OBTAINING STATES INVARIANTS FROM CLASS DIAGRAM IN UML.P

Marcel Lira Gomes*, Marcelo Udo*, Tiago Stegun Vaquero†, José Reinaldo Silva†, Flávio Tonidandel*

* *Centro Universitário da FEI*
IAAA - Artificial Intelligence Applied in Automation Lab.
São Bernardo do Campo, Brazil

†*Escola Politécnica - Universidade de São Paulo*
Design Lab. - PMR - Mechatronic and Mechanical Systems Department
São Paulo, Brazil

Emails: mmgomes@fei.edu.br, m_udo@fei.edu.br, tiago.vaquero@poli.usp.br, reinaldo@usp.br, flaviot@fei.edu.br

Abstract— States invariants can be obtained from implicit knowledge found in planning domains. through them, planners' search process and their performance have been improved. There are tools as TIM, DISCOPLAN and Rintanen that can obtain states invariants from initial state and correct operators of domains described in PDDL Language. UML.P language also allows a general domain description in the Class Diagram in addition to the initial state and operators given in the Object Diagrams and State Machine Diagrams. Different from the known tools, we will show that through an analysis of the Class Diagram we also get states invariants that further can be used to complement the ones obtained through the analysis of a domain initial state and operators or even use it to get inconsistencies through a cross validation of the states invariants.

Keywords— UML.P, UML.P Semantics, State Invariants, Domain Validation.

Resumo— Os estados invariáveis podem ser obtidos dos conhecimentos implícitos encontrados em descrições de domínios de planejamento. Através da utilização dos estados inveriáveis, o processo de busca dos planejadores e sua performance tem sido aperfeiçoados. Existem ferramentas tais como TIM, DISCOPLAN e Rintanen, que são capazes de obter os estados invariáveis do estado inicial e dos operadores corretos de domínios descritos em PDDL. A UML.P também permite uma descrição geral do domínio pelo Diagrama de Classes além do estado inicial e dos operadores dados pelos Diagramas de Objetos e os Diagramas de Estado de Máquina. Diferentemente das ferramentas conhecidas, nós iremos mostrar que através da análise do Diagrama de Classes nós também podemos obter estados invariáveis que futuramente podem ser utilizados para complementar os estados invariáveis obtidos através da análise do estado inicial e dos operadores de um domínio ou até serem utilizados para obter inconsistências através da análise cruzada dos estados invariáveis.

Keywords— UML.P, Semântica da UML.P, Estados Invariáveis, Validação de Domínios.

1 Introduction

At the beginning of the Artificial Intelligence (AI) planning research domain models used to be simple and, up to certain point, easily verified. The desire and necessity to develop domains close to real problems have let the domains description more complex and full of information. Following these necessities, domain modelling languages have grown resulting in complex languages with plenty of resources.

In spite of being the standard model language adopted by the AI Planning community, the PDDL (McDermott, 1998) language is not intuitive nor easy to be learned. Modeling a domain in PDDL is time consuming and difficult for who does not have familiarity with the language, and even for who already has, what may turn the language improper for a common market use. Several efforts have been done to ease the development of domain models, from which UML.P (Vaquero et al., 2006) has emerged as a promising language suitable for the AI planning community and possibly for market use. Moreover, UML.P is based on UML (Booch et al., 1998) that is widely known and flexible to describe a large variety of processes and domains. In collaboration with UML.P, the ItSimple (Vaquero et al., 2005) Knowledge Engineering play an important role helping users modeling domains intuitively in a friend environment.

Comparing UML.P with PDDL we note that the former has different resources not available in the latter. In addition to the resources available to describe initial and final states and operators, the UML.P has the Class Diagram that allows a general description of a domain. The Class Diagram gives different kinds of information, including implicit information, that are useful for the process of planning.

Implicit knowledge found in domain descriptions has been used to speed up planners, aiming the improvement of search process by feeding them with states invariants. States invariants play an important role during the search process of a plan, since they can help planners to avoid redundant analysis and search. Some preplanning tool as TIM (Long and Fox, 2000), DISCOPLAN (Gerevini and Schubert, 2000) and Rintanen (Rintanen, 2000) are already able to find correct states invariants from initial states and operators of correct PDDL domain descriptions.

Further than analyzing operators and initial state, this paper addresses the extraction of states invariants from UML.P analyzing the Class Diagram. Some of states invariants that we get from the Class Diagram are Typing Constraints, Simple Association Constraints, Symmetry Constraints, Multiplicity Constraints and XOR Constraints.

To show the states invariants extraction, we start this paper explaining briefly the syntax that has been developed to the UML.P. A syntax definition is necessary since the UML language have a semi-formal syntax which can lead to misinterpretation of domain descriptions. Next, we show how the states invariants can be extracted based on the Class Diagram. We finalize it discussing the advantages that the Class Diagram knowledge extraction will bring for the AI Planning community, concluding with future works.

2 UML.P

The Unified Modeling Language (UML) is a de facto industry standard visual language for modeling a wide variety of domain models. Its broad scope covers a large and diverse set of application domains (OMG, 2007). Although its wide embrace, there is a semantic problem that should be solved for a successful UML applicability in AI Planning. Its semantics is defined in informal written English that can lead to problems of: misinterpretation, analysis and design (Berardi et al., 2003). In order to solve this problem UML.P has been defined as a subset of UML with a more precise semantic description.

2.1 UML.P Characterization

UML.P does not intend redefine the UML language to create domain models for planning, however, we intend use its definition to support the AI Planning community with a new language for modeling domains. For this purpose the semantics of UML.P must be precisely defined by the light of planning, avoiding any problem of misinterpretation. An explanation of each diagram can be seen in Vaquero et al. (2005).

An AI planner is able to search a plan given a set of a domain model and domain problem. This set will be called as Planning System Domain in UML.P. So the Planning System Domain is compounded by a Planning Domain and a Domain Problem, $\Psi = \{P_D, D_P\}$.

A planning domain in UML.P is a set of Abstract Domain, Domain's Objects and all possible states of a domain ($P_D = \{A_D, Obj, S\}$), however, the possible states are not listed during the domain modeling. Domain's Objects denominates a set of all objects that exist in a domain model (Obj) and the Abstract Domain is the set that gives the characteristics of the Domain's Objects. It is formed by a Class Diagram, State Machine Diagrams and a set of all Domain Constraints ($A_D = \{C_D, SM_D, D_C\}$).

A Domain Problem is a set of Initial State, Final State and a set of Operators that can be applied to a domain ($D_P = \{S_{INI}, S_{FIN}, OP_P\}$). Giving the set of operators that can be applied to a domain, we can ease the definition of a problem. To a better comprehension of this feature, suppose a simple logistics planning domain with a *Truck* and an *Airplane* objects. The only two operators defined in this simple domain are *drive* and *fly* which belong to *Truck* and *Airplane* respectively. But, if for any reason, the *Airplane* could not fly, we should define a new domain model without the *Airplane* or without the *fly* operator. Allowing the definition of which operators we could use in a problem, we can ease the process of defining a problem in a domain.

For the matter of this article, the following subsection will show briefly only the definition of the Class Diagram in order to support the states invariants obtained from this diagram.

2.1.1 Class Diagrams

The Class Diagram describes the static properties of a system, it is compounded of a set of classes related by association and generalizations. In Figure 1, there are the primitive diagrammatic elements allowed in the Class Diagram of the UML.P.

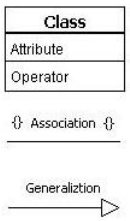

Figure 1: Diagrammatic Objects for Class Diagrams

For planning, a class defines a type for a set of objects that exist in a domain model. The classes may have attributes and operators prototypes that characterize them. Note that the operators are actions that objects of a type may perform in a domain. At this point, the operator prototype does not intend to define the behavior of an operator, but simply defines their owners.

An association is a relationship that links instances of classes. Note that in UML an association may relate more than one pair of classes, which is not allowed in UML.P. At each relationship's end there are the multiplicities, which inform how many objects can be associated through an association.

The generalization relationship defines a hierarchy for the classes related. Note that abstract classes are not allowed in UML.P since there is no function for them at this point.

Definition 1 (Class Diagram Set)
$C_D = \{T, At, O_P, G_R, A_R, \alpha, \beta, \delta, \phi\}$, *where:*

- T = *Finite set of Types. A type is any Class created in the Class Diagram* (T_{ABS}) *or any primitive type* (T_{PRIM}), *that are: Boolean, Integer and Float;*
- At = *Finite set of attributes, typed as* T_{PRIM};
- O_P = *Finite set of operators prototypes. The prototypes do not intend to define the behavior of operators, they are all defined in the State Machine Diagrams;*
- G_R = *Finite set of generalization relationship;*
- A_R = *Finite set of associations relationship;*
- $\alpha : At \rightarrow T_{ABS}$, *where* α *is a function that relate each attribute to an abstract type;*
- $\beta : O_P \rightarrow T_{ABS}$, *where* β *is a function that relate each operator prototype to an abstract type;*
- $(G_R, (T_{ABS} \times T_{ABS})) \in \delta$, *where* δ *is a set of generalization* (G_R) *related to pairs of abstract types that* $Dom(T_{ABS}) \neq Img(T_{ABS})$.
- $(A_R, (T_{ABS} \times T_{ABS})) \in \phi$, *where* ϕ *is the set of associations* (A_R) *related to pairs of abstract types.*

3 Knowledge Extraction

Through a simple analysis of the UML.P language we notice that there are explicit and implicit information in each diagram model. Among them the simplest one that can be easily noticed is the multiplicity information of an association relationship. Although the simplicity of this state invariant, we can use it and others information to check if an initial state and a final state are correct, based on the domain definition. Another important verification that we can do is the analysis of the domain's operators, checking if any of them could drive to an unexpected or incorrect state.

The advantage of these verifications is the possibility to check the internal consistence of a domain. An automatic analysis will improve the description time of a domain and also reduce the number of description errors. In the following subsections we will briefly show the states invariants that can be extracted from UML.P.

3.1 Typing

TIM and DISCOPLAN tools showed two distinct processes that infer the type structure from the initial state and operators. Although these tools showed a powerful process to get the hierarchy of a domain, we cannot check automatically if the initial state and operators were defined correctly, throwing the hierarchy checking process to a domain engineer.

UML.P is a typed language which all objects belongs to a class. Its structure and hierarchy are defined in the Class Diagram and it is given during the domain description process. The benefits of this characteristic are easily seen in a broad domain analysis, which we can check if all objects are correctly defined in the Initial and Final States. In the same way operators can be checked for the matter of its preconditions and post-conditions that must have correct values related to their attributes.

The set δ encloses the hierarchy defined in a Domain, through the Generalization Relationship it shows that there are supertypes and subtypes in the Domain. All types that do not have any mapping in δ set do not belong to any structure of classes. This characteristic allows the generation of states invariants related to the types in a domain.

Figure 2 shows an example of the Logistics Class Diagram. From the T_{ABS} set of this diagram we get the following types: *Package, Place, Location, Airport, Vehicle, Truck* and *Airplane*. For simplicity, we will not discuss about the *City* type in this paper. Following the characteristic described above we can generate the following states invariants.

$$\forall x.Truck(x) \rightarrow Vehicle(x)$$
$$\forall x.Airplane(x) \rightarrow Vehicle(x)$$
$$\forall x.Location(x) \rightarrow Place(x)$$
$$\forall x.Airport(x) \rightarrow Place(x)$$
$$\forall x.Place(x) \rightarrow \neg Vehicle(x)$$
$$\forall x.Vehicle(x) \rightarrow \neg Place(x)$$
$$\forall x.Package(x) \rightarrow \neg Place(x) \wedge \neg Vehicle$$
$$\forall x.Location(x) \rightarrow \neg Airport(x)$$
$$\forall x.Airport(x) \rightarrow \neg Location(x)$$
$$\forall x.Airplane(x) \rightarrow \neg Truck(x)$$
$$\forall x.Truck(x) \rightarrow \neg Airplane(x)$$

3.2 Associations

An association defines how two classes relate to each other, telling us which objects and how an objects can link to another one. In a Class Diagram Set the associations are defined in the ϕ set, where associations not related in this set are not allowed. In short, associations can be seen as predicates relating objects in AI Planning. As an example, in Figure 2 we have *isAt, isIn, at* and *parkedAt* associations that can be seen as predicates, so *isAt(x,y), isIn(x,y), at(x,y)*and *parkedAt(x,y)*which x and y are variables. From the set definition of ϕ we also get states invariants that we will describe in the following subsections.

3.2.1 Simple Association Constraints

As the invariants obtained by DISCOPLAN (Simple Implicative Constraints), through an analysis of the ϕ set, it is clear that comparable invariants can be obtained from the Class Diagram in a simple way. To get this kind of invariants from the Class Diagram we have to note that the navigability of the association plays an important

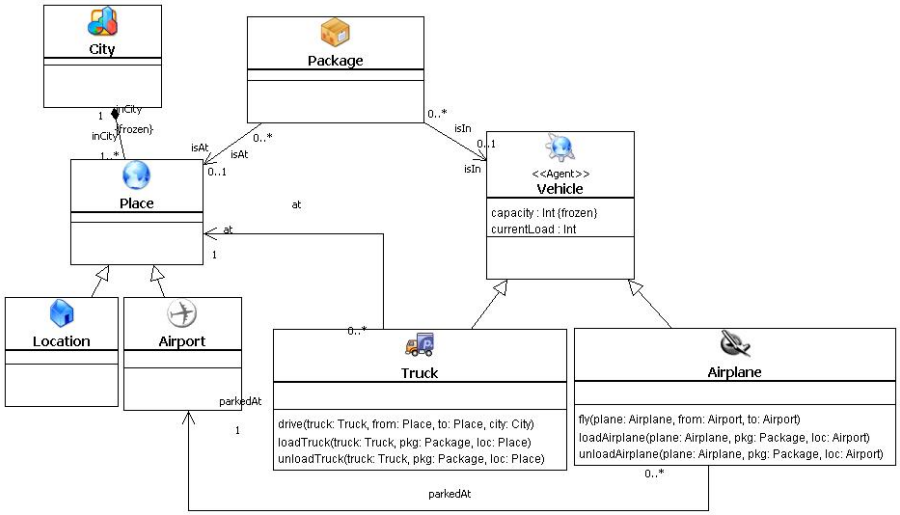

Figure 2: Logistics Class Diagram Example

role, which will dictates how the predicates will be constructed. The navigability is given by the direction of the arrow that represents an association, see Figure 1. From its construction the association origin and destiny will indicate the predicate order. So as an example lets get the *at* association from Figure 2, its origin is connected to the *Truck* class and destiny on the *Place* class, what will get the following state invariant $\forall x, y.at(x, y) \rightarrow Truck(x) \wedge Place(y)$. Below we list others invariants obtained from Figure 2.

$$\forall x, y.isAt(x, y) \rightarrow Package(x) \wedge Place(y)$$
$$\forall x, y.isIn(x, y) \rightarrow Package(x) \wedge Vehicle(y)$$
$$\forall x, y.parkedAt(x, y) \rightarrow Airplane(x) \wedge Airport(y)$$

There are some domain descriptions that already enclosures these features in predicate declaration.

3.2.2 Symmetry Constraints

As the Simple Association Constraints, through the ϕ set we also obtain Symmetry Constraints, which TIM and DISCOPLAN obtain them analyzing the domain operators. A Symmetry Constraint tells if a pair of objects can relate to each other by the same link, e.g. from the Figure 2 the association *isIn* tells that some packages can be in a vehicle but the contrary, a vehicle can be in some packages, is not true. From the association direction defined in the ϕ set we get the following constraints.

$$\forall x, y.isAt(x, y) \rightarrow \neg isAt(y, x)$$
$$\forall x, y.isIn(x, y) \rightarrow \neg isIn(y, x)$$
$$\forall x, y.at(x, y) \rightarrow \neg at(y, x)$$
$$\forall x, y.parkedAt(x, y) \rightarrow \neg parkedAt(y, x)$$

Although most of the associations have their origin and destiny, there are cases that associations do not have an origin and a destiny. In this case the association tells that independently of the objects origin or destiny the link is valid. As an example, suppose a domain that has an association called *beside* without any indication of its origin or destiny. The Symmetry Constraint obtained from this association is $\forall x, y.beside(x, y) \rightarrow beside(y, x)$.

This symmetry is very important to speed up any planner system. Symmetries can help the planners to avoid redundant analysis and search and this kind is not straightforward available in domain description.

3.2.3 Multiplicity Constraints

Multiplicity is an association property that informs how many links of the same kind can occur at same time between objects. Except for associations that have multiplicity unlimited (*0..** and ***) at both ends, there must exist at least one operator that change the domain state adding these links between objects and one operator that subtract them. This operators checking is automatic, which will prevent any careless at design time.

From associations having one of its ends with multiplicity equal to one, we get identity invariants. DISCOPLAN and TIM also obtain these invariants through the analysis of a domain description. Different from the process developed by them, through UML.P the extraction of this invariants are directly obtained from the ϕ set. As an example, lets get the *parkedAt* association from Figure 2, this association informs that none or plenty airplanes can park at an airport. So from this assertion we can conclude that $\forall x, y, z.ParkedAt(x, y) \wedge ParkedAt(x, z) \rightarrow y = z$. Note that this invariant is true just because one of the association end is one, otherwise we would not conclude this invariant. Below follows the others identity invariants obtained from Figure 2.

$$\forall x, y, z.isAt(x, y) \wedge isAt(x, z) \rightarrow y = z$$
$$\forall x, y, z.isIn(x, y) \wedge isIn(x, z) \rightarrow y = z$$
$$\forall x, y, z.At(x, y) \wedge At(x, z) \rightarrow y = z$$

3.2.4 XOR Constraints

All associations related to supertypes of a domain can be split to all its subtypes without losing its characteristics, e.g. from Figure 2 the association *isAt* relates the class *Package* and *Place* . Note that the class *Place* is the supertype of *Location* and *Airport* , so the *isAt* association can be split to the classes *Location* and *Airport* . It gives that some packages are at a Location and some packages are at an Airport.

The split associations that have any of their ends multiplicity equal to one revels an exclusive link between the associations. So from the Figure 2 the *isAt* link does not relate an object to others objects that have the same supertype. From this UML.P characteristic, we get the following constraints.

$$\forall \textit{ x: Package, y: Location, z: Airport. isAt(x,y) XOR isAt(x,z)}$$

$$\forall \textit{ x: Package, y: Truck, z: Airport. isIn(x,y) XOR isIn(x,z)}$$

DISCOPLAN also obtain XOR invariants from the analysis of the domain description. But at this point, we do not get all XOR constraints since we need to analyze the State Machine Diagrams to obtain more XOR constraints. In fact, DISCOPLAN extracts all XOR constraints from actions definitions in a PDDL model, while our knowledge extraction can obtain some of these XOR constraints directly from Class Diagram, without use actions definitions in State Machine Diagrams.

4 Discussion and Future Works

For a successful AI planning, a domain model should be consistent with actions description free of errors. Today domain model validation is a burden that has few tools to support the domain engineers. The consistency checking mainly depends on the designer to verify all domain description looking for possible errors.

TIM, DISCOPLAN and Rintanen tools intend to help domain engineers to find domain errors, however, they are not developed for this purpose. Although these tools are able to get states invariants, one or more sentences extracted may be incorrect if the domain description has any incorrect statement based on the domain logics. So, the domain engineer has to check all the sentences extracted by the tools and if any of them were logically incorrect, he will have to find the error, since the tools do not indicate its origin. Although these tools are not suitable to find errors in domain descriptions, the main lack is from the available languages that do not have suitable structure for redundant information, which would allow a cross information analysis to validate a domain.

A different approach can be applied to UML.P, mainly because its structure supports redundant information. This characteristic is easily seen in both main diagrams that root UML.P, the Class Diagram and the State Machine Diagram. As seen in this paper, through an analysis of the Class Diagram we can get states invariants as Typing Constraints, Simple Association Constraints, Symmetry Constraints, Multiplicity Constraints and XOR Constraints.

In the same way we also could get states invariants from the State Machine Diagram. Through a cross validation with the information extracted from the Class Diagram and State Machine Diagram we could find inconsistencies on the domain actions and structures. Implementing these methods would allow the ItSimple tool alerts the domain designer for potential errors and inconsistencies in a domain.

5 Conclusion

It is clear that UML.P is a powerful visual language with plenty of resources for AI Planning Community. But the lack of a formal semantics let the language fragile for use. Focused on this issue we are defining a formal semantic that will give a solid base for the use of the language in AI planning. Although the explanation given here, we still have to finish some proofs that are incomplete due to the development of the UML.P semantics. The development of the UML.P semantics is revealing other important characteristics that will improve the process of domain validation and knowledge acquisition for AI Planning community.

References

Berardi, D., Cal, A., Calvanese, D. and Giacomo, G. D. (2003). Reasoning on uml class diagrams. Available at http://citeseer.ist.psu.edu/article /berardi03reasoning.html.

Booch, G., Rumbaugh, J. and Jacobson, I. (1998). *The Unified Modeling Language User Guide*, Addison-Wesley.

Gerevini, A. and Schubert, L. (2000). Discovering state constraints in discoplan: Some new results, *AAAI Press*, pp. 761–767. Available at citeseer.comp.nus.edu.sg/gerevini00discovering .html.

Long, D. and Fox, M. (2000). Automatic synthesis and use of generic types in planning, *Artificial Intelligence Planning Systems*, pp. 196–205. Available at http://citeseer.ist.psu.edu/long00automatic .html.

McDermott, D. (1998). Pddl — the planning domain definition language, citeseer.ist.psu.edu/mcdermott97pddl.html.

OMG, O. M. G. (2007). Unified modeling language: Infrastructure. Available at http://www.omg.org/docs/formal/07-02-04.pdf.

Rintanen, J. (2000). An iterative algorithm for synthesizing invariants, *AAAI/IAAI*, pp. 806–811.

Vaquero, T. S., Tonidandel, F., de Barros, L. N. and Silva, J. R. (2006). On the use of uml.p for modeling a real application as a planning problem. Available at http://www.pmr.poli.usp.br/d-lab/artigos/ICAPS0602VaqueroT.pdf.

Vaquero, T. S., Tonidandel, F. and Silva, J. R. (2005). The itsimple tool for modeling planning domains, ICAPS 2005 Competition on Knowledge Engineering for Planning and Scheduling, Monterey, California, USA. Available at http://scom.hud.ac.uk/scomtlm/competition/papers/paper5.pdf.