On Modeling Formalisms for Automated Planning*

Jindřich Vodrážka and Roman Barták

Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
{vodrazka,bartak}@ktiml.mff.cuni.cz

Abstract. Knowledge engineering for automated planning is still in its childhood and there has been little work done on how to model planning problems. The prevailing approach in the academic community is using the PDDL language that originated in planning competitions. In contrast, real applications require more modeling flexibility and different modeling languages were designed in order to allow efficient planning. This paper focuses on the role of a domain modeling formalism as an interface between a domain modeler and a planner.

Keywords: knowledge modeling, automated planning.

1 Introduction

We can imagine problem solving as a journey from problem specification to problem solution. The approach taken by automated planning is a model-based one - we first design a model that formally describes our knowledge about a given area of interest (domain) and later we can exploit this knowledge to solve various instances of problems in the domain using a general purpose planner. The journey in this case can be divided to two phases:

1. *knowledge modeling* - performed by human experts. All relevant information about the problem is put together in order to create a domain model. The model is usually described within some knowledge-modeling formalism.
2. *planning* - performed by an automated planner. The planner is working only with the domain model specified in the first phase and with the description of one particular problem instance.

The knowledge modeling formalism is the dividing point between the two phases.

In this paper we will show that the position of this dividing point can define a tradeoff between simplicity and usability of the modeling formalism on one side and efficiency of the resulting model on the other side. Finally we will introduce a new planning formalism designed to balance this tradeoff.

* This work is supported by the Czech Science Foundation under the project No. P103/10/1287.

G. Agre et al. (Eds.): AIMSA 2014, LNAI 8722, pp. 270–277, 2014.
© Springer International Publishing Switzerland 2014

2 Planning Domain Example

In order to illustrate our point of view we will be refering to the Petrobras planning domain [6] which was one of the domains of the Challenge track of the International Competition on Knowledge Engineering for Planning and Schedulling, ICKEPS 2012, as part of the ICAPS 2012 conference.

In the Petrobras domain we have a fleet of ships, a list of locations (ports, waiting areas, and platforms), and cargo. Each ship has limited cargo capacity and a fuel tank. The ships consume fuel while navigating between the locations.

The main goal is to deliver cargo from ports to platforms. There are six operations, *navigate*, *dock*, *undock*, *load*, *unload* and *refuel*, that each ship can do. Only refueling can run in parallel with loading or unloading, while any other pair of operations for a single ship cannot overlap in time. The ship must be docked before loading, unloading, and refueling and it must be undocked before navigating. We are given the initial locations and fuel levels of ships and the initial location, weight, and destination of each cargo item. The task is to plan operations for ships in such a way that all cargo items are delivered.

Class hierarchy. When modeling a planning domain it is natural to describe classes of involved objects. The base class hierarchy can be described independently on the formalism used. We will be refering to the following class hierarchy for the Petrobras domain:

- `Ship` - a class for the transport ships
- `Cargo` - a class for the items of cargo
- `Location` - a generic class for points of interest. Distances between pairs of locations are part of the planning problem specification.
  - `WaitingArea` - areas designated for idle ships
  - `LogisticLoc` - locations where a ship can be loaded/unloaded
    - `Platform` - location with docking capacity for a single ship
    - `Port` - location with docking capacity for more than one ship. Refueling is possible only at ports and selected platforms.

3 Domain Designer Perspective

Design of the planning domain model can be done in a modeling formalism that is completely independent of the planning machinery. This is the case of the Planning Domain Definition Language (PDDL) [3].

PDDL model. We have already described the class hierarchy earlier. Now we will give the description of selected object properties with *predicates* and *fluents*. Both predicates and fluents represent properties that are subject to change during the planning process. *Predicates* in PDDL are used for boolean properties whereas *fluents* can take wider range of values (e.g. numeric fluents).

In PDDL we use instances of *predicates* and *fluents* to describe a state of the world and we use *actions* to model possible transitions between those states.

Predicates: In PDDL we can represent information about the location of each ship and item of cargo, using first-order-logic *predicates* with typed arguments (e.g. `(ship-at ?s - Ship ?l - Location)`).

Fluents: For each ship we can express its current fuel level and free cargo capacity as *numeric fluents* (e.g. `(fuel-level ?s - Ship) - number`).

Actions: Legal changes that can turn one state to another state are described by *actions*. An example of a PDDL code for action `load-cargo` follows[1]:

```
(:action load-cargo
  :parameters (?s - Ship ?c - Cargo ?loc - Location)
  :precondition (and
                 (at ?s ?loc)
                 (cargo-at ?c ?loc)
                 (>= (free-cargo-capacity ?s) (cargo-weight ?c))
                 (isDocked ?s ?loc))
  :effect (and
           (not (cargo-at ?c ?loc))
           (cargo-at ?c ?s)
           (decrease (free-cargo-capacity ?s) (cargo-weight ?c))))
```

Each predicate, fluent, and action declaration in PDDL actually represents a schema with variables. If we assign constants to these variables we can obtain many different predicates, fluents, and actions. This process is called *grounding*.

The set of grounded predicates, that are true at the moment, together with the values of all fluents are used to describe a world state.

A grounded action is *applicable* to a given state iff all *preconditions* are satisfied. The state changes according to the *effects* of the action (i.e. some predicates are made true/false and values of some fluents are changed).

The main idea of PDDL is to describe domain physics only. However, if we consider actions one by one, the physics alone may not suffice. For example a sequential plan can contain a docking action followed immediately by an undocking action, which is a valid but not reasonable sub-plan. Additional work has to be done in order to encode action ordering rules that would be useful for a planner. Therefore we conclude that the PDDL interface is closer to the modeler.

4 Planner Perspective

The New Domain Definition Language (NDDL) is an example of a modeling formalism that is strongly influenced by the target planner. NDDL is a part of the EUROPA2 planning system [1].

[1] PDDL uses Lisp-like syntax.

NDDL model. In addition to the base class hierarchy we need to define special classes that represent domain *attributes*. Each ship in the Petrobras domain can be described by two numeric attributes (`fuelLevel`, `cargoCapacity`) and one multi-valued state variable `shipState`, which can take values such as: `Navigating(loc1,loc2)` (ship is navigating from `loc1` to `loc2`) or `Loading(loc3,c1)` (ship is loading `c1` at `loc3`). All possible values are described with predicates.

```
class shipState extends Timeline {
   predicate Navigating { Location from; Location to; }
   predicate Loading { LogisticLoc dock; Cargo crate; }
   ... }
```

Tokens: Each predicate defined in this way can be used in a *token* which is a triple (O, P, I) where:

- O - is an object i.e. instance of some class (e.g. `shipState`)
- P - is a predicate of the class (e.g. `Navigating(X,Y)`)
- I - is a time interval $[s, t]$ where $s < t \leq H$ for some fixed value H (planning horizont)

Timelines: A sequence of *tokens* $T = (t_1, \ldots, t_k)$ with non-overlapping intervals:

$$\forall i \neq j : (I(t_i) = [a, b] \wedge I(t_j) = [c, d]) \Rightarrow (b \leq c \vee d \leq a)$$

is called a *timeline*, and it describes the history of a state variable. We can indicate this fact by the code `class X extends Timeline`. If the intervals defined in T cover all the time from 0 to H then T is a *completely specified* timeline. If there are some uncovered intervals the timeline is *partially specified*.

The planner starts with a set of *partially specified* timelines and its objective is filling in the gaps with matching tokens. Any set of partially specified timelines represents a *partial solution* of the original problem. In a *complete solution* all the timelines are *completely specified* in a way that satisfies all *constraints* defined in the domain model.

Constraints: To decide whether a token can extend a given partial solution, the NDDL model has to define temporal constraints (based on the Allen's interval algebra) among tokens. Figure 1 shows a diagram of token relations for action `LoadCargo`. Solid boxes represent tokens that have to exist in the solution in order to allow the addition of tokens represented by dashed boxes.

We have used only a small subset of NDDL features in our example to illustrate the position of the interface between a domain modeler and a planner. The language describes domain knowledge with timelines, tokens, and constraints – the structures used by the planner in the planning process. Valid sequences of tokens on a timeline can be deduced from the constraints described in the domain model. However, this requires an additional effort on the side of a domain modeler who has to design the constraints. This intuition leads us to conclusion that the NDDL interface is closer to the planner in this case.

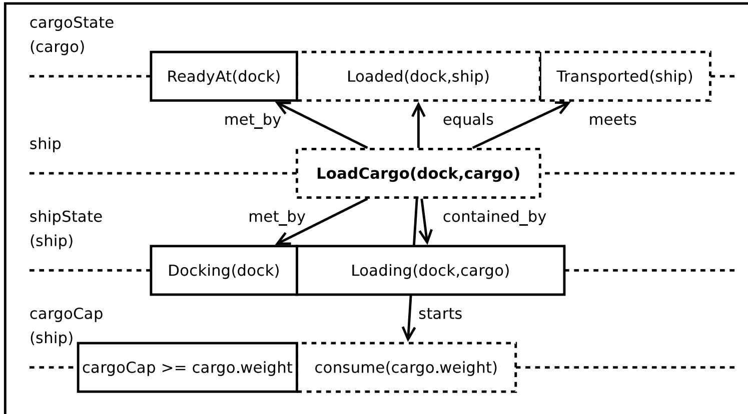

Fig. 1. Token relations in LoadCargo

5 Comparison

We have seen how two different modeling formalisms can be used to model one planning domain. Now we will analyze the action `LoadCargo` in more detail. Let us suppose that our knowledge can be summarized as follows.

There are three conditions: -

- both **ship** and **cargo** have to be at the same **location**,
- **ship** has to be docked,
- there has to be enough free cargo capacity on the **ship**.

If these conditions are satisfied, the action can be executed:

- **cargo** will be loaded on the **ship**,
- available cargo capacity on the **ship** will be decreased by the **cargo** weight.

Now we will review the modeling decisions taken when using PDDL and NDDL and we will discuss the differences.

PDDL: The code for the corresponding action can be found in Section 3. Once we declare the predicates and the fluents that are sufficient to encode all relevant domain knowledge we can use the PDDL syntax to describe action preconditions and effects in a way that is very close to the description at the beginning of this section.

NDDL: By using NDDL we need to take a different point of view. We suggest one possible sequence of modeling decisions:

1. Choose relevant timelines:
   - `cargoState(cargo)` - a timeline for the state of the loaded cargo,
   - `shipState(ship)` - a timeline for the state of the target ship,

   - `cargoCap(ship)` - a timeline for free cargo capacity of the target ship,
   - `ship` - a timeline for the target ship.
2. Allocate the token representing the action: `LoadCargo(dock,cargo)`.
3. Allocate tokens representing conditions and effects using temporal constraints. These tokens represent either conditions or effects as indicated in Figure 1.

In both cases we first need to declare some domain-specific notions (e.g. predicates and fluents in PDDL and timelines with predicates in NDDL). The difference between the two approaches is in the way that these notions are used.

It is possible to model actions in the PDDL without encoding any explicit knowledge about their possible ordering (e.g. dock and undock situation described earlier). The missing information can be difficult to obtain but there is a clear notion of action stated as a set of conditions and effects.

On the other side the NDDL model can provide the planner with some additional information but the notion of action, which is specified as a set of temporal constraints on tokens, can blur the semantics of the domain model.

6 New Knowledge Modeling Interface

We assume that a domain modeler can treat the planner as a black box. This is in accord with the *physics-only* principle employed in the PDDL. In contrast with this principle we want the modeler to give as much domain-independent information as possible.

Classes: In our formalism we distinguish between two types of classes:

1. *enumerative* classes can be used to represent discrete objects such as ships,
2. *numeric* classes allow description of quantities such as fuel tank capacity.

In addition to the class hierarchy from the Section 2, we need to specify at least one numeric class to be used for numeric values in the domain.

State variables: Domain properties that are subject to change are described as state variables of the following form:

$$s(a_1, \ldots, a_n) : r$$

where s is a name of an n-ary *state variable*, a_i represents classes of its parameters, and r is defined as a set of classes, which implicitly defines the range of values for the state variable (e.g. `cargoLocation(Cargo):{Ship,Location}` - an item of cargo can be either stored at some location or loaded on a ship).

The state of the world is described as a vector of values for all state variables defined in the planning problem instance.

Domain rules: Decisions made during the planning process often require some kind of computation (e.g. the fuel consumption depends on the trip distance). We suggest to describe each such computation as an n-ary function:

$$f : D_1 \times \ldots \times D_n \rightarrow D_{n+1}$$

called a *domain rule*, where D_i is a set of constant symbols compatible with some class C_i. Classes of both *numeric* and *enumerative* types are allowed.

Operators: Possible changes of the world state are described with operators. An operator is specified as a list of expressions. There are two types of expressions (we will refer to the example operator code below):

- *conditional expressions* are used to describe conditions among different state variable values and their parameters (e.g. line 4). The value of state variable `cargoWeight(C)` is constrained by `X`. The variable `X` is defined at line 5 where it represents the original value of the state variable `cargoCap(S)`.
- *transitional expressions* define both the condition and the change (e.g. line 6). The cargo item `C` is transfered from `D` to `S`.

```
1 loadCargo(D - LogisticLoc, C - Cargo, S - Ship)
2    shipLoc(S) = D
3    shipState(S) = docked
4    cargoWeight(C) <= X
5    cargoCap(S): X --> (X - cargoWeight(C))
6    cargoLoc(C): D --> S
```

Terms: The operator expressions use terms that can be constructed recursively from **constants** (e.g. `docked`), **variables** (e.g. `D,C,S,X`), **state variables** (e.g. `shipState(S)`), and **domain rule instances** (e.g. `(X - cargoWeight(C))`[2]).

Atomic conditions: For two terms within an *enumerative* class we use comparison relations $=$ and $\neq$ to build an atomic condition. In case of two terms within a *numeric* class we use the combinations of $=, <, >$. Standard operators of FOL $(\wedge, \vee, \neg, \forall, \exists)$ are used to construct more complex *conditional expressions.*

In the code above there are three *conditional expressions.* Two of them are asserting equality of *enumerative terms* (lines 2, 3) and one is asserting inequality of two *numeric terms* (line 4). There are two *transitional expressions* (lines 5, 6).

The action ordering can be enforced by introducing a *domain rule* `fsaCheck`, which references a finite state automaton (FSA) that describes all valid action sequences [2]. The FSA will have the following states: `undocked`, `navigating`, `waiting`, `loading`, `unloading`, and `refueling`. We replace the line 3 with:

```
3a    exists B: fsaCheck(A,B) = true
3b    shipState(S): A --> B
```

The conditional expression at line 3a constrains the variables A and B according to the FSA and the transitional expression at line 3b changes the state of the ship S from A to B. The value of the variable A is defined at line 3b.

[2] Arithmetic operators are binary functions.

We have showcased usage of *state variables*, *domain rules*, and *operators*. The resulting domain model allows user-defined extensions called domain rules to restrict action ordering. Arbitrary arithmetic functions can be defined. While NDDL permits a user to define functions as well, PDDL is less flexible which can be limiting for some real world applications [4].

7 Conclusion

The interface presented in this paper gives a different view on knowledge modeling which depends on multi-valued state variables. The value ranges of these variables are defined either by enumerative or numeric classes instead of the predicates as in the NDDL. The resulting domain model uses operators simmilar to actions from the PDDL. The closest formalism currently in existence is Action Notation Modeling Language (ANML) [5].

From the perspective of a domain modeler the interface provides an easy way to integrate various kinds of domain-specific knowledge using the domain rules.

On the planner side a problem instance is described by a finite set of state variables and domain rules, while the domain specific information is represented only in the operators.

References

1. Barreiro, J., Boyce, M., Do, M., Frank, J., Iatauro, M., Kichkaylo, T., Morris, P., Ong, J., Remolina, E., Smith, T., Smith, D.: EUROPA: A Platform for AI Planning, Scheduling, Constraint Programming, and Optimization. In: Proc. of ICKEPS (2012)
2. Barták, R., Zhou, N.-F.: On Modeling Planning Problems: Experience from the Petrobras Challenge. In: Castro, F., Gelbukh, A., González, M. (eds.) MICAI 2013, Part II. LNCS, vol. 8266, pp. 466–477. Springer, Heidelberg (2013)
3. Gerevini, A., Long, D.: BNF Description of PDDL 3.0, `http://www.cs.yale.edu/homes/dvm/papers/pddl-bnf.pdf`
4. Parkinson, S., Longstaff, A.P.: Increasing The numeric Expressiveness of the Planning Domain Definition Language. In: Workshop of the UK PlanSIG (2012)
5. Smith, D., Frank, J., Cushing, W.: The ANML Language. In: International Conference on Automated Planning and Schedulling - Poster Session (2008)
6. Vaquero, T.S., Costa, G., Tonidandel, F., Igreja, H., Silva, J.R., Beck, J.C.: Planning and Scheduling Ship Operations on Petroleum Ports and Platforms. In: Proceedings of the Schedulling and Planning Aplications Workshop (2012)