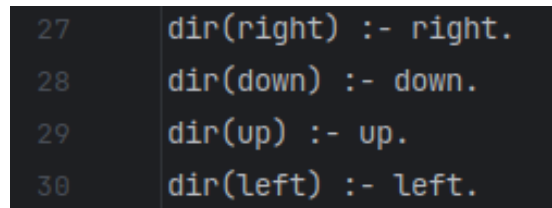
Since it is not feasible to manually configure the entire set of possible worlds, this had to be adjusted along the way. This scenario will therefore illustrate how to configure the reasoner's semantic model in a more in-depth and flexible manner.

For this example, an interesting thing to do might be to have a look at the `NavAgent` class while you are reading this.

4.1 Constraints generation and map reading

Another advantage of this approach is that it also solves the problem of generating the agent's behavioral rules from an omniscient structure. Indeed, all the lines corresponding to the constraints and the agent's behavior rules are directly declared in the agent's code. Furthermore, we no longer need one or more rule(s) indicating the direction to take in based on the agent's position as we now directly evaluate the directions.

These four rules are showed in Figure 21.



```
27    dir(right) :- right.
28    dir(down)  :- down.
29    dir(up)    :- up.
30    dir(left)  :- left.
```

Fig. 21. Agent rules to evaluate directions

The constraints for generating the worlds are the same as before, with just a few additional lines to introduce uncertainty about the goal.

Thus, in the semantic model initially generated none of the worlds contain any direction.

To add the directions, the process works as follows:

- First, it's important to understand that the `EpistemicAgent` class provides two functions: `modelCreated()` and `eventModelApplied(Event event)`, which are respectively notified when the model is first created and whenever an event is applied to it.
- These functions are meant to be overridden by the user to allow custom configuration.

- In this case, the `NavAgent` class extends `EpistemicAgent` and overrides these functions.
- When the model is created, the `modelCreated()` function is invoked. At that point, the agent retrieves the full set of possible worlds from the reasoner. For each pair <agent position, goal position>, it runs a **BFS** algorithm on the map to find the shortest path. The fastest directions for each pair <agent position, goal position> are stored in a **HashMap**, allowing the agent to avoid recalculating paths later on.
- Finally, the semantic model is replaced with a new one in which each pair <agent position, goal position> is associated with the corresponding set of precomputed directions.

Figure 22 shows the `modelCreated()` function from the `NavAgent` class.

```
//This function is called when the model has been created
@Override no usages @ethan
public void modelCreated(){
    System.out.println("Model created");
    this.readMap(MapType.LOCALIZATION_5x5);
    ModelResponse model = this.getWorldsResponseModel();
    for(World world : model.getWorlds()){
        Pair<String, String> locationsValues = getLocandLocGoalValues(world);
        shortestDirectionsFromLocationToOtherLocation.put(locationsValues, getAllShortestDirections(stringToLocation(locationsValues
    }
    try {
        replaceSemanticModel(model);
    } catch (RuntimeException e) {
        throw e;
    }
    // for the rule dir(direction) :- direction to be applied, so we can test if we know right, left, up or down,
    // the directions need to be in the BB. But as we don't want them all to be added to the reasoner worlds, we add them manually.
    try {
        this.addDirectionsToBB();
    } catch (JSONException e) {
        throw new RuntimeException(e);
    }
}
```

Fig. 22. Code snippet : the `modelCreated()`'s function

Figure 23 presents a snippet of the semantic model after the directions had been added.

4.2 Event Appliance

Then each time an event has been applied — specifically, the agent has moved, thereby changing its position values across the possible worlds — the directional consequences must also be updated. In this case:

- The full set of possible worlds is retrieved.
- All propositions related to directions are removed.
- For each pair <agent position, goal position>, the corresponding directions are retrieved from the **HashMap** and added to the matching world.

```

82  GETTING MODEL FOR AGENT
83  ***REPLACE-PROPS***
84
85  RESULT : simple_nav_optim
86  0 : left,loc(4,4),locGoal(2,3),up
87  1 : left,loc(4,3),locGoal(2,3)
88  2 : down,left,loc(4,2),locGoal(2,3)
89  3 : down,left,loc(4,1),locGoal(2,3)
90  4 : down,left,loc(4,0),locGoal(2,3)

```

Fig. 23. Snippet of the semantic model with directions

- As before, the semantic model is then replaced with a new model containing these updated worlds.

Figure 22 shows the `eventModelApplied(Event event)` function from the `NavAgent` class.

```

//This function is called each time an event has been applied to the model
@Override no usages @ethan
public void eventModelApplied(Event event){
    // we update the directions of the worlds in the model only if the agent moved and his possibles locations changed
    if(event.getTrigger().getOperator().equals(Trigger.TEOperator.add) && event.getTrigger().getLiteral().toString().startsWith("on(")){
        if(event.getTrigger().getLiteral().toString().startsWith("on(moved(") {
            try {
                ModelResponse model = this.getWorldsResponseModel();
                replaceSemanticModel(model);
            } catch (RuntimeException e) {
                throw e;
            }
        } else {
            try {
                ModelResponse model = this.getWorldsResponseModel();
                updateViewFromSemanticModel(model);
            } catch (RuntimeException e) {
                throw e;
            }
        }
    }
}

```

Fig. 24. Code snippet : the `eventModelApplied(Event event)`'s function

4.3 Evaluation and Code Execution

In this case the agent directly evaluates if he is whether certain of which directions he should take or if he considers a direction possible to be taken (Figure 25).

The rules related to obstacle perceptions around the agent remain in place, allowing it to determine its possible locations.

Finally, in this example different plans and rules are used to eliminate possible worlds (Figures 26 & 27):

```
[EpistemicReasoner - Metrics] Evaluating formula: (know left)
[EpistemicReasoner - Metrics] Formula JSON build time (ms): 0
[EpistemicReasoner - Metrics] Reasoner single formula evaluation time (ms): 1
[EpistemicReasoner - Metrics] Evaluating formula: (poss right)
[EpistemicReasoner - Metrics] Formula JSON build time (ms): 0
```

Fig. 25. Evaluation of directions by the agent

- If the agent does not perceive that he is at the goal, then all worlds in which the agent's position matches the goal's position are discarded (lines 107-109 in Figure 26 and line 70 in Figure 27). On line 109, we can see that a DEL belief deletion event is written just after the corresponding DEL belief addition event occurs/is written. This deletion won't have any effect as no onEvents/onRules has been written for it. Somehow, she is necessary because in Jason, a belief addition occurs only if the belief isn't in the Belief Base. Thus, we have to delete this belief in the agent's BB if we want to be able to apply the corresponding onEvent at each movement of the agent.
- If the agent is certain of its own position, it will generally choose a temporary goal (lines 93-94 in Figure 26). This helps prevent the agent from getting lost in its decision-making. In such cases, the agent decides to first move toward a specific target, even if it is not yet certain that this is the actual goal. In fact, in the context of a nav plan, line 76, if the agent has an objective, he will get the direction corresponding to this objective. And, this objective will be removed at line 101 if it's not the real goal.

```

76 +!nav : poss(dir(D) & locGoal(X2,Y2) & obj(X2,Y2))
77     <- move(D);
78     !checkIfGoal;
79     !nav.
80
81 +!nav : dir(D)
82     <- move(D);
83     !checkIfGoal;
84     !addTemporaryObjective;
85     !nav.
86
87 +!nav : poss(dir(D))
88     <- move(D);
89     !checkIfGoal;
90     !addTemporaryObjective;
91     !nav.
92
93 +!addTemporaryObjective : loc(X,Y) & poss(locGoal(X2,Y2))
94     <- +obj(X2,Y2).
95
96 +!addTemporaryObjective : true
97     <- .print("not sure about where I am...").
98
99
100 +!checkIfGoal : not at(goal) & loc(X,Y) & poss(locGoal(X,Y))
101     <- -obj(X,Y);
102     +~locGoal(X,Y).
103
104 +!checkIfGoal : not at(goal) & locGoal(X,Y) & poss(loc(X,Y))
105     <- +~loc(X,Y).
106
107 +!checkIfGoal : not at(goal)
108     <- +~na(goal);
109     -~na(goal).

```

Fig. 26. Different plans in agent's code

```
67      +on(~loc(X,Y)) : not loc(X,Y).  
68      +on(~locGoal(X,Y)) : not locGoal(X,Y).  
69  
70      +on(~na(goal)) : not (loc(X,Y) & locGoal(X,Y)).
```

Fig. 27. Different rules in agent's code