MECA Planner 0.2

User Manual

May 5, 2021

Planning problems for the MECA Planner are specified in the doxastic/epistemic planning language: *depl*. A single depl file describes both a planning domain and a planning problem. The grammar of depl is defined with the ANTLR meta-language in the file: `src/deplParser/Depl.g4`.

1 Example

We will use an example depl file, which is located at `problems/example.depl`.

This describes a planning domain involving two agents: a robot, whose behavior will be determined by the planner, and a human, whose behavior is estimated by a predictive model. There are two rooms and two hallways: each hallway connects the two rooms. The first room contains a coin and both agents, the second room contains a pizza. The robot does not know whether the coin lies heads- or tails-up, althought the human does. The robot knows about the pizza in the second room, but the human is unaware of the pizza. The robot's goal is to know how the coin lies without the human knowing that the robot knows this. The robot can determine how the coin lies by looking at it, but if the human is in the same room, she will observe this and know that the robot knows. The human is hungry, and would leave for the second room if she know about the pizza (according to the predictive model). A succesful plan is for the robot to announce to the human that there is pizza in the other room, wait for the human to leave by either of the halls, and then observe the coin. Here is the example depl file describing this problem:

```
types{
    Robot-Actor,
    Human-Actor,
    Actor-Locatable,
    Food-Locatable,
    Locatable-Object,
    Location-Object
}

objects{
    robot1-Robot,
    human1-Human,
    pizza-Food,
    roomA - Location,
    roomB - Location,
    hall1 - Location,
    hall2 - Location,
}

agents{
    robot1,
    human1{HumanModel},
}
```

```
fluents{
    at(Locatable, Location),
    heads(),
    hungry()
}

constants{
    !connected(Location, Location),
    connected(roomA,hall1),
    connected(roomA,hall2),
    connected(hall1,roomB),
    connected(hall2,roomB),
}

initially{
    *w1 <- {
        at(robot1, roomA),
        at(human1, roomA),
        at(pizza,roomB),
        heads()
    }
    *w2 <- {
        at(robot1, roomA),
        at(human1, roomA),
        at(pizza,roomB),
    }
    w3 <- {
        at(robot1, roomA),
        at(human1, roomA),
        heads()
    }
    w4 <- {
        at(robot1, roomA),
        at(human1, roomA),
    }

    B[robot1] <- {(w1,w1),(w1,w2),(w2,w2),(w2,w1),
                  (w3,w3),(w3,w4),(w4,w4),(w4,w3)},
    K[robot1] <- {(w1,w1),(w1,w2),(w2,w2),(w2,w1),
                  (w3,w3),(w3,w4),(w4,w4),(w4,w3)},
    B[human1] <- {(w1,w3),(w2,w4),(w3,w3),(w4,w4)},
    K[human1] <- {(w1,w1),(w1,w3),(w3,w3),(w3,w1),(w2,w2),(w2,w4),(w4,w4)
       ,(w4,w2)},
}

goals{heads()}
//goals{(B[robot1]heads() & !B[human1]B[robot1]heads()) |
//      (B[robot1]!heads() & !B[human1]B[robot1]!heads())}

actions{
    move(?a-Actor,?f-Location,?t-Location){
        owner{?a},
        precondition{at(?a, ?f)},
```

```
        precondition{connected(?f,?t)|connected(?t,?f)},
        //observes(?o-Actor){?o if at(?o,?f) | at(?o,?t)},
        observes(?o-Actor){?o if at(?o,?f)},
        causes{at(?a, ?t)},
        causes{~(at(?a, ?f))},
    }

    eat(?l-Location){
        owner{human1},
        precondition{at(human1, ?l)},
        precondition{at(pizza, ?l)},
        //observes(?o-Actor){?o if at(?o,?l)},
        observes(?o-Actor){?o},
        causes{~hungry()},
    }

    announcePizza(){
        owner{robot1},
        observes(?o-Actor){?o},
        announces{at(pizza,roomB)},
    }

    look(){
        owner{robot1},
        precondition{at(robot1, roomA)},
        //observes(?o-Actor){?o if at(?o, roomA)},
        observes(?o-Actor){?o},
        determines{heads()}
    }

    wait(?a - Actor,?l-Location) {
        owner{?a},
        precondition{at(?a, ?l)},
        //observes(?o-Actor){?o if at(?a,?l)},
        observes(?o-Actor){?o},
    }
}
```

2 Structure

A depl file contains the following sections, which must occur in the correct order, and are required unless specified as optional:

- types
- objects
- agents
- passive (optional)
- fluents
- constants (optional)
- initially

- goals
- actions

Each section begins with the section name, followed by "`{`", then the section contents, and then "`}`".

2.1 Whitespace and newlines

Whitespace and newlines are ignored, as are, C-style inline comments (anything following "`//`") and block comments ("`/*...*/`").

2.2 types

The *types* section defines a type heirarchy relating all objects in the planning problem. The content of the *types* section consists of a comma-separated list of type definitions. A type definition takes the form "`Subtype - Supertype`". The type "`Object`" is built-in, and the type heirarchy must be constructed such that every type is a subtype of "`Object`". A type name begins with an uppercase letter, followed by any number of upper- or lower-case letters, integers, and underscores.

The depl parser will use the type heirarchy defined in this section for two purposes. First, it will provide type checking of objects as a guard against errors. Second, it will automatically expand some statements that contain types into a set of statements instead containing objects of those types, providing a convenient short-hand. The type heirarchy is a strictly parse-time entity: no type information will be available to the planner.

2.3 objects

The *objects* section defines the objects in the planning problem. The content of the *objects* section consists of a comma-separated list of object definitions. An object definition takes the form "`object - Type`". Each type must either be "`Object`", or be defined in the *types* section. An object name begins with a lowercase letter, followed by any number of upper- or lower-case letters, integers, and underscores. Objects will be used as arguments to predicates to build a set of atomic boolean fluents (propositions).

2.4 agents

The MECA planner uses three types of *agents*. The single *system* agent and any number of *environment* agents are specified in this section. The third type, *passive* agents, are specified in the next section. An *agent* must be an object that has been defined in the *objects* section. An agent can have any type, but it is generally convenient to create a type for objects that will be agents, for example as we do with the "`actor`" type in our running example.

The *agents* section consists of a comma-separated list of agent definitions. There must be at least one agent definition. There must be exactly one *system* agent, definitions, the rest must be *environment* agent definitions. A system agent definition consists of only an agent name. An environment agent definition takes the form "`name{Model}`", where `name` is an agent name and `Model` is the name of a model class. The model class name must begin with an uppercase letter, followed by any number of lower- and upper-case characters, integers, and underscores. The model name must be the same as the name of a java class that extends `mecaPlanner.models.Model`, and the class should be defined in a `.java` file `src/mecaPlanner/models/`.

This example defines a system agent, `robot1`, and a single environment agent, `human1`. The order in which agents are defined determines the order in which they will act. thus `robot1` will act first, followed by `human1`, and then `robot1` again, etc. The planner will query `PizzaModel` to determine the predicted actions of `human1`, and will attempt to construct a plan that specifies the actions of `robot1`.

The planner considers two main things with *system* and *environment* agents. First, as discussed above, it considers their actions (as specified by the planner for *system* agents, as predicted by a model for *environment* agents. Second, it models their beliefs and knowledge, which are represented as part of the states over which the planner searches.

The depl parser and the Meca planner will use the information defined in this section for three purposes. First, agents will be associated with actions. Second, and the model assigned to each environment agent will be queried to predict that agent's actions. Third, the epistemic and doxiastic state and action systems will maintain representations of agents' mental states.

2.5 passive

This optional section defines *passive* agents, whose beliefs and knowledge are modeled by the planner (as with system and environment agents), but who do not act (unlike system and environment agents). The passive section consists of a comma-separated list of passive agent definitions. A passive agent definition takes the same form as a system agent definition: an agent name.

2.6 fluents

The *fluents* section defines all fluent atoms that will be available to the planner. A fluent atom consists of a predicate and a (possibly empty) orderd list of arguments. Arguments are objects, as defined in the *objects* section.

This section contains a comma-delimited list of fluent definitions. A fluent definition taks the form `name(p1,...,pn)`, where `name` is the predicate, which begins with a lowercase letter, followed by any number of upper- and lower-case letters, integers, and underscores, and each element of `(p1,...,pn)` is an argument. An argument is *either* an object name *or* a type name. If all arguments of a fluent definition are object names, a single fluent is defined. If any arguments are type names, the fluent definition is automatically expanded, substituting all objects that are of the specified type(s), in all combinations, to construct multiple fluents.

2.7 constants

This optional section defines constant (either true or false) atoms. This *constants* section consists of a comma-delimited list of constant definitions, where a constant definition takes either the form `name(p1,...,pn)` (true), or `!name(p1,...,pn)` (false). Automatic type-expansion is allowed as with fluent definitions.

If a constant is defined multiple times, its previously-defined values will be overriden. Thus, we could use type-expansion to construct a large number of false constants, and then override some of them to be true. As an example, separate from our running pizza-robot example (which does not use constants), consider a domain having many rooms, some (but not most) of which are connected to each other. A constants section specifing these constraints might look like this:

```
constants{
    !connected(Room, Room),
    connected(room1,room2),
    connected(room1,room3),
    connected(room3,room4),
    connected(room4,room5),
}
```

Similarly to the type heirarchy, constant definitions are a parse-time entity. The planner does not have access to them. Wherever a defined constant is found within a depl file, it is replaced withe a *true* literal or a *false* literal. Cosntants can be used to simplify and clarify some definitions, especially action definitions. For example, with the given example *constants* section, we might define a *move* action that transitions between two rooms (called, perhaps, `?from` and `?to`, see the *actions* section below) only if they are connected, specifying as a precondition that `connected(?from,?to) | connected(?to,?from)`. If this were done using fluents instead of constants, the parser would generate (and give to the planner) an action for every pair of rooms. As ther planner searched for a plan, it would repeatedly consider each of these move actions, only to discover that the preconditions are never satisfied for the vast majority of them. If the `connected` constraints are defined as constants, the parser determines that the preconditions for most of the possible *move* actions are constatnly false, and only generates (and passes to the planner) actions for movement between connected rooms.

2.8 initially

This section defines the start state (or possibly a set of start states) for the planner. A state definition has two parts. First, worlds and their valuations are defined. Then, per-agent knowledge and belief relations are defined over worlds.

Each world definition takes the form `w <- {f1,...,fn}` where `w` is a world name (used to reference the world when later definingrelations) and each of `f1` ... `fn` is a fluent that should be true in that world (unlisted fluents are false).

B[human1] ¡- (w1,w1),

The parser will reject Kripke structures that to not satisfy $S5_n$(on the knowledge relation) $KD45_n$(on the belief relation), and $KB1$and $KB2$(between the belief and knowledge relations).

2.9 The *goals* section

The *goals* section specifies the goals the planner tries to achieve. This section contains a comma-delimited list of goal formulae. The parser and's these together to construct a single goal formula. A goal g takes the form:

$$g :- \texttt{true} \mid \texttt{false} \mid f \mid (g) \mid \texttt{B}[a]g \mid g\&g \mid g|g \mid !g \mid \texttt{timestep}ei$$

where f is an (fluent or constant) atom, a is an agent, $e :-$ `==` | `!=` | `<` | `<=` | `>` | `>=`, and i is an integer.

$P[i]g$ (i believes possibly g) is provided as syntactic sugar for $!B[i]!g$.

Other supported syntax includes $K[i]g$, i knows g, and Cg, g is common knowledge. However, these or not really recommended. The knowledge relatin is intended primarily for performing belief revision when false beliefs are corrected, not to represent agent's beliefs. The implementation of common knowledge is equivalent to "everyone knows", not infinite iteration of shared belief.

An example goals section is:

```
goals{
    at(robot1,room1),
    timestep<=7
}
```

2.10 The *actions* section

2.11 Optional syntax

The depl syntax tries to be flexible. Any "`&`" (and) can instead be written as "`&&`". Any "`|`" (or) can instead be written as "`||`". Any "`!`" (not) can instead be written as "∼".

3 Test

The best way to test a depl file is with the `test` program.