MECA Planner 0.2

User Manual

April 2, 2021

1 The depl planning language

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.

For the complete depl grammar, defined with the ANTLR meta-language, consult `mecaPlanner/src/deplParser/Depl.g4`

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

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 "`}`".

1.1 The *types* section

The *types* section defines a type heirarchy relating all objects in the planning problem. Here is an example *types* section:

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

The content of the *types* section consists of a comma-separated list of type definitions. A type definition takes the form "`Subtype - Supertype`" (whitespace surrounding the "`-`" is optional). 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 main purposes. First, it will provide type checking of objects as a guard against errors. Second, as we will see in some later sections, 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. Note, however, that the type-heirarchy is a parse-time entity only, no type information will be available to the planner.

1.2 The *objects* section

The *objects* section defines the objects in the planning problem. Here is an example *objects* section:

```
objects{
    robot1-Robot,
    human1-Human,
    pizza-Food,
    hotdog-Food,
    room1-Location,
    room2-Location,
}
```

The content of the *objects* section consists of a comma-separated list of object definitions. An object definition takes the form "`object - Type`" (whitespace surrounding the "`-`" is optional). 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.

1.3 The *agents* section

The MECA planner considers three types of *agents*. The single *system* and arbitrarily-numberd *environment* agents are specified in this section. The third type, *passive* agents, are specified in the next section.

An *agent name* 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 contains only an agent name (i.e. an object that has been defined in the *objects* section). 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` (the class should be defined in a `.java` file within `mecaPlanner/src/mecaPlanner/models/`.

The order in which agents are defined determines the order in which they will act.

Here is an example *agents* section:

```
agents{
    robot1,
    human1{PizzaModel},
}
```

This example defines a system agent, `robot1`, and a single environment agent, `human1`. `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.

1.4 The *passive* section

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, i.e. an object name as has been defined in the *objects* section.

1.5 The *fluents* section

This section defines all fluent atoms that will be available to the planner. A fluent atom has a name and a (possibly empty) orderd list of parameters. Parameters are objects, as deinfed in the *objects* section.

This section contains a comma-delimited list of fluent definitions. A fluent definition taks the form `name(p1 \ldots pn)`, where `name` is the fluent name, which begins with a lowercase letter, followed by any number of upper- and lower-case letters, integers, and underscores, and each $p \in$`(p1 \ldots pn)` is a parameter definition. A parameter definition is *either* an object name *or* a type name. If all parameter definitions for a fluent definition are object names, a single fluent is defined. If any parameter definitions are type names, the fluent definition is expanded, substituting all objects that are of the specified type(s), in all combinations, to construct multiple fluents. For example, given the previous example *types* and *objects* example sections, the following two example *fluents* sections are equivalent:

```
fluents{
    at(robot1, room1),
    at(human1, room1),
    at(pizza,  room1),
    at(hotdog, room1),
    at(robot1, room2),
    at(human1, room2),
    at(pizza,  room2,
    at(hotdog, room2),
    human_hungry(),
    door_open(),
    robot_charged(),
}

fluents{
    at(Locatable, Location),
    human_hungry(),
    door_open(),
    robot_charged(),
}
```

1.6 The *constants* section

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 \ldots pn)` (true), or `!name(p1 \ldots 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 (the planner is concerned with change). Anywhere a defined constant is found, it is replaced withe a true literal or a false literal. depl uses constants 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 (and just being careful never to change their values), the parser would generate (and give to the planner) an aciton 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 we defining the `connected` constraints 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.

1.7 The *initially* section

This section defines a set of start states (*ed-states*) for the planner. Each start state can be defined directly (the *manual* option, currently the only option), or can be automatically generated from a set of propositions (the *automatic* otpion, not yet implemented). If a single start state is defined, there is no plan-time uncertainty. Multiple start states represent plan-time uncertainty: the planner will not know which is the actual start state, and will attempt to find a conformant solution.

This section contains a list of start-state definitions. Each start state definition is enclosed within braces (`{}`). Start state definitions are optionally delmited by commas. If only a single start state is defined, the enclosing braces (which are within the overall *initialy* section braces) may be ommitted.

1.7.1 Manual

A manual 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).

1.7.2 Automatic

Not implemented.

1.8 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:

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

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
}
```

1.9 The *actions* section

1.10 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 "∼".

2 Test

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