From Requirements and Analysis to PDDL in itSIMPLE$_{3.0}$

Tiago Stegun Vaquero[1,2] and **José Reinaldo Silva**[1] and **Marcelo Ferreira**[3]
Flavio Tonidandel[3] and **J. Christopher Beck**[2]
[1] Department of Mechatronics, Universidade de São Paulo, Brazil
[2] Department of Mechanical & Industrial Engineering, University of Toronto, Canada
[3] IAAA Lab, Centro Universitário da FEI - São Bernardo do Campo, Brazil
tiago.vaquero@poli.usp.br, reinaldo@usp.br, m_fer@uol.com.br, flaviot@fei.edu.br, jcb@mie.utoronto.ca

Abstract

Transforming requirements of real planning applications into a sound input-ready model for planners has been one of the main challenges in the study of Knowledge Engineering within AI planning. However, few tools and methods have been developed to facilitate this transformation process. itSIMPLE is a research project dedicated to support the design phases of such planning models. In this papers we describe how requirements in UML are translated to solver-ready PDDL models in itSIMPLE$_{3.0}$. We also present the translation from UML to Petri Nets for domain analysis. Finally, an overview of the tool support for analysis of plans returned by planners is exposed.

Introduction

It is well-known that real planning applications require careful design process, especially during the initial phases of a development project. Requirements gathering and modeling are two of the main challenges that usually impact directly on the final planning application. Extracting relevant knowledge from different sources (e.g. documents, experts, users, stakeholders) and then representing it in a sound model is indeed a hard task. Knowledge Engineering (KE) concepts have been investigated to help the designer. However, few tools and languages have been applied to facilitate the initial design phases, in which knowledge is gradually transformed from an informal representation to a formal model that can be sent to AI planners.

The itSIMPLE (Vaquero et al. 2007) project is a research effort to develop reliable KE tools for planning. Unlike other tools, itSIMPLE focuses on initial phases of a disciplined design cycle for creating sound models of real domains. The tool provides an integrated environment that combines languages and tools for supporting designers in the design process. In such environment, requirements gathering and modeling are performed using the *Unified Modeling Language* (UML) (OMG 2005), a general purpose language broadly accepted in Software Engineering and Requirements Engineering. The Petri Nets formalism (Murata 1989) is used to analyze the requirements in the UML models. A PDDL representation (up to 3.1) of the resulting UML model is automatically provided to be read by AI planners. The planning solutions given by these solvers are then simulated and evaluated in the tool.

This papers aims to expose the translation processes behind the itSIMPLE$_{3.0}$ framework. We focus on the translation from UML models to PDDL and also the representation of some UML components in Petri Nets. The main contributions of this paper are:

- A mapping process from UML to a solver-ready PDDL model;
- A Petri Nets representation of UML models for planning domain analysis;

This paper is organized as follows. First, we give an overview of itSIMPLE$_{3.0}$ and its language framework. Next, we describe the translation processes that guide the user from requirements in UML and Petri Nets-based analysis to a PDDL model. We then give a brief description of the tool support for analyzing plans returned by solvers. Finally, we present the conclusion and future work.

itSIMPLE$_{3.0}$ and its Language Framework

itSIMPLE is an open source project that aims to support designers in the knowledge engineering processes of real planning applications (Vaquero et al. 2007). itSIMPLE's integrated environment focuses on the crucial initial phases of a design such as requirements specification, modeling, model analysis, and plan evaluation (Vaquero et al. 2007). The tool has been applied and tested since 2005 in several real planning applications including petroleum supply ports (Sette et al. 2008), project management (Udo et al. 2008), manufacturing (Vaquero et al. 2006), information systems, and intelligent logistic systems. itSIMPLE$_{3.0}$, the latest version, brings new features such as the UML timing diagram for time-based models, new PDDL translation capabilities, flexibility in using planners, and an extended plan analysis tool.

The itSIMPLE environment allows users to follow a disciplined design process to create knowledge intensive domain models, from the informality of real world requirements to formal representations that can be directly read by solvers. The suggested design process, shown in Figure 1, follows a cycle of phases inherited from Software Engineering and Design Engineering, combined with modeling experiences of real planning domain.

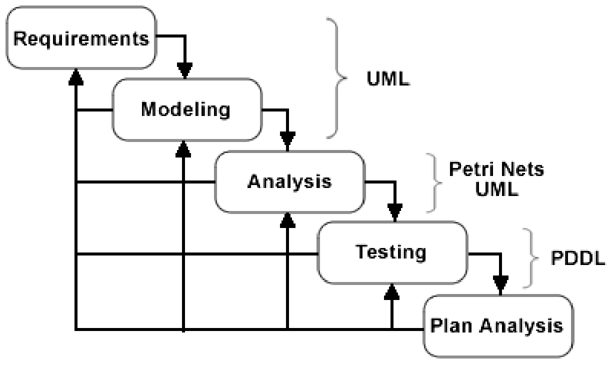


Figure 1: Design process in itSIMPLE$_{3.0}$

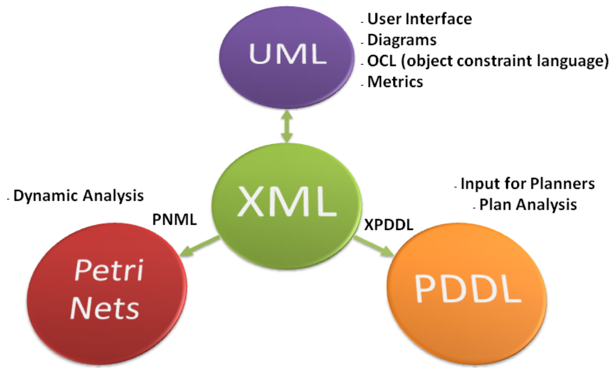


Figure 2: The architecture of the integrated languages

In the proposed process, requirements gathering and modeling are perform using UML. This diagrammatic language is an established notation for object-oriented design commonly used for modeling software applications, web applications, and business processes (OMG 2005). itSIMPLE allows designers to analyze UML models, including their dynamic characteristics, by using the Petri Nets (PN) formalism. Petri Nets are a well-known schema to represent workflow, discrete events and discrete dynamic processes in general (Murata 1989). The simulation of the resulting PN can reveal the need for refinements of the UML models. To deliver the analyzed UML model to a planner, itSIMPLE uses the standard PDDL representation which most planning systems support. As a final step, the tool supports designers during plan analysis, including simulation with UML diagrams and plan evaluation using acquired metrics. All adjustments and maintenance on the model, resulting from analysis, are performed manually in the UML representation.

In order to facilitate the translation between languages in the proposed design process, itSIMPLE uses the well-known language XML (Bray et al. 2004) as a core language for storing all information from UML diagrams (reflecting directly the UML model). Petri Nets and PDDL have direct representation in XML. For example, *Petri Nets Markup Language* (PNML) (Billington et al. 2003) is a XML-based representation of PNs while *eXtensible Planning Domain Definition Language* (XPDDL) (Gough 2004) is a XML-based representation of PDDL. The tool utilizes these two XML-based languages as means of achieving the PN representation and the PDDL model. All internal verifications and translations are performed in the data available in the core XML file, as shown in Figure 2.

Translation Processes

The main translators available in itSIMPLE$_{3.0}$ are based on mapping processes. Knowledge inserted in the tool using UML diagrams are first stored as an XML representation which is then mapped to PN and PDDL. Depending on which translation is requested by users, the tool extracts the necessary data from the central XML-based representation.

From UML to XML

itSIMPLE$_{3.0}$ provides designers a set of UML diagrams as a front-end to requirements elicitation and domain modeling. Five diagrams are available: (1) use case diagram for requirements; (2) class diagrams for static characteristics of the domain; (3) state machine diagrams for dynamics; (4) the new timing diagrams for time-based domain features; and (5) the object diagrams for problem and constraint definitions. Besides these diagrams, UML provides a predefined formal language called *Object Constraint Language* (OCL) (OMG 2003) to describe expressions on UML models, especially in class diagrams, state machines and object diagrams. OCL was designed to specify domain invariants, pre- and post-conditions of actions, and application-specific constraints.

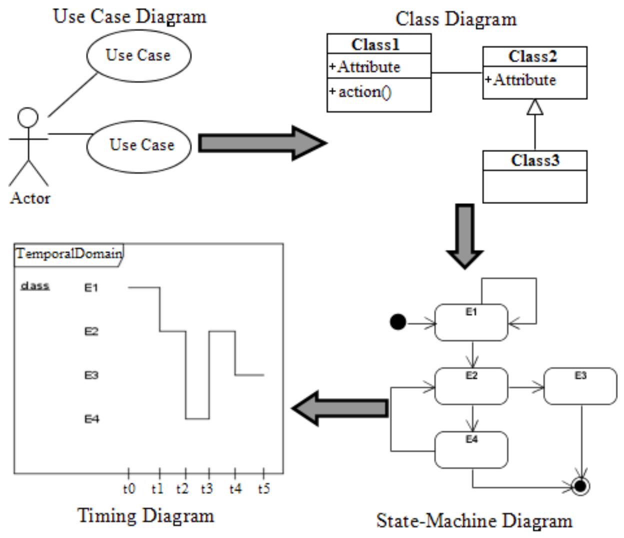


Figure 3: UML diagrams in itSIMPLE$_{3.0}$

All UML diagrams and expressions are stored directly in an XML representation. In fact, these diagrams are just a diagrammatic view of the knowledge in the XML model. The data input through the diagrams are represented using proper

XML tags that can be easily mapped back into a graphical representation of UML. The following example is a simplified XML representation of a class from UML.

```
<class id="6">
 <name>Truck</name>
 <description>class Truck</description>
 <stereotype>agent</stereotype>
 <attributes> ...  </attributes>
 <operators> ... </operators>
 <generalization .../>
 <constraints> ... </constraints>
</class>
```

Since the XML model reflects the UML diagrams in essence, the following descriptions of the translation processes use the term UML/XML as the central model of the planning domain in itSIMPLE.

From UML/XML to Petri Nets

In this translation, which is specific to dynamic analysis, itSIMPLE captures the data from state machines diagrams of UML/XML. These diagrams contain the knowledge directly related to dynamics. The tool creates a PNML representation of the state machines that will be graphically expose to the user as Petri Nets for visualization and simulation. While simulating the PNs, designers can validate the flow of the tokens in order to identifying deadlocks, parallelism, concurrency and inconsistent sequences of actions.

The PNML fits properly in itSIMPLE's analysis process not only because it is a XML-based representation, but also because it provides the concept of modules, called modular PNML (Kindler and Weber 2001), which is similar to object-oriented concepts. The *modular PNML* allows the definition of modules. A module in PNML encapsulates a set of places (states) and transition that defines the behavior of an object or artifact. A Petri Net in PNML can be created by instantiating and combining modules.

The following sections describe the mapping process from state machine diagram to modular PNML based on two analysis processes provided by itSIMPLE$_{3.0}$: *Modular Analysis* and *Interface Analysis* (Vaquero et al. 2007). It is important to note that these dynamic analysis are still in a preliminary stage and so far the tool uses only structural elements of the state machine diagrams to produce Petri Nets in PNML. Pre and post conditions of actions (described in OCL) are not considered yet. However, even with some limitations, the translation to PNML is still useful since it provides a simulation mechanism and also opens the opportunity to apply model checking techniques available in the Petri Nets literature (Murata 1989).

Modular Analysis

The Modular Analysis supports users in verifying the behavior of each class individually, taking into account dependencies with other classes. In order to perform this analysis, every state machine (representing a class) is a module in the modular PNML. Each state in a UML state machine is converted to a place in the PNML module while every action (arc) is converted to a transition element in the PNML module. Since the actions connect states in the UML diagram, the resulting transitions will connect the corresponding places in the module. The 'initial state' element in state machine indicates the initial token position in the module.

As an example of a PNML module, lets suppose a state machine diagram representing a class C$_D$ where an action t (that does not depend of any other class) leaves a state $s1$ and goes to the state $s2$. In this example, an 'initial state' points to $s1$. The state machine diagram would be translated as a module d in the PNML in the following simplified form:

```
<module name="d">
     <interface> ...
     </interface>
     <place id="{s1}">
        <initialMarking>
          <text>1</text>
        </initialMarking>
     </place>
     <place id="{s2}"> ... </place>
     <transition id="{t}"/>
     <arc source="{s1}"  target="{t}" />
     <arc source="{t}"  target="{s2}" />
</module>
```

In this modular PNML approach, actions that belongs to other classes are distinguished graphically, creating a dependency relationship in the modules. Furthermore, actions that depend on other classes receive an extra state, as a precondition, also to represent dependency. Figure 4 shows an example of Petri Net modules derived from *Package* and *Airplane* state machines of the Logistic domain. All transitions in the *Package*'s module (a) indicate that they affect the behavior of such class; however, they belong to and depend on other classes (modules), in this case the classes *Truck* and *Airplane*. On the other hand, *Airplane*'s module shows actions *load* and *unload* represented differently. This graphical difference shows that the actions belongs to the *Airplane* but they depend on other modules, such as *Package*. The action *fly* does not depend on other modules and it is then represented as a simple transition.

Interface Analysis

The Interface Analysis investigates the dynamic interactions among modules. During this analysis, designers can verify not just one but many modules together, visualizing their dependencies. In this analysis, state machines are translated individually as modules, following the same approach of Modular Analysis. However, the chosen modules are joined in a single PNML representation following the approach described in (Kindler and Weber 2001). When modules are combined, actions that appears in different diagrams are merged graphically as shown in Figure 5. As a result, a PNML file is generated and shown to users for simulation.

From UML/XML to PDDL

As opposed to the previous translation process, translating from UML/XML to PDDL requires that all knowledge contained in UML/XML must be represented in the PDDL model. In this procedure, the UML/XML model is represented as a PDDL model by means of XPDDL. Since

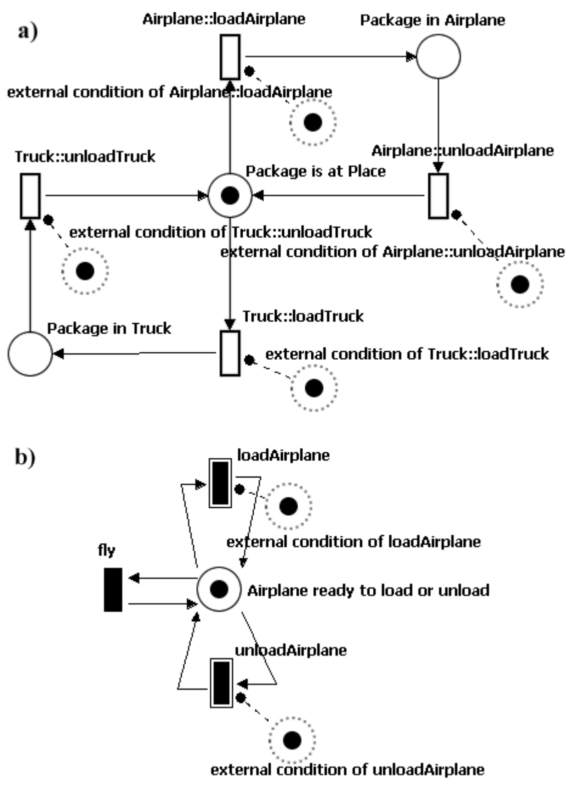

a)
Airplane::loadAirplane
Package in Airplane
external condition of Airplane::loadAirplane
Truck::unloadTruck
Package is at Place
Airplane::unloadAirplane
external condition of Truck::unloadTruck
external condition of Airplane::unloadAirplane
Truck::loadTruck
Package in Truck
external condition of Truck::loadTruck
b)
loadAirplane
external condition of loadAirplane
fly
Airplane ready to load or unload
unloadAirplane
external condition of unloadAirplane

Figure 4: Modular Analysis of (a) Package and (b) Airplane

XPDDL to PDDL translation is a straightforward process (as seen in (Gough 2004)), in this section we will directly refer to the mapping from XML/UML to PDDL.

Because PDDL models are divided in two files, domain and problem, the following descriptions focus on each translation process individually.

Domain Translation

A PDDL domain file contains static information about the model and the specification of actions/operators. This information is found in the UML/XML representation in the class diagrams, state machine diagrams, and timing diagrams.

Mapping Types

The mapping process starts from the class diagrams, in which all defined classes are extracted and represented in the *:types* section of the domain file. The hierarchy relationship is respected and represented in PDDL. For example, a class *Truck*, which is a specialization of a class *Vehicle*, would be represented as *Truck - Vehicle* in PDDL types. Figure 6 shows the mapping rules for types in PDDL.

Mapping Predicates and Functions

Predicates and functions are also mapped to PDDL from class diagrams, specifically from classes' attributes and associations. Generally, attributes defined as Boolean are represented as predicates in the *:predicates* section of PDDL (for example, attribute *clear* of class Block is mapped as *(clear ?x - Block)*). Integer or Float are represented as fluents in *:functions* of PDDL.

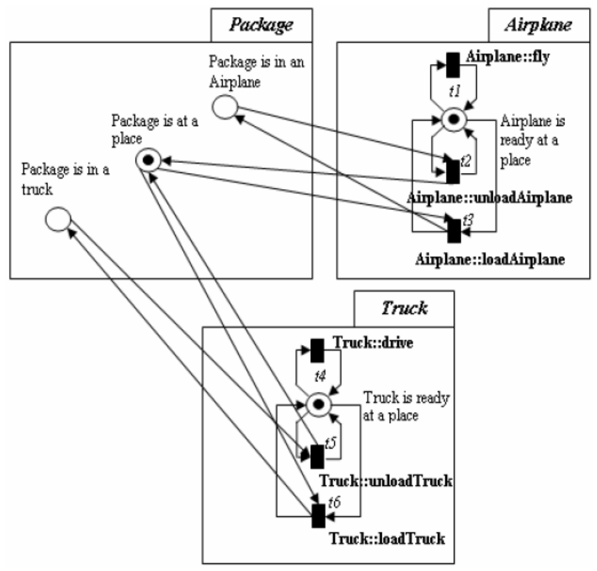


Figure 5: Interface Analysis of the classes Package, Truck and Airplane

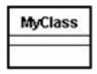
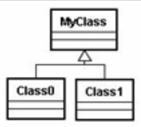
UML	PDDL
	`(:types MyClass - object ... )`
	`(:types Class0 - MyClass Class1 - MyClass MyClass - object ...)`

Figure 6: Mapping PDDL types from class diagrams

Some of the attributes are treated distinctly based on the chosen version of PDDL. For example, attributes that are defined as having their types of another class may depend on PDDL version (e.g., attribute *onTable* of a *Block* class is of type *Table*). In version 2.1, 2.2, and 3.0 these attributes are represented as predicates, for instance *(onTable ?x - Block ?y - Table)*. Conversely, in version 3.1[1], these cases are naturally mapped as fluents in *:functions*, for example *(onTable ?x - Block) - Table*.

Parameterized attributes are also used in UML class diagrams. For example, the attribute *distance(from:City, to:City)* of type Float from a class *CityMap* is a possible parameterized attribute in a class diagram. In these cases, the representation depends on the attribute's type as describe above. For the distance example, it would be represented in the *:function* section of PDDL as *(distance ?from - City ?to - City)*.

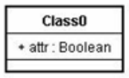
Attributes defined in classes that have stereotype "utility" are treated as global variables. The translation of these

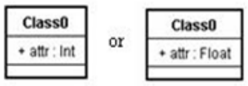
[1]PDDL3.1 http://ipc.informatik.uni-freiburg.de/PddlExtension

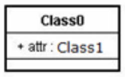
attributes to PDDL follows the same approach described above, but the class's name is not mentioned. For example, attribute *totalfuel* (Float) from a class called *Global* (utility) would be mapped as *(totalfuel)* in the section *:functions* of PDDL.

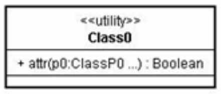
Associations are treated as predicates in PDDL as well. For example, lets suppose that class *Truck* has an association, called '*at*', with class *Place*. This association is mapped as *(at ?x - Truck ?y - Place)* in PDDL predicates. Figure 7 illustrates some of the main rules for translating predicates and functions in PDDL.

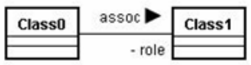
UML	PDDL

	`(:predicates`
	`  (attr ?cla - Class0))`

	`(:functions`
	`  (attr ?cla - Class0))`

	`(:predicates`
	`  (attr ?cla0 - Class0`
	`    ?cla1 - Class1))`
	or
	`(:functions`
	`  (attr ?cla0 - Class0)`
	`    - Class1))`

	`(:predicates`
	`  (attr ?p0 - ClassP0 …`
	`    ?pn - ClassPN))`

	`(:predicates`
	`  (role ?cla0 - Class0`
	`    ?cla1 - Class1))`

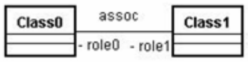
	`(:predicates`
	`  (role1 ?cla0 - Class0`
	`    ?cla1 - Class1)`
	`  (role0 ?cla1 - Class1`
	`    ?cla0 - Class0))`

Figure 7: Mapping PDDL predicates and functions from class diagrams

Mapping Actions
Since classes hold the name and parameters of their operators, itSIMPLE represents each action in the PDDL domain based on this information. As an example, operator *Truck::drive(t:Truck, from:Place, to:Place)* from a UML class is mapped as *(:action drive :parameters ?t - Truck ?from - Place ?to - Place))*. However, pre and post conditions are generally not specified in class diagrams, but in the state machine diagrams. Figure 8 shows the mapping of the name and parameters of actions.

In itSIMPLE's state machine diagrams, the pre- and post-conditions of each operator and the states are defined in OCL expressions (OMG 2003). These expressions are used in the actions to represent their conditions in the diagram. Moreover, each state is defined by the possible values of the class's attributes using conjunctive and disjunctive OCL expressions.

UMP	PDDL

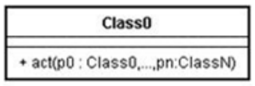
	`(:action act`
	`  (:parameters`
	`    ?p0 - Class0`
	`    ?p1 - Class1`
	`    ...`
	`    ?pn - ClassN)`
	`  (:precondition …)`
	`  (:effect …))`

Figure 8: Mapping PDDL action's name and parameters from class diagrams

Since an operator can affect different classes of objects, the conditions of such operators can be spread among the state machine diagrams. The mapping process collects all preconditions and postconditions of each operator, merging all state machines. In order to translate merged conditions expressed in OCL, itSIMPLE has a map that correlates OCL expressions and PDDL conditions. For example, if the expression '*block.clear = true*' is found in an operator's precondition, the tool would add the following expression into the proper action of PDDL: (clear ?block). Another example: '*truck.currentLoad = truck.currentLoad + 1*' in a postcondition would be represented as *(increase (currentLoad ?truck))* in PDDL.

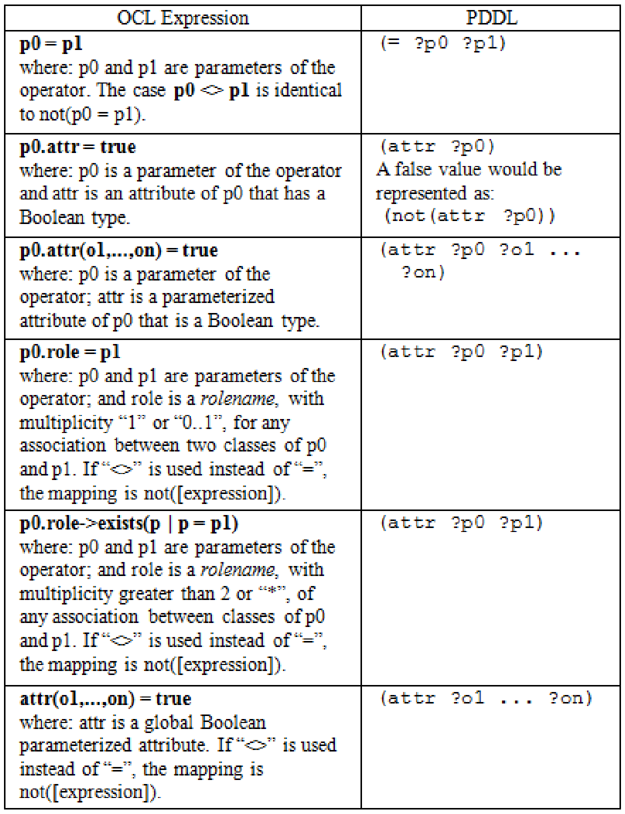
It has been observed that some of itSIMPLE's users define pre- and post-conditions of actions directly in the class diagrams using OCL expression. In this case, itSIMPLE does not perform the state machine merging process; instead, the tool performs the expression mapping directly. Figure 9 shows some examples of the mapping rules for translating OCL expressions into PDDL conditions. A complete map of pre and post-conditions in OCL into XPDDL and PDDL is described in the user documentation available in the project's website.[2]

Since OCL expressions on post-conditions work with variables attribution, itSIMPLE's translator must treat the cases where negating predicates are necessary. For example, if the OCL expression '*truck.at = from*' appears in the precondition and '*truck.at = to*' appears at the postcondition of an action *drive*, the tool would automatically add the condition *(not (at ?truck ?from))* in the *:effect* of a PDDL action, along with *(at ?truck ?to)* condition.

Mapping Temporal Characteristics of Actions
With the new timing diagrams added in itSIMPLE$_{3.0}$, temporal characteristics of actions can also be modeled and translated to PDDL. The timing diagram was added in order to address challenging domains such as those involving time. This diagram is a timeline based approach to capture temporal aspects of actions. When this diagram is used in a planning approach, it is intrinsically connected to the state machine diagrams which supply all significant states and attributes of an object.

There are two approaches to modeling temporal aspects in a domain using timing diagrams. The first one has a more

[2]User documentation. http://dlab.poli.usp.br



OCL Expression	PDDL
p0 = p1 where: p0 and p1 are parameters of the operator. The case **p0 <> p1** is identical to not(p0 = p1).	`(= ?p0 ?p1)`

p0.attr = true
where: p0 is a parameter of the operator and attr is an attribute of p0 that has a Boolean type.
	`(attr ?p0)`
	A false value would be represented as:
	`(not(attr ?p0))`

p0.attr(o1,...,on) = true where: p0 is a parameter of the operator; attr is a parameterized attribute of p0 that is a Boolean type.	`(attr ?p0 ?o1 ... ?on)`
p0.role = p1 where: p0 and p1 are parameters of the operator; and role is a *rolename*, with multiplicity "1" or "0..1", for any association between two classes of p0 and p1. If "<>" is used instead of "=", the mapping is not([expression]).	`(attr ?p0 ?p1)`
p0.role->exists(p | p = p1) where: p0 and p1 are parameters of the operator; and role is a *rolename*, with multiplicity greater than 2 or "*", of any association between classes of p0 and p1. If "<>" is used instead of "=", the mapping is not([expression]).	`(attr ?p0 ?p1)`
attr(o1,...,on) = true where: attr is a global Boolean parameterized attribute. If "<>" is used instead of "=", the mapping is not([expression]).	`(attr ?o1 ... ?on)`

Figure 9: Mapping OCL expression to PDDL conditions

general view of the model in which all objects are presented in a single diagram. This approach shows the how long the objects remain in each of their states in a possible sequence of actions. Each object in this diagram receives a frame that is linked to a shared timeline. The timeline represents the general duration of the possible sequence of actions. Each object's state is linked to time points that represent its duration. Figure 10 shows an example of a timing diagram using two objects. This diagram illustrates that each one has its own life-cycle that contains all states of the object and also the duration of each state.

The second approach shows the temporal details of a specific action. The goal is to describe how attributes and properties change during an action execution. OCL expressions are also used for defining the evolution of these attributes. Only the objects related to such action can participate in the diagram, as shown in Figure 11.

Currently, only the second approach is considered in the translation process to PDDL. If an action is represented in a timing diagram, this action will be a durative-action in PDDL. Accordantly to the latest version of PDDL, there are three types of temporally annotated conditions and effects: (1) *at start*, specifies that a variable must have a specific value when the action is triggered; (2) the *over all* specifies that a variable has to hold its values during the execution of the action; and (3) the *at end* specifies that the variable must has a specific value at the end of the action. When

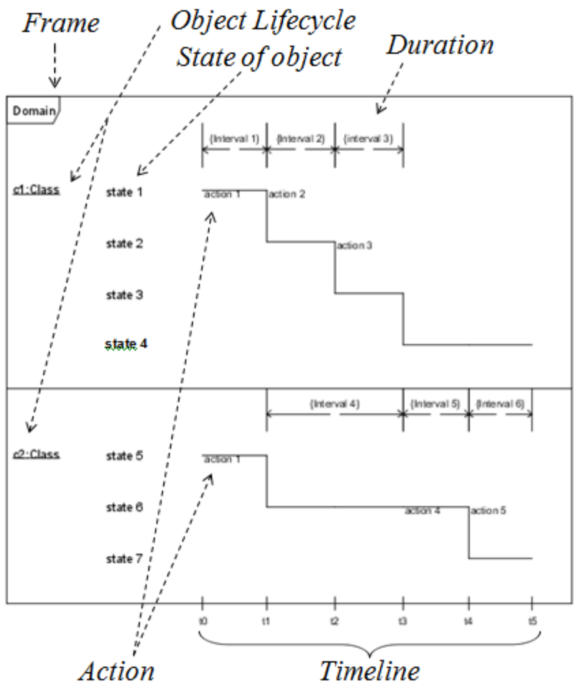


Figure 10: Timing diagram in itSIMPLE$_{3.0}$

an OCL expression is defined in the timing diagram, each property is translated to PDDL surrounded by one of these three temporal operators, depending on how they appear in the diagram. OCL expressions are also translated to PDDL following the processes described previously. For example, expression '*attribute* = *attribute* + *number*' in the effects would be interpreted as *(increase (attribute) number)* in PDDL surrounded by a temporal operator (e.g., *at end*) (Fox and Long 2003).

Mapping Constraints

The first constraint treated in itSIMPLE's translator is related to the association multiplicity on the class diagrams.

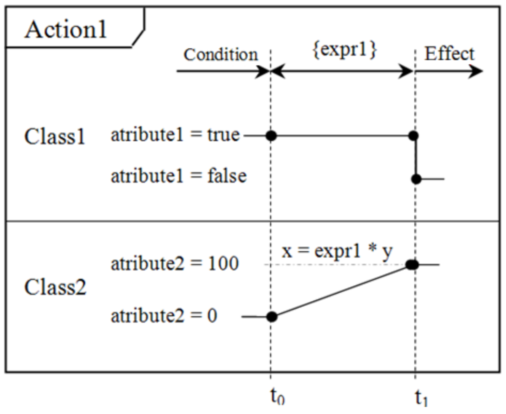


Figure 11: Timing diagram of an action in itSIMPLE$_{3.0}$

In the association '*at*' mentioned before, a *Truck* can only be at one *Place* at a time. The multiplicities are represented as constraints (using '*always*' operator) in the section *:constraints* of a PDDL 3.0 domain(Gerevini and Long 2006). In fact, these constraints reinforce (make explicit in the model) what most of time are implicit on action's conditions and effects. Figure 12 illustrates this mapping rule.

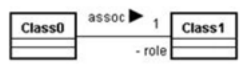
UML Multiplicity	PDDL
	`(:constraints (forall (?cla11 - Class1 ?cla12 - Class1 ?cla0 - Class0) (always (imply (and (role ?cla0 ?cla11) (role ?cla0 ?cla12)) (= ?cla11 ?cla12))) ))`

Figure 12: Mapping association multiplicity to PDDL constraints

As a new feature in itSIMPLE$_{3.0}$, users can also specify constraints on classes, attributes or associations in class diagrams by using OCL. For example, one could want to constrain the battery power level of a *Robot* inserting the following OCL expression in the class: *inv: self.powerlevel <20*. This expression would be represented in the domain section *:constraints* from PDDL 3.0 as *(always (forall (?r - Block) (<(powerlevel ?r - Robot) 20)))*. The translation process of these constraints is restricted to the available mapping of OCL expression to PDDL described previously.

Problem Translation

A PDDL problem file considered in itSIMPLE$_{3.0}$ can contain five main elements: objects, initial state, goal conditions (objective state), metrics, and problem constraints. This information is found in the object diagrams of the UML/XML representation.

Mapping Objects

The tool provides a dedicated object diagram (called *object repository*) to hold all objects used in a set of planning problems. Every object's name, along with the respective class's name, in the repository is translated and inserted to the section *:object* of a PDDL problem file. The mapping rule for objects in PDDL is shown in Figure 13.

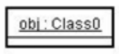
UML	PDDL
	`(:objects (obj - Class0))`

Figure 13: Mapping PDDL objects from object repository

Mapping Initial and Goal States

In every problem, there are at least two object diagrams, the *init* and the *goal*. The translation process for both diagrams follows the same mapping approach. Starting from the *init* snapshot, the tool seeks an object's attributes and their values in order to represent them in PDDL. For example, a graphical object *truck1* with attribute *capacity* equals to *100* would be represented as *(= (capacity truck1) 20)* in section *:init* of PDDL. Another example is an object *blockA* with attribute *clear* equal to *true* that would be represented as *(clear blockA)* in PDDL. Since associations are also treated as predicates, as previously described, their translation is also straightforward. For instance, *truck1* associated with *place1* through association '*at*' in the object diagram would be mapped as *(at truck1 place1)*. Another general example of creating the PDDL initial state from object diagrams is presented in Figure 14. The goal state (*:goal*) follows the same translation process as for the *:init*.

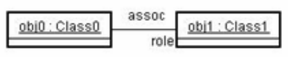
UML - Initial State	PDDL
	`(:init (role obj0 obj1))`

Figure 14: Mapping PDDL init from object diagram

Mapping Problem Constraints

Additional object diagrams can be used to represent the *Timed Initial Literal* concept from PDDL 2.2 (Edelkamp and Hoffmann 2004) and also the *State Trajectory Constraints* concept from PDDL 3.0 (Gerevini and Long 2006). When considering *timed initial literal*, users usually create specific situations or facts in time using object diagrams for describing exogenous events. Each diagram is attached to a specific point in time representing that all containing elements (associations and attributes with their respective values) will be true at such time point. This approach follows the same translation process as the init state; however, elements in the diagram are translated to the section *:init* preceded by the specified point in time. An example would be *(at 5 (clear blockA))* in PDDL 2.2.

In order to model *state trajectory constraints*, users can also create object diagrams that represent situations to be constrained. These object diagrams are attached to the desired constraint using basic modal operators such as *always, sometime, at-most-once*, and *at end* or *never* (*always* not). In this case, the whole snapshot is first translate to PDDL as described for init and goal states. Then, it is inserted in the *:constraints* section of a PDDL 3.0 problem file surrounded by the desired basic modal operator such as *(:constraints (and (always (<facts from the object diagram>))*.

Mapping Metrics

Finally, user can insert in the model OCL expressions containing an object's attributes that affect directly the quality of plans. These expressions must be generally maximize or minimize, depending on the problem. Following the approach of mapping OCL expression to PDDL conditions, itSIMPLE inserts the translated expression in the *:metric* section of PDDL. For example, a minimization of the expression '*robot1.traveldistance + robot1.powerusage*' would be translate to *(:metric minimize (+ (traveldistance robot1) (powerusage robot1)))* in a PDDL representation.

With both PDDL files created, itSIMPLE can deliver the PDDL model to a chosen planner.

Plan Analysis Support

For complex problems, lack of knowledge or ill-defined requirements can propagate to specifications and then to the problem submitted to the planner. Either of these scenarios (and others) may lead to the generation of poor quality plans. In these cases, bad plans and defects to a set of requirements must be spotted and fixed. Following these principles, itSIMPLE$_{3.0}$ allows users to test the generated PDDL model with a set of modern planners (Metric-FF, FF, SGPlan, MIPS-xxl, LPG-TD, LPG, hspsp, and SATPlan) in order to analyze the quality of the produced plans. Plan analysis starts from plan visualization and simulation in UML to a plan evaluation based on domain metrics.

Plan visualization and simulation, provided by the functionality called "Movie Maker" (Vaquero et al. 2007), are performed by capturing the response of a planner and executing the plan from the initial state to the goal state. This process creates a sequence of UML object diagrams that simulates the plan, state by state. Plan evaluation can be performed by selecting the domain metrics that directly effect the plan quality and observing their evolution during the simulation. Preference on values of these metrics can be defined by correlating the values to grades. These preferences allow itSIMPLE to evaluate the plan and produce a plan report that provides an overall grade for the plan, along with the charts showing the evolution of the metrics.

Conclusion

In this paper, we presented the translation processes available in itSIMPLE$_{3.0}$, a KE tool that has been developed to support designers in the development of real planning domains. We have described how requirements modeled in UML can be analyzed by Petri Nets and translated to an input-ready PDDL model for planners. The UML to Petri Nets translation opens the possibility to validate and analyze the model by using simulation and model checking techniques available in the PNs literature. The translation from UML to PDDL 3.1 provides a mechanism for testing and analyzing models with planners. The analysis gives the opportunity to improve models and, consequently, the quality of plans.

As future work, we have been investigating new methods for plan analysis such as virtual prototyping of plans and plan comparison that will reinforce the gradual improvement of domain models. New UML diagrams have been also studied to be included in itSIMPLE. The first candidate is the activity diagram which will represent HTN domains and some predefined strategies for planning. Finally, we plan to include the *System Modeling Language* (SysML[3]) to the itSIMPLE's framework for model verification and validation.

[3]www.omgsysml.org.

Acknowledgments

The authors are especially grateful to the researchers and students involved directly and indirectly in the project, as well as the many others who have, from time to time, sent suggestions on improvements to the itSIMPLE tool. This work has been supported in part by CNPq and CAPES.

References

Billington, J.; Christensen, S.; van Hee, K.; Kindler, E.; Kummer, O.; Petrucci, L.; Post, R.; Stehno, C.; and Weber, M. 2003. The petri net markup language: concepts, technology, and tools. In *Proceedings of the 24th Int Conf on Application and Theory of Petri Nets, LNCS 2679, Springer*, 483–505.

Bray, T.; Paoli, J.; Sperberg-McQueen, C. M.; Maler, E.; and Yergeau, F. 2004. Extensible Markup Language (XML) 1.0 (Third Edition). Technical report.

Edelkamp, S., and Hoffmann, J. 2004. Pddl 2.2: The language for classical part of the 4th international planning competition. Technical report, Fachbereich Informatik and Institut fr Informatik, Germany.

Fox, M., and Long, D. 2003. Pddl2.1: An extension of pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)* 20:61–124.

Gerevini, A., and Long, D. 2006. Preferences and soft constraints in pddl3. In Gerevini, A., and Long, D., eds., *Proceedings of ICAPS workshop on Planning with Preferences and Soft Constraints*, 46–53.

Gough, J. 2004. Xpddl 0.1b: A xml version of pddl.

Kindler, E., and Weber, M. 2001. A universal module concept for petri nets. In *Proceedings of the 8th Workshops Algorithmen und Werkzeuge fr Petrinetze*, 7–12.

Murata, T. 1989. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, 541–580.

OMG. 2003. *UML 2.0 OCL Specification m Version 2.0.*

OMG. 2005. *OMG Unified Modeling Language Specification, m Version 2.0.*

Sette, F. M.; Vaquero, T. S.; Park, S. W.; and Silva, J. R. 2008. Are automated planners up to solve real problems? In *Proceedings of the 17th World Congress The International Federation of Automatic Control (IFAC'08), Seoul, Korea*, 15817–15824.

Udo, M.; Vaquero, T. S.; Silva, J. R.; and Tonidandel, F. 2008. Lean software development domain. In *Proceedings of ICAPS 2008 Scheduling and Planning Application woRKshop. Sydney, Australia.*

Vaquero, T. S.; Tonidandel, F.; Barros, L. N.; and Silva, J. R. 2006. On the use of uml.p for modeling a real application as a planning problem. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS)*, 434–437.

Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itSIMPLE2.0: An integrated tool for designing planning environments. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS 2007). Providence, Rhode Island, USA.*