Reading PDDL, Writing an Object-Oriented Model

Flavio Tonidandel[1], Tiago Stegun Vaquero[2], and José Reinaldo Silva[2]

[1] Centro Universitário da FEI
IAAA Lab – EE - São Bernardo do Campo, Brazil
[2] Escola Politécnica – Universidade de São Paulo
Design Lab. – PMR – São Paulo, Brazil
flaviot@fei.edu.br, tiago.vaquero@poli.usp.br,
reinaldo@usp.br

Abstract. There are many efforts towards a combination of planning systems and real world applications. Although the PDDL is in constant evolution, which improves its capability to describe real domains, it is still a declarative language that is not so simple to be used by the non-planning community. This paper describes a translation process that reads a domain specification in PDDL and transforms it into an object-oriented model, more specifically into a version of UML for planning approaches. This translation process can let a designer read PDDL domains and verify it with some powerful tool like itSIMPLE or GIPO, or it can allow a planning system that only reads object-oriented models to run in domains described in PDDL originally.

1 Introduction

The PDDL (Planning Domain Description Language) [3] is a standard language to describe planning domains and problems since 1998 and it becomes a reference language for planning and non-planning community committed to present new models for real planning domains. However, as highlighted in some works [1], the PDDL is not intuitive enough to be an easy-to-use language. On the contrary, it has become more and more complex whereas new versions are released.

Concerning on the complexity and the lack of a tool that can support knowledge engineering processes, the first international event that focus in this subject was the ICKEPS 2005 (International Competition on Knowledge Engineering for Planning and Scheduling) where new proposals for tools and modeling languages emerged as well as the discussion about solving real planning problems.

On ICKEPS 2005, two works on object-oriented languages appeared: itSIMPLE [7] and GIPO [5]. They showed that they are promising and valuable tools for modeling, verification, validation and analysis of planning domains.

In fact, the object-oriented model is more intuitive for planning designers than PDDL. Since the planning community intends to build a bridge over the gap between real applications and planning simulation domains, it is suitable to have some flexibility in the modeling language as well as an easy-to-use processing model.

Many domains have been described in PDDL and many planners can read only domain and problem specifications in PDDL. The GIPO tool, as well as itSIMPLE,

J.S. Sichman et al. (Eds.): IBERAMIA-SBIA 2006, LNAI 4140, pp. 532 – 541, 2006.

can export their models to PDDL. However, there is only one work that translates PDDL into an object-oriented model [6] that is very specific for OCLh language.

This paper aims to define a translation process that can read a PDDL model and write an object-oriented model in a general way, more specifically into UML.P (UML in a Planning Approach). This paper is the first attempt to perform this translation and, therefore, it just considers the classical domain described in STRIPS-like PDDL.

The translation process was implemented and tested in the itSIMPLE tool, and this process can be adapted in the future to generate object-oriented models for any tool.

2 Overview of UML for Planning

The UML (Unified Modeling Language) [2] is one of the most used languages to model a great variety of applications and we believe that most engineers in several application areas are somehow familiar with the UML language [4]. The purpose of UML is to be a general modeling language, thereby we will consider the UML in the Planning Approach, which will conveniently be called UML.P throughout this paper.

2.1 Class Diagram

The class diagram is a representation of the planning domain structure showing the existing entities, their relationships, their features, methods (actions) and constraints.

In order to simplify the planning modeling process, *UML.P* proposes a General Class Structure composed by *Agent* class and a domain *Environment* class. The *Agent* changes the arrangement of objects, which compose the environment.

In the class diagram, there are relationships between classes where any relationship or dependency between two or more classes is an association. There are three kinds of relations: Association, Aggregation and Composition.

Associations just connect classes identifying some appropriate meaning, for instance, association “at” between the class *Vehicle* and *Location* in the Logistic domain. Each association between entities has its respective multiplicities. For example, one or many trucks (1…*) can be at one and only one place at a time.

Aggregation and Composition are a special kind of association which can be read as a “have”, “belong” or “made of”. For this paper, we will consider only Aggregations and Associations.

In *UML.P*, every class that can perform actions in the domain will be a specialization (subclass) of the class Agent, for convenience. These classes with dynamic behavior can have methods (actions) defined in class specifications. All the others classes will be associated to the Environment with an aggregation relation. The StateChart Diagram models the dynamic behavior of action.

2.2 StateChart Diagram

The StateChart Diagrams are very useful to represent entities that perform dynamic behavior. All methods (actions) in the class diagram are specified in these diagrams.

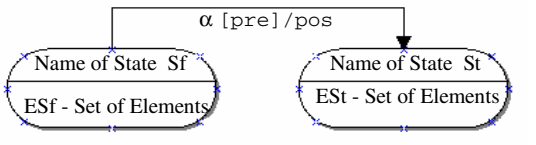


Fig. 1. The General Structure of the StateChart Diagram. The `pre` and `pos` are sets of elements characterizing preconditions and post-conditions of the action α. On the right there is an example of a Package StateChart Diagram of the Logistic Domain.

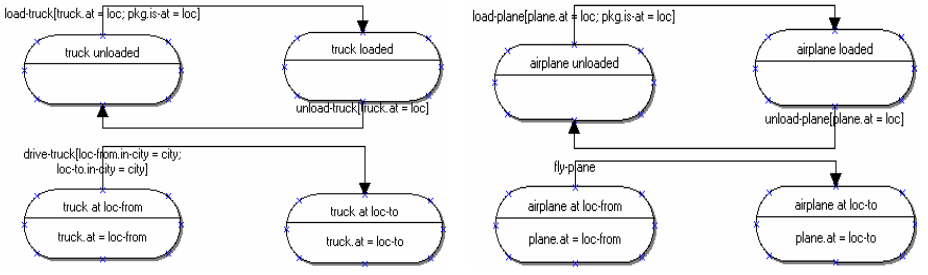


Fig. 2. Truck and Airplane StateChart Diagrams modeled in *UML.P*

Any class in Class Diagram can have its own State Diagram. A StateChart Diagram is unique for a class and it does not intend to specify all changes caused by an action, but only specify dynamic behaviors for an object of the class caused by one or more actions. This dynamic behavior is expressed by actions that affect the objects of the class. An action (method), therefore, can appear in many StateChart Diagrams.

For the planning context, the *UML.P* has special remarks during StateChart diagram construction:

- There is one diagram for each class capable of dynamic behavior. For instance, the truck class in the Logistic domain is a class with dynamic behavior;
- The state elements in the diagram represent the possible states of an object. The state is defined by the main values of its attributes.
- The state elements are not necessary mutually exclusive;

Summarizing, the states in StateChart Diagram describe the possible states of the class. The actions are transitions among these states. The general structure of a StateChart Diagram is shown by figure 1.

In a StateChart Diagram of a specific class, we have some considerations. First, the states lie about an object of the class that directs performs the action. The action has its own pre and post definition (as in figure 1) described explicitly in the diagram. These pre and post definitions can lie about objects of the StateChart Diagram´s class or another class in Class Diagram that can receive or suffer an indirect effect of this specific action (transition) but does not have a StateChart Diagram.

The structure in figure 1 has two states (named Sf and St) that are considered the State-form and State-to of the action α. Esf are Elements of the state Sf , similar for ESt. It is important to realize that when the action α is performed, the ESf elements

become false and the elements of ESt become true. For convenience, the elements of ESf and Est lie about the objects of the StateChart Diagram´s class.

```
Function translation_process(PDDL: PDDL file)
  Extract_types(PDDL);
  Create_specializations from types ;
  Create_associations(PDDL);
  For each action α in F:
       ABS_precond(α) and ABS_effect(α);
  Endfor
  Gather_States ;
  For each class in Class Diagram   Create_StateChartDiagram;
  Endfor.
```

Fig. 3. General Algorithm for the translation process

Considering and example, since the classical Logistic Domain has only two entities with dynamic behavior (*Truck* and *Airplane*) and one entity that can receive effects of actions (*Package*), it is necessary three StateChart Diagram. Figure 1 focused on the Package, Figure 2 focused on the class Truck and the class Airplane.

Observe that in the Plane StateChart Diagram, for instance, the States lie about the plane class and the pre and post definitions lie about an indirect effect for the plane (`plane.at=loc`) and another indirect effect of the transition between loaded and unloaded that is `pkg.at=loc`.

3 PDDL Translation into Object-Oriented Model

The translation of PDDL into an object-oriented model is a process that must make some semantic considerations in PDDL description and that will provide an elegant way to describe PDDL domains as object classes and transition of states.

The following section will describe the translation process used by the itSIMPLE tool. It translates each peace of a PDDL domain into each Diagram of the *UML.P*.

The figure 3 shows the entire process of the translation to *UML.P* from PDDL in a higher level algorithm language. Definitions for each function described in the figure 3 will be in the following sections.

3.1 Reading PDDL and Creating Class Diagram

The first step is to create a Class Diagram in *UML.P* by identifying classes in the PDDL specification. The PDDL has a structure of types that appears in the domains with a tag `:typing` in the requirements. In fact, there is a semantic correspondence between types in PDDL and classes in *UML.P*.

Therefore, the translation process, through the function **Extract_types(PDDL)** just reads the types in a PDDL file (F) and creates the classes in *UML.P*.

However, in order to define which class (environment or agent) will be a super-class of a given one, some analyses are necessary under the following definition:

Definition 1 (action structure). In PDDL, an action α has the following structure:

```
(:action α
  :parameters (?v1 - tp1 ... ?vn - tpn)
  :precondition ρ
  :effect (and  δ τ ))
```

Where p_i = $?v_i$ - tp_i is the i^{th} parameter of the action; δ is the delete list (negative effects) and τ is the add list (positive effects) of the action.

Aiming the translation to object-oriented model, some consideration must be made because PDDL action definition has no information about the most important object of the action. In fact, there is no imposed rule for definition of action parameter order in PDDL. We will call the order of the parameters in the :parameter tag in PDDL as a parameter list (PL). Formally:

Definition 2 (Parameters List). The Parameters List, denominated PL, of an action α is a list that takes into account the given order (from left to right) of the parameters in the :parameter tag in PDDL.

We will consider, therefore, the order of the PL as an indicative of the importance of each type (class) in the action definition. For instance the PL of following parameters of action Load-Truck (:parameters (?truck – truck ?pkg - package ?loc - place)) would be PL = {truck: truck, pkg: package, loc: place}.

The most important class of the action Load-Truck above is the truck, that is the first class of PL. With the consideration above, we can define the class where each action will be specified in the Class Diagram as methods.

The Class Diagram has a main structure that incorporates two default classes: *Agent* and *Environment*. Any action defined in PDDL will be allocated as a method of the class of its first parameter in PL and this class will be a subclass of *Agent*.

Besides Classes we have associations and aggregations in the Class Diagram. The aggregations are extracted directly from type definitions in PDDL by the function by also creating specializations through the function called **Create_specializations.** For example, truck is a vehicle in PDDL (described as truck – vehicle) and in the Class Diagram there will be a specialization relation between class Truck and class Vehicle.

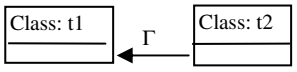
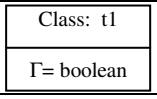
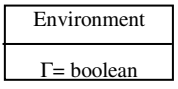
The associations are relations between classes with a name and multiplicity. These associations correspond semantically to a predicate of two arguments in the PDDL description. For example, *in(package, airplane)* means an association, called *in*, from package class to airplane class. The function **Create_associations(PDDL**) creates the respective associations under the following definitions:

Definition 3 (predicate structure). A predicate Γ in PDDL has the following structure: (Γ [?v1 – tp1] [?v2 – tp1]) where [ ] indicates optional argument, vi indicates variable i and tpi indicates the type of the variable i.

The predicates describe relations between classes. The main class of the predicate is the first argument of the predicate. For example, in the predicate (at ?truck1-truck ?city1-city) means that the predicate main argument is the truck1.

Concerning the number of arguments (arity) of the predicates, we can consider P_0 as a Predicate without argument (arity = 0), P_1 as a predicate with one argument (arity = 1) and P_2 as a predicate with two arguments (arity = 2).

Table 1. The correspondence between PDDL and UML for predicates

Predicate in PDDL	UML description	Graphical UML
(Γ ?v1 – tp1 ?v2 –tp2) Arity/2	Γ.v1 = v2	
(Γ ?v1 – tp1) Arity/1	Γ.v1 = true or Γ.v1 = false	
Γ Arity/0	Γ = true or Γ = false	

We do not consider predicates with more than 2 arguments although it can be expressed in UML. Table 1 defines the translation function between predicates and associations. Observe that predicates with less than two arguments are described as parameters of a class or the default class environment.

In the translation process, predicates P_1 and P_0 in PDDL are expressed in boolean type in UML.P. Therefore, we can define the negation of a predicate of P_1 or P_0 as:

Definition 4 (Negation of P). For a predicate $P \in P_0 \cup P_1$, $\overline{P}$ is the negation of P by negating the Boolean value of the predicate description. For a predicate $P \in P_2$, in the form p1.v1=v2, $\overline{P}$ will be defined as p1.v1=NULL.

Using the definition above, we can show the following example: if P is "a.p=true" then $\overline{P}$ would be "a.p=false".

In UML.P, there are multiplicity definitions on associations. However, since the PDDL description has no information of quantities between arguments of a predicate, the translation process just considers the multiplicity 0…* for all associations.

3.2 Reading PDDL and Creating StateChart Diagram

In order to understand the semantic relation between actions definitions in PDDL and StateChart Diagrams in *UML.P*, we must realize that the action definition in PDDL is a composition of States (Sf and St) and `pre` and `post` elements for such action in all StateChart Diagrams. In one hand, suppose an action α that goes from Sf to a St in one StateChart Diagram. Predicates that describe Sf are related to precondition and to a delete list of α in PDDL. All predicates of the St will compose the Add list (positive effects) in the effects list in PDDL.

On the other hand, the information in `pre` and `post` that comes together with actions name in StateChart Diagram are posted in the precondition and effects of the action definition in PDDL.

In order to perform the translation of the dynamic behavior of actions, we consider the delete list as a subset of the precondition list. The key to translate PDDL actions into transitions of states is the parameter list (PL) of each action.

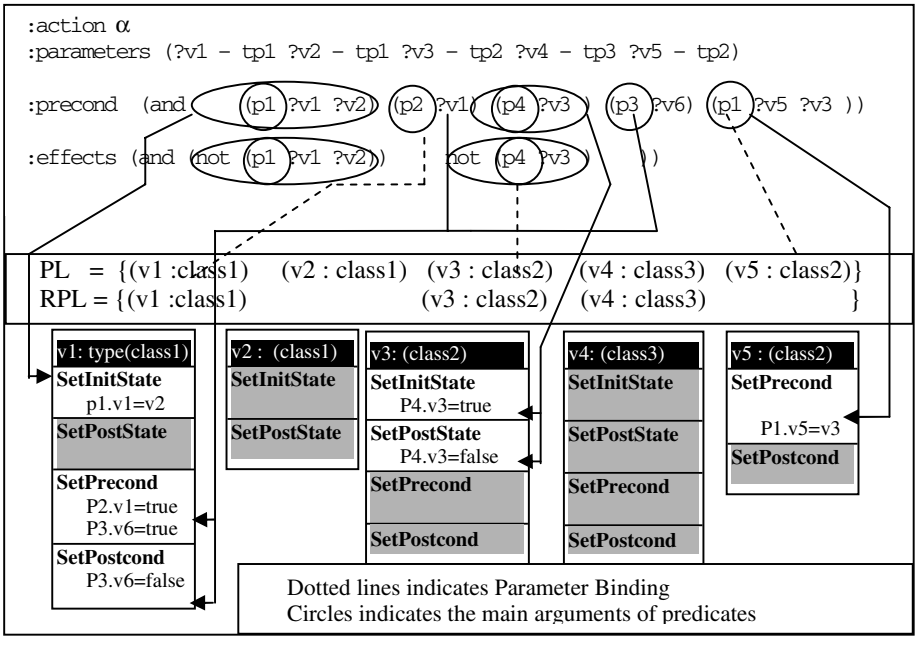


Fig. 4. The general schema of the action analysis and the split of predicates in their correspondent class and set

Considering the predicates in an action, each of them lies to one class or an association. In fact, they are grouped by their correspondent classes.

We must consider that only the first occurrence of each type in the action's parameter list will be considered relevant for action purposes. In order to define relevant classes (types in PDDL) of an action, we will consider the types of the action parameters in PDDL. Since PDDL does not express the relevant types explicitly, we will consider the order of the parameters in an action extracted by the PL list (def. 2).

Then, we will consider the first occurrence of each type in a list of parameters of the action as the relevant reference of that type in such action. In fact, we will create a new list called Relevant Parameters List (RPL). By definition, we have:

Definition 5 (Relevant Parameters List). Let be an ordering of parameters in an action α called PL. The Relevant Parameters List, denominated RPL, is a subset of PL, RPL $\subseteq$ PL, and their elements are the first occurrence of any type in the PL ordered list.

The analysis of actions intends to classify each action in accordance with the parameters in the parameter list (PL). In order to precisely define which class can have a StateChart Diagram, we will define the relevant classes of the parameter list of the action as the classes that can have a StateChart Diagram.

Performing the action analysis, we will consider that each parameter in PL will have two sets of predicates: SetPrecond and SetPostcond. The parameters in the RPL will have two more sets: SetInitState and SetPostState.

```
Function ABS_precond (a: action)
For each P in :precond of a
   i ← Parameter Binding of P;
   if P is a Relevant Predicate and P ∈ δ
        insert P in SetInitState(i)
        If (P ∈ P0∪P1 ) insert P̄ in SetPostState(i)
   endif;
   if P is a Relevant Predicate and P ∉ δ
        insert P in SetPrecond(i)
   endif;
   if P is not a Relevant Predicate
        insert P in SetPrecond(i)
        insert P̄ in SetPostcond(i)
   endif;
```

Fig. 5. Algorithm for the ABS structure

The SetInitState is related to Sf in figure 1. In fact, SetInitState of an action is a subset of the Sf elements. Similarly, SetPostState is related to St and SetPrecond and SetPostcond are related to pre and post definitions. This structure of Sets for each parameter in PL is called ABS(Action Behavior Structure).

Therefore, the process of action analysis extracts the predicates that can fit into each set of each parameter. Each predicate will be inserted in a specific set as shown by an example in Figure 4. In order to explain the process in Figure 4, we must define Relevant Predicate and Parameter Binding of a predicate in an action:

Definition 6 (Parameter Binding for predicates with arguments). Consider a predicate $\Gamma \subseteq P_1 \cup P_2$ and its first argument v_1 into the action α definition. The Parameter Binding will be a parameter that is the same variable or value of the first argument v_1.

Definition 7 (Parameter Binding for Predicates with no arguments). Consider a predicate $\Gamma \subseteq P_0$ into the action α definition. The Parameter Binding will be the first parameter of the PL list of the action α.

Definition 8 (Relevant Predicate). A predicate Γ is considered relevant iff its first argument v_1 is in the RPL (Relevant Parameter List).

The most important feature is to identify the parameter in the PL list of the action that each predicate is related to. It is done by considering the first argument of the predicate. This analysis that creates a relation between predicates and parameters of the action is necessary to identify the correct states in the StateChart Diagram.

From now on, we will consider the following notation to denote sets of each parameter of the action:

- SetPrecond(i) and SetPostCond (i) – The Precondition and Postcondition Sets of the parameter i of the PL list.
- SetInitState(i) and SetPostState(i) – The Initial State and Post-State Sets of the parameter i of the PL list. This parameter is a relevant parameter.

```
Function ABS_effect (a: action)
For each P as positive effect in :effects
   i ← Parameter Binding of P;
   if P is a Relevant Predicate
        insert P in SetPostState(i)
        If (P ∈ P0∪P1) insert P̄ in SetInitState(i)
   endif;
   if P is not a Relevant Predicate
      insert P in SetPostcond(i)
      If (P ∈ P0∪P1) insert P̄ in SetPrecond(i)
   endif;
endfor;
```

Fig. 6. Algorithm for the ABS structure

The Figure 4 shows a general schema for the application of the algorithm in figure 5. The algorithm in Figure 5 and 6 shows the process to create the ABS structure.

ABS_precond(α): The algorithm considers the predicates in the precondition list of an action α in PDDL. Each predicate P and its negation $\overline{P}$ can be inserted into the SetInitState or in SetPrecond of the structure of a parameter in PL.

The first operation is to find the parameter 'i' in PL that correspond to each predicate P in precondition. In one hand, if P is relevant and it is in the delete list, it will be added in the SetInitState of the parameter i and $\overline{P}$ will be added to SetPostState if is not P_2.

On the other hand, if P is relevant but it is not in the delete list, P can not be in a State definition, so it will be added in the Setprecond list of parameter i. Since only relevant predicates can be in a state of a class, if P is not relevant, it will only be added to Setprecond of parameter i and its negation will be added to SetPostcond if P is not P_2.

ABS_effects(α): The algorithm is similar to the ABS_precond(α). However, it considers the predicates in the add list and do not compare with the delete list.

After the process of analysis and the extraction of SetInitState and SetPostState of each action, we must gather the states for each class.

Gather_States: The translation process groups by class all SetInitStates and all SetPostStates in a list of states. Then, the process gathers all states which are the same, i.e., that have the same set of predicates.

In order to discover that one state is the same of another state, they must have predicates and variables with the same name and order. Ex: the argument of both on(x,y) in one state and on(x,y) in another state must be equals. If they are not the same, the states will not be comparable and they will be considered different states.

Create_StateChartDiagram: The translation process examines all actions and groups of each class presented in the PL list in order to define the StateChart Diagrams.

For one class, we will have a set of SetIntiStates and SetPostState and the actions related to them. Each state description in SetInitState and SetPostState will become a state in the StateChart Diagram, Sf and St respectively in accordance to figure 1. Then, each action related to these states will be described as a transition between

these states. This action will have incorporated in its `pre` and `post` definition all Setprecond and SetPostcond list of predicates for that class.

This process occurs to all classes and all actions. However, some Setprecond and Setpostcond of some action lie in a parameter that does not have a StateChart Diagram. In these cases, the SetPrecond and SetPostcond will be jointed to the SetPrecond and Sertpostcond of the class of the first parameter of the action.

4 Evaluation

The translation process described in this paper was implemented in itSIMPLE tool. Many STRIPS-like domains like Block World, Logistic, Free Cell, and others were used to evaluate the translation process. In all cases the translation was successfully done and the translation back to PDDL was correctly performed, which certify that the translation process keeps the same semantic in both models.

5 Conclusion

This paper showed a translation process that can read a STRIPS-like domain model described in PDDL and write it in UML.P. The translation process made some considerations in PDDL description because the *UML.P* approach model, which relies on object-oriented specifications, needs more information than the declarative description of the PDDL can provide. However, with these considerations, a model described in PDDL can be easily extracted as object-oriented model and can be read by tools like itSIMPLE to visualize, analyze, and verify the domain description.

References

1. Boddy, M. Imperfect Match: PDDL2.1 and Real Applications. *Journal of Artificial Intelligence Research* 20 (2003) 133 – 137.
2. D'Souza, F. D. and Wills, A. C. *Object, Components, and Frameworks with UML – The Catalysis Approach*. Addison-Wesley. USA and Canada (1999).
3. McDermott, D *et al*. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Committee (1998).
4. OMG - Object Management Group. *Unified modeling language specification: version 1.4.* URL http://www.omg.org/uml (2001).
5. Simpson, R.M, McCluskey, T. L., Zhao, W., Aylett, R. S. and Doniat, C. *An Integrated Graphical Tool to support Knowledge Engineering in AI Planning*. Proceedings, 2001 European Conference on Planning, Toledo, Spain (2001).
6. Simpson, R.M, McCluskey, T.L, Liu and Kitchin. "Knowledge Representation in Planning: A PDDL to OCLh Translation", LNCS 1932 (2000).
7. Vaquero, T. S., Tonidandel, F., Silva, J. R. *The itSIMPLE tool for Modeling Planning Domains*. ICAPS 2005 Competition on Knowledge Engineering for Planning and Scheduling, Monterey, California, USA (2005).