Extracting State Constraints from UML Planning Models

Marcel Lira Gomes[1] and **Tiago Stegun Vaquero**[2] and **José Reinaldo Silva**[2] and **Flavio Tonidandel**[1]
[1]Centro Universitário da FEI
IAAA - Artificial Intelligence Applied in Automation Lab - São Bernardo do Campo, Brazil
[2]Escola Politécnica - Universidade de São Paulo
Design Lab, Mechatronic and Mechanical Systems Department - São Paulo, Brazil
mmgomes@fei.edu.br, tiago.vaquero@poli.usp.br, reinaldo@usp.br, flaviot@fei.edu.br

Abstract

The knowledge Engineering for Planning and Scheduling requires techniques and tools that can assist a designer to better understand, specify, verify and validate a domain model. There are some tools, like itSIMPLE and GIPO, that can meet the modeling process requirements but none of them can reason about the domain in order to validate model descriptions automatically. This paper, therefore, aims to enhance itSIMPLE tool with a formal interpretation of UML models in F-Logic and a consequent formal and valid extraction of State Constraints which can help designers to verify their models. This extraction is the first step towards a robust system that can provide a consistency validation of Planning UML models.

Introduction

From the first International Competition on Knowledge Engineering for Planning and Scheduling - ICKEPS 2005 - much attention has been given to the tools that help the modeling process for planning engines. The modeling process requires functionalities that assist designers to better understand, specify, visualize, verify and validate their planning domain models (Vaquero *et al.* 2007). Knowledge Engineering (KE) tools, such as itSIMPLE (Vaquero, Tonidandel, & Silva 2005), ModPlan (Edelkamp & Mehler 2005), GIPO (Simpson 2005) and others, have been developed to meet the modeling process requirements.

The itSIMPLE tool was designed to assist the user during the life cycle of a planning domain modeling. It is an enhanced integrated environment based on languages such as UML (OMG 2001), XML (Bray *et al.* 2004), Petri Nets (Murata 1989) and PDDL (Fox & Long 2003), which leads designers from the informality of real world requirements to formal domain models (Vaquero *et al.* 2007). The knowledge acquisition process is performed based on the UML.P (Unified Modeling Language in a Planning Approach), a special use of UML (Unified Modeling Language). The modeling process done in itSIMPLE is mainly based on four diagrams: Use Case, State Machine, Class and Object. The Use Case and State Machine diagrams are behavioral diagrams, depicting the dynamic aspects and the interactions of domain's objects. The State Machine diagram models the object's actions, permitting their main characteristics description, such as pre and post conditions. Opposed to behavioral diagrams, the structural Class and Object diagrams show the object's static characteristics. The Class diagram describes domain's classes and their relationships. A class represents a set of objects that have same characteristics, constraints and semantics. The Object diagrams represent facts or examples of domain resources, being used to depict the initial and final state of a planning problem.

Despite of the facilities provided by the UML in itSIMPLE, it still does not have a method to validate the internal consistency of a domain. The internal consistency is the process that gathers internal information and state constraints, by inference or information explicitly inserted in a model, from the diverse set of diagrams, by crossing them in order to check if conflicting information is obtained. When the system is capable to perform what we call cross validation among diagrams, it becomes a powerful Knowledge Engineering tool that make possible the consistency verification of the entire domain model.

However, the UML semantics is defined in informal written English turning the application of automatic inference methods directly in the diagrams inappropriate. So, in order to overcome this issue, the UML.P models must be interpreted to a formal logic that allows the reasoning on the diagrams. For this purpose the F-Logic (Kifer, Lausen & Wu 1995), a frame based language that meet with UML object oriented structure, was chosen to interpret the Class and Object diagrams in this paper.

Tools such DISCOPLAN (Gerevini 1998) and TIM (Fox & Long 1998) are already able to infer state constraints from actions and initial state of a domain problem modeled in PDDL. The same structure is also available in the UML.P State Machine diagram, so an adaptation of these tools could be applied to infer the same state constraints from this diagram. However, the UML.P, available in itSIMPLE, also has the Class diagram with several information available for analysis; however there is no method available to infer state constraints from it. Therefore, the focus in this paper is to present a formal inference of state constraints from the Class and Object diagrams in order to let, as a future work, the possibility to perform internal consistency checking of a domain model described in UML.P.

F-Logic

The F-Logic is a first order logic based on frames. The language is composed of constructors, variables, auxiliary symbols and usual first order connectives and quantifiers.

A simple term, that is a term without connectives, is called F-Molecules. As an example, the term *block[onBlock => block]* is a single scalar signature F-Molecule from the classic Block World domain, it asserts that an object typed as *block* has a method *onBlock* that can hold an object typed as *block*.

Beyond scalar signatures molecules, the language has IS-A assertions that associate objects to class sets (e.g.: *blckA:block*, affirms that *blckA* is an object of class *block*) and build subclass structures (e.g.: *car::vehicle*, affirms that *car* is a subclass of the class *vehicle*); scalar data expressions that give values to methods (e.g. *blckA[onBlock -> blckB]*, asserts that *blckA* has the value *blckB* in the method *onBlock*); set valued signatures assert that methods can hold more than one object (e.g.: *table[blocksOver =>> block]*, affirms that the objects typed as *table* has a method *blocksOver* that can hold a set of objects typed as *block*). The language also sets valued data expressions (e.g. *tbl1[blockOver ->>* $\{$*blckA, blckB*$\}$*]*, affirms that the objects *blckA* and *blckB* is hold by the method *blockOver* of the object *tbl1*]); and inheritable data expressions (e.g.: *block[onBlock *-> blckB]*, asserts that all objects typed as *block* will have the method *onBlock* with initial value of *blckB*) that allow inheritance of data expressions for the objects attached to a class.

The data expressions and signatures may also have parameters attached to methods, e.g.:

signA[methd@param$_1$*,...,param*$_n$ *=> signC]*,
objA[methd@val$_1$*,...,val*$_n$ *-> objC]*
signB[methd@param$_1$*,...,param*$_n$ *=>> signC]*,
objB[methd@val$_1$*,...,val*$_n$ *->> objC]*.

As seen, the terms *signX* and *objX* denote the class name and object name respectively and *methd* denotes any method of a class. The *param* parameters define a typed object that is passed as a parameter and *val* is the object that respects the type of the parameter.

Connecting the F-Molecules with the usual first order connectives will result in F-Formulas, e.g.: *block[onBlock => block]* $\wedge$ *block[onTable => table]*. However, when the same identifier for a signature or object is used for terms connected via $\wedge$ it will be abbreviated as *block[onBlock => block; onTable => table]*.

A model described in F-Logic keeps the characteristic and methods concentrated around the objects, presenting a different structural organization compared to the traditional first order logic, which arrange information around attributes. This structure is well adjusted to UML.P that is also based in object structure, what let easier an interpretation from one to another.

The F-Logic Language has twelve inference rules and one axiom. The axiom, called *reflectivity axiom*, asserts that a class is a subclass of itself, i.e. *X::X*, this axiom assists some inference rules showed below. The inference rules are grouped as: main, IS-A, type and others.

The main inference rules is composed by the following rules:

- Resolution ($\frac{\neg L \vee C, L' \vee C', \theta = mgu(L,L')}{\theta(C \vee C')}$)
- Factoring ($\frac{L \vee L' \vee C, \theta = mgu(L,L')}{\theta(L \vee C)}$, $\frac{\neg L \vee \neg L' \vee C, \theta = mgu(L,L')}{\theta(\neg L' \vee C)}$)
- Paramodulation ($\frac{L[T] \vee C, (T' \doteq T") \vee C', \theta = mgu(L,L')}{\theta(L[T \backslash T"] \vee C \vee C')}$)

The symbols *L* and *L'* are positive literals, *C* and *C'* are clauses and the others are *id-terms*, i.e. identity term. The *mgu* function is the most general unifier of the terms.

IS-A inference rules is composed by:

- IS-A acyclicity ($\frac{(P::Q) \vee C, (Q'::P') \vee C', \theta = mgu(\langle P,Q \rangle, \langle P',Q' \rangle)}{\theta((P \doteq Q) \vee C \vee C')}$)
- IS-A transitivity ($\frac{(P::Q) \vee C, (Q'::R') \vee C', \theta = mgu(Q,Q')}{\theta((P::R') \vee C \vee C')}$)
- Subclass Inclusion ($\frac{(P:Q) \vee C, (Q'::R') \vee C', \theta = mgu(Q,Q')}{\theta((P:R') \vee C \vee C')}$)

These rules capture the subclass relationship and its interaction with class membership.

Type inference rules capture:

- the properties of Inheritance ($\frac{P[Mthd@Q_1,...,Q_k => T] \vee C, (S'::P') \vee C', \theta = mgu(P,P')}{\theta(S'[Mthd@Q_1,...,Q_k => T] \vee C \vee C')}$)
- Input Restriction ($\frac{P[Mthd@Q_1,...,Q_i,...,Q_k => T] \vee C, (Q"_i::Q'_i) \vee C', \theta = mgu(Q_i,Q'_i)}{\theta(P[Mthd@Q_1,...,Q"_i,...,Q_k => T] \vee C \vee C')}$)
- Output Relaxation ($\frac{P[Mthd@Q_1,...,Q_k => R] \vee C, (R'::R") \vee C', \theta = mgu(R,R')}{\theta(P[Mthd@Q_1,...,Q_k => R"] \vee C \vee C')}$).

These rules are similar to set-valued methods.

From the others rules inference, the requirement that a scalar method must return at most one value is given by the Scalarity inference rule ($\frac{P[Mthd@Q_1,...,Q_k -> R] \vee C, P'[Mthd'@Q'_1,...,Q'_k -> R'] \vee C'}{\theta(R \doteq R' \vee C \vee C')}$, $\theta = mgu(\langle P, Mthd, Q_1, ..., Q_k \rangle, \langle P', Mthd', Q'_1, ..., Q'_k \rangle)$), this inference rule is similar for inheritable scalar expression. The merging property is captured by the Merging inference rule ($\frac{P[...] \vee C, P'[...] \vee C', \theta = mgu(P,P'), L" = merge(\theta(P[...]), \theta(P'[...]))}{L" \vee \theta(C \vee C')}$).

The Elimination inference rule state that $\frac{\neg P[] \vee C}{C}$, but if *C* is an empty clause then the elimination rule will derive an empty clause as well.

Interpretation

The interpretation must maintain all the information found in the Class and Object diagrams in order to enable the reasoning in the model. Not all available information in these diagrams are interpreted directly to F-Logic, so it is necessary to state a method for this process.

Since the F-Logic is a general purpose logic, it is necessary a definition of some patterns to make the interpretation simpler. So, for the aim of the interpretation the following signatures are defined: *typePrimData* (defines all the primary data, i.e. boolean, integer, and others); *limSet* (allows to get and maintain the multiplicity information of each

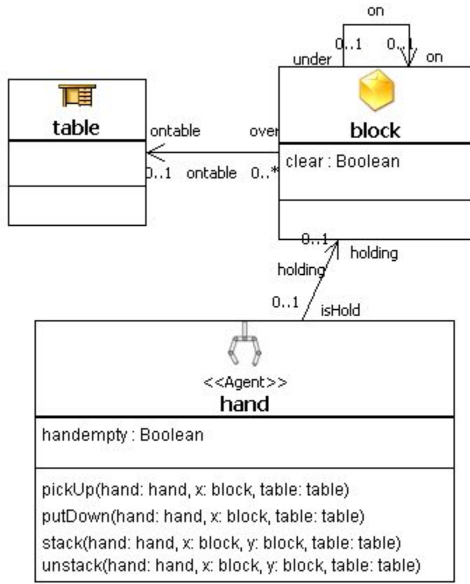

Figure 1: UML Class Diagram of a Blocks World Domain

extremity of an association); *mutability* (permits to get and maintain the mutability information of an association); *associationTypeEx* (means to get and maintain the type of an association extremity, i.e. simple, aggregation and composition associations); *typeStereotype* (gets and maintain the stereotype of a class); *characteristic* (allows to get a diverse set of the associations); and *classFund* (defines a fundamental class for all classes used in the interpretation).

These signatures define classes that will be used in all interpretation, but it is also necessary the definition of some fixed objects, connected to the signatures, that will assist the interpretation. As known from UML.P, the multiplicity of an association extremity have lower and upper limits, so the objects *limLower* and *limUpper* typed as *limSet* is defined to be used as parameter and maintain the limits of the multiplicity for each extremity. The characteristic of the extremity type is also interpreted as objects, i.e. *assoc_Simple*, *assoc_Aggregation* and *assoc_Composition*.

There are three types of mutability used in the UML.P: Changeable, Static and Add Only. They are also objects typed as *mutability*, but interpreted as *mut_Changeable*, *mut_Static* and *mut_AddOnly*. The mutability information state how an association is affected during its life.

From now on, to exemplify the interpretation process the Figure 1 will be used. The figure shows a simple Class diagram of the classic Blocks World. It represents the characteristics of three types of objects, called classes, which this domain has: *table*, *block* and *hand*. The group of objects belonging to the *table* class have no attributes and may have relationship to *block* objects. The relationship is limited by the multiplicity, in the example the *block* objects may connect to none or one table, while table may connect to none or lots of blocks. The *block* objects have the *clear* attribute typed as boolean, beyond the relationship with *table* objects, it also may have relationship with others *block* objects and *hand* objects. They are also limited by the multiplicity, which limits the relationship with only none or one block and the same for the hand. The *hand* objects have the attribute *handempty* typed as boolean and also the actions that these objects may perform in a domain. The actions are: *pickUp*, *putDown*, *stack* and *unStack*. These actions declarations do not intend to define what the action will cause in a domain, but just show which objects may perform actions in a domain problem. Details about the construction of this and others diagrams can be found in (Vaquero *et al.* 2007).

From the UML.P Class diagram, the interpretation of a simple UML.P class to F-Logic is quite direct, since each class defines a class in F-Logic. All classes interpreted from UML.P to F-Logic will be descendent of the *classFund*, as stated for the interpretation. Interpreting the three classes from Figure 1 in UML.P to F-Logic: $table :: classFund$, $block :: classFund$ and $table :: classFund$. The stereotype of the classes is captured through the *typeStereotype* signature, so all classes signature will be added with the method *stereotype* that enables this information maintenance. From the Figure 1, results:

$$table[stereotype => typeStereotype; \\ stereotype *-> ster_null], \\ block[stereotype => typeStereotype; \\ stereotype *-> ster_null] \text{ and} \\ table[stereotype => typeStereotype; stereotype *-> \\ ster_agent].$$

As the stereotype does not have a pattern, its interpretation will be done adding the suffix *ster_* to the stereotype name, as can be seen in the interpretation above.

Attributes defined in classes are interpreted as methods to F-Logic, in the form:

$$class[attribute => class] \text{ or} \\ class[attribute =>> class].$$

Some attributes also have multiplicity characteristics, so it will be captured adding up the signature:

$$class[attribute@limSet => integer].$$

It is important to recall that there exist two object typed as *limSet*: *limLower* and *limUpper*, getting the bounds of the attributes. It is also limited with mutability characteristic, captured by:

$$class[attribute@characteristic => mutability].$$

The following signatures are an example part of the interpretation from the Figure 1:

$$block[clear => boolean; \\ clear@limSet => integer; \\ clear@limUpper *-> 1; \\ clear@limLower *-> 1; \\ clear@characteristic => mutability; \\ clear@atrib_charac-> mut_changeable]$$

$$hand[handempty => boolean; \\ handempty@limSet => integer; \\ handempty@limUpper *-> 1; \\ handempty@limLower *-> 1; \\ handempty@characteristic => mutability; \\ handempty@atrib_charac-> mut_changeable].$$

The actions interpretation will not be considered in this paper because it will not be used to get state constraints (process described in the next section), however there already exist a process for their interpretation and its description will be left for future works.

The most complex interpretation follows from the association. In fact, concerning the UML.P representation, there are lots of information that are not directly interpreted to F-Logic. An association just has its name and mutability as general characteristic, the extremity is particular to a specific connection. For the general characteristics, the interpretation will follow the form:

$$class[associationName_oppositeExtremity_className \\ => mutability;].$$

Association is limited of two extremities in the UML.P, each extremity have a name, multiplicity, type and extremity type. Except by the extremity type, the extremity of an association has most of the characteristics that an attribute have, so for convenience the interpretation will follows the same process used for the attributes except by the mutability. From the example showed in Figure 1 the following interpretation of the associations if formed:

$$
\begin{gathered}
table[ontable_block => mutability; \\
ontable_block *-> mut_changeable; \\
ontable_block_over@characteristic => \\
associationTypeEx; \\
ontable_block_over@assoc_TypeEx *-> \\
assoc_Simple; \\
ontable_block_over@limSet => integer; \\
ontable_block_over@limUpper *-> *; \\
ontable_block_over@limLower *-> 0] \\
block[ontable_table => mutability; \\
ontable_table *-> mut_changeable; \\
ontable_table_onTable@characteristic => \\
associationTypeEx; \\
ontable_table_onTable@assoc_TypeEx *-> \\
assoc_Simple; \\
ontable_table_onTable => table; \\
ontable_table_onTable@limSet => integer; \\
ontable_table_onTable@limUpper *-> 1; \\
ontable_table_onTable@limLower *-> 0; \\
on_block => mutability; \\
on_block *-> mut_changeable; \\
on_block_on@characteristic => associationTypeEx; \\
on_block_on@assoc_TypeEx *-> assoc_Simple; \\
on_block_on => block; \\
on_block_on@limSet => integer; \\
on_block_on@limUpper *-> 1; \\
on_block_on@limLower *-> 0; \\
on_block_under@characteristic => \\
associationTypeEx; \\
on_block_under@assoc_TypeEx *-> assoc_Simple; \\
on_block_under@limSet => integer; \\
on_block_under@limUpper *-> 1; \\
on_block_under@limLower *-> 0; \\
holding_hand => mutability; \\
holding_hand *-> mut_changeable; \\
holding_hand_isHold@characteristic => \\
associationTypeEx; \\
holding_hand_isHold@assoc_TypeEx *-> \\
assoc_Simple; \\
holding_hand_isHold@limSet => integer; \\
holding_hand_isHold@limUpper *-> 1; \\
holding_hand_isHold@limLower *-> 0]; \\
hand[holding_block => mutability; \\
holding_block *-> mut_changeable; \\
holding_block_holding@characteristic => \\
associationTypeEx; \\
holding_block_holding@assoc_TypeEx *-> \\
assoc_Simple; \\
holding_block_holding => block; \\
holding_block_holding@limSet => integer; \\
holding_block_holding@limUpper *-> 1; \\
holding_block_holding@limLower *-> 0].
\end{gathered}
$$

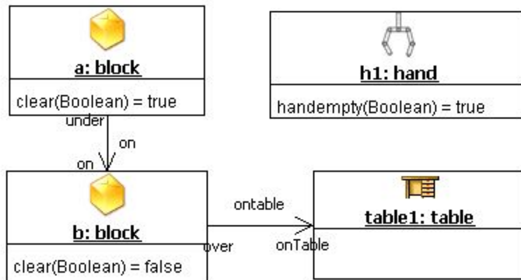

Figure 2: UML Object Diagram of a Block World Domain

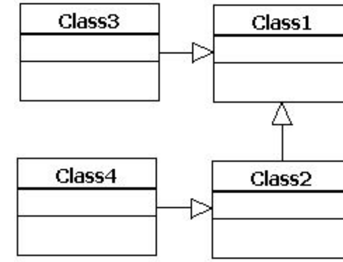

Figure 3: UML Object Diagram Hierarchy Sample

Beyond the association, the class hierarchy is got by the generalization relationship. The interpretation is get directly from the Class diagrams and follows the form: $classA :: classB$. From Figure 3, which states that *class4* is subclass of *class2*, *class2* is subclass of *class1* and *class3* is subclass of *class1*, the following interpretation is got:

$$class1 :: classFund, class2 :: class1 , class3 :: class1 \\ \text{and} , class4 :: class2.$$

The F-Logic signatures get from the Class diagram assist the interpretation of the objects from Object diagram. All objects interpretation will inherit their characteristics from the F-Logic signatures, i.e. $obj : class$. The attributes defined in the Object diagrams will be interpreted following a signature method in F-Logic got from the Class diagram in-

terpretation. The following interpretation was get from the example showed in Figure 2, which is an initial state of a simple problem of Block World Domain:

$$a : block, b : block, h1 : hand, table1 : table, \\ a[clear- > true; on_block_on- > b], \\ b[clear- > false; ontable_table_onTable- > table1] \\ \text{and } h1[handempty- > true]$$

Due to the characteristic of the class properties given by the F-Logic and compatible with the UML.P, the objects *a*, *b*, *h1* and *table1* will have all the methods that have been defined as inheritable (*->), as also only the methods that have signatures can be used in the domain. The property that limits the use of methods defined in the signatures is called *well typed programs*(Kifer, Lausen & Wu 1995). This property is an important characteristic that guarantee that all data expressions respect a signature defined in F-Logic and since all signatures is obtained from the Class diagram there must have signatures defined for all objects in a domain problem. This concept is important because it will support the reasoning definitions to get state constraints from the Class and Object diagrams.

State Constraints

Until this point we only have the translation of the Class and Object diagrams to F-Logic. This translation brings the advantage of a formal representation to the UML.P allowing the process of reasoning about the model. In this section it will be shown the extraction of state constraints from the types, associations, multiplicities and attribute mutability.

Type State Constraint

The types are obtained directly from the UML.P Class diagram interpreted to F-Logic, and also from the types fixed for the interpretation. The classes' hierarchy are depicted explicitly from the Class diagram and let clear the types and subtypes of each member of them. However, to validate this assertion, the definition of two functions that will assist the constraints extraction is necessary.

Definition 1: The set of all superclasses of a class will be given by the function $\upuparrows (class)$.

This function will return the set of all superclasses that a class descends, whereas the function given in the definition below will return the set of all subclasses that a class defines.

Definition 2: The set of all subclasses of a given class will be obtained by the function $\downdownarrows (class)$.

To make clear the role of these definitions, consider the Figure 3. The result of the function $\downdownarrows (Class1)$ will be $Class2 \wedge Class3 \wedge Class4$, whereas the result for the function $\upuparrows (Class2)$ will be $Class1$.

In addition, it is necessary the definition of the minimum requirement for a model in UML.P to be interpretable to F-Logic. It must be clear that this interpretation is restricted to get state constraints from the Class and Object diagrams, so these diagrams set is the minimum requirement to get an UML.P model interpretable to F-Logic.

Definition 3: A UML.P model is liable to be interpreted into F-Logic if it is composed by a Class diagram *C* and one or more Object diagrams $\overrightarrow{O}$. All the resources used in $\overrightarrow{O}$ must be defined in *C*, i.e. $\overrightarrow{O} \in_c C$. All models that respect this condition will be called of $\widehat{M}$.

Although the Definition 3 restricts the interpretation of UML.P models for any model that has a Class diagram and Object diagrams, it is not sufficient to permit a correct analysis. It is also necessary that the domain model has the class models of all the objects modeled in the Object diagram.

Definition 4: A domain model $\widehat{M}$ will belong to the set of models possible to interpret to F-Logic, $\widehat{M} \in F$, if and only if its interpretation results in a *well typed* F-Logic *program*.

So, the Definition 4 guarantee that only domain models that have all the static characteristics well defined in the Class diagram will be interpretable to F-Logic in the context of state constraint analysis.

Since the interpretation requires that all objects have their classes defined, we can assert that a UML.P domain that respect the Definition 4 is strongly typed and this assertion is straightly obtained from the F-Logic interpretation.

It is easy to get the classes hierarchy and others characteristics directly from the Class diagram, however its semi-formal semantic may let margin for great number of interpretations. From the example of the figure 3 it is possible to get that *Class1* is superclass of *Class2*, *Class3*, and *Class4* as also the *Class2* is superclass of *Class4*. To denote this assertion symbolically the symbol $\lessdot$ will be used, i.e. from the example of the figure 3 follows $Class1 \lessdot Class2 \wedge Class3 \wedge Class4$ and $Class2 \lessdot Class4$. The following theorem proofs that the hierarchy structure is also possible to get from the F-Logic interpretation.

Theorem 1: For a model $\widehat{M} \in F$, that *C* is the class set defined in the Class diagram in UML.P, so the assertion $\forall Cl \in C, Cl \lessdot \downdownarrows (Cl)$ is true. The $\lessdot$ symbol denotes that a class is superclass of the class returned by the function $\downdownarrows$.
Proof: The proof follows from the F-Logic's semantics definitions and the application of the Transitivity inference rule, the details of this proof will be left for a future publication.

The following theorem allow us know all the classes that an object belongs to. It makes use of the symbol $\lhd$ denoting that an object belongs to a class, i.e. $A \lhd block$ from figures 1 and 2 denotes that the object A belongs to the class $block$. It is also used the symbol $\rightarrow$ denotes a simple implication.

Theorem 2: For a model $\widehat{M} \in F$, that *C* is the class set defined in the Class diagram in UML.P and *O* the object set defined in the Object diagram, so the assertion $\forall Obj \in O$ *and* $Cl \in C, Obj \lhd Cl \rightarrow Obj \lhd \upuparrows (Cl)$ is true.
Proof: From the Transitivity and Subclass Inclusion inference rules and F-Logic semantics it is possible to proof the validity of this theorem, that will not be shown completely due to the reduced space available.

The concept presented in the theorem 2 is not defined clearly in the UML definitions, so the interpretation to F-Logic gives a strict concept to generalization so we can as-

sure what classes an object belongs, stating type constraints for all objects. From the Blocks World example, in figures 1 and 2, it is possible to get that objects *a* and *b* are typed just as *block*, *h1* is typed just as *hand* and *table1* is typed just as *table*. For a more complex Class diagram structure it will be possible to get the complete set of classes that an object belongs to, generating state constraints as $\forall X.Class4(X) \rightarrow Class2(X)$ and $\forall Y.Class2(X) \rightarrow Class(1)$ from the example of figure 3.

The Theorems 3 and 4 shows the propagation of superclass attributes and associations, allowing to get state constraints from the types of the F-Logic signatures.

Theorem 3: For a model $\widehat{M} \in$ *F*, that *C* is the class set defined in the Class diagram in UML.P, so the assertion $\forall Cl \in C, Atr(Cl) \supset Atr(\Uparrow (Cl))$ is true. The *Atr* is a function that gets the attributes and actions of a class and $\supset$ denotes "contain". So the assertion can be read as "For all classes in a Class diagram, a class will contain all attributes and actions of yours superclasses".
Proof: Since the inference rule of Inheritance state that any sub class inherits all properties from its superclass, the proof runs directly from the inference rule.

Since the interpretation makes the attributes and associations work in a similar mode, the theorem can be expanded to the associations of the classes.

Theorem 4: For a model $\widehat{M} \in$ *F*, that *C* is the class set defined in the Class diagram in UML.P, so the assertion $\forall Cl \in C, Assoc(Cl) \supset Assoc(\Uparrow (Cl))$ is true. The *Assoc* is a function that gets the associations of a class.
Proof: The proof runs directly from the Inheritance inference rule of the F-Logic.

Since the theorems 3 and 4 show that the simple inheritance is applicable to a UML.P domain interpreted to F-logic, it is possible to get state constraints based on the types of F-Logic signatures. From the figure 1 it is possible to get state constraint as $\forall X, Y.holding(X,Y) \wedge hand(X) \rightarrow block(X)$ given in first order logic. In a more general constraint form given in F-Logic:
$\forall X, Y.X[Meth \rightsquigarrow Y] \wedge Y : Class' \rightarrow X : Class$"
and $\forall X, Y.X : Class'[Meth \rightsquigarrow Y] \rightarrow X : Class$", which means that for all X and Y, considering that X has an attribute Meth with a value Y, and Y is a member of Class', implies that X is a member of Class", i.e., X and Y must belong to different classes. The symbol $\rightsquigarrow$ denotes $->$ and $->>$. These are possible since a signature is defined for each class of a domain and the attributes and associations is propagated to all sub classes. The theorem 5 below, gets this concept.

Theorem 5: For a model $\widehat{M} \in$ *F*, that *C* is the class set defined in the Class diagram in UML.P and *O* is the object set defined in the Object diagram in UML.P. The set of classes and objects interpreted to F-Logic will be called *C'* and *O'*, so the assertion $\forall XandY \in O', ClandCl' \in C', X : Cl \wedge Cl[atr \Rightarrow Cl'] \wedge X[atr \rightsquigarrow Y] \rightarrow Y : Cl'$ is true. The symbols $\Rightarrow$ denotes $=>$ and $=>>$, $\rightsquigarrow$ denotes $->$ and $->>$, and *atr* denotes an attribute or association interpreted to F-Logic. **Proof:** Its proof follows from the definition of signatures given in the F-Logic.

Applying the above theorem in the example of the figures 1 and 2, state constraint of the type $a[on_block_on-> b] \wedge b : block \rightarrow a : block$ and $b : block[ontable_table_onTable-> table1] \rightarrow table1 : table$ are obtained.

Multiplicity State Constraint

It is clearly noticeable that the association multiplicity is a state constraint modeled explicitly in a domain. The simplest one can be got from the F-Logic data scalar expressions generated when the multiplicity of an association is has its higher value equal to one, e.g. from the figure 1 a block can be at maximum in one table, so the higher multiplicity for the *onTable* association is 1.

Theorem 6: For a model $\widehat{M} \in$ *F*, that *C* is the class set defined in the Class diagram in UML.P and *O* is the object set defined in the Object diagram of UML.P. The set of classes and objects interpreted to F-Logic will be called *C'* and *O'*, lets X, Y and Z be variables and M and P_k be attributes components of an object. So the assertion $\forall X, YandZ \in O', X[M@P_1, ..., P_k-> Y] \wedge X[M@P_1, ..., P_k-> Z] \rightarrow Y = Z$ is true.
Proof: The proof runs directly from the F-Logic Scalarity Inference rule.

Since the scalar expression state that only one object can be hold by a method, it is quite simple to get this constraint. From the example of Blocks World domain given by figures 1 and 2, it is possible to get state constraint of the form:$\forall X \in O', a[on_block_on-> b] \wedge a[on_block_on-> X] \rightarrow X = b$. However, the Class diagram also states multiplicity constraints that result in data expression with upper and under limit different of 1, resulting in set valued data expressions. Different of the scalar data expressions, the set valued data expressions do not allow constraints like the Theorem 6. But the multiplicity limit allows the checking of the model as defined below.

Definition 5: For a model $\widehat{M} \in$ *F*, that *O* is the object set defined in the Object diagram interpreted to F-Logic. For all objects *Obj* that have under and upper limits for an attribute *Atr*, $Obj[Atr@limLower-> LowerValue]$ and $Obj[Atr@limUpper-> UpperValue]$, we will call a model well limited the models that the restriction $LowerValue \leq AssociationNo(Obj[Atr \rightsquigarrow Obj') \leq UpperValue$. And $\forall Obj' LowerValue \leq AssociationSum(Obj'[complementaryAssoc(Atr) \rightsquigarrow Obj]) \leq UpperValue,$, for the cases where $Obj[Atr \rightsquigarrow Obj']$ is not defined. The function *AssociationNo* gives the number of active elements of an attribute in an object, *AssociationSum* is a function that gives the total amount of active attributes complementary and have its value equal to *Obj* and the symbol $\rightsquigarrow$ denotes $->$ and $->>$.

This definition allows the internal checking for the limits imposed by the Class diagram models in the Object diagram and also in all steps in a plan generation process, since this limits cannot be disrespected.

Mutability Constraint

The association mutability is an indirect constraint. Its meaning does not cause a direct state constraint but limits how an action will affect a relationship.

Definition 6: For a model $\widehat{M} \in F$, that O is the object set defined in the Object diagram in UML.P interpreted to F-Logic. For all Objects $Obj \in O$ that have the static mutability restriction for an attribute *Atr*, i.e. $Obj[Atr@mutability- \ > \ mut_Static]$ implies that $Obj[Atr \rightsquigarrow Val]$
must not be changed during the life cycle of an object.

The static mutability characteristic can be used to check all the actions modeled in a domain if they are affecting a association restricted by a mutability constraint. As its restriction limits the association change, no action may have it as a parameter in the post condition of an action. The same concept can be applied to the *add only* mutability.

Definition 7: For a model $\widehat{M} \in F$, that O is the object set defined in the Object diagram in UML.P interpreted to F-Logic. For all Objects $Obj \in O$ that have the *add only* mutability restriction for an attribute *Atr*, i.e. $Obj[Atr@mutability- > mut_AddOnly]$ implies that $Obj[Atr \rightsquigarrow Val]$ must not be reduced during the life cycle of an object.

Since it is limited to be add only, it is also possible to make a simple check in all action for this restriction.

Discussion & Conclusion

The UML.P Class and Object diagrams, available in itSIMPLE tool, are source of a diverse set of information that enriches a planning model description. The proximity of concepts allow a UML.P model to be interpreted in F-Logic. The interpretation has kept all the information of these diagrams in order to exploit the information available in a model.

It is clear that the DISCOPLAN and TIM still get more state constraints from the actions and the initial state than the process used in this paper to get state constraint from the Class and Object diagrams. But the main objective of getting state constraint from the UML.P diagrams is to allow the cross validation of the information. This first step opens a new horizon directed to the static validation of a model by using TIM and DISCOPLAN similar tools to extract constraints from State machine Diagram and the process described in this paper to extract constraint from Class and Object Diagrams. The constraints extracted from different diagrams can be used to detect conflicts in a model.

It is also possible to use the state constraints obtained from the UML.P Class diagram to improve the planner performance, this approach had already been applied to the state constraints get from the TIM and DISCOPLAN tools showing important improvements to planners.

Another important result from this paper is the possibility to state formal correspondence between UML.P models and PDDL. It will let a soundness method to translate PDDL to UML.P and vice-versa.

In fact, it is still needed an interpretation of the State Machine diagram for a formal logic, since it has weak and informal semantics. For this reason the Transaction F-Logic (Kifer 1995) is being studied for this purpose, as it allows the capture of the domain dynamics and a possible translation to PDDL.

From now on, the results of this paper can be considered in the ItSIMPLE tool to assist the development of models in AI planning. The main goal of itSIMPLE project is to provide a bridge over the gap between real applications and planning systems by letting available a plethora of techniques and tools to support the entire model design life cycle. However much effort still have to be done to make it possible and this paper is just a little step toward this goal.

References

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

D'Souza, D. F., and Wills, A. C. 1995. Logical Foundations of Object Oriented and Frame Based Languages. *Journal of the ACM* 42(4):741–843.

Edelkamp, S., and Mehler, T. 2005. Knowledge acquisition and knowledge engineering in the modplan workbench. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning, Monterey, California, USA*.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence* 9:367–421.

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. 1998. Inferring State Constraints as Control Knowledge for Domain-Independent Planning. In *Proceedings of the Workshop on Planning as Combinatorial Search, AIPS-1998*.

Kifer, M. 1995. Deductive and Object Data Languages: A quest for Integration. *Lecture Notes in Computer Science* 1013:187–212.

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

OMG. 2001. *OMG Unified Modeling Language Specification, m Version 1.4*.

Simpson, R.M. 2005. Gipo graphical interface for planning with objects. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning, Monterey, Califormia, USA*.

Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itSIMPLE$_{2.0}$: An Integrated Tool for Designing Planning Domains In *Proceedings of the 17th International Conference on Planning and Scheduling, AAAI Press*.

Vaquero, T. S.; Tonidandel, F.; and Silva, J. R. 2005. The itsimple tool for modelling planning domains. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning, Monterey, Califormia, USA*.