Knowledge Modeling for Computer Games: Comparative Study

J. Vodrážka
Charles University in Prague, Faculty of Mathematics and Physics, Prague, Czech Republic.

Abstract. This paper presents a comparsion of knowledge modeling languages for automated planning. It is focused on three features, namely external events, explicit time and concurrent actions, which are commonly found in planning domains from computer gaming. These features are described with three modeling languages PDDL, NDDL and ANML. Resulting descriptions are compared and their strong points are summarized in the end.

Introduction

One of the areas suitable for application of automated planning is the area of computer game development. In order to use some planner in a computer game we first have to describe a planning domain — that is the abstract description of the game world. This abstract description should be accurate enough in order to capture all the important aspects of the game dynamics.

From the point of knowledge engineering this is a challenging task. If we do not want to limit ourselves to logic puzzles or desk games such as chess, where the description is usually simple, we will need to describe planning domains that are often very similar to real-world domains and require some complex features.

First we are going to describe three selected features by illustrating them with a simple example. Then we will look at three different domain modeling languages and we will see if they can express these features. In the end we will compare the three languages and highlight their strong points.

Example domain

Let us consider simple planning domain that is inspired in computer games. The domain features an agent which moves around in a maze. The maze is dynamic because it features moving platforms. The goal of the agent is to move to certain location in the maze. In order to reach it the agent has to use the platforms. The movement of the platforms is continuous and independent of the agent. Each platform moves back and forth between two extreme positions and changes its direction automatically. Particular problem specification determines extreme positions for all platforms as well as their speed and starting positions. The starting position and goal position of the agent and the map of the maze itself is also part of the problem specification.

In the planning domain previously described we will focus on the following features:

External events: In order to create appropriate model for the domain we need a way to describe movement of the platform. The feature we need here is called an external event.

Figure 1. Simple maze with one moving platform.
Basically this is a special kind of action that can not be planned. For example one particular event the arrival of the platform to one of its extreme positions. In this particular case we know *certainly* that the event will occur.

The general case when there is only a *possibility* of event occurrence will be not discussed in this paper.

**Explicit time:** Let us suppose that the movement of the platform and the agent is continuous and for the purpose of the game it is important to minimize the amount of time when the agent is idle (e.g. waiting for the platform to arrive). In such case we need to reason about *explicit time* in our model.

**Concurrent actions:** Let us have two actions $A$ and $B$ that may happen in the same time (e.g. agent getting on a platform and movement of this platform). We call such actions *concurrent*. Any domain model should be able to describe the actions in such manner that gives answers to following questions:

- Is it possible to execute actions $A$ and $B$ together?
- What is the result of the execution?

**Existing approaches**

It is possible to recognize three main approaches among knowledge modeling techniques used today. The following paragraphs provide simplified descriptions that aim at explaining how the state of the modeled system and transition between two states is described. In order to illustrate the main differences we will refer to the example domain.

**Propositions:** The first approach is based on *propositional logic*. Current state of game world is described as a set of propositions that holds — such as $at(Agent1, Loc5)$. Valid state transitions are described by actions. For example action $move(Agent1, Loc5, Loc3)$ describes valid transition from any state where $at(Agent1, Location5)$ holds. If we apply the action, the predicate $at(Agent1, Loc5)$ is replaced by $at(Agent1, Loc3)$ in the set that represents the state of the system.

**Timelines:** The second approach works with a finite set of *timelines*. One timeline describes history for one dynamic element (e.g. $Agent1$). In the domain model we specify possible predicates for each such element. For example we can allow the agent to move between distinct locations ($move(X, Y)$) and to get on/off the platform ($getOn(X, P)/getOff(X, P)$). Complete timeline for the agent is later composed as a sequence of adjacent intervals that span over some fixed time horizon. Each interval is associated with one predicate so that we can describe the state of the system in a given timepoint by enumerating active predicates on all timelines. A set of constraints is used to restrict possible sequences for each timeline and to define compatibility conditions for different timelines.

**State-variables:** The third approach is using *state-variables* to describe state of the game world. There is a finite number of such variables and each variable has its domain of values specified. One such variable in our example domain can be defined as $Agent1.pos \in L$; where $L$ is a set of all possible locations. Vector of values over all state variables in a given timepoint represents a state. Transitions between states are described through actions that can modify values of some state variables. For example action $move(Agent1, Loc5, Loc3)$, which is applicable in a state where $Agent1.pos = Loc5$, changes the value of $Agent1.pos$ to $Loc3$.

In following text we will see if the three features described earlier can be captured in following modeling languages:

- **PDDL+** [*Fox and Long* [2002]] — an extension of language PDDL that was originally based on the first approach

---

1. Note that these predicates actually matches actions.
(:action getOn
 :parameters (?a − agent ?p − platform ?l − location)
 :precondition (and (at ?a ?l) (< (dist ?l ?p) RANGE))
 :effect (and (not (at ?a ?l)) (at ?a ?p))
)

Figure 2. PDDL action.

• NDDL [Bedrax-Weiss et al. [2005]] — modeling language which builds on the second approach
• ANML [Smith et al. [2008]] — modeling language that is based on the third approach

Short description of each language will be followed by description of language constructs that are relevant for modeling of selected features from example planning domain.

PDDL+
The language PDDL+ is one of the extensions of well established domain modeling language PDDL that was originally created in order to enable international planning competitions. Although there are some planners in existence that supports majority of PDDL features, there is no planner yet that would support features introduced in PDDL+.

The main contribution of PDDL+ is in the possibility to model occurrences of environment dependent events such as the repeated arrivals of the platform in our example.

Central construct for description of change in PDDL is referred to as action. Example code for action getOn is displayed in Figure 2.

It is important to note that action described in PDDL code actually describes only a template with variables. Particular instance of given action can be obtained by assigning values to all parameters.

External events:

In order to describe automatic movement of the platform we can use following language constructs:

• event — describes an instantaneous change which is not under control of the executive and is relevant only for logical state of the system.
• process — description of continuous change of numeric fluents. Processes are triggered when certain condition become true and terminated when it become false. Logical state of system is not affected by processes.

It is possible to accomplish the task by describing a single event–process pair. The idea is to keep platform going in one direction by process movePlatform(Pla,Loc1,Loc2) triggered by predicate direction(Pla,Loc1,Loc2) and the fact that the distance\(^2\) of the platform from Loc2 is bigger than zero: (> (dist Pla Loc2) 0).

In the moment when this distance becomes zero the event changeDir is triggered and predicate direction(Pla,Loc1,Loc2) is replaced by direction(Pla,Loc2,Loc1) which stops process movePlatform(Pla,Loc1,Loc2) and starts its reverse. The syntaxe of events and processes is very similar to that of actions.

Explicit time: In PDDL+ there is a possibility to define durative actions. In durative action one can specify the amount of time needed to complete the action. It is also possible to specify conditions with limited temporal extent. This means that preconditions and effects may be temporally bound either to the start or end of the action or to the time interval between these two timepoints. During the execution of a durative action or a process one can access special variable \#t that represents amount of time since the start of the action execution.

\(^2\)Represented as a numeric fluent.
Concurrent actions: In order to check whether two given action instances can be executed in parallel the language rely on so called no-moving-targets rule which states that no two action instances can make use of a logical value if one of the two is accessing the value to update it. Such actions are considered as mutually exclusive.

NDDL

The language NDDL is used in EUROPA [Barreiro et al. [2012]] which is a framework for modeling and solving problems in planning, scheduling and constraint programming. This framework is built on CAIP\(^3\) paradigm [Frank and Jonsson [2003]]. Domain description in NDDL contains:

- **classes** that describes attributes of the world (e.g. `agent` which describes the position of the agent). The attributes represents dynamic properties of the domain. History of each attribute can be captured within a timeline.
- **predicates** that define possible states of the attributes (e.g. `Going(X,Y)` — the agent is on its way from location $X$ to $Y$). A notion of interval is introduced in order to give temporal extent to predicates. For example we can use interval `(agent,0,10,Going(Loc1,Loc2))` to express that `agent` is going from `Loc1` to `Loc2` between time 0 and 10.
- **configurations** that can restrict possible evolutions for one attribute (e.g. it is possible for the attribute `agent` to have value `Going(X,Y)` only immediately after `At(X)` or `Going(Z,X))` or enforce synchronization of different attributes. For example we may require following synchronization between the attributes `agent` and `platform`: `agent = GetOn(P,Y)` is possible iff `platform = ChangeDir(Y)`. According to the CAIP paradigm, the configurations are used to describe the constraints.

Figure 3 shows parts of timelines for two attributes (agent and platform). Constraints for the predicate `GetOn(P,Y)` are denoted with arrows.

External events: It is possible to describe external events in NDDL. One can include predefined timelines for all platforms as a part of a goal description. In this way the external events become part of particular planning problem instance. Goal description in NDDL is always specified as a set of timelines over a fixed horizon.

Explicit time: In NDDL the time is modeled with intervals. The language allows to specify relations between intervals before their particular start and end times are specified. The simplified version of interval algebra described in [Allen and Koomen [1983]] is used.

We can see the examples of three interval relations (`meets`, `met_by` and `contains`), used to specify temporal constraints for predicate `GetOn`, in Figure 3.

Concurrent actions: Since there are no actions in NDDL there is no need to handle concurrency at all — different timelines are used to represent concurrent activities as we can see in Figure 3. The domain rules expressed as configurations forbids inconsistent behavior by design.

\(^3\)Constraint-based Attribute and Interval Planning
action getOn(Agent a, Platform p) {
    duration := JUMP_TIME;
    [start] {
        p.nearestDist < 0.3; /* platform is close enough to the location */
        p.nearest == a.pos; /* the agent is at the same location */
    };
    [all] {
        a.pos == a.pos := p;
    };
}

Figure 4. ANML action.

ANML

The language is being developed as an alternative to existing modeling languages. Although there is no planner available yet that would support ANML directly, modeling capabilities of the language were tested on complex domains [Boddy and Bonasso [2010]].

Variable-value model used in ANML makes it possible to view one possible evolution of the world state as a chronicle. In a chronicle all actions are instantiated with fixed start and end times. To this point we can understand ANML domain model as a set of restrictions that defines a set of valid chronicles.

There are five basic constructs that can be used in ANML code:

1. **Declarations** of constants, variables, objects, functions and predicates

2. **Propositions** which are used to define various constraints

3. **Action descriptions** which specify problem independent rules that are specific in given planning domain

4. **Assertions** which are used to describe initial state for particular problem instance

5. **Goals** which are used to describe desired state and domain invariants for particular problem instance

First three of them are used to describe the domain itself (i.e. only the domain rules and classes without specification of any particular instances) while assertions and goals are used to describe particular planning problem instance (e.g. particular maze with initial positions of particular platforms and agents).

**External events:** There is no explicit language construct that would enable direct description of external events. The task can be accomplished by including actions that describe the movement of the platform in the goal description. For example list of assertions such as: goal [0,20] contains PlatformCycle(P1); would be part of our problem specification. The code [0,20] defines an interval and PlatformCycle(P1) is a composite action that describes the movement of the platform. Apparently this is possible only in a fixed time horizon.

**Explicit time:** In ANML it is possible to describe durative actions with temporally qualified expressions. ANML code for action getOn is displayed in Figure 4.

For convenience ANML defines keywords for important timepoints like start and end or intervals like all. There are two temporally qualified conditions which has to be fulfilled at start and one combined condition and variable assignment that goes on during interval all. Duration of the action can be defined using constant JUMP_TIME.

Temporal qualification is not restricted to predefined timepoints. It is possible for to describe delayed effects or conditions by using reference to arbitrary timepoints such as [start+3]. All temporal variables in ANML are positive integers but the resolution is not fixed to any particular time unit.
Concurrent actions: In most cases the semantics of ANML makes it possible to determine the result of execution of two parallel actions. The issue of concurrency is important only if two actions assign value to one variable at the same time. Let us consider a situation when there are two different platforms available for the agent to get on. In ANML this means that two instances of action getOn are available — each with different platform (for example P1 and P2). In theory there is no way to tell whether the agent gets on the platform P1 or P2. It is however possible to prevent such situations by designing particular problem instance in a way that does not allow more than one platform arrive at the same location.

Final comparison and discussion

Three knowledge modeling languages were discussed in this study. Each of them proved to be expressive enough to model given planning domain. Table 1 provides a quick comparison of discussed languages with respect to target modeling features.

Table 1. Language comparison.

<table>
<thead>
<tr>
<th>Language</th>
<th>external events</th>
<th>explicit time</th>
<th>concurrent actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDDL+</td>
<td>yes</td>
<td>yes (timepoints)</td>
<td>no-moving-target rule</td>
</tr>
<tr>
<td>NDDL</td>
<td>through goal spec.</td>
<td>yes (intervals)</td>
<td>compatibilities</td>
</tr>
<tr>
<td>ANML</td>
<td>through goal spec.</td>
<td>yes (timepoints)</td>
<td>minor issues</td>
</tr>
</tbody>
</table>

Strongest points of each languages can be summarized as follows:

- PDDL+: modeling of external events
- NDDL: only language with planner available
- ANML: flexible modeling of time

It takes a great deal of effort to design a planning domain. From the point of knowledge engineering the languages that uses the notion of action (PDDL+, ANML) seems easier to work with.

Although there are some tools designed to help user through the modeling process it would be beneficial to have a tool that would be easier to use for non-experts in planning. This might be possible if we build such tool on top of a lightweight modeling language that would enable modeling of the three features examined in this paper. We aim to create such language by extending the formalism described in [Vodrážka and Barták [2012]].

References


