

ICKEPS 2009

**Proceedings of the Third
International Competition on
Knowledge Engineering for
Planning and Scheduling**

**Thessaloniki, Greece
September 20th, 2009**

*Edited by
Roman Barták, Simone Fratini, and Lee McCluskey*

The 2009 ICAPS conference is sponsored by

University of Macedonia, Greece

Institute of Cognitive Sciences and Technologies, National Research Council
(ISTC-CNR), Italy

Information and Communication Technology Department,
National Research Council (ICT-CNR), Italy

Planning and Scheduling Team, ISTC-CNR, Italy

National Science Foundation (NSF), USA

Artificial Intelligence Journal, Elsevier

International Joint Conference on Artificial Intelligence (IJCAI)

European Coordinating Committee for Artificial Intelligence (ECCAI), Europe

Willow Garage, USA

David E. Smith, USA

NICTA, Australia

European Science Foundation, COST Action, Europe

Marathon Data Systems, Greece

Hellenic Artificial Intelligence Society (EETN), Greece

Prefecture of Thessaloniki, Greece

Hellenic Ministry of Culture, Greece

Klidarithmos Publishing, Greece

*Held in cooperation with the Association for the
Advancement of Artificial Intelligence*

ICKEPS 2009

Proceedings of the Third International Competition on Knowledge Engineering for Planning and Scheduling

Edited by

Roman Barták, Simone Fratini, and Lee McCluskey

Cover painting courtesy Yannis Stavrou.

Contents

Preface / iv

Roman Barták, Simone Fratini, and Lee McCluskey

Organizing Committee / v

Programme Committee / v

Papers

LOCM: A tool for acquiring planning domain models from action traces / 1

Stephen Cresswell

On Compiling Data Mining Tasks to PDDL / 8

Susana Fernández, Fernando Fernández, Alexis Sánchez, Tomás de la Rosa, Javier Ortiz, Daniel Borrajo, David Manzano

Modeling E-Learning Activities in Automated Planning / 18

Antonio Garrido, Eva Onaindia, Lluvia Morales, Luis Castillo, Susana Fernández, Daniel Borrajo

JABBAH: A Java Application Framework for the Translation Between Business Process Models and HTN / 28

Arturo González-Ferrer, Juan Fernández-Olivares, Luis Castillo

PORSCE II: Using Planning for Semantic Web Service Composition / 38

Ourania Hatzi, Georgios Meditskos, Dimitris Vrakas, Nick Bassiliades, Dimosthenis Anagnostopoulos, Ioannis Vlahavas

Augmenting Instructable Computing with Planning Technology / 46

Clayton T. Morrison, Daniel Bryce, Ian R. Fasel, Antons Rebguns

From Requirements and Analysis to PDDL in itSIMPLE3.0 / 54

Tiago Stegun Vaquero, José Reinaldo Silva, Marcelo Ferreira, Flavio Tonidandel, J. Christopher Beck

Preface

The International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS) has been running since 2005 as a bi-annual event promoting the development and importance of the use of knowledge engineering methods and techniques within Planning and Scheduling (P&S). While past events focused in general on knowledge engineering, this year the focus of ICKEPS was on a specific aspect: tools, translators and techniques that when input with a model described in an application-area specific language, output solver-ready domain models.

Many solvers have been developed within the P&S community which accept domain models encoded in a domain-independent language (such as those in the PDDL family). It is vital that these solvers are visible and accessible to potential users outside of the P&S community. Translators are one way to spread the P&S culture and technology and to broaden their catchments area. By allowing the use of languages designed by and for application domain experts, translators bring the P&S technology „into the field“, broadening the user community and giving the chance of demonstrating the applicability to real problems.

The participants of this third ICKEPS edition contributed with tools and translators from a wide range of application areas such as:

- Web services and Semantic web,
- Workflows and Business Process Modeling,
- Databases and Data Mining,
- UML,
- E-learning

A major goal of AI competitions has always been to accelerate development in some specified area. We hope to contribute with ICKEPS in pushing the research on the processes that deal with the acquisition, validation and maintenance of planning domain models, fostering also the connection with KE research in different areas. Investigating how modeling languages from different areas can be translated into each other furthers research into the principles of KE, and provides a valuable, reciprocal contribution.

– *Roman Barták, Simone Fratini, Lee McCluskey*
Workshop co-Chairs

Organizing Committee

Roman Barták, Charles University, Czech Republic

Simone Fratini, ISTC-CNR, Italy

Lee McCluskey, University of Huddersfield, UK

Programme Committee

Sara Bernardini, London Knowledge Lab, UK

Amedeo Cesta, ISTC-CNR, Italy

Stephen Cresswell, University of Huddersfield, UK

Stefan Edelkamp, University of Bremen, Germany

Susana Fernandez Arregui, Universidad Carlos III de Madrid, Spain

Jeremy Frank, NASA, USA

Antonio Garrido, Universitat Politecnica de Valencia, Spain

Robert Goldman, SIFT, USA

Arturo Gonzalez Ferrer, Universidad de Granada, Spain

Peter Jarvis, NASA, USA

Ugur Kuter, University of Maryland, USA

Clayton T. Morrison, The University of Arizona, USA

Julie Porteous, University of Teeside, UK

Tiago S. Vaquero, University of Sao Paulo, Brazil

Dimitris Vrakas, Aristotle University of Thessaloniki, Greece

LOCM: A tool for acquiring planning domain models from action traces

Stephen Cresswell

School of Computing and Engineering
The University of Huddersfield, Huddersfield HD1 3DH, UK

Abstract

This paper describes *LOCM*, a system which carries out the automated induction of action schema from an input language describing sets of example action sequences. The novelty of *LOCM* is that it can induce action schema without being provided with any information about predicates or initial, goal or intermediate state descriptions for the example action sequences. We envisage *LOCM* being applied in tasks for which example sequences can easily be collected, e.g. by logging workflows or moves in a computer game. In this paper we describe the implemented *LOCM* algorithm, and analyse its performance by its application to the induction of domain models for several domains. To evaluate the algorithm, we used random action sequences from existing models of domains, as well as solutions to past IPC problems.

NB: this paper is an extended version of a short ICAPS paper

Introduction

In this paper we describe a generic tool called (*LOCM*¹) which we believe can be used in a range of (rather than one specific) application areas. For application areas in which *LOCM* is effective, it inputs a sentence within an abstract language of observed instances and outputs a solver-ready PDDL domain model. The strength of *LOCM* lies in the simplicity of its input: its observed instances are descriptions of plans or plan fragments within the application area. *LOCM* relies on four assumptions:

- there are many observations for it to use;
- the observations are (sub)sequences of possible action applications within the domain;
- each action application is made up of an identifier, and names of objects that it affects;
- objects in the application can be grouped into sorts, where each object of each sort behaves in the same way as any other.

Working under the assumptions of Simpson et al's object-centric view of domain models (Simpson, Kitchin, and McCluskey 2007), we assume that a planning domain consists

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

¹Learning Object-Centred Models

of sets (called *sorts*) of object instances, where each object behaves in the same way as any other object in its sort. In particular, sorts have a defined set of states that their objects can occupy, and an object's state may change (called a state transition) as a result of action instance execution. *LOCM* works by assembling the transition behaviour of individual sorts, the co-ordinations between transitions of different sorts, and the relationships between objects of different sorts. It does so by exploiting the idea that actions change the state of objects, and that each time an action is executed, the preconditions and effects on an object are the same. Under these assumptions, *LOCM* can induce action schema without the need for background information such as specifications of initial/goal states, intermediate states, fluents or other partial domain information. All other current systems e.g. *Opmaker* (Richardson 2008), ARMS (Wu, Yang, and Jiang 2005), and the system of (Shahaf and Amir 2006) require some of this background knowledge as essential to their operation.

This first version of *LOCM* which we describe in this paper is aimed at applications which have little structure. In future work we aim to develop *LOCM* to be effective in applications which have more complex static structures such as maps, spatial rules or hierarchy. Currently, the kinds of applications domains we are experimenting with are in Games and Workflow.

The *LOCM* System

LOCM Inputs and Outputs

The input to *LOCM* are a set of sound sequences of action instances. An outline, abstract specification of the input language to *LOCM* is as follows:

```
<SequenceList> ::=  
    { "(" <SequenceId> "," <Sequence> ")" }  
<SequenceId> ::= <Id>  
<Sequence> ::= <ActionInstance>+  
<ActionInstance> ::=  
    <ActionName> "(" <Obj> { "," <Obj> } ")" ";"  
<ActionName> ::= <Id>  
<Obj> ::= <Id>
```

Using the well known *tyre-world* as an example, the following is a sequence containing four action instances, where an action is a name followed by a sequence of affected objects:

```
open(c1); fetch_jack(j,c1); fetch_wrench(wr1,c1); close(c1);
```

These sequences, which are akin to *workflow event logs*, may be observed from an existing process or supplied by a trainer. In the empirical evaluation below, we have tested the approach using example sequences from existing solvers and from a random walk generator. The output of *LOCM* (given sufficient examples) is a domain model consisting of sorts, object behaviour defined by state machines, predicates defining associations between sorts, and action schema in solver-ready form.

The *LOCM* Method

Phase 1: Extraction of state machines In our approach, we assume that an object of any given sort occupies one of a fixed set of states. Initially, we assume an object’s state can be defined without reference to the associations it has with any other specific objects. *LOCM* starts by first collecting the set of all transitions occurring in the example sequences. A transition is defined by a combination of action name and action argument position. For example, an action *fetch_wrench(wr1,ctrnr)* gives rise to two transitions: *fetch_wrench.1*, and *fetch_wrench.2*. Each transition describes the state change of objects of a single sort in isolation. For every transition occurring in the example data, a separate *start* and *end* state are generated.

The trajectory of each object is then tracked through each training sequence. For each pair of transitions T_1, T_2 , which are consecutive for an object Ob , we assume that $T_1.end = T_2.start$.

Using a training set from the tyre world, suppose some object *c1* goes through a sequence of transitions given in the example used above:

```
open(c1); fetch_jack(j,c1); fetch_wrench(wr1,c1); close(c1);
```

Let us assign state names to the input and output states of transitions affecting *c1*:

$$\begin{array}{lcl} S_1 & \implies \text{open.1} \implies & S_2 \\ S_3 & \implies \text{close.1} \implies & S_4 \\ S_5 & \implies \text{fetch_jack.2} \implies & S_6 \\ S_7 & \implies \text{fetch_wrench.2} \implies & S_8 \end{array}$$

Using the example action sequence, and the constraint on consecutive pairs of transitions, we can then deduce that $S_2 = S_5, S_6 = S_7, S_8 = S_3$.

Suppose our example set contains another action sequence:

```
open(c2); fetch_wrench(wr1,c2); fetch_jack(j,c2); close(c2);
```

We deduce that $S_2 = S_7, S_8 = S_5, S_6 = S_3$, and hence $S_2, S_3, S_5, S_6, S_7, S_8$ all refer to the same state. If additionally we have the sequence:

```
close(c3); open(c3);
```

We deduce that $S_4 = S_1$, hence we have tied together individual states to partially construct a state machine for containers (Fig. 1). A more formal description of the algorithm follows ²:

²Whereas our system is designed to use multiple training sequences, for simplicity the presentation here uses only a single sequence.

```
procedure LOCM ( Input action sequence Seq)
For each combination of action name A and
argument pos P for actions occurring in Seq
Create transition A.P, comprising
new state identifiers A.P.start and A.P.end
Add A.P to the transition set TS
Collect the set of objects Obs in Seq
For each object Ob occurring in Obs
For each pair of transitions  $T_1, T_2$ 
consecutive for Ob in Seq
Equate states  $T_1.end$  and  $T_2.start$ 
end
end
Return TS, transition set
OS, set of object states remaining distinct
```

At the end of phase 1, *LOCM* has derived a set of state machines, each of which can be identified with a sort.

Phase 2: Identification of state parameters Each state machine describes the behaviour of a single object in isolation, without consideration of its association with other objects, e.g. it can distinguish a state of a wrench corresponding to being in *some container*, but does not make provision to describe *which* container it is in.

In the object-centred representation, the dynamic associations between objects are recorded by *state parameters* embedded in each state. Phase 2 of the algorithm identifies parameters of each state by analysing patterns of object references in the original action steps corresponding to the transitions. For example, consider the state *wrench_state0* for the *wrench* sort (Fig. 2). Considering the actions for *putaway_wrench(wrench,container)*, and *fetch_wrench(wrench,container)*. For a given wrench, consecutive transitions *putaway_wrench, fetch_wrench*, in any example action sequence, always have the same value as their *container* parameter. From this observation, we can induce that the state *wrench_state0* has a state variable representing *container*. The same observation does not hold true for *wrench_state1*. We can observe instances in the training data where the wrench is fetched from one container, and put away in a different container.

This second phase of the algorithm performs inductive learning such that the hypotheses can be refuted by the examples, but never definitely confirmed. This phase generally requires a larger amount of training data to converge than Phase 1 above. Phase 2 is processed in three steps, shown below in the algorithmic description. The first two steps generate and test the hypothesised correlations in ac-

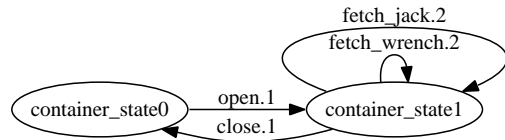


Figure 1: An incomplete state machine for containers in tyre-world

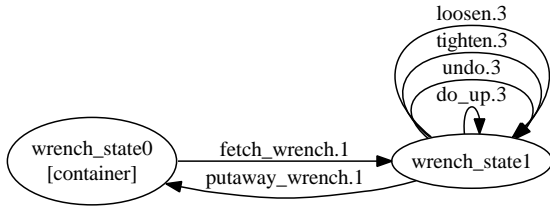


Figure 2: Parameterised states of wrench.

tion arguments, which indicate the need for state parameters. The third step generates the set of induced state parameters.

procedure **LOCM-II** (Input action sequence *Seq*,
Transition set *TS*, Object set *Obs*)
Object state set *OS*)

Form hypotheses from state machine

For each pair $A_1.P_1$ and $A_2.P_2$ in *TS*
such that $A_1.P_1.end = S = A_2.P_2.start$
For each pair $A_1.P'_1$ and $A_2.P'_2$ from *TS* and *S* in *OS*
with $A_1.P'_1.sort = A_2.P'_2.sort$
and $P_1 \neq P'_1, P_2 \neq P'_2$
(i.e. a pair of the other arguments
of actions A_1 and A_2 sharing a common sort)
Store in hypothesis set *HS* the hypothesis
that when any object *ob* undergoes sequentially
the transitions $A_1.P_1$ then $A_2.P_2$,
there is a single object ob' ,
which goes through both of the corresponding
transitions $A_1.P'_1$ and $A_2.P'_2$
(This supports the proposition that state *S*
has a state parameter which can record
the association of *ob* with ob')

end
end

Test hypotheses against example sequence

For each object *Ob* occurring in *Obs*
For each pair of transitions $A_1.P_1$ and $A_2.P_2$
consecutive for *Ob* in *Seq*
Remove from hypothesis set *HS* any hypothesis
which is inconsistent with example action pair

end
end

Generate and reduce set of state parameters

For every hypothesis remaining in *HS*
create the state parameter supported by the hypothesis
Merge state parameters on the basis that
a transition occurring in more than one transition pair
is associated with the same state parameter in each occurrence
end
return: state parameters and correlations with action arguments

Phase 3: Formation of action schema Extraction of an action schema is performed by extracting the transitions corresponding to its parameters, similar to automated action construction in the OLHE process in (Simpson, Kitchin, and McCluskey 2007). One predicate is created to represent each object state. The output of Phase 2 provides correlations between the action parameters and state parameters

occurring in the start/end states of transitions. For example, the generated *putaway_wrench* action schema in PDDL is:

```
(:action putaway_wrench
:parameters (?wrench1 - wrench ?container2 - container)
:precondition (and (wrench_state1 ?wrench1)
                   (container_state1 ?container2))
:effect (and (wrench_state0 ?wrench1 ?container2)
             (not (wrench_state1 ?wrench1))))
```

The generated predicates *wrench_state0*, *wrench_state1*, *container_state1* can be understood as *in_container*, *have_wrench* and *open* respectively. The generated schema can be used directly in a planner. It would also be simple to extract initial and final states from example sequences, but this is of limited utility given that solution plans already exist for those tasks.

Evaluation of LOCM

LOCM has been implemented in Prolog incorporating the algorithm detailed above. In this paper we attempt to analyse and evaluate it by its application to the acquisition of existing domain models. We have used example plans from two sources:

- Existing domains built using GIPO III. In this case, we have created sets of example action sequences by random walk.
- Domains which were used in IPC planning competitions. In this case, the example traces come from solution plans in the publicly released competition solutions.

We have used *LOCM* to create state machines, object associations and action schema for 4 domains. Evaluation of these results is ongoing, but initial results show that state machines can be deduced from a reasonably small number of plan examples (30-200 steps), whereas inducing the state parameters requires much larger training sets (typically > 1000 steps).

Tyre-world (GIPO version). A correct state machine is derived, corresponding closely to the domain as constructed in GIPO. The induced domain contains extra states for the *jack* sort, but this model is valid. After training to convergence there are 3 parameter flaws. See the end of this section for a discussion of flaws and their automated repair, and fig. 3 for a diagram of the repaired model, Appendix A for action schema).

Blocks (GIPO version). A correct state machine is derived. After training to convergence there are 3 parameter flaws. The low number of steps needed to derive the state machine is due to there being only 2 sorts in the domain, both of which are involved in every action.

Driverlog (IPC strips version). State machines and parameters are correct for all sorts except trucks. For trucks, the distinction of states with/without driver is lost, and an extra state parameter (driver) is retained. The state machine for driver is shown in fig. 4

Freecell (IPC strips version). This is a version of the well-known patience card game used in the IPC3 competition. There are three sorts discovered in the freecell domain -

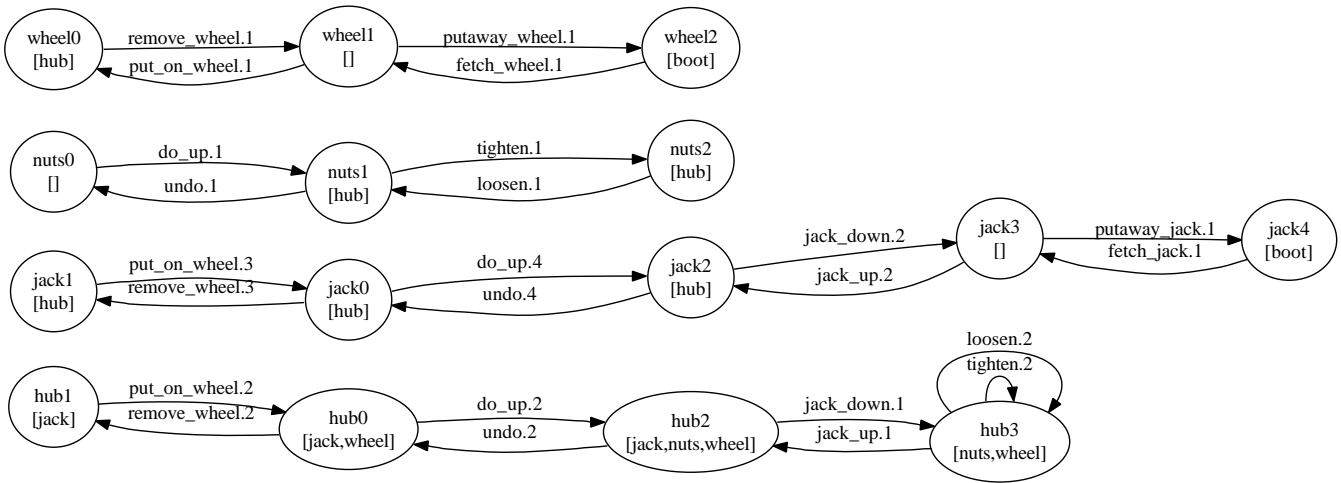


Figure 3: Other state machines induced from the tyre-world.

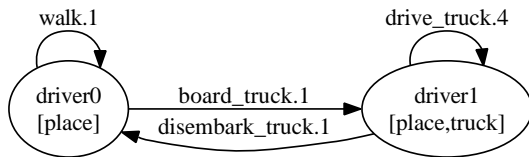


Figure 4: Induced state machine for driver in driverlog domain.

suits, cards and numbers. In the competition version of the domain, number objects are used to represent denominations of cards and to count free cells and free columns. The state machine derived for the cards has 7 states. The states (see fig. 5) can be understood as follows:

- card3 - in a column and covered by another card
- card4 - in a column and not covered
- card5 - in a free cell
- card0 - in a home cell
- card1, card2, card6 - in a home cell and covered

It is not helpful to distinguish the 3 final states, but *LOCM* cannot determine that they are equivalent. Whilst the *LOCM* results from Freecell are amongst the more interesting we have found, there are a number of problems which need to be overcome in future versions of *LOCM* to extract a usable domain model from freecell plans:

- The distinction is lost between cards which are the bottom of a column and other cards which are in a column. Solving this problem requires weakening of the strong assumptions underpinning phase I.
- *LOCM* doesn't detect background relationships between objects — the adjacency of pairs numbers, and the alternation of black cards on red cards. This could be achieved by inductive learning on the set of all actions which ever occur.

Randomly-generated example data can be different in character from purposeful, goal-directed plans. In a sense, random data is more informative, because the random plan is likely to visit more permutations of action sequences which a goal-directed sequence may not. However, if the useful, goal-directed sequences lead to induction of a state machine with more states, this could be seen as useful heuristic information.

Where there is only one object of a particular sort (e.g. gripper, wrench, container) all hypotheses about matching that sort always hold, and the sort tends to become an internal state parameter of everything. For this reason, it is important to use training data in which more than one object of each sort is used.

The induced models may contain detectable flaws: the existence of a state parameter has been induced, but there are one or more transitions into the state which do not set the state parameter. The flaws usually arise because state parameters are induced only by considering pairs of consecutive transitions, not longer paths.

The inconsistency may indicate that an object reference is carried in from another state without being mentioned in an action's argument. In this case a repair to the model can be proposed, which involves adding the "hidden" parameter to some states, but a further cycle of testing against the example data is required to check that the repair is consistent. The parameters in the state machine shown in fig. 3 and the example operators in Appendix A have been generated from the algorithms described above, together with an initial implementation of an algorithm for detecting, repairing and testing parameter flaws. This was successful at completing a correct and consistent model for the tyre domain. This will be further developed in future work.

The most fundamental limitation is whether it is possible to correctly represent the domain within the limitations of the representation that we use for action schema.

- We assume that an action moves the objects in its argu-

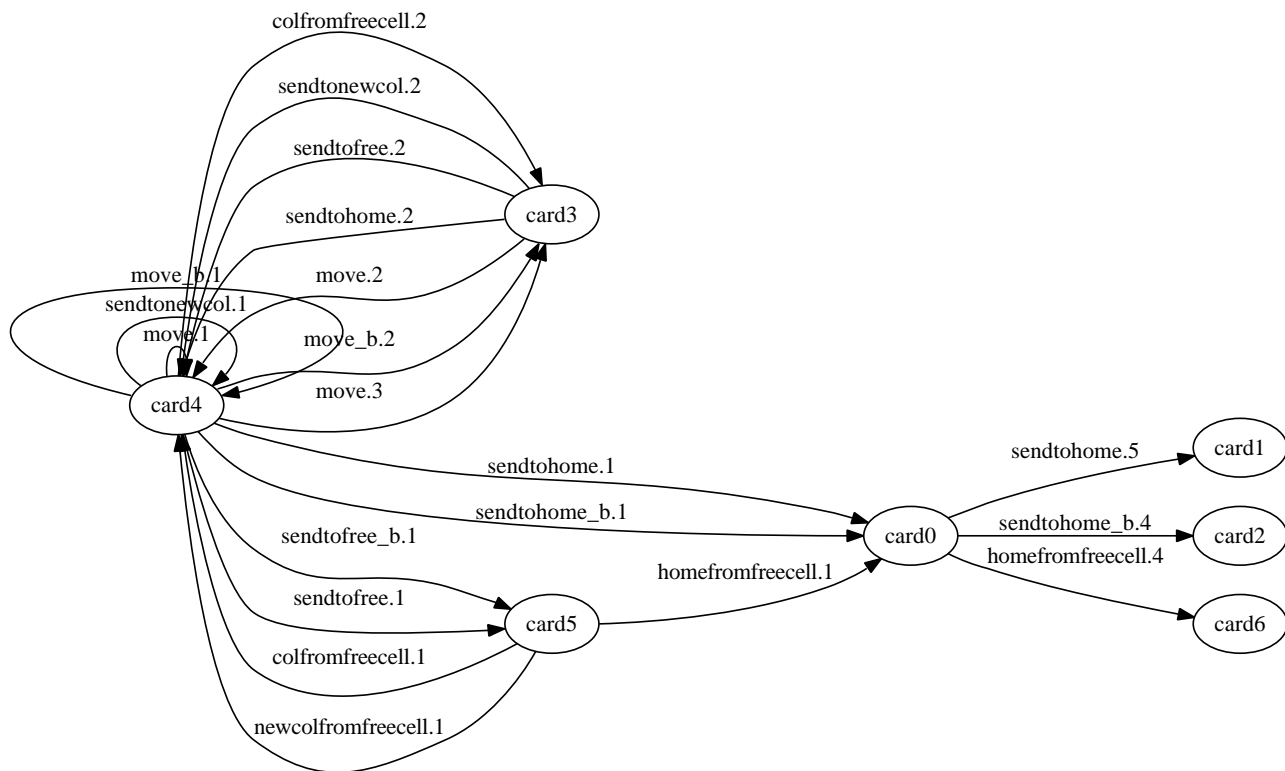


Figure 5: Induced state machine for cards in Freecell domain.

ments between clearly-defined substates. Objects which are passively involved in an action may make a transition to the same state, but cannot be in a *don't care* state.

- Static background information, such as the specific fixed relationships between objects (e.g. which places are connected), is not analysed by the system. In general, this can lead to missing preconditions. The *LOCM* algorithm assumes that all information about an object is represented in its state and state parameters. In general, this form of information may vary anyway between training examples.

Related Work

LOCM is distinct from other systems that learn action schema from examples in that it requires **only** the action sequences as input; its success is based on the assumption that the output domain model can be represented in an object-centred representation. Other systems require richer input: *ARMS* (Wu, Yang, and Jiang 2005) makes use of background knowledge as input, comprising types, relations and initial and goal states, while the system of (Shahaf and Amir 2006) appears to efficiently build expressive actions schema, but requires as input specifications of fluents, as well as partial observations of intermediate states between action executions. The *Opmaker* algorithm detailed in (McCluskey et al. 2009) relies on an object-centred approach similar to *LOCM* but it too requires a partial domain model as input as well as a training instance.

The *TIM* domain analysis tool (Fox and Long 1998) uses a similar intermediate representation to *LOCM* (i.e. state space for each sort), but in *TIM*, the object state machines are extracted from a complete domain definition and problem definition, and then used to derive hierarchical sorts and state invariants.

Learning expressive theories from examples is also a central goal in the Inductive Logic Programming community. We lack space to discuss this literature here, but work by for example (Benson 1996) is very relevant to the induction of planning domain models.

Conclusion

In this paper, we have described the *LOCM* system and its use in learning domain models (comprising object sorts, state descriptions, and action schema), from example action sequences containing no state information.

Although it is unrealistic to expect example sets of plans to be available for all new domains, we expect the technique to be beneficial in domains where automatic logging of some existing process yields plentiful training data, e.g. games, workflow, online transactions.

The work is at an early stage, but we have already obtained promising results on benchmark domains, and we see many possibilities for further developing the technique. In particular, we expect to be able demonstrate *LOCM* in the competition acquiring usable domain models from ac-

tion traces of humans playing computer games such as card games.

References

- Benson, S. S. 1996. *Learning Action Models for Reactive Autonomous Agents*. Ph.D. Dissertation, Dept of Computer Science, Stanford University.
- Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *J. Artif. Intell. Res. (JAIR)* 9:367–421.
- McCluskey, T.; Cresswell, S.; Richardson, N.; and West, M. M. 2009. Automated acquisition of action knowledge. In *International Conference on Agents and Artificial Intelligence (ICAART)*, 93–100.
- Richardson, N. E. 2008. *An Operator Induction Tool Supporting Knowledge Engineering in Planning*. Ph.D. Dissertation, School of Computing and Engineering, University of Huddersfield, UK.
- Shahaf, D., and Amir, E. 2006. Learning partially observable action schemas. In *AAAI*. AAAI Press.
- Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning Domain Definition Using GIPO. *Journal of Knowledge Engineering* 1.
- Wu, K.; Yang, Q.; and Jiang, Y. 2005. ARMS: Action-relation modelling system for learning acquisition models. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning*.

APPENDIX A

The operators induced for the tyre domain are shown below in a simplified form of OCL syntax.

```
operator(close_container(Boot1),
[
  tr(Boot1:boot,
    boot_state0(Boot1) =>
    boot_state1(Boot1)
  ])
).
```

```
operator(do_up(Nuts1,Hub2,Wrench5,Jack3),
[
  tr(Nuts1:nuts,
    nuts_state0(Nuts1) =>
    nuts_state1(Nuts1,Hub2))
  tr(Hub2:hub,
    hub_state0(Hub2,Jack3,Wheel4) =>
    hub_state2(Hub2,Jack3,Nuts1,Wheel4))
  tr(Wrench5:wrench,
    wrench_state1(Wrench5) =>
    wrench_state1(Wrench5))
  tr(Jack3:jack,
    jack_state0(Jack3,Hub2) =>
    jack_state2(Jack3,Hub2)
  ])
).
```

```
operator(fetch_jack(Jack1,Boot2),
[
  tr(Jack1:jack,
    jack_state4(Jack1,Boot2) =>
    jack_state3(Jack1))
  ])
).
```

```
tr(Boot2:boot,
  boot_state0(Boot2) =>
  boot_state0(Boot2)
]).
```

```
operator(fetch_wheel(Wheel1,Boot2),
[
  tr(Wheel1:wheel,
    wheel_state2(Wheel1,Boot2) =>
    wheel_state1(Wheel1))
  tr(Boot2:boot,
    boot_state0(Boot2) =>
    boot_state0(Boot2)
  ])
).
```

```
operator(fetch_wrench(Wrench1,Boot2),
[
  tr(Wrench1:wrench,
    wrench_state0(Wrench1,Boot2) =>
    wrench_state1(Wrench1))
  tr(Boot2:boot,
    boot_state0(Boot2) =>
    boot_state0(Boot2)
  ])
).
```

```
operator(jack_down(Hub1,Jack2),
[
  tr(Hub1:hub,
    hub_state2(Hub1,Jack2,Nuts3,Wheel4) =>
    hub_state3(Hub1,Nuts3,Wheel4))
  tr(Jack2:jack,
    jack_state2(Jack2,Hub1) =>
    jack_state3(Jack2)
  ])
).
```

```
operator(jack_up(Hub1,Jack4),
[
  tr(Hub1:hub,
    hub_state3(Hub1,Nuts2,Wheel3) =>
    hub_state2(Hub1,Jack4,Nuts2,Wheel3))
  tr(Jack4:jack,
    jack_state3(Jack4) =>
    jack_state2(Jack4,Hub1)
  ])
).
```

```
operator(loosen(Nuts1,Hub2,Wrench4),
[
  tr(Nuts1:nuts,
    nuts_state2(Nuts1,Hub2) =>
    nuts_state1(Nuts1,Hub2))
  tr(Hub2:hub,
    hub_state3(Hub2,Nuts1,Wheel3) =>
    hub_state3(Hub2,Nuts1,Wheel3))
  tr(Wrench4:wrench,
    wrench_state1(Wrench4) =>
    wrench_state1(Wrench4)
  ])
).
```

```
operator(open_container(Boot1),
[
  tr(Boot1:boot,
    boot_state1(Boot1) =>
    boot_state0(Boot1)
  ])
).
```

```

operator(put_on_wheel(Wheel1,Hub2,Jack3),
[
  tr(Wheel1:wheel,
    wheel_state1(Wheel1) =>
    wheel_state0(Wheel1,Hub2))
  tr(Hub2:hub,
    hub_state1(Hub2,Jack3) =>
    hub_state0(Hub2,Jack3,Wheel1))
  tr(Jack3:jack,
    jack_state1(Jack3,Hub2) =>
    jack_state0(Jack3,Hub2)
  ]).

operator(putaway_jack(Jack1,Boot2),
[
  tr(Jack1:jack,
    jack_state3(Jack1) =>
    jack_state4(Jack1,Boot2))
  tr(Boot2:boot,
    boot_state0(Boot2) =>
    boot_state0(Boot2)
  ]).

operator(putaway_wheel(Wheel1,Boot2),
[
  tr(Wheel1:wheel,
    wheel_state1(Wheel1) =>
    wheel_state2(Wheel1,Boot2))
  tr(Boot2:boot,
    boot_state0(Boot2) =>
    boot_state0(Boot2)
  ]).

operator(putaway_wrench(Wrench1,Boot2),
[
  tr(Wrench1:wrench,
    wrench_state1(Wrench1) =>
    wrench_state0(Wrench1,Boot2))
  tr(Boot2:boot,
    boot_state0(Boot2) =>
    boot_state0(Boot2)
  ]).

operator(remove_wheel(Wheel1,Hub2,Jack3),
[
  tr(Wheel1:wheel,
    wheel_state0(Wheel1,Hub2) =>
    wheel_state1(Wheel1))
  tr(Hub2:hub,
    hub_state0(Hub2,Jack3,Wheel1) =>
    hub_state1(Hub2,Jack3))
  tr(Jack3:jack,
    jack_state0(Jack3,Hub2) =>
    jack_state1(Jack3,Hub2)
  ]).

operator(tighten(Nuts1,Hub2,Wrench4),
[
  tr(Nuts1:nuts,
    nuts_state1(Nuts1,Hub2) =>
    nuts_state2(Nuts1,Hub2))
  tr(Hub2:hub,
    hub_state3(Hub2,Nuts1,Wheel3) =>
    hub_state3(Hub2,Nuts1,Wheel3))
  tr(Wrench4:wrench,
    wrench_state1(Wrench4) =>
    wrench_state1(Wrench4)
  ]).

operator(undo(Nuts1,Hub2,Wrench5,Jack3),
[
  tr(Nuts1:nuts,
    nuts_state1(Nuts1,Hub2) =>
    nuts_state0(Nuts1))
  tr(Hub2:hub,
    hub_state2(Hub2,Jack3,Nuts1,Wheel4) =>
    hub_state0(Hub2,Jack3,Wheel4))
  tr(Wrench5:wrench,
    wrench_state1(Wrench5) =>
    wrench_state1(Wrench5))
  tr(Jack3:jack,
    jack_state2(Jack3,Hub2) =>
    jack_state0(Jack3,Hub2)
  ]).

```

On Compiling Data Mining Tasks to PDDL *

Susana Fernández and Fernando Fernández and Alexis Sánchez
Tomás de la Rosa and Javier Ortiz and Daniel Borrajo
Universidad Carlos III de Madrid. Leganés (Madrid). Spain

David Manzano
Ericsson Research Spain
Madrid, Spain

Abstract

Data mining is a difficult task that relies on an exploratory and analytic process of large quantities of data in order to discover meaningful patterns and rules. It requires complex methodologies, and the increasing heterogeneity and complexity of available data requires some skills to build the data mining processes, or knowledge flows. The goal of this work is to describe data-mining processes in terms of Automated Planning, which will allow us to automatize the data-mining knowledge flow construction. The work is based on the use of standards both in data mining and automated-planning communities. We use PMML (Predictive Model Markup Language) to describe data mining tasks. From the PMML, a problem description in PDDL can be generated, so any current planning system can be used to generate a plan. This plan is, again, translated to a KFML format (Knowledge Flow file for the WEKA tool), so the plan or data-mining workflow can be executed in WEKA. In this manuscript we describe the languages, how the translation from PMML to PDDL, and from a plan to KFML are performed, and the complete architecture of our system.

Introduction

Currently, many companies are extensively using data-mining tools and techniques in order to answer questions as: who would be interested in a new service offer among your current customers? What type of service a given user is expecting to have? These questions, among others, arise nowadays in the telecommunication sector and many others. Data mining (DM) techniques give operators and suppliers an opportunity to grow existing service offers as well as to find new ones. Analysing the customer generated network data, operators can group customers into segments that share the same preferences and exhibit similar behaviour. Using this knowledge the operator can recommend other services to users with a high probability for uptake. The problem is not limited to operators. In every business sector, companies are moving towards the goal of understanding their cus-

tomers' preferences and their satisfaction with the products and services to increase business opportunities.

Date mining is a difficult task that relies on an exploratory and analytic process of large quantities of data in order to discover meaningful patterns and rules. It requires a data mining expert able to compose solutions that use complex methodologies, including problem definition, data selection and collection, data preparation, or model construction and evaluation. However, the increasing heterogeneity and complexity of available data and data-mining techniques requires some skills on managing data-mining processes. The choice of a particular combination of techniques to apply in a particular scenario depends on the nature of the data-mining task and the nature of the available data.

In this paper, we present our current work that defines an architecture and tool based on Automated Planning that helps users (non necessarily experts on data-mining) on performing data-mining tasks. Only a standardized model representation language will allow for rapid, practical and reliable model handling in data mining. Emerging standards, such as PMML (Predictive Model Markup Language), may help to bridge this gap. The user can represent and describe in PMML data mining and statistical models, as well as some of the operations required for cleaning and transforming data prior to modelling. Roughly speaking, a PMML file is composed of three general parts: the data information, the mining build task and the model. But, during the complete data mining process, not all of them are available. The data part describes the resources and operations that can be used in the data mining process. The mining build task describes the configuration of the training task that will produce the model instance. It can be seen as the description of the sequence of actions executed to obtain the model, so from the perspective of planning, it can be understood as a plan. This plan would include the sequence of data-mining actions that should be executed over the initial dataset to obtain the final model that could be also added to the third part of the PMML. Therefore, a complete PMML file contains all the information describing a data mining task, from the initial dataset description to the final model, through the knowledge flow that generates that model. We use a PMML file containing only the data information, and leaving empty the other two parts, as an input to create PDDL (Planning Domain Definition Language) problem files. These files allow,

*This work has been partially supported by the Spanish MICINN under project TIN2008-06701-C03-03, the regional project CCG08-UC3M/TIC-4141 and the Automated User Knowledge Building (AUKB) project funded by Ericsson Research. Copyright © 2009, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

together with the PDDL domain file and a planner, to automatically create a plan or knowledge flow. This knowledge flow will be executed by a machine learning engine. In our case, we use one of the most used data mining tools, the WEKA system (Witten & Frank 2005). In WEKA, knowledge flows are described as KFML files. The results of this process can be evaluated, and new plans may be requested to the planning system. The plans generated by the planner for solving the translated PDDL problem are encoded in XML and can be directly added to the PMML mining building task. However, so far, stable versions of WEKA do not encode models in XML, so we leave empty the model part of the PMML file. Instead, the system returns to the user a result zip file containing all the model files together with statistical information files in WEKA format. In this paper, we describe the first implementation of such a tool, emphasizing the compilations between PMML and PDDL.

There has been previous work whose aim is also to apply planning techniques to automatize data mining tasks (Amant & Cohen 1997; Morik & Scholz 2003; Provost, Bernstein, & Hill 2005), but any of them use standard languages to represent the knowledge, as we have done in the work presented in this paper. Next section presents an introduction to data mining. The remainder sections describe the general architecture, the languages used in the application, the modelization of the DM tasks in PDDL, the implemented translators and the conclusions.

Introduction to KDD and Data Mining

Knowledge Discovery in Databases (KDD) concerns with the development of methods and techniques for analyzing data. The first part of the KDD process is related to the selection, preprocess and transformation of the data to be analyzed. The output of this part of the KDD process is a set of patterns or training data, described in terms of features, typically stored in a table. Data mining is the following step in the KDD process, and consists of applying data analysis and discovery algorithms that generate models that describe the data (Fayyad, Piatetsky-Shapiro, & Smyth 1996). The modelling task is sometimes seen as a machine learning or statistical problem where a function must be approximated. The KDD process finishes with the interpretation and evaluation of the models.

Learning algorithms are typically organized by the type of function that we want to approximate. If the output of the function is known “a priori” for some input values, we typically talk about supervised learning. In this case, we can use regression (when the approximated function is continuous), or classification (the function is discrete) (Mitchell 1997). There are many models for supervised learning. Some examples are neural networks, instance-based, decision and regression rules and trees, bayesian approaches, support vector machines, etc. When the output of the function to approximate is not known “a priori”, then we talk about unsupervised learning. In this kind of learning, the training set is composed only of the list of input values. Then, the goal is to find a function which satisfies some learning objectives. Different learning objectives can be defined, like clustering,

dimensionality reduction, association, etc, and they all may require different models and learning algorithms.

The evaluation of a data mining problem typically requires to apply the obtained model over data that were not used to generate the model. But depending on the amount of data, or the type of function modelled, different evaluation approaches may be used, such as split (dividing the whole data set in training and test sets), cross-validation, or leave-one-out (Mitchell 1997).

Therefore, In the data mining process, there are four main elements to define: the training data, the model or language representation, the learning algorithm, and how to evaluate the model. The number of combinations of those four elements is huge, since different methods and techniques, all of them with different parameter settings, can be applied. We show several examples of these operators in the paper. Because of that, the data mining process is sometimes seen as an expert process where data mining engineers transform original data, execute different mining operators, evaluate the obtained models, and repeat this process until they fit or answer the mining problem. Because of the complexity of the process, it is suitable as a planning problem that can be solved with automated approaches (Fernández *et al.* 2009).

Architecture

Figure 1 shows the general architecture of the approach. There are four main modules; each one can be hosted in a different computer connected through a network: the *Client*, the *Control*, the *Datamining* and the *Planner*. We have used the Java RMI (Remote Method Invocation) technology that enables communication between different servers running JVM's (Java Virtual Machine). The planner incorporated in the architecture is SAYPHI (De la Rosa, García-Olaya, & Borrajo 2007) and the DM Tool is WEKA. Although, any others could be used. Next, there is a description of each module.

The Client Module

It offers a control command console interface that provides access to all application functionalities and connects the user to the *Control* module using RMI. It captures the PMML file and sends it to the *Control* module. At this point, the PMML file only contains the information concerning the dataset and operations that can be executed in the data mining process. The other two parts are empty. Before performing any DM task, the corresponding dataset must be placed in the DM Tool host, through this module.

The Control Module

It is the central module of the architecture that interconnects and manages the planning and DM modules, serving requests from the user module. This module is also responsible of performing the conversions between the PDDL and PMML formats invoking the *PMML2PDDL* external application. It also transforms the output plan generated by the planner to a KFML format used by the WEKA Knowledge Flow. This KFML format transformation is performed

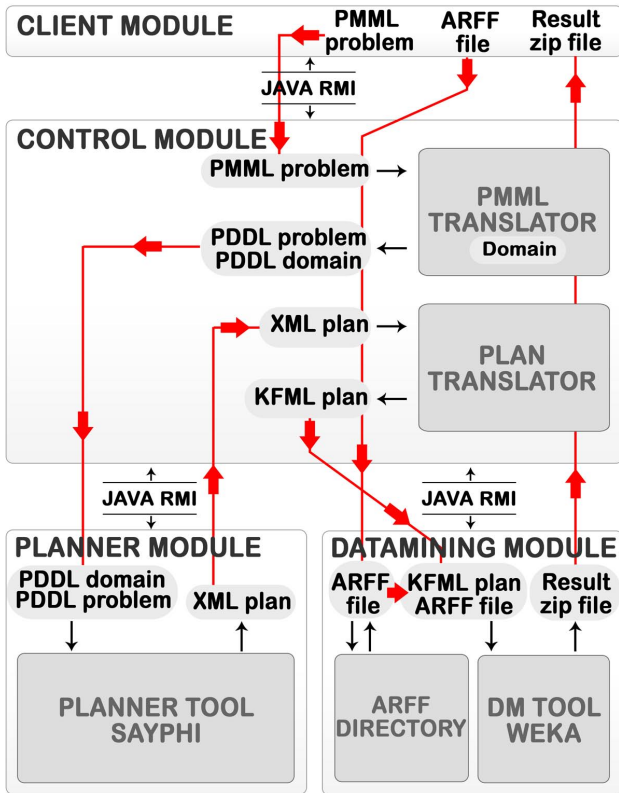


Figure 1: Overview of the architecture.

through the invocation of the external *PlanReader* application.

The input to the module is the user choice together with the required files to carry out that option. In case the user's option is to perform a DM request, the algorithm of Figure 2 is executed. First, the **PMML2PDDL** translator generates the PDDL problem file from the PMML file. Then, the planner is invoked through RMI to solve the translated problem. The returned plan is translated to a KFML file. Finally, the DM Tool is invoked to execute the translated KFML file. The *result* is a compressed file containing the model generated by the DM Tool and the statistics.

The Datamining Module

This module provides a Java wrapper that allows the execution of DM tasks in the WEKA DM Tool through Knowledge Flow plans. The Java wrapper is able to obtain the model output and the statistics generated as a result of the Knowledge Flow execution. This module also contains an *Arff* directory for managing the storage of the datasets that are necessary for the *PlanReader* and WEKA executions. The inclusion or removal of *arff* files are managed by the user through the options offered in the command control user interface.

DM-Request(*pmml-file, domain*):*result*

pmml-file: PMML file with the DM task
domain: PDDL domain

problem=**PMML2PDDL**(*pmml-file*)
plan=**RMI-Planner**(*domain, problem*)
kfml-file=**Plan2KFML**(*plan*)
result=**RMI-DMTool**(*kfml-file*)
return *result*

Figure 2: Algorithm for executing a DM request.

The input to the module is a KFML file and the output is a compressed file including the model and the statistics generated during the KFML execution.

The Planner Module

The planning module manages requests from the control module receiving the problem and the domain in PDDL format. It returns the result to the *Control* module in XML format ready for the conversion to a KFML format. Currently, planning tasks are solved by the planner SAYPHI, but the architecture could use any other planner that supports fluents and metrics. We have used SAYPHI because: i) it supports PDDL (requirements *typing* and *fluents*); ii) it incorporates several search algorithms able to deal with quality metrics; iii) it is implemented in Lisp allowing rapid prototyping of new functionalities; and iv) it is in continuous development and improvement in our research group.

Standard Languages of the Tool

This section describes the two languages used in this work (apart from PDDL). First, we describe PMML (Predictive Model Markup Language), an XML based language for data mining. Then, we describe KFML (Knowledge Flow for Machine Learning), another XML based language to represent knowledge flows in data mining defined in the WEKA tool (Witten & Frank 2005). The PDDL language is not explained since it is a well-known standard in the planning community. We use the PDDL 2.1 version (Fox & Long 2003) for handling classical planning together with numeric state variables.

The Predictive Model Markup Language (PMML)

PMML is a markup language for describing statistical and data mining tasks and models. It is based on XML, and it is composed of five main parts:¹

- The header contains general information about the file, like the PMML version, date, etc.
- The data dictionary defines the meta-data, or the description of the input data or learning examples.

¹The language is being developed by the Data Mining Group (DMG). See www.dmg.org for further information

Models	Functions
AssociationModel	
ClusteringModel	
GeneralRegressionModel	
MiningModel	
NaiveBayesModel	AssociationRules
NeuralNetwork	Sequences
RegressionModel	Classification
RuleSetModel	Regression
SequenceModel	Clustering
SupportVectorMachineModel	
TextModel	
TreeModel	

Table 1: Models and Functions defined in the PMML standard

- The transformation dictionary defines the functions applicable over the input data, like flattening, aggregation, computation of average or standard deviation, normalization, principal component analysis (PCA), etc. In our case, this knowledge defines the actions that can be applied over the data, which will be defined in the planning domain file.
- The mining build task describes the configuration of the training task that will produce the model instance. PMML does not define the content structure of this part of the file, so it could contain any XML value. This mining build task can be seen as the description of the sequence of actions executed to obtain the model, so from the perspective of planning, it can be understood as a plan. This plan would include the sequence of data-mining actions that should be executed over the initial dataset to obtain the final model.
- The model describes the final model generated after the data mining process, i.e. after executing the mining build task. There are different models that can be generated depending on the data analysis technique used, ranging from bayesian, to neural networks or decision trees. Depending on the type of model the description will use a different XML format.

The models implement some functions, and depending on the function, they may introduce different constraints over the data (which will have to be considered in the planning domain file). For instance, a `TreeModel` can be used both for classification or regression, but depending on the implementation itself, only one of such functions may be available. If, for instance, only regression has been implemented, then the class attribute of the dataset must be continuous if we want to apply the `TreeModel`. The complete list of models and functions defined in the PMML standard are defined in Table 1. Different models can implement different functions and, as introduced above, the applicability of the model to a function may depend on the implementation of the model itself and the advances of the state of the art, so they are not constrained “a priori”.

A complete PMML file contains the information about the five parts. However, during the data mining process, not all the data is available. At the beginning, the PMML file contains only the header, the data dictionary and the transformation dictionary. In addition, it includes the different models that may be used to find a solution. This can be considered as the input of the data mining process. The mining build task contains the steps or process that should be followed to obtain a model using the data described in the PMML initial file. In our case, we generate this version of the PMML file from the result of planning by encoding the plans in XML. Finally, the model could be included after the data mining tool executes the data-mining workflow generated by the planner. In our case, we generate the model using the WEKA Knowledge Flow tool that receives as input a KFML file. The tool automatically translates the plan into the KFML file, but the model is not incorporated in the PMML file, because WEKA does not encode it yet in XML.

WEKA and the Knowledge Flow Files (KFML)

WEKA (Witten & Frank 2005) is a collection of machine learning algorithms to perform data mining tasks. It includes all the software components required in a data mining process, from data loading and filtering to advanced machine learning algorithms for classification, regression, etc. It also includes many interesting functionalities, like graphical visualization of the results. WEKA offers two different usages. The first one is using directly the WEKA API in Java. The second one consists of using the graphical tools offered: the Explorer permits to apply data mining algorithms from a visual interface; the Experimenter permits to apply different machine learning algorithms over different datasets in a batch mode; the Simple CLI, which permits to make calls to WEKA components through the command line; and the Knowledge Flow.

WEKA Knowledge Flow is a data-flow inspired interface to WEKA components. It permits to build a knowledge flow for processing and analysing data. Such knowledge flow can include most of the WEKA functionalities: load data, prepare data for cross-validation evaluation, apply filters, apply learning algorithms, show the results graphically or in a text window, etc. Knowledge flows are stored in KFML files, that can be given as input to WEKA.

A KFML file is an XML file including two sections. The first one defines all the components involved in the knowledge flow, as data file loaders, filters, learning algorithms, or evaluators. The second one enumerates the links among the components, i.e. it defines how the data flows in the data mining process, or how to connect the output of a component with the input of other components. WEKA Knowledge Flow permits to load KFML files, edit them graphically, and save them. KFML files can be executed both by using the graphical interface or the WEKA API.²

A knowledge flow can be seen as the sequence of steps that must be performed to execute a data mining process. From our point of view, the knowledge flow is not more

²The WEKA API can be used to execute KFML files only from version 3.6.

than the plan that suggest to perform a data mining process. Later, we will show how a plan generated in our architecture can be saved in the KFML format and executed through the WEKA API.

Modelling Data-Mining Tasks in PDDL

The PDDL domain file contains the description of all the possible DM tasks (transformations, training, test, visualization, ...). Each DM task is represented as a domain action. The PDDL problem files contain information for a specific dataset (i.e. dataset schema, the suitable transformation for the dataset, the planning goals, the user-defined metric, etc.). Domain predicates allow us to define states containing static information (i.e. possible transformations, available training or evaluation tasks, etc.) and facts that change during the execution of all DM tasks (e.g. `(preprocess-on ?d - DataSet)` when the dataset is pre-processed). Fluents in the domain allow us to define thresholds for different kinds of characteristics (e.g. an execution time threshold, or the readability of a model) and to store numeric values obtained during the execution (e.g. the total execution time, the mean-error obtained, etc.).

There are different kinds of actions in the domain file:

- **Process Actions:** for specific manipulations of the dataset. For instance, `load-dataset` or `datasetPreparation` for splitting the dataset or preparing it for cross-validation after finishing the data transformation. Preconditions verify the dataset and test mode availability with the predicates `available` and `can-learn`, respectively. Figure 3 shows the PDDL `datasetPreparation` action. It adds the effect `(eval-on)` for allowing the training and testing. We use fluents for estimating the execution time of actions, whose values are defined in the problem. We assume the execution time is proportional to the number of instances (in fact thousands of instances) and depends on the test mode. For example, we estimate that the preparation factor for splitting is 0.001 and for cross-validation is 0.005. The `(loaded ?d)` precondition is an effect of the `load-dataset` action.

```
(:action datasetPreparation
:parameters (?d - DataSet ?t - TestMode)
:precondition (and (loaded ?d)
                  (can-learn ?d ?t))
:effect (and (eval-on ?d ?t)
            (not (preprocess-on ?d))
            (not (loaded ?d))
            (increase (exec-time) (* (thousandsofInstances)
                                     (preparationFactor ?t))))))
```

Figure 3: Example of PDDL process action.

- **Transformation Actions:** in the form of `apply-transformation-<filter>`. Preconditions verify field type constraints and whether the filter has been included as a DM task for the dataset or not. The action usually adds a fact indicating that the task has been done (e.g. the `normalize` transformation adds to the state the fact `(normalized DataSet)`). Figure 4 shows the attribute selection PDDL action. We assume this is the last transformation we can perform because

afterwards we stop knowing the remaining attributes and the metric estimation would return an unknown value. Precondition `(known-fields)` checks it. Precondition `(transformation ?i AttributeSelection)` verifies the filter has been included in the PMML file and `(preprocess-on ?d)` prevents from applying the transformation before performing the pre-process and after preparing the data for splitting or cross-validation. Again, we assume an execution-time increment that is computed from several fluents whose values are defined in the problem.

```
(:action apply-transformation-AttributeSelection
:parameters (?d - DataSet ?i - TransfInstance)
:precondition (and (preprocess-on ?d)
                  (known-fields ?d)
                  (transformation ?i AttributeSelection))
:effect (and (applied-instance ?d ?i)
            (applied-transformation ?d AttributeSelection)
            (not (known-fields ?d))
            (increase (exec-time)
                      (* (* (filter-time AttributeSelection)
                          thousandsofInstances)
                        (* (dataDictionaryNumberOfFields)
                           (dataDictionaryNumberOfFields))))))
```

Figure 4: PDDL action representing the attribute selection transformation.

- **Training Actions:** in the form of `train-<model-type>`, where `<model-type>` can be `classification`, `regression` or `clustering`. In each type, the action parameters indicate different models previously defined in the PMML file. There is a precondition for verifying if the model instance is learnable, predicate `learnable`, and another one to verify the model, predicate `is-model`. These actions add to the state that the selected option is a model for the dataset with the predicate `is-<model-type>-model`. Figure 5 shows the PDDL action for training in a classification task. Training tasks always increment errors, in case of classification they are the classification error, *percentage-incorrect*, and the readability of the learned model, *unreadability*. And, we assume values for these increments.

```
(:action train-classification
:parameters (?mi - ModelInstance ?m - Model ?n -
             ModelName ?d - DataSet ?fi - FieldName ?t - TestMode)
:precondition (and (learnable ?mi)
                  (is-model ?mi ?m)
                  (implements ?m classification ?n)
                  (is-field ?fi ?d)
                  (dataDictionaryDataField-otype ?fi categorical)
                  (eval-on ?d ?t))
:effect (and (is-classification-model ?mi ?d ?fi)
            (not (preprocess-on ?d))
            (not (learnable ?mi))
            (increase (unreadability)
                      (model-unreadability ?m))
            (increase (percentage-incorrect)
                      (model-percentage-incorrect ?m))
            (increase (exec-time)
                      (* (* (model-exec-time ?m)
                          thousandsofInstances)
                        (* (dataDictionaryNumberOfFields)
                           (dataDictionaryNumberOfFields))))))
```

Figure 5: PDDL action for training.

- **Testing Actions:** in the form of `test-<model-type>`.

These actions usually follow their corresponding training action. For instance, cross-validation or split. They are separated from the training actions, because they are needed when handling the process flow in the KFML. Testing actions add the fact that the learned model has been evaluated.

- **Visualizing and Saving Actions:** in the form of `visualize-<result-option>`. These actions are related to goals defined in the problem file. Action parameters indicate the preferred option depending on the learned model, the pre-defined options in the PMML file and the goal defined for the specific planning task. There is a precondition to verify if the visualization model is allowed for that model, predicate `allow-visualization-model`.

The domain actions represent specific DM tasks. A matching between domain actions and the DM tasks defined in the KFML file is required. To have an idea of the complexity of this planning domain, we have to take into account that WEKA implements more than 50 different transformation actions or filters, more than 40 training actions for classification and regression, and 11 training actions for classification. Each of these transformation and training actions can be parameterized, so the number of different instantiations of those actions is huge. Obviously, not all of them must be included in the PDDL files to generate a plan, and the user can define in the PMML which of them s/he is interested in using.

There are different ways to define the domain actions. For instance, in our case the actions `train-classification` and `train-regression` are coded separately because they correspond to different DM tasks in WEKA. However, they could have been merged in only one action, obtaining a more compact domain representation with an easier maintenance. With the current implementation we gain the opportunity of faster integration with other DM tools different than WEKA. The performance of the planner is not affected, since it performs the search using grounded actions, so both domain definitions will become equivalent in search time.

Information in the problem file is automatically obtained from the PMML file using the translator described in the next section. This information, specific to a dataset, is used by the planner to instantiate all the possible actions specified in the PMML file by the user and to handle constraints imposed to the planning task.

The whole domain contains the following actions. `load-dataset` is always the first action in all plans. Then, the plans can contain any number of filter actions such as `apply-transformation-DerivedField-2op`, `apply-transformation-normalize`, `apply-transformation-discretization` or `apply-transformation-AttributeSelection`. There could be no filter action, as well. After the attribute selection filter is applied no other filter is allowed. The following action is the `datasetPreparation` action that prepares the dataset for splitting or for a cross-validation. Then, there are three possible actions for training, `train-classification`, `train-regression` and `train-clustering`, another three for testing,

`test-classification`, `test-regression` and `test-clustering`, and three more for visualizing the model `visualize-classification-model`, `visualize-regression-model` and `visualize-clustering-model`. The last action in all plans is the `visualize-result` for knowing the learning results (errors, execution time, ...).

Translators

PMML to PDDL

The PMML2PDDL translator automatically converts parts of a PMML file with the DM task information into a PDDL problem file. The problem file together with a domain file, that is assumed to stay fixed for all the DM tasks, are the inputs to the planner. The problem file contains the particular data for each DM episode, including the dataset description, the transformations and models available for that problem, and the possible preferences and constraints the user required. An example of preference is minimizing the total execution time. Obtaining an error less than a given threshold is an example of constraint.

A PDDL problem file is composed of four parts: the *objects*, the *inits*, the *goals* and the *metric*. *Inits* represents the initial state of the problem and there is one proposition for each fact in the state. There are some propositions common to all problems (static part of the problem specification) and others are translated from the PMML file (dynamic part). *Goals* represent the problem goals. So far, they are fixed for all the problems, and consist of visualizing the result, (`visualized-result result text`), for saving the statistics required to analyze the generated DM models. *Metric* represents the formula on which a plan will be evaluated for a particular problem. A typical DM metric consists of minimizing the classification error, but others could be used, as minimizing execution time or maximizing the readability of the learned model. SAYPHI, as the planners based on the enhanced relaxed-plan heuristic introduced in Metric-FF (Hoffmann 2003), only work with minimization tasks. So, we transform maximizing the understandability of the learned model for minimizing *complexity*. The other preferences considered in our model are: *exec-time*, for minimizing the execution time; *percentage-incorrect*, for minimizing the classification error; and *mean-absolute-error*, for minimizing the mean absolute error in regression tasks and clustering. We can handle similar constraints.

The static part of the problem specification includes all the propositions concerning the predicates `is-model`, `learnable`, `allow-visualization`, `available` and `can-learn`, explained in the previous section, and the initial values of the fluents. The dynamic part includes the information about the dataset, the constraints and preferences, the transformations and the models available for the DM request. Figure 6 shows an example of PMML file representing the well-known *Iris* DM set, allowing only one transformation (a discretization) and two models (a neural network and a decision tree). In general, for most DM tasks, a user would include in the PMML file all WEKA DM techniques (there are plenty of them). In the example of PMML

file, the user also defines: two constraints, concerning the execution time and the complexity; and one preference, to minimize the percentage-incorrect value, or classification error. We obtain information from the following parts of the PMML: *Header*, *DataDictionary*, *TransformationDictionary* and *Models*. *Header* includes the constraints and the preferences. Constraints are translated by adding numerical goals to the PDDL problem. Preferences are translated by changing the metric of the problem. *DataDictionary* includes one field for each attribute in the dataset and they are translated into propositions in the initial state and objects of the problem. *TransformationDictionary* is translated by adding some objects and propositions in the initial state of the problem. Finally, each model defined in the *Models* part is translated by adding a proposition in the initial state.

Figure 6 shows an example of the translation from a PMML file to a PDDL problem file. It is easy to see how some information is translated. For instance, the tag *DataField* in the PMML file generates some predicates in the PDDL file, like `dataDictionaryDataField-type`, `dataDictionaryDataField-otype`, one for each attribute of the initial dataset. The transformation *Discretize* of the PMML file is translated by adding the predicate `(transformationInstance-type discretize continuous)`. When a model appears in the PMML, as `<NeuralNetwork modelName="nnmodel1" functionName="classification" algorithmName="Neural Net" activationFunction="radialBasis">`, it enables the planner to use neural networks. The parameters of that model are also defined in the PMML file, so the predicate `(implements NeuralNetwork classification nnmodel1)` and other related predicates are added to the problem file. Similarly, the preference described in the header of the PMML file, `<Constraint variable="percentage-incorrect" value="minimize"/>`, is translated by including a metric `(:metric minimize (percentage-incorrect))` in the problem file; while the constraints `<Constraint variable="exec-time" value="30"/>` and `<Constraint variable="complexity" value="8"/>` are translated by including the numerical goals `<(exec-time) 30>` and `<(complexity) 8>`.

The translator makes the following conversions from the PMML file to the PDDL problem file (`<variable>` represents the value of field *variable* in the PMML file):

1. Translates entry `<DataDictionary>` into the following propositions in the *inits*:
 - (a) `(= (DataDictionaryNumberOfFields) <numberOfFields>)`
 - (b) `(dataDictionaryDataField-type <name> <dataType>)`, for each `<DataField>` entry
 - (c) `(dataDictionaryDataField-otype <name> <optype>)`, for each `<DataField>` entry
 - (d) `(is-field <name> DataSet)`, for each `<DataField>` entry
 - (e) Creates an object `<name>` of type `SchemaFieldName` in *objects*
2. Adds the following proposition in the *inits* for each `<DefinedFunction>` entry:

- (a) `(transformationInstance-type <name> <opType>)`
 - (b) `(transformation <name> valueX)`, where `valueX` is in `<DefinedFunction.Extension.WekaFilterOptions.option.value>` when `variable="filter" value=valueX`
 - (c) Create an object `<name>` of type `transfInstance` in *objects*
3. Adds the following proposition in the *inits* for each `<DerivedField>` entry:
 - (a) `(dataDictionaryDataField-type <name> <dataType>)`
 - (b) `(dataDictionaryDataField-type <name> <optype>)`
 - (c) `(derivedField-operation <name> <Apply.function>)`
 - (d) `(derivedField-source <name> <Apply.FieldRef.field>)` for each `<FieldRef>` in each `<DerivedField>`
 - (e) Create an object `<name>` of type `DerivedFieldName` in the *objects*
 4. Adds the following proposition in the *inits* for each `<modelName>` entry:
`(implements <model> <functionName> <modelName>)`
 5. Adds a numeric goal `<(<Constraint variable> <value>)>` for each `<DataMiningConstraints>` entry
 6. Translates entry `<DataMiningPreferences>` for `(:metric minimize (<Constraint variable>))` in *metric*.

In order to make the translation process easier and more general, we have defined an intermediate XML file, *translation.xml*, to drive the translation explained above. The translator interprets this file each time together with the input PMML file and generates a PDDL problem file. So, it is possible to include, modify or eliminate PMML entries from the translation process without changing the program, but modifying this file. This intermediate XML file permits to configure the information we need from the PMML file, so that in case the PMML file or the domain change we only have to change this file without modifying the translator. For example, the translation 1.(b) explained above is represented with the following entry in the *translation.xml* file:

```
<elements value="dataDictionaryDataField-type"
  where="DataDictionary/DataField">
  <field type="attribute">name</field>
  <field type="attribute">dataType</field>
  <format>(/0 /1 /2)</format>
</elements>
```

Planning for DM Tasks

As we mentioned in the previous section, SAYPHI solves the planning task depending on the metric specified in the PMML file. SAYPHI has a collection of heuristic algorithms. For this application, the planner performs a Best-first Search that continues exploring nodes in order to find multiple solutions. Figure 7 shows an example of a best-cost solution plan when the translated PDDL indicates the percentage error as the problem metric. Likewise, Figure 8 shows a best-cost solution plan for the same problem, but using the execution time metric. In the first plan a *Neural Network* is preferred

```

<?xml version="1.0" encoding="UTF-8"?>
<PMML xmlns="http://www.dmg.org/PMML-3_2"
  version="3.2"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.dmg.org/PMML-3_2 pmml-3-2.xsd">
  <Header copyright="Copyright UC3M Ericsson LME 2009">
    <Extension>
      <DataSources DataSourceFormat="file">
        <DataFile location="iris.arff"/>
      </DataSources>
      <DataMiningConstraints constraints="yes">
        <Constraint variable="exec-time" value="30"/>
        <Constraint variable="complexity" value="8"/>
      </DataMiningConstraints>
      <DataMiningPreferences preferences="yes">
        <Constraint variable="percentage-incorrect"
          value="minimize"/>
      </DataMiningPreferences>
    </Extension>
  </Header>

  <DataDictionary numberOfFields="5">
    <DataField name="sepalength" displayName="sepalength"
      optype="continuous" dataType="double"/>
    <DataField name="sepalwidth" displayName="sepalwidth"
      optype="continuous" dataType="double"/>
    <DataField name="petalength" displayName="petalength"
      optype="continuous" dataType="double"/>
    <DataField name="petalwidth" displayName="petalwidth"
      optype="continuous" dataType="double"/>
    <DataField name="class" displayName="class"
      optype="categorical" dataType="string">
      <Extension name="class" value="yes"/>
      <Value value="Iris-setosa" property="valid"/>
      <Value value="Iris-versicolor" property="valid"/>
      <Value value="Iris-virginica" property="valid"/>
    </DataField>
  </DataDictionary>

  <TransformationDictionary>
    <!-- WEKA Filters: -->
    <DefineFunction name="discretize" optype="continuous">
      <Extension>
        <WekaFilterOptions options="yes">
          <option variable="supervised" value="yes"/>
          <option variable="type" value="attribute"/>
        </WekaFilterOptions>
      </Extension>
      <ParameterField name="x" optype="continuous"/>
    </DefineFunction>
  </TransformationDictionary>

  <!-- Models -->
  <NeuralNetwork modelName="nnmodel1" functionName="classification"
    algorithmName="Neural Net" activationFunction="radialBasis">
    <Extension>
      <WekaModelOptions options="yes">
        <option variable="model" value="functions"/>
        <option variable="algorithm" value="RBFNetwork"/>
        <option variable="num-clusters" value="2"/>
      </WekaModelOptions>
    </Extension>
  </NeuralNetwork>

  <TreeModel modelName="treemodell" functionName="classification"
    algorithmName="C4.5">
    <Extension>
      <WekaModelOptions options="yes">
        <option variable="model" value="tree"/>
        <option variable="algorithm" value="J48"/>
        <option variable="unpruned" value="true" />
      </WekaModelOptions>
    </Extension>
  </TreeModel>
</PMML>

```

```

(define (problem p1)
  (:domain mole)
  (:objects sepalength sepalwidth petalength
    petalwidth class - SchemaFieldName)
  (discretize AttributeSelection - transfInstance)
  (:init

    ;; Common to all problems
    (is-model tree treeModel)
    (learnable tree)
    (is-model nn NeuralNetwork)
    (learnable nn)

    (allow-visualization-model treeModel graph)
    (allow-visualization-model treeModel text)
    (allow-visualization-model NeuralNetwork graph)
    (allow-visualization-model NeuralNetwork text)

    (allow-visualization-result result text)
    (allow-visualization-result result graph)
    (allow-visualization-result result2 text)
    (allow-visualization-result result2 graph)

    (= (exec-time) 0)
    (= (function-time-discretize) 0.05)

    (= (model-exec-time NeuralNetwork) 0.01)
    (= (model-exec-time treeModel) 0.001)

    (= (complexity) 0)
    (= (model-complexity NeuralNetwork) 5)
    (= (model-complexity treeModel) 9)

    (= (percentage-incorrect) 0)
    (= (model-percentage-incorrect NeuralNetwork) 5)
    (= (model-percentage-incorrect treeModel) 9)

    (= (mean-absolute-error) 0)
    (= (model-mean-absolute-error NeuralNetwork) 5)
    (= (model-mean-absolute-error treeModel) 9)

    (= (exec-time-test-regression) 5)
    (= (exec-time-test-clustering) 5)

    ;; depend on the PMML
    ;; PMML Inserted by the translator
    (dataDictionaryDataField-type sepalength double)
    (dataDictionaryDataField-type sepalwidth double)
    (dataDictionaryDataField-type petalength double)
    (dataDictionaryDataField-type petalwidth double)
    (dataDictionaryDataField-type class string)
    (dataDictionaryDataField-optype sepalength continuous)
    (dataDictionaryDataField-optype sepalwidth continuous)
    (dataDictionaryDataField-optype petalength continuous)
    (dataDictionaryDataField-optype petalwidth continuous)
    (dataDictionaryDataField-optype class categorical)
    (is-field sepalength initialDataSet)
    (is-field sepalwidth initialDataSet)
    (is-field petalength initialDataSet)
    (is-field petalwidth initialDataSet)
    (is-field class initialDataSet)
    (transformationInstance-type discretize continuous)
    (implements NeuralNetwork classification nnmodel1)
    (implements TreeModel classification treemodell)
    (= (DataDictionaryNumberOfFields) 5)
    ;; PMML Inserted by the translator
    )
  (:goal (and (visualized-result result text)
    (< (complexity) 8)
    (< (exec-time) 30)
    ))
  (:metric minimize (percentage-incorrect))
)

```

(a) Example of PMML file encoding a DM request for the *Iris* domain. The PMML file has been simplified to eliminate non-relevant information for the purpose of generating the problem file.

(b) Problem file in PDDL generated from the PMML file shown in (a). Again, irrelevant information has been eliminated.

Figure 6: Simplified PMML and PDDL problem files.

```

0: (LOAD-DATASET INITIALDATASET)
1: (DATASETPREPARATION INITIALDATASET CROSS-VALIDATION)
2: (TRAIN-CLASSIFICATION NN NEURALNETWORK
   NNMODEL4 INITIALDATASET CLASS CROSS-VALIDATION)
3: (TEST-CLASSIFICATION NN INITIALDATASET NEURALNETWORK
   CLASS CROSS-VALIDATION RESU)
4: (VISUALIZE-RESULT NN INITIALDATASET CROSS-VALIDATION
   RESU TEXT)

```

Figure 7: An example plan minimizing the percentage error.

```

0: (LOAD-DATASET INITIALDATASET)
1: (DATASETPREPARATION INITIALDATASET SPLIT)
2: (TRAIN-CLASSIFICATION TREE TREEMODEL
   TREEMODEL1 INITIALDATASET CLASS SPLIT)
3: (TEST-CLASSIFICATION TREE INITIALDATASET
   TREEMODEL CLASS SPLIT RESU)
4: (VISUALIZE-RESULT TREE INITIALDATASET
   SPLIT RESU TEXT)

```

Figure 8: An example plan minimizing the execution time.

in the `train-classification` action because in the default definition it has a lower cost than applying the same action with another model. On the other hand, the second plan has two differences. The *Split* parameter is selected instead of *Cross-validation*, and *Tree Model* is preferred instead of *Neural Network*. Both differences arise because these actions are cheaper in terms of execution time. The best-cost solution using the readability metric coincides with the second plan, since the *Tree Model* is the parameter the minimizes the cost for the training action.

Most of the action costs rely on formulae affected by pre-defined constants for each parameter (e.g. a constant fluent defined in the initial state, such as `(= (model-percentage-incorrect NeuralNetwork) 5)`). These constant values were set by a data mining expert in order to reproduce common results depending on different executed tasks. Accordingly, the total cost for a solution is taken as an estimation, since the real cost in this application can not be known in advance. Sensing the environment after each action execution could establish real costs, but we have not included this process at the current state of the project.

Plan to KFML

This section describes the translator that generates the KFML files. Once the planner generates a plan, it has to be translated into a KFML file, so it can be executed by the WEKA Knowledge Flow. The translator reads a plan in an XML format. Figure 9 shows a plan (in a standard format, not XML) generated by SAYPHI using the PDDL problem shown in Figure 6.

The translator generates as output a new KFML file with an equivalent plan plus some new actions that the WEKA Knowledge Flow can execute. Each action in the PDDL Domain corresponds to one or many WEKA components. Therefore, the translator writes for each action in the plan the corresponding set of XML tags that represent the WEKA

```

0: (LOAD-DATASET INITIALDATASET)
1: (APPLY-TRANSFORMATION-ATTRIBUTESELECTION
   INITIALDATASET ATTRIBUTESELECTION4)
2: (DATASETPREPARATION INITIALDATASET CROSS-VALIDATION)
3: (TRAIN-CLASSIFICATION TREE TREEMODEL TREEMODEL1
   INITIALDATASET CLASS CROSS-VALIDATION)
4: (TEST-CLASSIFICATION TREE INITIALDATASET
   INITIALDATASET CLASS CROSS-VALIDATION RESULT)
5: (VISUALIZE-CLASSIFICATION-MODEL TREE
   TREEMODEL GRAPH INITIALDATASET CLASS)
6: (VISUALIZE-RESULT TREE INITIALDATASET
   CROSS-VALIDATION RESULT TEXT)

```

Figure 9: A example plan generated by SAYPHI.

```

...
<object class="weka.gui.beans.BeanInstance" name="2">
  <object class="int" name="id" primitive="yes">2</object>
  <object class="int" name="x" primitive="yes">450</object>
  <object class="int" name="y" primitive="yes">145</object>
  <object class="java.lang.String" name="custom_name">
    CrossValidationFoldMaker
  </object>
  <object class="weka.gui.beans.CrossValidationFoldMaker" name="bean">
    <object class="int" name="seed" primitive="yes">1</object>
    <object class="int" name="folds" primitive="yes">10</object>
  </object>
</object>
...

```

Figure 10: A part of the KFML file corresponding to the `DATASETPREPARATION` action in the solution plan presented in Figure 9.

component. For instance, Figure 10 shows a brief section of the KFML file generated from the plan in Figure 9. This part corresponds to the third action in the plan, i.e., the `DATAPREPARATION` action.

Actions appearing in a plan can correspond to one of these cases:

- The action corresponds to exactly one WEKA component. For instance, the `LOAD-DATASET` action corresponds to the `ArffLoader` component of the knowledge flow. Some additional information may be needed from the PMML file (e.g., the URI of the dataset to be loaded).
- The action corresponds to many WEKA components and the action parameters decide which component needs to be selected. For instance, in the `DATASETPREPARATION` action, the second parameter indicates the type of operation. Thus, `CROSS-VALIDATION` corresponds to the `CrossValidationFoldMaker` component in the knowledge flow, and `SPLIT` corresponds to the `TrainTestSplitMaker` component.
- The action corresponds to many WEKA components and the action parameters only specify the name of a technique or model. In these cases, the translator needs to extract that information from the PMML file in order to decide which KFML components should be selected. The information extracted from the PMML file includes the name of the component and the parameters that has to be written in the XML code. For instance, the `TREEMODEL` parameter of the `TRAIN-CLASSIFICATION` action corresponds to the J48 algorithm and some other component

properties defined in the PMML file (See Figure 6).

After writing all components into the KFML file, the translator connects them in the order specified by the plan using the linking primitives of KFML. Finally, the translator adds some extra components in order to save the information generated during the execution. That information are the learned models and the results of the plan execution. Figure 11 shows the knowledge flow diagram that the WEKA GUI presents when reading the KFML file generated for this example.

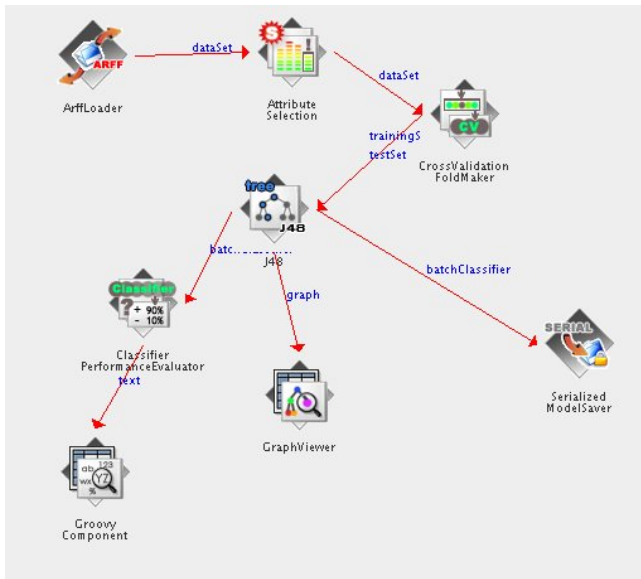


Figure 11: The knowledge flow diagram of the KFML file translated from the plan in Figure 9

Conclusions and Future Work

This paper presents a proposal for modelling data mining tasks by using automated planning, based on extensive use of standard representation languages. The contribution of the work is twofold: modelling the data mining task as an automated planning task and implementing translators able to compile any data mining episode represented in the standard language PMML into the planning standard language PDDL. We have defined a PDDL domain that contains actions to represent all the possible DM tasks (transformations, training, test, visualization, ...). The domain is assumed to stay fixed for all the DM episodes, but each action contains preconditions to control its activation. The PDDL problems are automatically translated from a PMML file representing a DM episode adding the propositions for activating the allowed actions in the particular DM episode. This model allows to deal with plan metrics as minimizing the total execution time or the classification error. During the planning process the metrics are computed from a planning point of view giving some estimated values for their increments experimented in the actions, but we cannot know their true value until a DM Tool executes them. Once the planner solves a

problem the solution plan is translated into a KFML file to be executed by the Knowledge Flow of WEKA. The generated model and the statistics are returned to the user. We have implemented a distributed architecture to automate the process.

In the future, we would like to generate several plans to solve the same problem, to execute them in WEKA and to return all the models to the user. Probably, the best models according to the planner are not necessarily the best models according to the user. Thus, we would also like to apply machine learning for improving the plan generation process and the quality of the solutions.

References

- Amant, R. S., and Cohen, P. R. 1997. Evaluation of a semi-autonomous assistant for exploratory data analysis. In *Proc. of the First Intl. Conf. on Autonomous Agents*, 355–362. ACM Press.
- De la Rosa, T.; García-Olaya, A.; and Borrajo, D. 2007. Using cases utility for heuristic planning improvement. In *Case-Based Reasoning Research and Development: Proceedings of the 7th International Conference on Case-Based Reasoning*, 137–148. Belfast, Northern Ireland, UK: Springer Verlag.
- Fayyad, U.; Piatetsky-Shapiro, G.; and Smyth, P. 1996. From data mining to knowledge discovery in databases. *AI Magazine* 17(3):37–54.
- Fernández, F.; Borrajo, D.; Fernández, S.; and Manzano, D. 2009. Assisting data mining through automated planning. In Perner, P., ed., *Machine Learning and Data Mining 2009 (MLDM 2009)*, volume 5632 of *Lecture Notes in Artificial Intelligence*, 760–774. Springer-Verlag.
- Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 61–124.
- Hoffmann, J. 2003. The Metric-FF planning system: Translating “ignoring delete lists” to numeric state variables. *Journal of Artificial Intelligence Research* 20:291–341.
- Mitchell, T. M. 1997. *Machine Learning*. McGraw-Hill.
- Morik, K., and Scholz, M. 2003. The miningmart approach to knowledge discovery in databases. In *In Ning Zhong and Jiming Liu, editors, Intelligent Technologies for Information Analysis*, 47–65. Springer.
- Provost, F.; Bernstein, A.; and Hill, S. 2005. Toward intelligent assistance for a data mining process: An ontology-based approach for cost-sensitive classification. *IEEE Transactions on Knowledge and Data Engineering* 17(4).
- Witten, I. H., and Frank, E. 2005. *Data Mining: Practical Machine Learning Tools and Techniques*. 2nd Edition, Morgan Kaufmann.

Modeling E-Learning Activities in Automated Planning *

Antonio Garrido and Eva Onaindia

Universidad Politecnica de Valencia.

Camino de Vera s/n, 46071 (Valencia). Spain

Lluvia Morales and Luis Castillo

Universidad de Granada.

Granada. Spain

Susana Fernández and Daniel Borrajo

Universidad Carlos III de Madrid.

Leganés (Madrid). Spain

Abstract

This paper presents three approaches to generate learning designs using existing domain-independent planners. All of the approaches compile a course defined in a standard e-learning language into a planning domain, and a file containing student's learning information into a planning problem. The learning designs are automatically generated from the plans that solve the problems. The approaches differ in the kind of planning domain generated, thus increasing the possibilities of using existing planners: i) hierarchical, ii) including PDDL actions with conditional effects, and iii) including PDDL durative actions. We also analyse the pros and cons on the knowledge engineering procedures used in each approach.

Introduction

Sequencing of learning activities, according to different student's profiles and pedagogical theories, has been a widely studied subject by planning community for at least a decade (Brusilovsky & Vassileva 2003; Castillo *et al.* 2009; Vrakas *et al.* 2007). This sequencing depends on temporal conditions given by the needs of each student, the course duration, the available resources and even collaboration between tutors and students, which makes the problem very interesting for the AI P&S community.

However, acquiring enough information on educational domains to represent them as a planning domain is not an easy work at practice. Until few years ago, there was no standard language to represent most of the many aspects involved in learning activities sequencing. Thanks to the recent rise and widespread use of specification languages based on XML schemata, such as IMS-MD, IMS-LIP, and IMS-LD (IMS-GLC 2001 2009), the e-learning community can now represent information on educational domains in full detail. Despite this, designing generic translators from the information of those standards into a planning domain representation can be difficult because of two main reasons: i) people give different meanings and uses to the fields of the standards, given that the standards provide some flexibility on how to represent knowledge, and ii) there is a great variety of planning paradigms.

*This work has been partially supported by the Spanish MICINN under projects TIN2008-06701-C03-03 and TIN2005-08945-C06-05, and the regional project CCG08-UC3M/TIC-4141.

This paper focuses on how to model learning scenarios (learning objects and students) as planning domains+problems, and on the automated translation from standard e-learning languages to planning models. More particularly, the paper presents a general architecture which consists of three translation approaches to compile learning designs, based on IMS-MD and IMS-LIP standards (IMS-GLC 2001 2009), into planning domains, as depicted in Figure 1. These domains, together with the file compilations that contain students' learning information, are later solved by existing domain-independent planners. Thus, the resulting plans represent tailored sequences of learning activities that students must follow. And this represents an important advantage: each learning design comprises a personalised plan that fully fits each student's necessities and preferences, learning styles and lets him/her work at his/her own pace. Finally, the plan is translated into another standard representation, called IMS-LD, that displays the learning design on different on-line learning platforms. In essence, this paper contributes with:

- An automated translation of IMS-MD and IMS-LIP e-learning templates into three different planning compilations: i) hierarchical, ii) PDDL-conditional, and iii) PDDL-temporal.
- An intuitive graphic tool that enriches the metadata labelling of the learning objects. This enrichment plays an important role, as it serves to complete information that is not always described from an educational point of view, but still needed for AI planning.
- An effective use of planning technology to generate learning designs that best suit students' learning goals, thus promoting a more personalised access to the learning objects.
- An additional translator that parses the resulting plans, and generates the input resources (learning objects) and learning design to be included in state-of-the-art learning platforms, such as dotLRN and Moodle, thus closing the e-learning cycle.

The paper is organised as follows. First, we briefly describe the e-learning basis on which our work is based. Next, we include some related work. After that, we analyse how to model learning designs in planning, and present the translation section with the templates used to convert e-learning

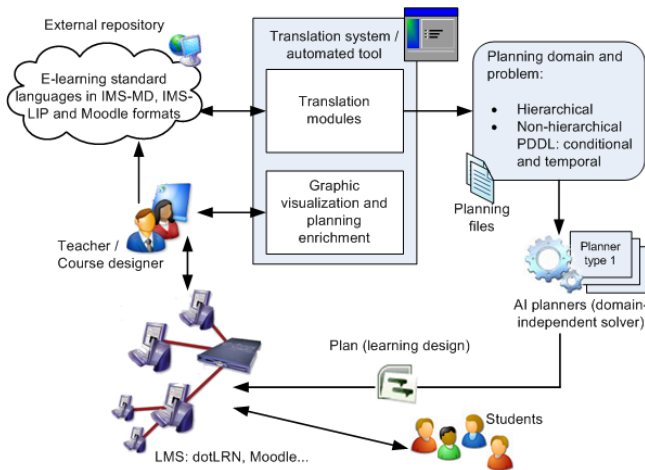


Figure 1: Overview of our system's architecture.

standards into planning domains. Then, we introduce our graphic tool that allows us to enrich the courses from a planning standpoint. Later, we discuss the options to use the resulting plans. Finally, we conclude the paper and motivate the future work.

Basic Background on E-Learning

There are many standards that the e-learning community uses to deal with learning activities and their sequencing. The most famous one is SCORM (Wisher 2009) which integrates some of the IMS (IMS-GLC 2001 2009) family standards but not those that model student profiles and sequencing personalization. The IMS family standards are composed, among others of the following languages:

- IMS-MD, which is responsible for describing learning activities and relations among them. It uses an XML schema with nested labels such as *title*, *resourceType*, *learningTime*, etc.
- IMS-LIP, which integrates every student's profile data in a single document. The standard has also nested labels; e.g. *identifier*, *name*, *preferences*, *competencies*, etc.
- IMS-LD, which includes the relations between information and sequencing of learning activities, and one or several students' profiles.

These standards are supported by some learning management systems (LMS). We have established an XML-RPC and SOAP communication protocol with two of them, Moodle and dotLRN respectively, capable to retrieve and provide information about them in order to be used in the real world. Moodle supports IMS-MD and IMS-LIP standards and has an API that returns them in a special format shown in Figure 2. dotLRN can additionally display learning activities sequences using the IMS-LD standard.

The first two standards are relatively easy to use and that is the reason why its extensive spread over the past years. However, despite the IMS-LD's first stable version is already six years old, it has not been widely accepted in the

```

- <row>
  <identifier>ITEM-task3</identifier>
  <title>Algorithms</title>
  <language />
  <otherPlatformRequirements />
  <learningResourceType />
  <difficulty>es</difficulty>
  <typicalLearningTime />
  <resource />
  <resourceId />
  <honeyAlonsoStyle />
- <isPartOf>
  <IP00>ORG-s2ctest2</IP00>
</isPartOf>
<isBasedOn />
- <requires>
  <REQ0>ITEM-action6</REQ0>
  <REQ1>ITEM-task5</REQ1>
</requires>
<references />
</row>
- <row>
  <identifier>ITEM-action6</identifier>
  <title>Basic Algorithms Lecture</title>
  <language>es</language>
  <coverage>Mandatory</coverage>
  <otherPlatformRequirements>NoOne</otherPlatformRequirements>
  <learningResourceType>Narrative Text</learningResourceType>
  <difficulty>very easy</difficulty>
  <typicalLearningTime>9</typicalLearningTime>
  <resource>test-src/one.html</resource>
  <resourceId>RES-2DB0D82CFCB32EBC9B9D961486EC04A6</resourceId>
  <honeyAlonsoStyle />
- <isPartOf>
  <IP00>ITEM-task3</IP00>
</isPartOf>
<isBasedOn />
<requires />
<references />
</row>

```

Figure 2: Two learning objects of an XML course in Moodle format.

e-learning community. It is really difficult for one person to design a plan that considers most of the variables to sequence learning activities in a course and adapt that sequence to every student's profile. And this is the main motivation for using P&S techniques, as they can be very powerful to automatically generate IMS-LD documents containing sequences of learning activities fully adapted to the students.

Related Work

The application of integrated P&S techniques to improve the sequencing of learning activities from Intelligent Tutoring Systems was first introduced in (Peachy & McCalla 1986). The underlying idea for using P&S techniques is to deal with causal (planning) and temporal-resource constraints (scheduling) capabilities in an e-learning setting. Since then, some works have appeared but proposing *ad-hoc* approaches that do not consider standard labellings of learning objects. In (Mohan, Greer, & McCalla 2003), an exhaustive IMS-MD standard labelling over learning objects for the definition of a domain model was proposed, which acted as a precursor for more standard approaches.

The approach proposed in (Camacho, R-Moreno, & Obieta 2007) works over e-learning courses and adapts the plan sequencing to the IMS-LD standard. However, it does not take advantage of the entire standards for domain modelling. Also, it does not support temporal constraints on particular learning activities, but on the entire course. The PASER system (Kontopoulos *et al.* 2008) uses ontologies over learning activities curricula, after the planning process, to simulate the *lack* of information about its causal relations. However, that system does not support scheduling features, such as temporal reasoning, or adaptation capabilities to learning styles and tutor interaction. On the other hand, the approach presented in (Ullrich & Melis 2009) combines both IMS-MD metadata for domain modelling and a set of competencies required by the students. This approach generates an activities sequence over a domain-dependent intelligent tutoring system, which unfortunately makes it lose the ability to generalise the translation process and a further application of its results.

The work presented in this paper shares the ideas on e-learning standard usage and generation of sequence of activities. But, we address that by using a knowledge engineer-

ing algorithm to directly generate the planning domain of a course in a standard Planning Domain Definition Language (PDDL), so that we can easily use state-of-the-art planners to solve the problem.

Modelling Learning Design in Planning

Learning designs keep a strong resemblance with AI planning models. After all, they both rely on the underlying idea of using a sequence of activities that are *linked* by cause-effect relations and make it possible to achieve some (learning) goals. From a general perspective, most learning designs share the following features:

- A course is defined by a set of different learning activities, also known as *learning objects*. Usually, they are represented as XML schemata (see Figure 2). For example, in the definition of an AI course, there may be a generic task for reading the introduction to planning. And, there could be several learning objects to accomplish it, such as viewing a slide presentation, reading the introduction text from the text book, searching for the concept of Automated Planning in the Web and reading a couple of pages, or seeing a graph about planning. It is enough for the student to follow any one of them to accomplish the task, but more than one can be performed as well. This set of options is usually called the *Metadata (MD)* set.
- Each activity can be more or less appropriate to each student depending on the student's profile. Therefore, we need mechanisms to determine this profile. There are many theories that classify students into a set of profiles. Two well-known examples are the Felder's learning styles¹ (Felder 1996) and the Honey-Alonso ones (HoneyAlonso 2002). These learning styles can be translated as order rules or utilities according to the planning paradigm used.
- Learning activities can have dependency relations among them. For instance, before reading about PDDL, the student should have some knowledge on predicate logic. More formally, we support four types of relations that include hierarchical structures and ordering relations based on content dependencies. The hierarchical structures use the *IsPartOf* IMS-MD relation, which represents a hierarchical aggregation of learning objects. Additionally, there are three types of causal dependencies, *Requires*, *IsBasedOn* and *References*, as in LOM terminology (LOM 2002). We interpret the first two relations as hard requirements. In the case of the *Required* elements, all of them have to be completed before initiating a new learning object. Let us assume that 'A Requires B' and 'A Requires C'. In this case, B and C need to be finished before doing A. In the case of the *IsBasedOn* elements, at least one of them has to be completed. Assuming 'A IsBasedOn B' and 'A IsBasedOn C', only B or C must be completed

¹The learning styles model developed by Richard Felder incorporates four dimensions: the Perception dimension (sensitive/intuitive), the Processing dimension (active/reflective), the Input dimensions (visual/verbal) and the Understanding dimension (sequential/global).

before initiating A. Intuitively, the *Requires* and *IsBasedOn* relations represent the idea of conjunctive and disjunctive requirements. On the other hand, the course designer might also recommend other previous objects by means of the *References* relation. This relation does not denote a hard requirement but a recommendation (soft requirement) to complete a learning object before proceeding with the next one.

- Each activity takes a standard time (duration) to fulfill, commonly known as *typical learning time*. In Figure 2, however, ITEM-task3 has no duration (*typicalLearningTime*) as it is derived from its aggregated learning objects. In other words, duration is only specified for *primitive* activities.
- Each learning activity belongs to a source type, such as a lecture, a diagram, an exercise, etc. Although this does not seem to be very relevant for planning, it really has a positive or negative impact in the outcome of the activity. According to education experts, the source type highly interacts with the student's profile. For instance, a lecture is very recommendable for Felder's verbal students but not for visual ones, and just the opposite holds for a diagram.
- The *Metadata* set of learning objects is translated into a planning domain where each learning object is represented as one or several planning actions. As a general rule, we can state that one planning domain is defined per course, but we can use course composition and create more general and larger domains, thus providing more opportunities to reuse the learning objects in other contexts. The planning problem comprises one or more students, where the students' profiles and initial background are encoded as propositions included in the initial state. Finally, the goals usually consist in attaining some levels of knowledge in particular topics or even in accomplishing the whole course.

It is important to note that this information is part of the IMS standard and not specially introduced for AI planning. But, to the specific purpose of using planning techniques, it is essential not to have missing components and to deal always with coherent information. For instance, a situation like 'A Requires B' and 'B Requires A' would entail a failure to find a plan².

As seen from above, learning objects, with their duration, student profile's dependence and the relations defined in their metadata can be metaphorically considered as traditional actions used in AI planning domains. In particular, each learning object can be simply modelled as an action, its dependency relations as preconditions, and its outcomes as effects. For instance, the learning objects of Figure 2 could be modelled by means of a PDDL-like structure similar to the next one:

²The tool that we present below allows the user to visually notice this situation and, therefore, helps validate metadata labelling.

```

(:action ITEM-task3 ;; Algorithms
 :parameters (?s - student)
 :duration ... ;; defined by its aggregated actions
 :precondition (and (ITEM-action6)
                   (ITEM-task5)
                   student's profile requirements...)
 :effect (and (ITEM-task3-done)
              other student's profile-dependent effects...)
)

(:action ITEM-action6 ;; Basic Algorithms Lecture
 :parameters (?s - student)
 :duration 9
 :precondition (student's profile requirements...)
 :effect (and (ITEM-action6-done)
              other student's profile-dependent effects...)
)

```

The basic elements of an action, such as preconditions, duration and effects, are easily recovered from each learning object's metadata and generated in the planning translation. However, there are other elements that are not easy nor intuitive, such as the *IsPartOf* (hierarchical structure) or *References* (soft preconditions) relations, the conditionality/interaction that appears when dealing with different source types and students' profiles, etc. All these elements impose important challenges during the planning compilation, which are more or less significant depending on the planning approach to be used. This reflects the need of emphasising the knowledge engineering methods to perform such compilations.

Translators

This section describes in detail the three different compilations to be subsequently used by domain-independent planners. The most intuitive approach is the *hierarchical* one, where a learning design is modelled as a task hierarchy containing durative actions. Next one is the *PDDL-conditional* that includes PDDL actions with conditional effects and, finally, the *PDDL-temporal* approach that models durative actions. We also analyse the pros and cons on the knowledge engineering procedures used in each approach.

Hierarchical Domain Compilation

This hierarchical domain compilation is based on an extended version of PDDL for handling temporal knowledge in HTN Planners described in (Castillo *et al.* 2006).

When using a hierarchical approach there are two main structures to take into account, tasks and durative actions. Their characteristics, according to the compilation of an e-learning domain, are the following:

To define tasks, the main subject (also called the main task of the course) is formed by ordered subsets of subtasks that have an *IsPartOf* relation with the main one. These subtasks contain others and so on until the subtasks are related to learning activities which are represented as primitive durative actions.

Each task has one or more methods that contain ordered tasks and/or durative actions. Their order is given either by the *IsBasedOn* relation or by additional preconditions based on order rules according to the Honey-Alonso learning style (HoneyAlonso 2002) of each student and its relation with the resource types of the learning activities in the method. For example, if a task is formed by three durative actions, not ordered by *IsBasedOn* relation, and its resource types are

exercise, *narrativeText* and *simulation*, then we have two methods with the next preconditions:

```

If the student's learning style is Pragmatic then
  the sequence order is: exercise, simulation and narrativeText
If the student's learning style is Theoretical then
  the sequence order is: narrativeText, simulation and exercise,

```

where *Pragmatic* and *Theoretical* are constants related to the kinds of learning styles for a student according to the Honey-Alonso theory.

In the next lines, we describe the compilation of the *Algorithms* subject (with identifier item-TASK3 in Figure 2). It is related by *IsPartOf* relation with a subject identified as item-TASK5 that *Requires* the durative action item-action6 which is part of Algorithms too. As they are completely related through a *Requires* relation, then item-TASK3 has a unique method with no preconditions and tasks in the explicit order mentioned later.

```

(:task item_TASK3
 :parameters (?id - stId)
 (:method unique
 :precondition ()
 :tasks (
 (item_action6 ?id)
 (item_TASK5 ?id)))
)

(:task Optional_item_action12
 :parameters (?id - stId)
 (:method yes
 :precondition (availability ?id much)
 :tasks ((item_action12 ?id)))
 (:method no
 :precondition ()
 :tasks ()))

```

On the other hand, and taking into account that this paradigm is based on temporal deadlines, if any of the actions aggregated in the task is related to it through the *References* relation, then an auxiliary task with two methods is created. As in *Optional_item_action12*, the first method does not contain any action or subtask and has no preconditions. The second method must 'invoke' the referenced action only if the student has enough time according to his/her profile.

Finally, to define durative actions we consider that each of them has also conditions related to the student's profile, e.g. a required language level, a high performance, or multimedia availability. Usually, these conditions are assigned to actions with soft preconditions or with the same name but different preconditions and durations. They help adapt a sequence according to the deadlines for each student, which are imposed to each action related to the goal of the course in the problem definition.

PDDL-Conditional Domain Compilation

This approach assumes each learning activity has an utility value that depends on two factors: the student's profile and the learning source type of the activity. According to pedagogical theories, each learning source type is related to the Felder's learning styles, represented in the student's profile (Baldiris *et al.* 2008). So, for each pair <student learning style, activity source type> there is a corresponding utility. We represent learning styles with predicates and the activity utility with a fluent, as it is explained next. The model also assumes that there is a learning object named *fictitious-finish-course-name* that contains the tasks required to fulfil the entire course.

We use one predicate for each Felder's learning style. For example, (*sequential ?s - student ?p - profile_level_type*). The *profile_level_type* can take the value *strong*, *moderate* or *balanced*. If the system determines that the student is, for

example, *strong sensitive* and *strong active* we would add in the initial state of the PDDL problem the propositions (*active student1 strong*) and (*sensitive student1 strong*).

Each learning object is translated into a PDDL action in the following way³:

- The XML label `<title>` is used as the action name.
- We define a predicate with the same action name, but adjoining the prefix *task_* and the suffix *_done*. It is added to the action effects and represents the fact that the student has performed such activity and prevents him/her from repeating it.
- The XML label `<typicallearningtime>` represents the activity duration. We use a fluent to represent the time, (*total_time_student ?s*), that is increased in the amount of this label in the action effects.
- The XML label `<learningsourcetype>` represents the activity source type. Its possible values are *lecture*, *narrativetext*, *slide*, *table*, *index*, *diagram*, *figure*, *graph*, *exercise*, *simulation*, *experiment*, *questionnaire*, *problem-statement*, *selfassessment* and *exam*. We have used the fluent (*reward_student ?s*) to represent the activity utility. Given that it depends on both the student’s learning style and the activity source type, we use conditional effects. For example, when the learning source of an activity is a *lecture*, Felder’s pedagogical theory says that it is *very good* for *reflective*, *intuitive* and *verbal* students. So we add the following conditional effects to the PDDL action:

```
(when (reflective ?s strong)
      (increase (reward_student ?s) 40))
(when (intuitive ?s strong)
      (increase (reward_student ?s) 40))
(when (verbal ?s strong)
      (increase (reward_student ?s) 40))
```

To compute the increasing values of the *reward_student* fluent, we base on a table defined in (Baldiris *et al.* 2008), where rows represent learning source types, columns are the Felder’s learning styles, and intersections can take the values: *very good*, *good* or *indifferent*, depending on how the source type adapts to the Felder’s style. And, we have converted them into numbers, by some kind of normalization.

- The XML label `<relation>` defines a relation between two learning activities. We use two of the four types of causal relations defined in the IMS-MD: *Requires* and *IsBasedOn*, with the meaning defined above. In fact, a learning object with an *IsBasedOn* relation is considered as a fictitious action, because the student has to perform only one of the actions in the *or*-condition and both the reward and the total.time remain the same.

Figures 3 and 4 show PDDL actions translated from learning objects with relations of type *Requires* and *IsBasedOn* respectively. The first action describes the activity *simulates-strips-problem*. It requires that the student has already performed activity *reads-classical-planning*, it takes 30 minutes, and it adds the corresponding rewards. The learning source type is *problem* that is *very good* for

³This compilation takes as input an IMS-MD *Metadata* set.

strong active, *sensitive* and *visual* students and *good* for *strong global* students. We add the precondition (`(not (task_strips_done ?s))`) to avoid including twice the same action in the plan. The second action represents that a student could perform the activity *simulates-strips-problem* or *experiments-strips-problem* to accomplish the task *task_strips_done*.

```
(:action simulates-strips-problem
:parameters (?s - student)
:precondition (and (task_reads-classical-planning_done ?s)
                  (not (task_simulates-strips-problem_done ?s)))
:effect (and (task_simulates-strips-problem_done ?s)
             (increase (reward_student ?s) 5)
             (increase (total_time_student ?s) 30)
             (when (active ?s strong)
                   (increase (reward_student ?s) 30))
             (when (sensitive ?s strong)
                   (increase (reward_student ?s) 30))
             (when (global ?s strong)
                   (increase (reward_student ?s) 15))
             (when (visual ?s strong)
                   (increase (reward_student ?s) 30))))
```

Figure 3: Example of a PDDL action translated from a learning object with a *Requires* relation.

```
(:action OR-fictitious-strips
:parameters (?s - student)
:precondition (and (not (task_strips_done ?s))
                  (or (task_simulates-strips-problem_done ?s)
                      (task_experiments-strips-problem_done ?s)))
:effect (and (task_strips_done ?s)))
```

Figure 4: Example of a PDDL action translated from a learning object with a *IsBasedOn* relation.

As we said before, the *fictitious-finish-course-name* learning object contains, as a *Requires* relations, the tasks required to fulfill the course. This learning object is translated into a fictitious PDDL action with one effect, (*task_course-name_done ?s*), and its preconditions are the tasks required to complete the course, plus the predicate (`(total_time_student ?s) (time_threshold_student ?s)`), to avoid the plan to exceed the time limit. This threshold is defined in the planning problem and represents the total time the student can devote to the course. The planning problem has only the goal (*task_course-name_done ?s*).

This representation allows that any planner that supports full ADL extension (including conditional effects) and fluents can find a solution.

PDDL-Temporal Domain Compilation

This approach follows the same thread presented in the previous conditional compilation *w.r.t.* a non-hierarchical representation of actions in PDDL. However, there are some differences *w.r.t.* the action model:

- As conditional effects are not supported by all existing planners, we do not generate actions with unbound parameters, but fully grounded actions. That is, actions where all the parameters have been instantiated. This means that the name of each predicate needs to include now information about the student. All this process is done automatically. Although this entails rather larger

domains when dealing with many students (only one operator for all the students *vs.* as many grounded actions as students), it simplifies the generation of both preconditions and effects that depend on the student's profile. Now, we do not need preconditions like `(active ?s strong)` nor conditional effects because the action (with all its effects) is generated only if the student is *strong* in the *active* dimension of the learning style. In other words, through a previous automated grounding process, the planning domain will be formed only by those actions that are actually applicable for each student.

- All predicates are generalised to numeric fluents, i.e. all the variable information in the domain is encoded as functions. This means that, instead of using a STRIPS model of actions where preconditions and effects are bi-valued (true/false) predicates, now we can deal with a broader domain of values that allows to keep different levels of knowledge. This increases the expressivity of the model, by allowing us not only metric rewards (e.g. `(increase (reward_Student1) 30)`, like in the conditional compilation) but also having preconditions such as `(>= (Task_Reads-classical-planning_Student1) 50)`. This represents better the fact of: i) achieving marks after executing the tasks, and ii) requiring successful scores before executing tasks.
- This model encodes durations as defined in PDDL2.1, and its successors, by using the `:duration`. This value is directly taken from the *typical learning time* metadata of the learning object. Thus, the PDDL domain can be subsequently used by any temporal planner. Nevertheless, this compilation also has the ability to model time as in the conditional compilation; that is, by means of an artificial fluent (`total_time`) that represents the time-line. The advantage of doing this is that this approach provides a domain compilation valid for existing temporal and non-temporal metric planners.
- The hierarchical structure is flatly encoded by means of two dummy actions, *Start* and *End*, that represent the aggregation activity. *Start* contains the preconditions of the aggregation activity and *End* its effects. On the other hand, the actions generated for all the aggregated objects have that *Start* as precondition. Obviously, both *Start* and *End* have duration 0. Recalling the example depicted in Figure 2, Figure 5 shows an example of the three actions that are generated when encoding the hierarchical relations in a flat structure.
- The utilisation of numeric fluents in actions makes the inclusion of metric resources and their cost easier, as traditionally used in P&S. Particularly, this approach can also model the cost of each action, in terms of the resources used, by simply adding a new effect such as `(increase (resource_cost_Computer) value)`. The inclusion of the resources cost will later allow the user to define more flexible metrics to be optimised in the planning problem.

```
(:durative-action Start_ITEM-task3_Std1 ;; Algorithms
:parameters ()
:duration (= ?duration 0)
:condition (and (at start (= (Start_ORG-s2ctest2_Std1_done) 1))
(at start (= (Start_ITEM-task5_Std1_done) 1))
(at start (= (Start_ITEM-task3_Std1_done) 0)))
:effect (and (at end (increase (Start_ITEM-task3_Std1_done) 1))))

(:durative-action End_ITEM-task3_Std1 ;; Algorithms
:parameters ()
:duration (= ?duration 0)
:condition (and (at start (= (Start_ITEM-task3_Std1_done) 1))
(at start (= (Start_ITEM-task6_Std1_done) 1))
(at start (= (End_ITEM-task3_Std1_done) 0)))
:effect (and (at end (increase (End_ITEM-task3_Std1_done) 1))
increase other numeric expressions or resource_costs))

(:durative-action ITEM-action6_Std1 ;; Basic Algorithms Lecture
:parameters ()
:duration (= ?duration 9)
:condition (and (at start (= (Start_ITEM-task3_Std1_done) 1))
(at start (= (ITEM-action6_Std1_done) 0)))
:effect (and (at end (increase (ITEM-action6_Std1_done) 1))
increase other numeric expressions or resource_costs))
```

Figure 5: Durative actions generated for the learning objects of Figure 2.

Problems Compilation

Once the domain is generated, we need to define problems in such a way that, when the planner solves them, each plan represents a learning design for a particular student. That is, the sequence of learning actions a student should perform in order to complete the course. Usually, LMS have mechanisms based on standards to access the relevant student information for the designs. IMS-LIP has become a standard for storing such student information. But, again, this standard is too generic and tries to cover too many aspects. Therefore, it is necessary to select the relevant student's characteristics required for our planning problems and the XML fields that contain them.

This section describes a proposal of IMS-LIP schema for translating it into a planning problem. The proposal is valid for the hierarchical domain and the PDDL domain with fluents and conditional effects, although the translated propositions, obviously, differ in each domain in relation to the domain predicates. So far, the planning problems for the temporal domain must be defined separately because the grounded domain makes an automatic translation difficult. After all, when working with grounded actions, both the planning domain and problem are packed together as the grounded actions in the domain are only valid for that particular planning problem.

On the one hand, planning problems include propositions to represent the objects, the initial state, the goals and a metric to optimise. In our domains, the objects represent the student's information for the learning design. The initial state represents the student's profile, the initial values of the fluents, the previous knowledge of the student, the language of the course, and some other information (e.g. performance, equipment, availability, etc.). The goal is usually to pass the entire course or a part of it.

On the other hand, the core structures of the IMS-LIP are based upon: accessibility information, activities, affiliations, competencies, goals, identifications, interests, qualifications, certifications and licences, relationship, security

keys, and transcripts. Within each category several data elements and structures are defined. Some of these are specified explicitly as data types (language strings, for the most part) and others are defined as recursive hierarchical structures. Thus, the question is how to match both structures, the IMS-LIP and the planning problem ones, so that, an automatic translation compiles an IMS-LIP file into a planning problem. Table 1 shows the XML fields we have used to allow this compilation. The first column represents the IMS-LIP code and the second column represents the corresponding translation into the planning problem.

IMS-LIP	Planning Problem
<identification><name> <contenttype><referential> <indexid>student1	:objects student1
<accessibility><preference> <typename><typevalue> Learner_Style_Processing <prefcode>reflective_strong	:inits (reflective student1 strong)
<accessibility><preference> <typename><typevalue> Learner_Style_HoneyAlonso <prefcode>theoretical_strong	:inits (honeyAlonso student1 theoretical)
<goal><typename> <tyvalue>AI-course <contenttype><temporal> <typename>Time_Threshold <temporalfield>3881	:inits (= (time_threshold student1) 3811) :goals (task_AI-course_done student1)
<activity><typename> <tyvalue>Task <learningactivityref> <text>graph_theory	:inits (task_graph_theory_done student1)
<accessibility><language> <typename><tyvalue>English	:inits (language_level English student1 high)
<competency><contenttype> <referential><indexid> performanceLevel <description> <short>High	:inits (performance_level student1 high)

Table 1: Example of a problem compilation from an IMS-LIP. Irrelevant information has been eliminated.

Approaches Comparison

Table 2 shows the differences between the three approaches regarding some characteristics. The first rows represent where or how each planning feature is defined. For example, the *Hierarchical* domain is translated from a course defined in Moodle format and the *PDDL-Temporal* one can be defined either in Moodle or in IMS-MD format. The translation is fully automated in all cases. 'Tool' means that the characteristic is defined through the Tool we have implemented. The planning goals in the *Hierarchical* approaches are defined in the IMS-LIP, while in the *PDDL-Conditional* one are defined in a learning object of the Metadata set. The row *Deadline definition* represents where the time limit each student can devote to the course is defined. For example, in the *Hierarchical* and *PDDL-Conditional* approaches, it is a field in the IMS-LIP that is automatically translated. The row *Prerequisite definition* refers to the previous knowledge the student should have in order to follow the course. The row *LOM relations* means the types of relations, according to LOM terminology, supported for the approaches. *Soft preconditions* refers to the fact that the learning design can contain activities that, without being mandatory, provide some benefit to the student. This is possible through the methods in the *Hierarchical* approaches and through the pre-

condition ($< (total_time_student ?s) (time_threshold_student ?s)$) in the *PDDL-Conditional* representation. *Time management* represents how the approaches deal with time. The *Hierarchical* and *PDDL-Temporal* approaches use durative actions while the *PDDL-Conditional* uses fluents. The *PDDL-Temporal* can also compile the domain using a fluent to represent time instead of durative actions. The row *Metrics* means whether the approach can manage quality metrics or not. The last row represents the planner required to solve the problems modelled by the approach. SIADEX (Castillo *et al.* 2006) is the only planner able to generate learning designs in the *Hierarchical* approach, because of the input language. So far, there is no standard language for representing hierarchical domains in PDDL. The other two approaches compile the domains into PDDL, so any planner that supports fluents, metrics, and conditional effects, in the case of *PDDL-Conditional*, or durative actions in the case of *PDDL-Temporal*, can solve the problems.

Characteristic	Hierarchical	PDDL-Conditional	PDDL-Temporal
Domain definition	Moodle	IMS-MD	Both
Problem definition	Moodle and IMS-LIP	IMS-LIP	Tool
Goal definition	IMS-LIP	LO in MD	Problem (Tool)
Deadline definition	IMS-LIP	IMS-LIP	Problem (Tool)
Prerequisite definition	IMS-LIP	IMS-LIP	Problem (Tool)
LOM relations	All	IsBasedOn Requires	All
Students' profile	Honey-Alonso	Felder	Both
Soft preconditions	Method	Domain	-
Time management	Durative actions	Fluent	Both
Metric	No	Yes	Yes
Planner	SIADEX	Conditional effect Metrics	Temporal Metrics

Table 2: Approaches comparison.

The *Hierarchical* approach permits modelling more learning features, including durative actions, but only the planner SIADEX can solve the problems. Also, the methods have to be manually defined. Also, it cannot deal with quality metrics. The other approaches use PDDL and can deal with metrics such as minimizing the total time the student devotes to the course. However, current state-of-the-art planners cannot manage maximizing metrics, so a metric for maximizing the total utility that the learning activities report to the student is not easily applicable. The *PDDL-Temporal* approach automatically generates grounded domains avoiding the use of conditional effects, but it makes ulterior domain modifications difficult, as, for example, trying to find ways for maximizing the utility.

Tool

Once the three translation modules have been presented, we describe the tool that supports the user on generating the planning files. As indicated in Figure 1, the tool comprises two parts and its main goals are twofold. First, the tool acts as an interface for the translators, thus making this process simple and transparent to the user. Second, the tool provides a graphic visualization of the learning objects and their relations, and also allows the designer to modify and tune them by means of intuitive drag&drop graphic components and user-friendly input forms.

Support Interface for Translation

The tool contains options for both importing and exporting files in standard e-learning formats, together with translation support to planning files. Particularly, we can easily import/export the learning objects encoded as IMS-MD in dotLRN or in Moodle XML files. As an example, Figure 6 shows a snapshot of the tool when importing a simple Moodle course with the objects depicted in Figure 2. The possibility of importing learning objects from common standards is very convenient as it allows the designer to reuse many of the objects available in web repositories. After that, the tool uses the three different compilation methods described in the previous section to generate the planning domains and problems accordingly.

Graphic Visualization and Tuning of the Learning Objects

The second part of the tool focuses on modelling e-learning courses, acting as a complementary module to specify and facilitate the completion and extension of metadata records of learning objects, specially those related to the structural and logical relationships that are essential for planning. Loosely speaking, the tool offers a much more intuitive representation of the learning objects by using graphic elements (see Figure 6) rather than the XML files (see Figure 2). This is interesting as we can see at a glance the hierarchical structure, the aggregated objects, the students' profiles, their relationships and also helps notice some inconsistencies, such as circular dependencies between learning objects (e.g. A and B Require each other).

A clear advantage of our tool is that it can be used to improve the quality of the learning objects, at least from the planning perspective. The e-learning standards include much information within the objects, in the form of metadata, but they are not always directly usable in planning. Actually, some of the items are not concerned with planning, like the keywords, the format or the source of the objects. Others are not equivalent to the same named items in planning, such as the resources: a resource in e-learning may be a URL the student needs to visit, but not a shared resource that imposes additional constraints and costs to the plan. Consequently, we can use the tool to complete and tune the metadata labelling of the learning objects, making them more accurate *w.r.t.* i) information about the student's profile, ii) required resources, iii) typical learning time, i.e. duration, and iv) relationships among objects and their types. After all, the more accurate the metadata of the objects is, the better for the planner—it will have more opportunities to find a plan better adapted to the student. Figure 7 shows one of the forms that allow to input basic information about the learning object (from the planning point of view), min and max duration, requirements on profiles and previous concepts, necessary resources, etc. With all this information, and once the students' information is modelled, the tool performs a temporal domain compilation and generates both the domain and the problem files in PDDL format, ready to use by any existing domain-independent planner.

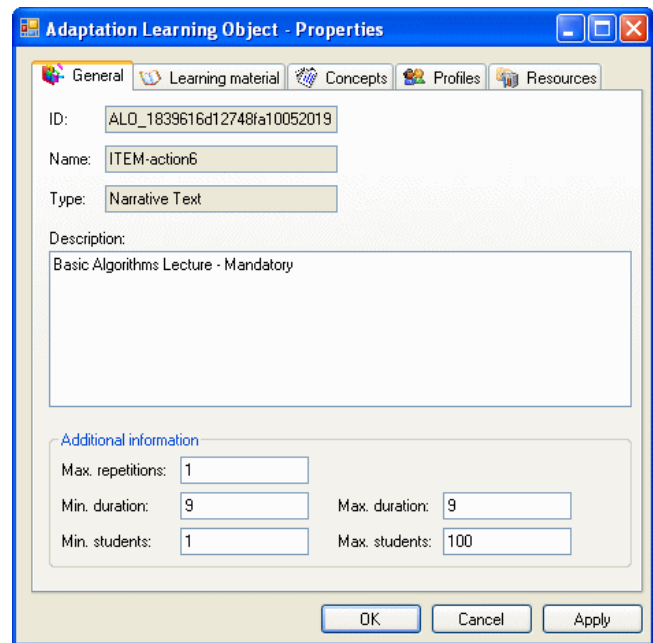


Figure 7: Input form with information about the learning object 'Basic Algorithms Lecture'.

Use of Plans

Each plan generated by a domain-independent planner, using a domain and a problem as described above, represents the learning design that best suits the student whose characteristics are modeled in the problem. Learning platforms include tools for executing the designs when the design is represented in a specific language. For example, dotLRN interprets IMS-LD, whereas Moodle has its own templates. We have implemented two more translators: from the plans of non-hierarchical planners to IMS-LD and from the plans of a hierarchical planner to the Moodle templates.

The first translator compiles a plan into an IMS-LD that dotLRN is able to execute. This is a zip file that contains the input resources (learning objects), as well as the learning design (output of the planner). The first part is basically a copy of the input resources that is usually contained in a directory. The second part consists of an XML file. Next, we describe the main fields of this file (some others are easily filled in from this information) and how they are generated from the domain, problem and plan:

- Objectives: these are filled with the name of the goals of the problem. The problem goals are always the main effects of the final action of any domain.
- Prerequisites: these are the initial conditions on previous knowledge that is required to follow this course. They are the links to learning objects of other courses, or objectives of other courses. This will allow us to perform multiple course planning in the future.
- Roles: in this case, the only role is that of the learner, the student for whom the learning design is generated.

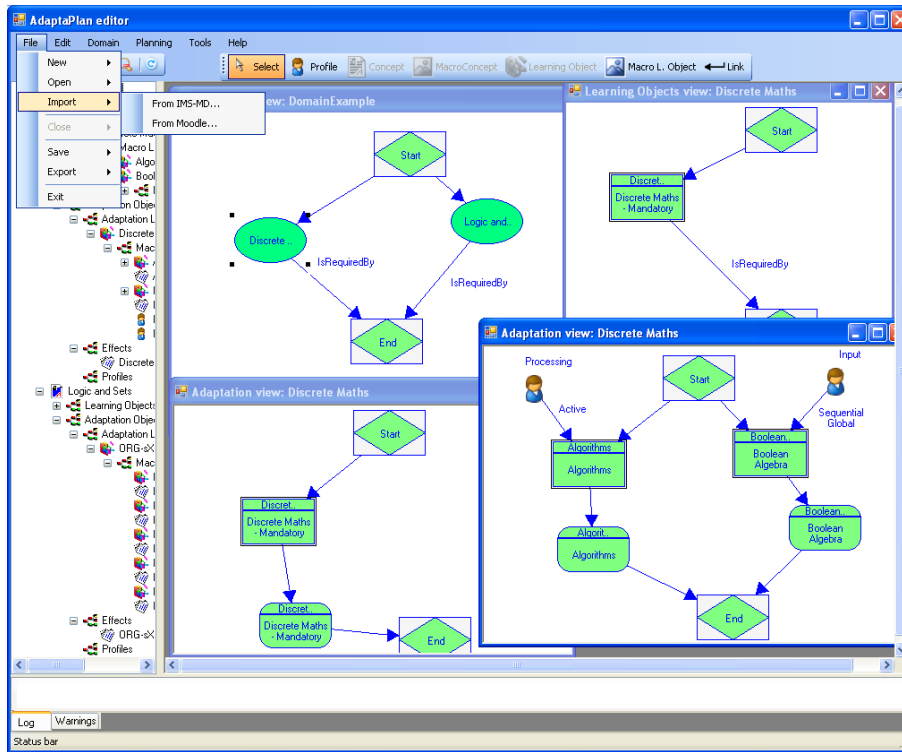


Figure 6: Snapshot of the tool.

- **Activities:** for each action in the plan, an IMS-LD activity entry is generated. This field is just an enumeration of those actions, and each one also includes a link to the corresponding learning object. Fictitious actions are omitted.
- **Activity-structure:** this IMS-LD concept relates to the plan itself. So, here the sequence of actions in the plan is represented as such. Given that the standard allows other control structures, such as conditional plans with branches, in the future we will study how to generate conditional plans and the effect it has on the fact that students follow different alternatives.
- **Resources:** for each learning object in the input IMS-MD, a resource, that can be or not used by the plan, is defined.

Once the translator generates the zip file, it can be uploaded to dotLRN and be used by any student after a sequence of bookkeeping activities: defining the student's preferences, defining relevant roles of the course, initiating the execution of the course and so on.

The second translator compiles the hierarchical plan into a Moodle template. This template describes an XML file which is automatically related with a course by Moodle. The XML document contains several items with a student's identifier and a related action, as in next lines.

```
<item>
  <studentId> student1 </studentId>
  <actionId> ITEM-action1 </actionId>
</item>
<item>
  <studentId> student1 </studentId>
  <actionId> ITEM-action6 </actionId>
</item>
```

The actions related with each student's identifier are not subjects of a course, but learning activities, i.e. durative actions which were previously stored in Moodle database using IMS-MD standard. The order in which items appear in the document correspond to the plan with the learning activities sequencing obtained for each student. Internally, hierarchy of subjects is provided by Moodle according to the information previously stored.

Moodle permits us to deal with collaborative plans. If items of several students are interspersed, then dependency between actions of a previous student and the next one must be taken into account by the platform. Obtaining collaborative plans to take advantage of this characteristic is a task to be done in a short future.

Conclusions and Future Work

E-learning is about designing a sequence of learning activities a student needs to perform in order to complete a course. Principally, this involves three main issues: course definition, student's learning information and learning design execution. There are languages to represent all of them based on XML schemata, but an important effort needs to be done to be fully automated.

This paper has proposed a three-approach procedure to interpret and translate e-learning tasks into automated planning. A course definition is represented as a planning domain, the student's learning information as a planning problem for that domain and the learning design as the plan generated by a domain-independent planner when solving that

problem. We have implemented translators from the corresponding e-learning languages into three different kinds of planning domains and problems. These three domain reasoners allow different planners to automatically generate valid learning designs in a few seconds. However, we have detected three main drawbacks. First, e-learning languages are too generic and try to cover too many aspects, making the implementation of general and suitable translators for all LMS very difficult. Second, in spite of the expressive power of e-learning languages, there are still few aspects that cannot be represented and are essential in P&S. For example, the definition of the resources involved in tasks, their costs, the temporal constraints on availability and how these resources are to be managed are important lacks in e-learning languages. Finally, our domain modelling allows us to generate valid learning designs, but they cannot guarantee optimal plans in terms of utility to the student.

In the future we want to find solutions to overcome the previous drawbacks by addressing two parallel lines. Firstly, we are interested in coming up with more expressive models of actions for planning e-learning activities. This will increase the opportunities to: i) deal with course composition, and ii) validate and resolve courses with similar but incommensurate learning objects. Secondly, we want to extend the tool to assist the course designer in making sure that the same naming conventions are used. As one of the anonymous reviewers suggested, the adoption of a common ontology can be very useful (Kontopoulos *et al.* 2008).

References

- Baldiris, S.; Santos, O.; Barrera, C.; J.G., J. B.; Velez, J.; and Fabregat, R. 2008. Integration of educational specifications and standards to support adaptive learning scenarios in adaptaplan. *Special Issue on New Trends on AI techniques for Educational Technologies. International Journal of Computer Science and Applications (IJCSA)*.
- Brusilovsky, P., and Vassileva, J. 2003. Course Sequencing Techniques for Large-Scale Web-Based Education. *International Journal Continuing Engineering Education and Lifelong Learning* 13(1/2):75–94.
- Camacho, D.; R-Moreno, M.; and Obieta, U. 2007. CAMOU: A Simple Integrated e-Learning and Planning Techniques Tool.
- Castillo, L.; Fernández-Olivares, J.; García-Pérez, O.; and Palao, F. 2006. Efficiently handling temporal knowledge in an htn planner. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006*, 63–72.
- Castillo, L.; Morales, L.; Gonzalez-Ferrer, A.; Fdez-Olivares, J.; Borrajo, D.; and Onaindia, E. 2009. Automatic generation of temporal planning domains for e-learning problems. *Journal of Scheduling*. Accepted.
- Felder, R. M. 1996. Matters of style. *ASEE Prism* 6(4):18–23.
- HoneyAlonso. 2002. Honey alonso learning style theoretical basis in spanish. Available at <http://www.estilosdeaprendizaje.es/menuprinc2.htm>.
- IMS-GLC. 2001-2009. Ims specifications. Available at <http://www.imsglobal.org>.
- Kontopoulos, E.; Vrakas, D.; Kokkoras, F.; Bassiliades, N.; and Vlahavas, I. 2008. An ontology-based planning system for e-course generation. *Expert Systems with Applications* 35:398–406.
- LOM. 2002. Draft standard for learning object metadata. IEEE. 15 July 2002. 6 Oct. 2007. Available at http://ltsc.ieee.org/wg12/files/LOM_1484_12.1_v1_Final_Draft.pdf.
- Mohan, P.; Greer, J.; and McCalla, G. 2003. Instructional Planning with Learning Objects.
- Peachy, D., and McCalla, G. 1986. Using Planning Techniques in Intelligent Tutoring Systems. *International Journal of Man-Machine Studies* 24(1):77–98.
- Ullrich, C., and Melis, E. 2009. Pedagogically Founded Courseware Generation Based on HTN-planning. *Expert Systems with Applications* 36(5):9319–9332.
- Vrakas, D.; Tsoumakas, G.; Kokkoras, F.; Bassiliades, N.; Vlahavas, I.; and Anagnostopoulos, D. 2007. PASER: a curricula synthesis system based on automated problem solving. *Int. Journal on Teaching and Case Studies, Special Issue on "Information Systems: the New Research Agenda, the Emerging Curriculum and the New Teaching Paradigm"* 1(1/2):159–170.
- Wisher, R. 2009. *Sharable Content Object Reference Model(SCORM) 2004 4th Edition Documentation Suite*. ADL.

JABBAH: A Java Application Framework for the Translation Between Business Process Models and HTN

Arturo González-Ferrer

Centro de Enseñanzas Virtuales
University of Granada
c/ Real de Cartuja 36-38, Spain 18071
arturogf@ugr.es

Juan Fernández-Olivares and Luis Castillo

Departamento de Ciencias de la Computación e IA
University of Granada
c/ Periodista Daniel Saucedo s/n, Spain 18071
{faro, l.castillo}@decsai.ugr.es

Abstract

HTN planning paradigm has been widely used during the last decade to model and solve planning and scheduling problems. Even so, little research have been oriented to represent and generate these planning domains automatically with the help of software tools. In this paper we present an extensible software framework directed to cover this goal, proposing an innovative knowledge engineering method that transform a workflow graph into an equivalent nested process model, which simplifies the subsequent mapping to HTN-PDDL. Some results in the field of e-learning management are also exposed.

1. Introduction

The difficulty of writing Planning and Scheduling (P&S) domains is well known by the AI community, and usually a lot of human effort is necessary to explore the real problems that are likely to be modeled, capturing the acquired knowledge with accuracy into a planning domain, that is usually coded using non-intuitive languages as PDDL (Long and Fox 2003) or any of its flavours. Despite being a difficult task, still little work has focused in helping to do it in a convenient way. However, this kind of problem is specially suitable for the Knowledge Engineering (KE) discipline.

Even though there are already some approaches (Simpson, Kitchin, and McCluskey 2007; Vaquero et al. 2007; Bouillet et al. 2007) devoted to the field of KE for P&S, they are rather directed to be helpful for planning experts (dealing with the modeling of world objects and actions). The approach here presented is more aligned with (Barták et al. 2008), and deals with the automatic generation of planning domains from expert knowledge introduced by using existing tools and standard languages that are close to IT architects and organization stakeholders.

Concretely, we propose in this paper the development of a software framework that is able to automatically map this acquired knowledge into P&S domain and problem definitions. Our work is focused on the reuse of Business Process Modeling (BPM) tools (Havey 2005). They are able to deal with goals and tasks specification, environmental analysis,

design, implementation, enactment, monitoring and evaluation of business processes (Muehlen and Ho 2006).

Exploiting the common field between BPM and P&S is interesting, not only because we could use robust, formal and mature software tools to capture the knowledge we want to represent into the corresponding P&S domain (data, activities, rules, performers, etc), but also because we could reuse existing process models that have already been designed by software architects for a specific problem, offering P&S as a possible solution for a very wide range of application fields, taking as input a pre-existing process model. Moreover, introducing an automated P&S system into the BPM life cycle (Muehlen and Ho 2006) of a company, capable of both interpreting and reasoning about an initial workflow model representation, can provide support for decision making on key issues like tasks organization, resources allocation, or even requirement and use cases analysis.

The work here presented is based on the hypothesis that the process structure, the ordering constraints and the control flow structures of a BPM model, can be captured by an HTN knowledge representation language. Hence, we could use an state-of-art HTN planner that takes this domain representation as input and use its output in order to obtain action plans helpful for management tasks. This existing equivalence between both BPM and HTN made us consider the development of a software tool to carry on the transformation of one model into the other.

So, the contribution of this paper is that, having a BPM design of any organizational process, modeled under some previous requirements, we can extract the corresponding HTN planning domain and problem files directly from the original process diagram without the interaction from any planning expert. Moreover, this is done keeping the process control-flow restrictions as well as the data model reflected on it. Furthermore, some experiments have been carried on to support the organization and management of e-learning course development requests, allowing to check the usefulness of our approach.

The paper is structured as follows. Section 2 introduces some concepts and technical background about the problem. Section 3 details the Knowledge Engineering procedure developed. Section 4 exposes some requirements on the input process diagram. Section 5 exposes some results and Section 6 describes some conclusions and lessons learned.

2. Technical Background

In this section we introduce the BPMN/XPDL business modeling languages, which has been chosen for our work. We also introduce *workflow patterns*, in order to convey why we decide to use them as the main background concept for our transformation, and which are also the basis for choosing the *Hierarchical Task Network* (HTN) paradigm to model the resulting planning domain.

2.1 BPMN/XPDL

XPDL stands for *XML Process Definition Language*. The goal of XPDL (WfMC 2008) is to store and exchange a process definition, offering an XML serialization of the *Business Process Management Notation* (BPMN) graphical representation of the process diagram. The main advantage of using XPDL as modeling language is that it is commonly used among business analysts, and it can be used to represent the organization activity easily. There are a lot of modeling tools that already incorporate XPDL natively or as an additional plug-in. Although some used directly BPEL (*Business Process Execution Language*) (García-Bañuelos 2008) to design the process diagram, ideally this should be done in XPDL, as it was thought for modeling, not for execution (Palmer 2007). Next, an overview of the XPDL entities and attributes considered in our work is exposed:

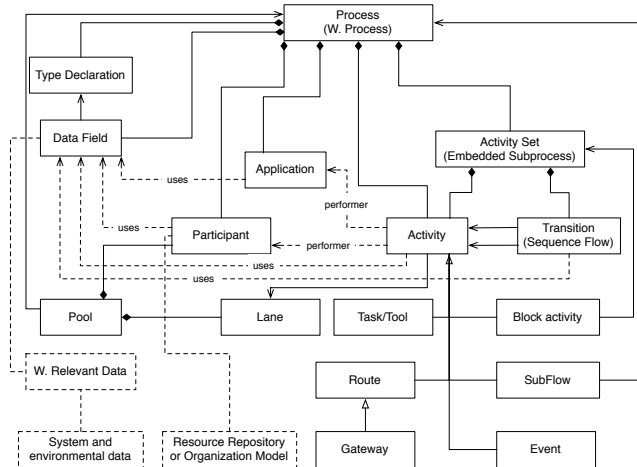


Figure 1: XPDL metamodel

Activities. They comprise a logical, self-contained unit of work, which will be carried out by *participants* and/or computer applications. Activities are related to one another via *transitions*.

Gateways. Special activities used to implement decisions that affect the sequence flow path through the process.

Transitions. Transitions may be either *conditional* (involving expressions which are evaluated, driving the sequence flow path) or *unconditional*, and may result in the sequential or parallel operation of individual activities.

Lanes. They denote areas or departments of the organization or process, and every activity contained within that particular lane will be done within that area. So, we can use

them to establish a way to define capabilities that a particular activity requires. So, a participant also needs to belong to a specific lane, in order to be able to complete this activity.

Participants. They can be differentiated by their definition scope: a) Those defined at *process level* can be considered as possible resources that can be allocated to one or more activities. They will have associated precondition definitions, established by the lanes the participant belongs to. This membership can be specified as an *extendedAttribute* tag for every participant at modeling time. b) Those defined at *activity level* will force that specific activity to be done by the participant specified.

Parameters and DataFields. These entities are used to define the process data model. We can control flow from a gateway by creating a parameter (which has an associated data type) and values to be used in an associated rule. On a conditional transition exiting the gateway, we can specify that the transition will be followed only when parameter values match the expression specified in the rule. Information that is internal to the process is represented as *Data Fields* and information required outside the process is represented by *Parameters*.

XPDL modeling tool We have used TIBCO Business Studio, a modeling tool that use the BPMN graphical notation, is offered for free and includes support for latest release of XPDL v2.1.

Taking advantage of our expertise on e-learning management, we have analyzed a specific organisational process to manage the collaborative creation of e-learning courses within the virtual learning center of the University of Granada. This process implies the participation and interaction of different roles (instructional designers, graphic designers, HTML developers, sysadmins, tutors, etc.), and it has an explicit activity time ordering (see figure 2). The transformation of this process into a P&S domain will help to make the most of the e-learning center workload, also offering an estimation (to both managers and customers) about the time needed to deploy the requested course.

2.2 Workflow Patterns

Workflow Patterns (van der Aalst et al. 2003) are those generic structures found in a graph representation that capture frequently-used relationships between tasks in a process, and that are typically nested to form the whole process model. The XPDL language can represent some of them, although it lacks of some power for the correct representation of complex patterns (van der Aalst 2003). Therefore, only the most basic ones are going to be considered throughout the paper, those that can be well represented and are expressive enough for the definition of most processes: *serial*, *parallel split-join*, and *parallel exclusive-OR* (usually used to capture conditional structures). As shown later, our mapping process will work by detecting these workflow patterns in a process model and translating each of them to its corresponding HTN structure.

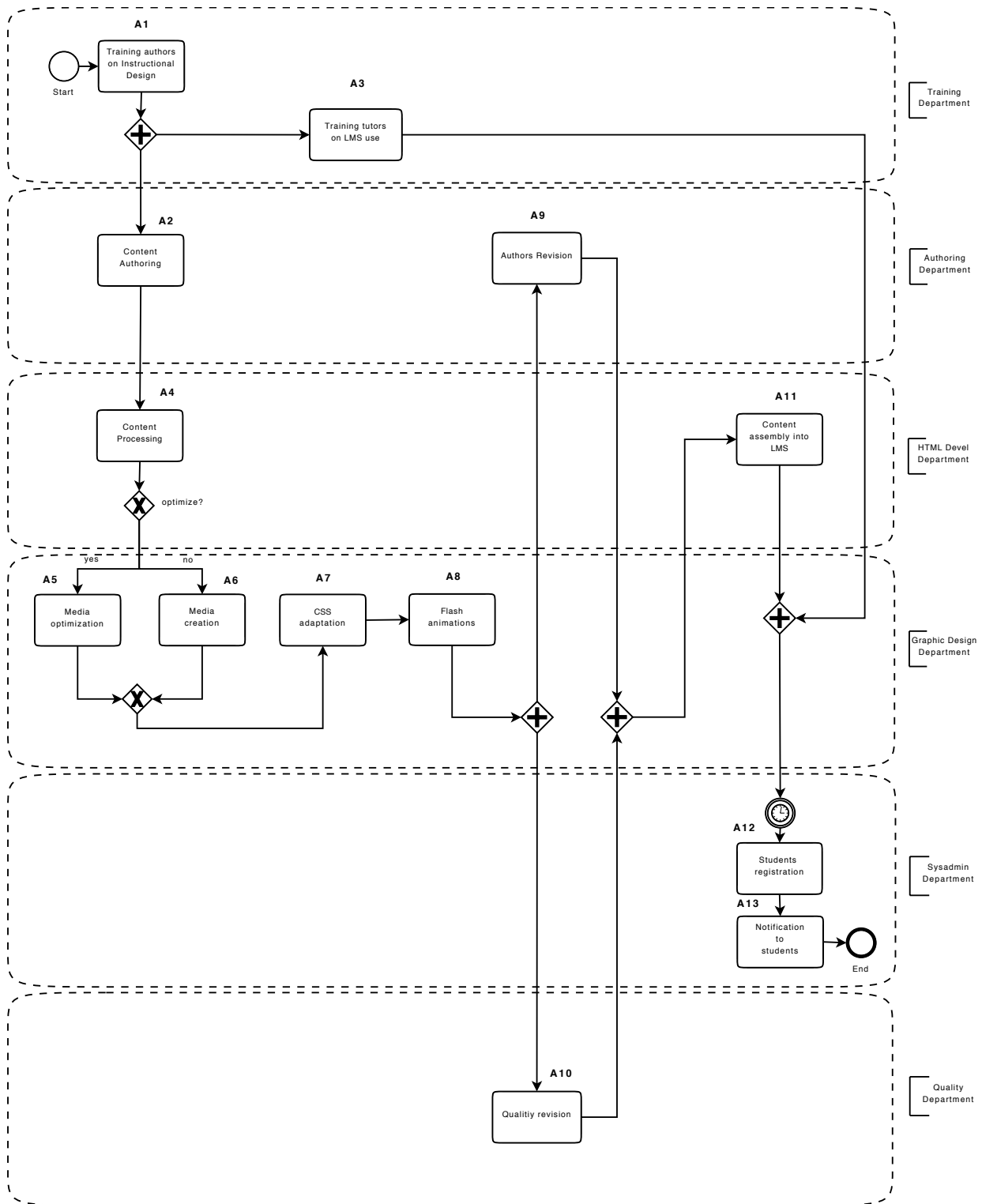


Figure 2: An example organisational process modeled using the BPMN graphical notation, describing the development of courses in a specific e-learning center. On the right side, text annotations boxes show the lane represented with point lines on the left; the boxes, named A1-A13, represent the activities; the arrows represent the transitions; the elements with the symbols 'x' and '+' are the exclusive-OR and parallel gateways; the participants are not represented graphically, and can only be explored within the BPM modelling tool. This model can be serialized as a XPD L stream, and later analyzed by JABBAH.

2.3 Hierarchical Task Network Planning

HTN planning domains are designed in terms of a hierarchy of compositional activities. Lowest level activities, named actions or primitive operators, are non-decomposable activities which basically encode changes in the environment of the problem. On the other hand, high level activities, named tasks, are compound actions that may be decomposed into lower level activities. Every task may be decomposed following different schemas, or methods, into different sets of sub-activities. These sub-activities may be either tasks, which could be further decomposed, or just actions. HTN paradigm is able to represent the hierarchical structure of the domain and it is also expressive enough to capture the expert knowledge in order to drive the planner to a desirable solution.

HTN-PDDL notation The HTN planning domain language used in this work is a hierarchical extension of PDDL (Long and Fox 2003) that uses the following notation.

Types, constants, predicates, functions, and durative-actions are used in the same way that in original PDDL language. In addition, the *task* element is introduced to express compound tasks. Its definition can include *parameters*, different decomposition *methods* with associated *preconditions* (that must hold in order to apply the decomposition method) as well as *tasks* to represent its corresponding lowest level task decomposition.

At the problem definition, *objects* is used to define objects that are present in the problem, *init conditions* to define the set of literals that are initially true, and *task-goals* to define the set of high level tasks to achieve.

Compound tasks, decomposition methods and primitive actions represented in a planning domain mainly encode the procedures, decisions and actions that are represented in the original BPM model. More concretely, the knowledge representation language, as well as the planner used, are also capable of representing and managing different workflow patterns present in any BPM process model. A knowledge engineer might then represent control structures that define both, the execution order (sequence, parallel, split or join), and the control flow logic of processes (conditional and iterative ones). For this purpose the planning language allows sub-tasks in a method to be either sequenced, and then they appear between parentheses (T1,T2) , or splitted, appearing between braces [T1,T2].

We have used the IACTIVE™ planner for this paper, as it is already known how to translate workflow patterns for semantic web services composition (J.Fernandez-Olivares et al. 2007), as well as its adaptation to temporal knowledge (Castillo et al. 2006). In addition, it has already been used in several applications (Castillo et al. 2007; Fdez-Olivares et al. 2008).

Next section describes the KE procedure needed to extract the P&S domain and problem from a process diagram.

3. Translation Overview

Roughly speaking, what we want to do is to identify common patterns in a workflow model (which can be clearly seen as a graph), so that we can generate a tree-like structure,

much similar to HTN domains. This entails the resolution of two main problems: a) analyze the workflow model to get a corresponding graph, b) interpret the resulting graph, mapping it to a tree-like structure. To do this, a collateral challenge, out of AI Planning scope but necessary, is the transformation of the graph into a tree-like structure, which has been done using an algorithm described later at section 3.2.

So, our Knowledge Engineering proposal consists of three different stages (see figure 3) which are necessary in order to develop a sound approach for the problem of capturing knowledge from a BPM model that will finally be represented into an HTN planning domain:

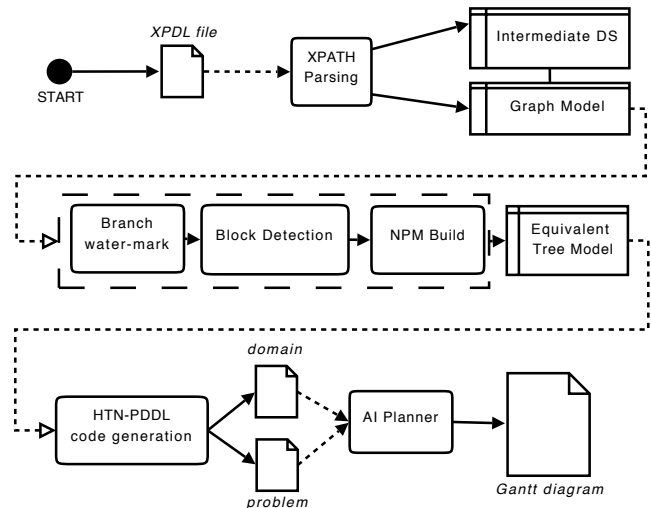


Figure 3: The different stages of the translation process

a) Firstly, we need to parse the source XPD document, storing it into an intermediate data structure and graph model that can be easily managed throughout the next stages.

b) Then, we need to detect the different blocks of workflow patterns (parallel and serial blocks), distinguishing their kind from the knowledge acquired in the previous parsing stage, and build up an equivalent tree-like model. This is carried on by arranging those workflow patterns hierarchically, but also keeping the semantic information (about control flow and decisions) present in the process diagram.

c) Finally, we need to do a code generation phase, where we analyze the tree model that has been populated previously, trying to generalize common patterns found in the graph (i.e. serial or parallel split-joins patterns are always coded in the same way), and writing the HTN-PDDL code that corresponds to the tree-graph fragment analyzed.

Next, we proceed to give further insights on the development of these 3 steps.

3.1 Mapping to an Intermediate Graph Model

This step takes as input a standard XPD file (previously exported from the BPM modeling tool used), reading it by using XPATH(W3C 1999) parsing technology, which allows searching only the XML entities we are interested in. Then, it obtains as result a graph in which every node represents

an activity (or gateway) and every edge represents a transition between two activities (conditional or unconditional, as exposed previously at BPMN/XPDL subsection). Furthermore, we will keep all the relevant information about participants, lanes, parameters, etc. by using an associated data structure that will be used throughout the mapping process.

It's important to note that both *gateways* and *transitions* elements are the main elements that drive the control flow in XPDL workflow graphs. They are also the main elements considered for our work and, from a workflow patterns perspective, they will define how to map organizational processes into planning and scheduling domains, so that the future plans developed by our software framework will mainly act according to their definition in the process diagram.

At this point, we have developed a graph model of the original process diagram that can be further worked out in order to achieve our goal.

3.2 Block Detection: Mapping to a Tree Model

The goal of this stage is to build an equivalent tree model from the graph obtained in the previous phase. Our work for this level of the mapping process is based on previous research done in (Bae et al. 2004), where an algorithm was developed to generate a tree representation of a workflow process which was later used to derive ECA (event-condition-action) rules, helpful for controlling the workflow execution.

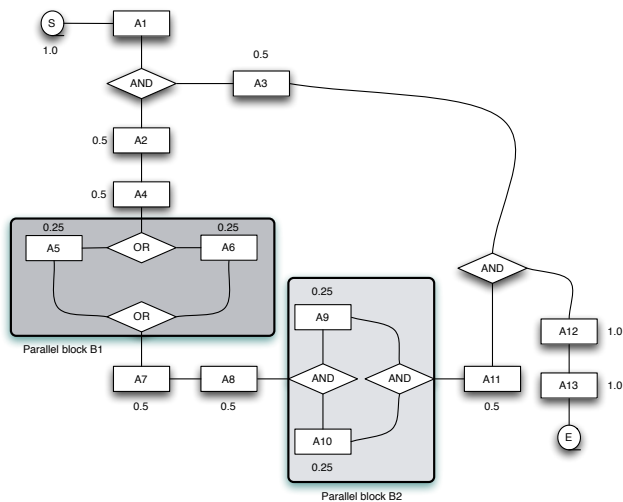


Figure 4: Part of the block detection algorithm applied to graph of figure 2. We can appreciate the branch-water mark procedure as well as the workflow pattern detection.

The tree representation obtained is called a *nested process model* (NPM) (Bae et al. 2004). It describes how to build up a process model in a top-down manner, representing a root node which is decomposed into a set of subprocesses and tasks, and so on. It adopts and generalizes a hierarchical model, allowing to express a parent-child relationship between subprocesses. We adapted this tree-like model to our particular problem, the representation of P&S domains, taking into account the control-flow information included in

gateways and *transitions*, as expressed before, adding additional information about the process and data model as well. Thus, the algorithm for block detection described has the next three steps:

1. The first step is to mark every node of the graph with a weight, based on a *branch-water* procedure (see figure 4). It simulates a pipeline network carrying water, being 1.0 the quantity of water poured at the start node, and branching the quantity through the pipe. If the water-level at a specific node is l , and the flow is branched into k alternatives, then l/k quantity of water is propagated through every alternative node. The water-level measure is the method used to identify the most inner block in the graph, which is important for the next steps. It allows to build a NPM in a bottom-up approach, as exposed next.

2. The second step is to identify serial and parallel workflow patterns (we call them *blocks* here) consecutively, using the weight to identify the most inner block. Every time we identify a serial or parallel block, we substitute all the nodes that constitutes that block with a special SERIAL_BLOCK (SB) node or PARALLEL_BLOCK (PB) node, obviously linking the new node with the preceding and successors nodes, in order to keep the graph being *connected* and *two-terminal* (it has an unique start and end node). If the workflow graph fulfills the requirements commented before, it is easy to see that this process ends having an unique SB or PB block node that constitutes the root node for the nested process model we want to build up.

3. Finally, if the root node is now expanded using the nodes it grouped originally, placing them as children, and we do this operation recursively with every SB or PB block node, we can draw the new tree-like structure that we were searching for. This is done as a typical breadth-first search algorithm. The result of the procedure constitutes what we called the *nested process model* (NPM) of the original BPM diagram, using a bottom-up approach (see figure 5). Observe that those nodes with minimum weight lay at the lower levels, and go up consecutively as their weight increase (this is the reason to look first for the most-inner blocks).

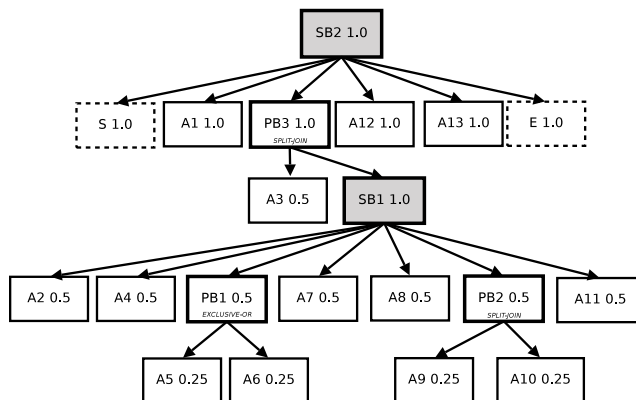


Figure 5: An example of a Nested Process Model generated from the previous BPMN process model. Note that leaf nodes correspond to activities and non-leaf nodes correspond to serial and parallel blocks

There are some considerations we must stress on. Firstly, we need to keep all the knowledge acquired in the parsing stage (lanes, participants, parameters and its association to gateways, etc.), being important to have custom implementations of graph nodes and edges. Second, we must keep the nodes that gave rise to the new special block nodes, introduced in the second step. And last, the knowledge present in gateways must be transferred to the new parallel block nodes (i.e., we must transfer the type of gateway, the parameters/rules that drives the flow, etc...), as soon as those gateways nodes are not going to be present on the new built NPM (but their semantic is maintained, including the relevant information mentioned into the newly created PB nodes). The algorithm complexity is $O(n^2)$, being n the number of edges of the workflow graph (Bae et al. 2004).

3.3 HTN-PDDL Code Generation

Now we give specific details about how we generate the HTN planning domain and problem files, taking as basis both the tree-like structure (the NPM, figure 5) and intermediate data structures, already developed in the previous phases.

As opposite to the bottom-up approach followed to create the NPM, the generation of HTN-PDDL code is going to follow a top-down approach. It is clear to see that, as we already have a tree-like model, all we need to do is a breadth-first search over the NPM, considering the information relevant to every node (described along this section), and considering also some patterns related with some kind of nodes (see figure 6).

Next, we expose how to express the different elements of an HTN-PDDL domain and file definitions. We also expose the underlying conceptual mapping from XPDL source elements, reflecting both the *process* and *data models*.

Domain name and requirements. These HTN-PDDL blocks are encoded as const strings (the requirements section is considered always the same).

Types. The basic types considered are those that are going to be useful in any planning domain: *activity*, *participant* and *lane*. Of course, parameters data types must be also generated (see the corresponding item below).

Constants. XPDL *activities* and *lanes* will be mapped as HTN-PDDL constants, which are going to be used later throughout the domain and problem files. This is automatically extracted from the intermediate data structure obtained in section 3.1, and they will be coded in lowercase characters (i.e. activities will be coded as a_x , being x the activity id).

Predicates. We must include, at least, two default predicates, useful in almost any process model mapping:

1) (*belongs_to_lane ?p - participant ?l - lane*). This predicate is used to express which lanes the participant belongs to, in other words, what abilities correspond to every participant. It will be used to encode both initial conditions of the problem (one predicate instance for every ability a participant possess) and preconditions for the durative actions (a precondition for every activity within a lane).

2) (*completed ?a - activity*). This predicate will encode initial conditions of the problem as well as preconditions and effects for durative actions.

There are also some predicates that should be added dynamically, those that are related to parameters/rules matching pairs (described later at *parameters* item).

Durative Actions. Every activity of the process diagram corresponds to a leaf-node in the NPM and it is mapped as a *primitive durative action* on the planning domain, as a fragment following the next pattern:

```
(:durative-action Ax
:parameters(?w - participant)
:duration(= ?duration D)
:condition(belongs_to_lane ?w L)
:effect (completed ax))
```

For every k , being k an activity of the NPM, a corresponding durative action Ax is generated, being x the id number, whose effect is the completion of the activity ax (which was coded as a constant previously, and the associated predicate named *completed*). The duration of the activity, D , which is coded in XPDL using an *extendedAttribute* tag, and the lane the activity belongs to, L , are mapped directly from the corresponding XML attributes present on the XPDL activity.

Realize that order constraints among activities, in non-hierarchical planning paradigms, are coded through the use of preconditions in durative actions, being necessary an extra cause-effect analysis. However, in HTN planning paradigm, order constraints are directly mapped into the corresponding syntactic structures developed to that end. So, our approach does not need to abuse of precondition definition, simplifying the process, as exposed next in the definition of compound tasks.

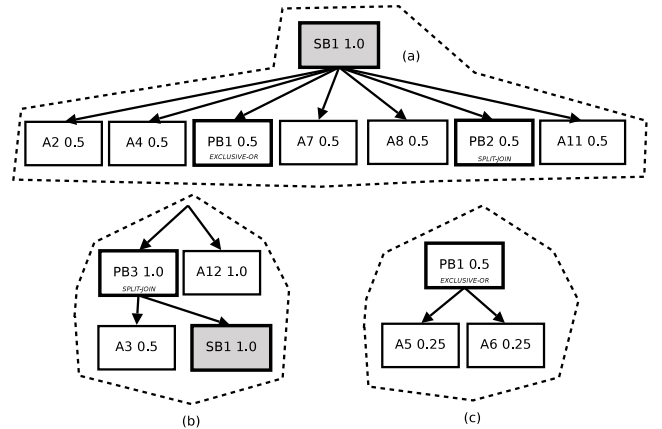


Figure 6: Different patterns identified in the NPM representation that are mapped as HTN compound tasks. (a) a serial block, (b) a split-join block, (c) a exclusive-OR block

Compound Tasks. The HTN-PDDL compound tasks are mapped from those intermediate nodes (non leaf-nodes) of the Nested Process Model. These nodes always correspond to workflow pattern blocks (see figure 6), that are actually specifications of different tasks with control flow mechanisms that are coded as order constraints (sequential/parallel) or as alternatives (if-then):

1. **Serial Blocks.** One activity must be executed after other, following a sequence in time. This can be expressed in HTN-PDDL as a sequence of primitive actions and/or tasks surrounded by parentheses. Next example represents the fragment of figures 6(a) and 7:

```
(:task SB1
  :parameters ()
  (:method blsb1
  :precondition ()
  :tasks ((A2 ?w1) (A4 ?w2)
  (PB1 ?optimize) (A7 ?w3)
  (A8 ?w4) (PB2)
  (A11 ?w5))))
```



Figure 7: a serial block fragment

Note that, on one hand, durative actions A_x must be generated with the corresponding parameter $?w_y$ which express a resource that has to be allocated at planning-time (the participant y is assigned the activity x). On the other hand, compound tasks that are also part of the decomposition can be generated with or without parameter, representing the formal *parameter* which drives the flow in the original XPDL process diagram (i.e. the parameter 'optimize' in the example above controls which flow to follow, as exposed next).

2. **Parallel Split-Join Blocks.** They represent a branch of the process flow into two or more flows (*split*) that are carried on simultaneously (without specifying which of them should be executed first), and that finally converge into the same flow again (*join*). These parallel split-join blocks are represented in HTN-PDDL enclosed by square brackets, as the following case, that represents the fragment of figures 6(b) and 8:

```
(:task PB3
  :parameters ()
  (:method blpb3
  :precondition ()
  :tasks ((A3 ?w1) (SB1)
  (A12 ?w2))))
```

Note that A12, the *right brother node* of PB3 in figure 6(b), is the activity executed after the *join* gateway. This scheme repeats for every parallel split-join block detected in the nested process model.

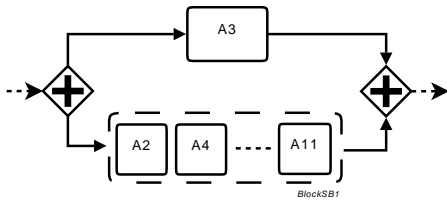


Figure 8: a parallel split-join block fragment

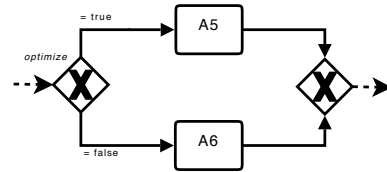


Figure 9: a parallel exclusive-OR block fragment

3. **Parallel Exclusive-OR Blocks.** They represent blocks which flow is controlled by a gateway node which has associated both a formal parameter and a corresponding logical expression that controls which alternative flow must be followed. We generate a method for every possible alternative to follow, using the expression as precondition of the defined method, as the next example, that represents the fragment of figures 6(c) and 9:

```
:task PB1
:parameters (?x - parameter)
(:method ifA5
  :precondition (value ?x true)
  :tasks (A5 ?w1))

(:method elseA6
  :precondition (value ?x false)
  :tasks (A6 ?w1))
```

It's clear that we should also map the parameters and expressions in such a way that different kind of parameters/expressions pairs and its associated data types can be added to the framework in a future. We have already done it for boolean data type, as described next.

Parameters. Parameters are usually associated to Exclusive-OR parallel blocks, and they can be initially expressed as follows, as soon as they have been modeled as *boolean* parameters:

- add an HTN-PDDL type 'parameter'.
- add a HTN-PDDL constant for every parameter (i.e. the parameter named *optimize*).
- add a predicate (i.e. named *value*) to check boolean values (true, false).
- pass the corresponding parameter to the Exclusive-OR block wherever it is used, as done in previous example with parameter *optimize*. This is very easy, as the parameters have been already stored in the intermediate data structure.
- in the problem file, define the parameter as an initial condition of the problem. Note that parameter values should be passed to the AI planner somehow before interpreting the domain and problem files generated (i.e. it can be given by the user outside the framework).

Other data types could be included using a similar methodology, but adding more powerful rule expressions (step c) is still one of the features to be improved in the JAB-BAH framework. Besides this mapping, we also tried referring to an external organizational data model stored in UML, using some of the capabilities of the BPM modeller, as the XPDL standard supposedly supports it, but this feature was somehow experimental in the modeller and we could not

complete it. Using UML for storing the data model would be ideal, as there are already authors (Vaquero et al. 2007) that worked out a methodology to express this model in PDDL.

Objects. Every *participant* is going to be defined at the problem file as an object (of 'participant' data type). **Init Conditions.** Besides parameter values mentioned above, we must include the abilities that every participant (previously defined as object) possesses, in other words, what lane the participant belongs to (using the predicate *belongs_to_lane* described before). **Goals.** The goal of the problem definition file will be the root node of the NPM, which is always a compound task, that can be iteratively decomposed in order to generate all the process plan.

So we have described in previous sections the whole KE process followed to map a BPM model to its corresponding P&S domain and problem definitions. In the next section, we introduce some restrictions on the input process model, necessary to guarantee the correctness of our solution.

4. Input process model requirements

For the sake of the framework usability, we need to establish some requirements on the input process model, owing to the fact that not always the designed diagrams have the desired properties for later processing (any BPM expert knows about this circumstance). Thus, we have considered the next three conditions on the input process model:

a) The input process model must include an unique start node *s* and an unique end node *e*. Extrapolating this property to the equivalent graph model, it must be *two-terminal*.

b) All the gateway nodes that split the flow in the input process model, must have a corresponding gateway node that joins the flow again. Extrapolating this property to the equivalent graph model, it must be *well-structured*.

c) The input process model must be connected between elements from start to end nodes, so that for every node, always exists a path from *s* to *e* that goes through that node. Extrapolating this property to the equivalent graph model, it must be *directed* and *connected*.

The proposed requirements are demanded in order to guarantee that the workflow pattern detection stage is carried on correctly (b), as well as the branch water-mark procedure included in that stage (a, c). Some inspiring works for the establishment of these requirements have been the SESE (single-entry single-exit) regions of a graph (García-Bañuelos 2008), and the *process structure tree* (Vanhatalo, Volzer, and Koehler 2009). It seems natural to delimit the sort of process models that can be worked out, as usually done in other research related to BPM (van der Aalst 1999).

Maybe, the most demanding requirement for the user are (a, b), but fortunately some transformations have been already developed in order to obtain *well-structured* graphs from unstructured ones (Vanhatalo et al. 2008) (which also eliminates the need for an unique end node). These transformations could be introduced either into BPM modeling tools or into the JABBAH framework itself, in order to increase the number of models it can analyze, being unnecessary to modify the diagram manually. Although no validation checking about the input has been developed yet in our

software tool, we would like to include it in a near future, so that it can be helpful to the IT architects during the design process by giving tips at design-time, similarly to any other simulation engine included within traditional BPM tools.

5. Experiments

The JABBAH framework described above has been developed following an object-oriented methodology. It is based on the Java graph library *jgraph*, which provides a complete and customizable library, covering to a large extent our needs for creating graph data structures, with fully customized nodes and edges implementation. Furthermore, its corresponding visualization libraries, *jgraph* and *jgraphlayout* help us to develop a visual interface, allowing to see the original graph extracted from the source XPDL document as well as its corresponding NPM.

We have done some experiments by using JABBAH over the process shown at figure 2. It represents the whole process to develop and deploy a specific course within the e-learning center at the University of Granada. Having an incoming course request, as well as some available workers with different capabilities each, we want to assign an activity to every worker, so that we can have a plan over time that tells the e-learning managers information about the workers allocation as well as the end time of the whole course development, which allows to do an anticipated decision-making upon the course request.

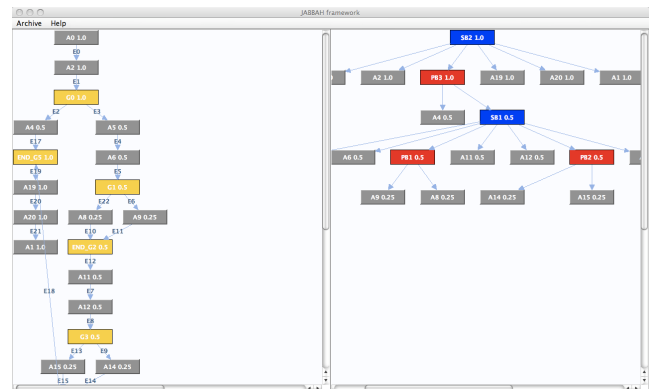


Figure 10: Screenshot of JABBAH framework running over the example process

The first thing to do is to design the process diagram (figure 2) with the help of the TIBCO Business Studio process modeler. The only inputs the managers have to provide once the process model has been designed are a) the estimated duration of every activity and b) the abilities every specific worker possesses. Both requirements are usually known by managers. Then, we export the diagram as an XPDL file, and import that file using JABBAH, which will show both the process diagram (on the left) and the corresponding nested process model (on the right), as shown in figure 10. Figures 4, 5 and 6 described the first two stages of our KE procedure applied to the mentioned process, and how the NPM is built up. At the same time, the HTN-PDDL

translation process is carried on over the NPM, saving the domain and problem files in an output directory. We can then interpret those files by using the IACTIVE™ planner, and get the corresponding plan.

Using different abilities assignment for real workers at the e-learning center, as well as estimated activities duration, we have checked the viability of the output plans. An example assignment as the following: Emilio (*training*), Storre (*authoring*), Miguel (*html*), JoseBa (*graphic*), Arturo (*admin*) and FMoreno (*quality*), would result on the output plan shown as a Gantt diagram at figure 11.

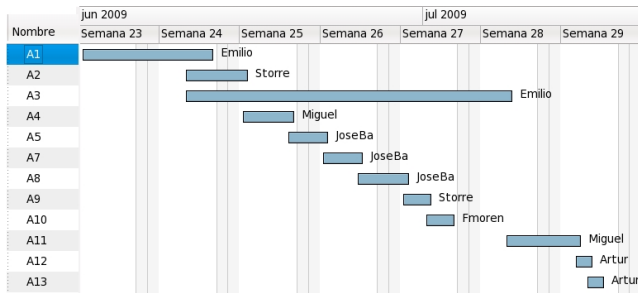


Figure 11: Output plan as a Gantt diagram

So, we end up checking that the KE procedure developed is useful within this particular scenario, and it can be extrapolated to multiple scenarios, as soon as the process diagram is represented in the terms specified in this paper.

6. Conclusions and Lessons Learned

This paper has made some innovative contributions in order to overcome the traditional drawbacks of P&S modeling, specifically for the HTN paradigm. Mainly, a sound KE procedure has been developed in order to express BPM process diagrams as HTN P&S domain, building up an intermediate data structure that organises the source process diagram as a nested process model, simplifying the subsequent transformation into HTN-PDDL code. This is very useful, not only by the number of application areas it can give support to, but also because most of the processes are being modeled with BPM tools, what increases the usability of JABBAH.

What's more, our paper hints at a future direction to follow on the automatic capture and generation of HTN planning problems from organizational processes, and it could give rise to any other profitable approaches, at least in the area of Enterprise Resource Planning (ERP) and Supply Chain Management (SCM), which has already been part of previous research in P&S (Pardoe and Stone 2006). Some results were obtained in the field of e-learning management, which was very useful and interesting for the IT personal at the e-learning center of the University of Granada. What's more, JABBAH can fill some existing gaps in BPM tools, as exposed next.

6.1 Business Prospects

Understanding and estimating the time and cost to complete a product development process is a key business challenge.

Typically, managers have relied in project management tools (PMT) for planning purposes, but the interdependencies between time and resource constraints make it very difficult to analyze activity costs and resource requirements using traditional PMTs. The introduction of computed-aided Business Process Simulation (BPS) tools traditionally helped to capture the resource constraints, decisions rules and stochastic behaviour of real situations. But, while the strength of BPS tools relay in their ability to incorporate stochastic situations in the model, their use imply that, to find the best resource allocation scenario, the manager has to determine various scenarios and simulate them (Tumay 1996). First, this is not very realistic, as the simulation relays on subruns for a specific scenario which is usually not repeatable, as the constraints evolve in time. Second, BPS tools are based on trial-and-error mechanisms that don't help the manager to do the correct allocation of resources to activities. Sometimes this circumstance can be a serious problem as the constraints get harder, making difficult to find a correct assignment.

It's important to note that the resource allocation feature is still a requirement to be improved in BPM/BPS tools (Castellanos et al. 2006). The JABBAH framework directly tackle both BPS inconvenients exposed above. On the one hand, JABBAH would be used every time that a new order request was received, being more realistic, as it would evaluate the existing constraints at that specific moment. On the other hand, the traditional trial-and-error mechanism will be replaced by JABBAH, as it helps the manager to decide a good scenario, while keeping all the constraints defined in the process.

Furthermore, the results obtained by the JABBAH framework could be incorporated back into the BPM simulation engine (usually, the simulation scenario is expressed using an XPD extension), so that the manager could simulate the process with the obtained assignment. Hence, after some executions, workflow mining tools could be used to investigate how much the processes have improved by using the new technique (in terms of resources under-utilization or over-utilization, in terms of production and benefits, etc).

JABBAH could be useful at project-based and customer service-based processes, that is, processes where there are customers which ask for a product which, after the corresponding development process carried on by a collaborative teamwork made up of different humans, departments or roles (and why not, software applications or web services), is finally supplied to the customer. This could be expanded to other kind of projects as long as we improve the representation of the data model, which is one of our future challenges.

6.2 Future Work

The expression of temporal dependencies is surprisingly poorly addressed by the different BPM standards, and can be very difficult to introduce not only on the modeling side, but also in the enactment side. A specific extension called Time-BPMN (Gagné and Trudel 2009) has been created recently to, on the one hand, simplify the temporal constructs of the original BPMN, and on the other hand, allow the specification of temporal constructs that were not possible in the original BPMN. Specifically, the temporal constructs

of time points, intervals/durations, temporal constraints and temporal dependencies have been considered in this extension, which is based on Allen's interval algebra. As exposed in (Castillo et al. 2006), the HTN-PDDL extension used in JABBAH is able to correctly represent these temporal constructs. So, it seems reasonable that the next step of our work will be the consideration of this extension, and the automatic translation into HTN-PDDL of the temporal constructs that can be depicted through Time-BPMN, so that our tool acquire a better capability to express other complex scenarios.

Furthermore, we must emphasize that, though we used XPDL in our work, the JABBAH framework has been developed with the idea of *extensibility* in mind, taking into consideration future growth. That means that if we would like to use the next BPMN 2.0 specification, that supposedly will include an XML serialization itself, we could implement a different parsing method, keeping the same intermediate data structure populated correctly, as exposed previously at section 3.1. Similarly, the block detection algorithm used (Bae et al. 2004), could be substituted by other similar approaches. We would like to add the RPST (Vanhatalo, Volzer, and Koehler 2009), in order to improve the efficiency of the block detection method ($O(n)$), checking also its behaviour for P&S domain generation. Last but not least, since we used a language independent tree-like output model, we could introduce a plug-in for any different planning language, as long as they respect the HTN paradigm.

This would provide us a sandbox environment where we could test different techniques, measuring how they behave and how far will their capacity of expression go, in terms of P&S modeling. The JABBAH framework has just started and hopefully it can be tested over some other different process models, so that we can enrich its design, which still needs stressing on the improvement of the process data model, as soon as the process model and control-flow perspective have been the ones that got an intense dedication at this early stage of development.

References

- Bae, J.; Bae, H.; Kang, S.; and Y.Kim. 2004. "Automatic Control of Workflow Processes Using ECA Rules". *IEEE Transactions on Knowledge and Data Engineering* 16(8).
- Barták, R.; Little, J.; Manzano, O.; and Sheahan, C. 2008. "From enterprise models to scheduling models: bridging the gap". *Journal of Intelligent Manufacturing*.
- Bouillet, E.; Feblowitz, M.; Liu, Z.; Ranganathan, A.; and Riabov, A. 2007. "A Knowledge Engineering and Planning Framework based on OWL Ontologies". In *ICKEPS 2007*.
- Castellanos, M.; Casati, F.; Sayal, M.; and U.Dayal. 2006. *LNCS 3811*. Springer. chapter "Challenges in Business Process Analysis and Optimization", 1–10.
- Castillo, L.; Fdez-Olivares, J.; García-Pérez, O.; and Palao, F. 2006. "Efficiently handling temporal knowledge in an HTN planner". In *Proceedings of 16th ICAPS*, 63–72.
- Castillo, L.; Fdez-Olivares, J.; García-Pérez, O.; and A. González, F. P. 2007. "Reducing the impact of AI Planning on end users". In *Workshop on Moving P&S Systems into the Real World (Keynote talk)*.
- Fdez-Olivares, J.; Castillo, L.; Cózar, J.; and García-Pérez, O. 2008. "Supporting clinical processes and decisions by hierarchical planning and scheduling". In *Proceedings of SPARK 08*.
- Gagné, D., and Trudel, A. 2009. "Time-BPMN". In *Proceedings of 1st International Workshop on BPMN*.
- García-Bañuelos, L. 2008. "Pattern Identification and Classification in the Translation from BPMN to BPEL". In *Proceedings of OTM 2008*, 436–444. Springer.
- Havey, M. 2005. "Essential Business Process Modeling". O'Reilly.
- J.Fernandez-Olivares; Garzón, T.; Castillo, L.; O.García-Pérez; and Palao, F. 2007. "A Middleware for the automated composition and invocation of semantic web services based on HTN planning techniques". In *LNAI*, volume 4788, 70–79. Springer.
- Long, D., and Fox, M. 2003. "PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains". *Journal of Artificial Intelligence Research* 20:61–124.
- Muehlen, M., and Ho, D. T.-Y. 2006. *Business Process Management Workshops, LNCS 3812*. Springer. chapter "Risk Management in the BPM Lifecycle", 454–466.
- Palmer, N. 2007. *BPM and Workflow Handbook*. Workflow Management Coalition. chapter "Workflow and BPM in 2007: Business Process standards see a new global imperative", 9–14.
- Pardoe, D., and Stone, P. 2006. "Predictive Planning for Supply Chain Management". In *Proceedings of ICAPS*.
- Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. "Planning domain definition using GIPO". *The Knowledge Engineering Review* 22:117–134.
- Tumay, K. 1996. "Business Process Simulation". In *Proceedings of Winter Simulation Conference*, 93–98.
- van der Aalst, W.; ter Hofstede, A.; Kiepuszewski, B.; and Barros, A. 2003. "Workflow Patterns". *Distributed and Parallel Databases* 14(1):5–51.
- van der Aalst, W. 1999. "Formalization and Verification of Event-Driven Process Chains". *Information and Software Technology* 41(3):639–650.
- van der Aalst, W. M. 2003. "Patterns and XPDL: A critical Evaluation of the XML Process Definition Language". *QUT Technical report FIT-TR-2003-06* 1–30.
- Vanhatalo, J.; Volzer, H.; Leymann, F.; and Moser, S. 2008. "Automatic Workflow Graph Refactoring and Completion". In *LNCS*, volume 5364. Springer. 100–115.
- Vanhatalo, J.; Volzer, H.; and Koehler, J. 2009. "The Refined Process Structure Tree". *Data & Knowledge Engineering* 9(68):793–818.
- Vaquero, T.; Romero, V.; Tonidandel, F.; and Silva, J. 2007. "itSIMPLE 2.0: An Integrated Tool for Designing Planning Domains". In *Proceedings of 17th ICAPS*.
- W3C. 1999. "XML Path Language, v1.0". <http://www.w3.org/TR/xpath>.
- WfMC. 2008. "XML Process Definition Language Specification, v2.1". *WFMC-TC-1025* 1–216.

PORSCE II: Using Planning for Semantic Web Service Composition

Ourania Hatzil¹, Georgios Meditskos², Dimitris Vrakas², Nick Bassiliades²,
Dimosthenis Anagnostopoulos¹, Ioannis Vlahavas²

¹Department of Informatics and Telematics, Harokopio University of Athens, Greece

²Department of Informatics, Aristotle University of Thessaloniki, Greece

{raniah, dimosthe}@hua.gr, {gmeditsk, dvrakas, nbassili, vlahavas}@csd.auth.gr

Abstract

This paper presents PORSCE II, an integrated system that performs automatic semantic web service composition through planning. In order to achieve that, an essential step is the translation of the web service composition problem into a planning problem. The planning problem is then solved using external domain-independent planning systems, and the solutions are visualized and evaluated. The system exploits semantic information to enhance the translation and planning processes.

1. Introduction

Web services nowadays are essential parts of the World Wide Web, as they accommodate interoperability between heterogeneous systems. However, in many cases, the need for complex and integrated service functionality cannot be fulfilled by a simple atomic web service, leading to the requirement for web service composition. The task of web service composition becomes significantly difficult, time-consuming and inefficient as the number of available atomic services increases continuously. Therefore, the possibility to automate the web service composition process is proved essential.

Automated web service composition is significantly facilitated by the development of the Semantic Web, since the existence of semantic information permits composition using intelligent techniques, such as AI Planning. Semantic description of web services is accommodated through the development of a number of standards such as OWL-S [6], WSMO [11], SAWSDL [13] and WSDL-S [12].

PORSCE II aims at automated semantic web service composition through planning with semantic relaxation. The first and very significant step in this process involves translation of the web service composition problem to a planning problem. This translation takes place between the most prominent standards in each area: OWL-S [6] for semantic description of web services and PDDL [5] for definition of planning domains and problems. According to user preferences, the translation process may take into account semantics, resulting from the semantic analysis of

the domain; if so, semantically equivalent or relevant concepts are also included, in order to cope with cases when no exact plans can be found. The result of the transformation process is a fully formulated planning problem which incorporates all the required semantic information. PORSCE II consequently exports the planning problem to PDDL and invokes external planning systems to acquire plans, which constitute descriptions of the desired complex service. Each plan is evaluated in terms of statistic and accuracy measures. Finally, the system integrates a visual component which accommodates plan visualization and modification.

The rest of the paper is organized as follows: Section 2 discusses some related work, Section 3 provides an overview of the OWL-S standard, while Section 4 outlines the system architecture. Section 5 elaborates on the translation process, including semantic analysis and relaxation performed in the system. Section 6 presents the rest of the system operations. Section 7 presents a case study and performance evaluation and finally, Section 8 concludes the paper and poses future directions.

2. Related Work

One of the first systems that attempted automatic web service composition is SHOP-2 [15]. The system uses services descriptions in DAML-S, the predecessor of OWL-S, and performs HTN planning to solve the problem. The disadvantage of this approach lies in the fact that the planning process, due to its hierarchical nature, requires given decomposition rules, or methods, as they are referred to, which have to be encoded in advance with the help of a DAML-S process ontology.

OWLS-Xplan [16] uses semantic descriptions of web services in OWL-S to derive planning domains and problems, and then invokes a planning module called Xplan to generate the complex services. The system is PDDL compliant, as the authors have developed an XML dialect of PDDL called PDDXML. However, semantic information provided from domain ontologies is not

utilized; therefore the planning module requires exact matching for service inputs and outputs.

Other approaches for automatic web service composition are not further discussed here either because they do not deal with the important issue of translating semantic web service descriptions into planning terms or because they require some prior, domain-specific knowledge of the composition issues.

The main advantage of the proposed framework with respect to the aforementioned systems is the extended utilization of semantic information, in order to perform planning under semantic relaxation and find approximate solutions. Furthermore, PORSCE II does not require any prior, domain-specific knowledge to form valid, desired complex services, apart from the OWL-S descriptions of the atomic web services and the corresponding ontologies. Finally, the system is able to handle cases of service failure through simple service replacement, without the obligation to perform planning again.

3. OWL-S

OWL-S is an upper ontology based on OWL [14], created in the context of the Semantic Web in order to describe knowledge concerning semantic web services. It is used in combination with additional ontologies, which organize the concepts appearing in the OWL-S descriptions. The use of OWL-S renders the semantics of the descriptions machine comprehensible; therefore it enables intelligent agents to discover, invoke and compose web services automatically. A web service description in OWL-S is comprised of three parts [6]:

- *Service Profile*: describes what the service accomplishes, limitations on service applicability and quality, and requirements that the service requester must satisfy to use the service.
- *Process Model*: describes the way a client can communicate and use the service.
- *Service Grounding*: specifies the details of how an agent can access a service, such as a communication protocols and message formats.

The approach presented here utilizes the semantic information contained in the *Service Profile* of a specific web service, along with the corresponding ontologies, in order to translate the description in planning terms. An example of an OWL-S Profile and the correspondences between its elements of interest and the planning terms they are translated into is provided in Section 5.1.

4. System Overview and Architecture

PORSCE II is the evolution of the prototype system PORSCE [7]. The core translation component exists in both systems; however, PORSCE II aims at a higher degree of integration as it additionally contains a visual interface, more elaborate relevance metrics, complex service accuracy assessment and the ability to modify

complex web services. Furthermore, PORSCE II adopts a different way of modeling the web service composition as a planning problem, which reduces the complexity of the planning problem, thus accelerating the planning process.

In order to highlight the planner independency of PORSCE II, which enables the use of any domain independent planning system based on PDDL, another external planner has been included in addition to the original one.

The main features of the integrated system are:

- Translation of OWL-S atomic web service descriptions into planning operators.
- Interaction with the user in order to acquire their preferences regarding the complex service and desired metrics for concept relevance.
- Enhancing the planning domain and problem with semantically similar concepts.
- Exporting the web service composition problem as a planning domain and problem in PDDL.
- Providing solutions by invoking external planners.
- Assessing the accuracy of the complex services.
- Visualizing and modifying the solution.

PORSCE II comprises of the OWL-S Parser, the Transformation Component, the OWL Ontology Manager (OOM), the Visualizer and the Service Replacement Component. An overview of the architecture and the interactions among the components is depicted in Fig. 1.

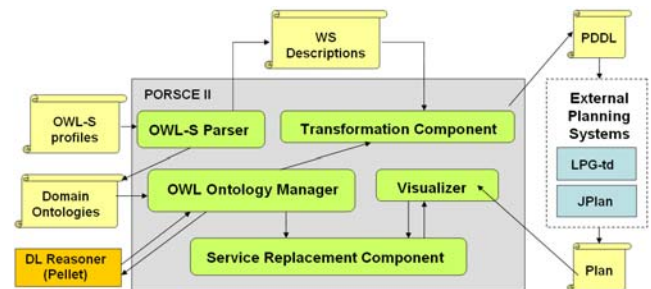


Fig. 1 The architecture of PORSCE II.

The OWL-S Parser is responsible for parsing a set of OWL-S web service profiles and determining the corresponding ontologies that the concepts appearing in the web service descriptions belong to. The OWL Ontology Manager (OOM), utilizing the Pellet DL Reasoner [2], applies the selected algorithm for discovering concepts that are similar to a query concept. The Transformation Component is responsible for a number of operations that result in the formulation of the planning problem from the initial web service composition problem, and its consequent solving. The purpose of the Visualizer is to provide the user with a visual representation of the plan, which in fact is the description of the complex service. Finally, the Service Replacement Component enables the user to replace a specific atomic web service in the complex service sequence. Details on the system functionality regarding the translation process and the rest of its operations are provided in Sections 5 and 6.

PORSCE II is implemented in Java and it is available online, along with example problems, at http://www.dit.hua.gr/~raniah/porsceII_en.html

5. Transformation Process

The transformation process includes the translation of the web service composition problem into a planning problem and its possible enhancement with semantic information. The process starts at the OWL-S Parser, which parses the OWL-S profiles of the available atomic web services and forwards them to the Transformation Component. The Transformation Component is responsible for a number of operations, including translating the web service descriptions received from the OWL-S Parser to planning operators and enhancing them with similar concepts derived from the OOM. Moreover, it interacts with the user in order to formulate the planning problem, and exports both the planning domain and problem to PDDL.

5.1. OWL-S to PDDL Translation

A planning problem in PORSCE II, in accordance with the STRIPS notation [4], is a tuple $\langle I, A, G \rangle$ where I is the initial state, A is the set of available actions that can be used to modify states, and G is the set of goals. Each action A_i has three lists of facts containing the preconditions of A_i , the facts that are added to the state and the facts that are deleted from the state after the application of A_i , denoted as $\text{prec}(A_i)$, $\text{add}(A_i)$ and $\text{del}(A_i)$ respectively

A straightforward solution adopted by PORSCE II for mapping the web service composition problem to a planning problem is the following: Let IC be the set of concepts that the user wishes to provide to the complex service and GC its desired outputs (goals). If O denotes the set of all the available concepts in the ontology, then $IC \subseteq O$, $GC \subseteq O$ and $IC \cap GC = \emptyset$. The inputs that the user wishes to provide formulate the initial state, while the desired outputs of the complex service formulate the goals of the problem: $I = IC$ and $G = GC$. Both the input and output sets are provided externally by the user.

The available OWL-S web service profiles are used in order to obtain the planning operators: each web service description WSD_i is translated to an operator A_i , using the information in the each profile (Fig. 2).

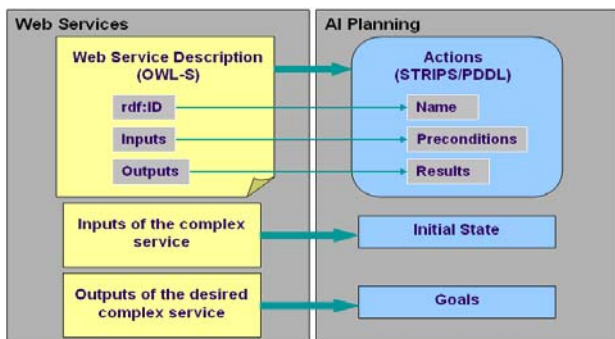


Fig. 2 Mapping web services to planning.

More specifically:

- The name of the action is the rdf:ID of the profile:
 $\text{name}(A_i) = WSD_i.\text{ID}$
- The preconditions of the action are formed based on the service's input definitions (concepts):

$$\text{prec}(A_i) \equiv \bigcup_{k=1}^n \{WSD_i.\text{hasInput}_k\}$$

- The add effects of the action comprises of the service's output definitions (concepts):

$$\text{add}(A_i) \equiv \bigcup_{k=1}^m \{WSD_i.\text{hasOutput}_k\}$$

- The delete list is left empty, since in the current study only services that do not have any negative effects in the world model are dealt with: $\text{del}(A_i) \equiv \emptyset$

An example of an OWL-S to PDDL transformation is presented in Fig. 3, where the mapping presented above is marked. The web service description at hand concerns a web service that accepts as input the activity Sightseeing and presents the user with areas that offer this activity.

5.2. Semantic Analysis

The step of semantic analysis, parallel to the transformation process, enables the system to translate the web service composition problem to a planning problem by taking into account semantic information. This step is implemented by the OWL Ontology Manager (OOM). During translation, the OOM is used extensively for performing semantic relaxation, which is useful in cases when an exact input/output matching plan is not available. The OOM locates equivalent and semantically relevant concepts; therefore, approximate plans can be created.

In our approach, two ontology concepts are considered semantically similar if and only if

- ❖ they have a *hierarchical relationship*
- ❖ their *semantic distance* does not exceed a threshold

As far as the *hierarchical relationship* is concerned, four hierarchical filters are used for its definition for two ontology concepts A and B :

- *exact*(A, B): The two concepts should have the same URI or they should be equivalent, in terms of OWL class equivalence, i.e. $A = B \vee A \equiv B$.
- *plugin*(A, B): The concept A should be subsumed by the concept B , i.e. $A \sqsubseteq B$.
- *subsume*(A, B): The concept A should subsume the concept B , i.e. $B \sqsubseteq A$. In both the *plugin* and the *subsume* filters the subsumption relationships of equivalent concepts are not considered.
- *sibling*(A, B): The two concepts should neither have a hierarchical relationship, nor be disjoint; instead, they should have a common superclass T , such as $A \sqsubseteq T \wedge B \sqsubseteq T$.



Fig. 3 Example of an OWL-S to PDDL

The *semantic distance* between two ontology concepts can be calculated in PORSCIE II using two methods:

The *Edge-Counting Distance* (ec) computes the distance of two concepts in terms of the number of edges found on the shortest path between them. An edge exists between two concepts A and B if A is the direct subclass of B , denoted as $A \sqsubseteq_d B$.

The implementation of the ec distance between two concepts, denoted as $d_{ec}(A, B)$, returns a value between 0 and 1, with 1 denoting absolute mismatch, and considers the following cases:

- $exact(A, B) \Rightarrow d_{ec}(A, B) = 0$: equivalent concepts
- $A \sqcap B \sqsubseteq \perp \Rightarrow d_{ec}(A, B) = 1$: disjoint concepts
- $plugin(A, B) \vee subsume(A, B) \Rightarrow d_{ec}(A, B) = p/p_{max}$.

If there is a hierarchical relationship between the two concepts, the distance is equal to the number of edges in their shortest path (p) normalized to $[0..1]$ using the maximum ec distance (p_{max}) found in the ontology, which can be approximated as $p_{max} = 2h - 1$, where h is the maximum edge distance from a leaf concept to the owl:Thing concept (T).

- $sibling(A, B) \Rightarrow d_{ec}(A, B) = \min[d_{ec}(A, T) + d_{ec}(B, T)]$: the concepts have a sibling relationship and the ec distance is the minimum of the sum of the ec distances of each concept from the least common ancestor T . Note that the owl:Thing is not considered as a common ancestor, since $\forall A, B : A \sqsubseteq T \wedge B \sqsubseteq T$ and therefore, no special structural knowledge is provided.

The *Upwards Cotopic Distance*, denoted as $d_{uc}(A, B)$, is defined in terms of the upwards cotopic measure, denoted as $uc(A)$ that represents the set of the superclasses of the concept A , including A itself [10]. In PORSCIE II, the upwards cotopic distance definition has been modified in order to incorporate the semantics of an ontology hierarchy. More specifically, the owl:Thing concept is not considered in the uc measure, while the union and intersection set operators take into account the concept equivalence semantics. In that way concept set multiplicity is ignored, that is, if $A \equiv B$, then $\{A, B, C\} = \{A, C\} \vee \{B, C\}$, and the concept set membership is semantically checked, that is, if $A \equiv B$ and $D = \{A\}$, then $A \in D \wedge B \in D$. Based on these remarks, the upwards cotopic distance is defined as

$$d_{uc}(A, B) = 1 - \frac{|uc(A) \cap uc(B)| - 1}{|uc(A) \cup uc(B)| - 1}$$

If two concepts have only the owl:Thing class as the common superclass or they are disjoint, then their distance equals to 1; otherwise, if the two concepts have a hierarchical relationship, then $d_{uc}(A, B) \in [0..1)$.

5.3. Semantic Awareness and Relaxation

The representation of the web service composition as a planning problem is empowered if the planning system is aware of semantic similarities among syntactically different concepts (semantic awareness). The solution adopted by PORSCCE II involves enhancing the domain and problem description with all the required semantic information in a pre-processing phase and letting the planner handle it as a classical planning problem. PORSCCE II adopts this solution in order to: a) be able to use any planner, compliant with PDDL, as the semantic enhancement applied to the domain remains transparent to the planner, and b) minimize the interactions between the planner and the OOM, which introduce an overhead on the planning time.

In the pre-processing phase, the system uses the OOM in order to acquire all the semantically relevant concepts for both the facts of the initial state and the outputs of the operators, discovered by the semantic analysis process described in the previous section. The enhancement of the problem by PORSCCE II is based on the following rules:

- The original concepts of the initial state together with the semantically equivalent and similar concepts form a new set of facts noted as the Expanded Initial State (EIS).
- The goals of the problem remain the same.
- The Enhanced Operator Set (EOS) is produced, by altering the description of each operator, while preserving the initial size of the set. More specifically, the effects list of each operator is enhanced by including all the equivalent and semantically similar concepts for the concepts in the initial effects list.

Suppose, for example, that the initial state I of the problem is the following:

```
I = {Sightseeing, Dates}
```

and that there are only the following two operators:

```
CityHotelMapService:
  prec={City, Hotel}, effect={Map}
SightSeeingAreaService:
  prec={Sightseeing}, effect={Area}
```

The OOM for a given distance metric and threshold discovers the following relevant concepts:

```
Dates  $\approx$  Duration
Area  $\approx$  County,
Map  $\approx$  GPSRoute
```

The pre-processor alters the problem definition to the following:

```
EIS: {Sightseeing, Dates, Duration}
EOS: CityHotelMapService:
  prec={City, Hotel}
  effect={Map, GPSRoute}
SightSeeingAreaService:
  prec={Sightseeing}
  effect={Area, County}
```

The new problem, namely $\langle EIS, EOS, G \rangle$ is encoded into PDDL and forwarded to the planning system in order to acquire a solution. Note that the semantic information is encoded in such a way that it is transparent to the external planning systems, which can solve the problem as any other classical planning problem.

6. Solution and Integration

PORSCCE II aims at integrating the composition process, including solving the problem through invocation of external planning systems, visualization, evaluation and modification of the solutions.

6.1. Acquiring Solutions

Since the transformation process results in the export of both the planning domain and problem in PDDL, any PDDL-compliant domain independent external planning system can be used.

Currently, two different planning modules have been incorporated in the system: JPlan [1], which is an open-source Java implementation of Graphplan and LPG-*td* [8]. Both planners proved to be remarkably fast and can handle a respectable number of operators, which is very important as the number of available web services is expected to increase significantly over time. After the planning process is completed, JPlan provides the plan, in its own format, which comprises of a simple sequential list of actions. LPG-*td*, on the other hand, provides the plan in a format that complies with PDDL+. The plan in this case might not be sequential, but structured in levels; actions belonging to the same level can be executed in an arbitrary sequence, however all actions of a certain level must be completed before any action of the following level can be executed. Subsequently, these plans are visualized and their accuracy is evaluated.

6.2. Complex Service Accuracy Assessment

Semantic relaxation and the use of multiple planners may produce a number of complex services, for which statistics and quality metrics have to be calculated. Such metrics include the number of actions and the number of levels in the plan, as well as a plan distance quality metric, which indicates the accuracy of the plan, when semantic relaxation takes place.

For the calculation of the plan semantic distance, each concept appearing in the inputs or outputs of the actions of

the plan is annotated by the OOM with a semantic distance d_i with respect to the original concept it was derived from, using the selected similarity metric. A concept distance of 0 reveals identical or equivalent concepts. Additionally, each concept is annotated with a weight w_i , which represents the kind of hierarchical relationship to the original concept, as in some cases certain hierarchical relationships might be more desirable than others.

These values are combined to form a plan semantic distance. For example, when the upwards cotopic distance metric is used, the plan semantic distance is calculated as a weighted product of these concepts, as the product represents better the semantic distance in this case:

$$PSD_{uc} = \prod_{i=0}^n w_i d_i, d_i \neq 0$$

The plan accuracy metric in both cases is calculated as $1 - PSD$; therefore, if there is exact input to output matching, or if only equivalent concepts are used, then the plan quality metric value is 1, while it decreases as the plan becomes less accurate.

6.3. Visualization and Modification

The Visualizer enhances comprehensibility by providing a visual representation of the complex service and facilitates plan manipulation. The complex service is represented as a schema of simple service invocations, showing inputs and outputs (Fig. 7, 8).

The Visualizer module invokes and interacts closely with another module of the system, the Service Replacement Component, which discovers all actions that could be used alternatively instead of the chosen one, using advice from the OOM for equivalent and relative concepts. An action A is considered an alternative for an action Q of the plan as far as it does not disturb the plan sequence and the intermediate states, that is $prec(A) \subseteq prec(Q)$ and $add(A) \supseteq add(Q)$. The selected alternative service substitutes the original one both in the plan and in the visualization.

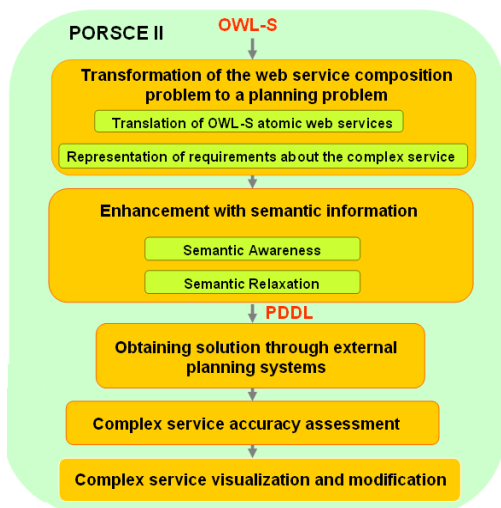


Fig. 4 The steps for the demonstration.

7. Demonstration and System Evaluation

This section aims at demonstrating the use and evaluating the performance of the system through a case study, following the general course depicted in Fig. 4.

The test sets used to perform experiments were obtained from the OWLS-TC version 2.2 revision 1 [3], while several service descriptions were modified or added to these domains, accommodating the demonstration of the capabilities of the system. The web services that were modified or added to the domain are depicted in Table 1.

Table 1. The added / modified web services.

Service	Inputs	Outputs
BookToPublisher	Book, Author	Publisher
CreditCardCharge	OrderData, CreditCard	Payment
ElectronicOrder	Electronic	OrderData
PublisherElectronicOrder	PublisherInfo	OrderData
ElectronicOrderInfo	Electronic	OrderInformation
Shipping	Address, OrderData	ShippingDate
WaysOfOrder	Publisher	Electronic
CustomsCost	Publisher, OrderData	CustomsCost

The transformation of the web service composition problem to a planning problem includes translating all available OWL-S atomic web services, including the aforementioned ones, to PDDL operators. It also incorporates the representation of the requirements about the complex service, which the user can express through a dialog interface such as the one depicted in Fig. 5.



Fig. 5. Defining initial and goal states and desired planners.

The scenario implemented here concerns the electronic purchase of a book. The user provides as inputs a book title and author, credit card info and the address that the book will be shipped to, and requires a charge to credit card for the purchase, as well as the shipping dates and the customs cost for the specific item. The initial state corresponds to the inputs of the complex service, while the goal state represents the desired complex service outcome.

The next step is optional semantic relaxation, performed through semantic enhancement. The user defines semantic metrics and thresholds through the interface depicted in Fig. 6.

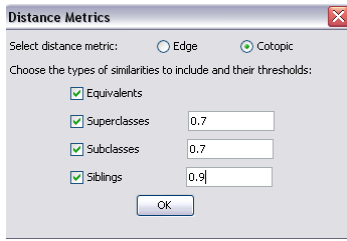


Fig. 6. The semantic enhancement interface.

At this point, the system exports the formulated (and possibly semantically enhanced) planning domain and problem to PDDL. Consequently, it invokes external planners to acquire solutions.

The plans produced by JPlan and LPG-td for the specific case study using the operator set described above,

without including any semantically relevant concepts are presented in Fig. 7.

While exact matching of input to output concepts is obligatory in the classical planning domains, in the web services world the case can be different, as it is preferable to present the user with a complex service that approximates the required functionality than to present no service at all. The semantically similar concepts obtained from the OWL Ontology Manager enable the system to compose alternative services that approximate the desired one in case there are no exact matches, by performing semantically relaxed concept matching. Such an approximate service for the specific case study is presented in Fig. 8. The calculated accuracy of this service is different from the accurate ones presented in Fig. 7.

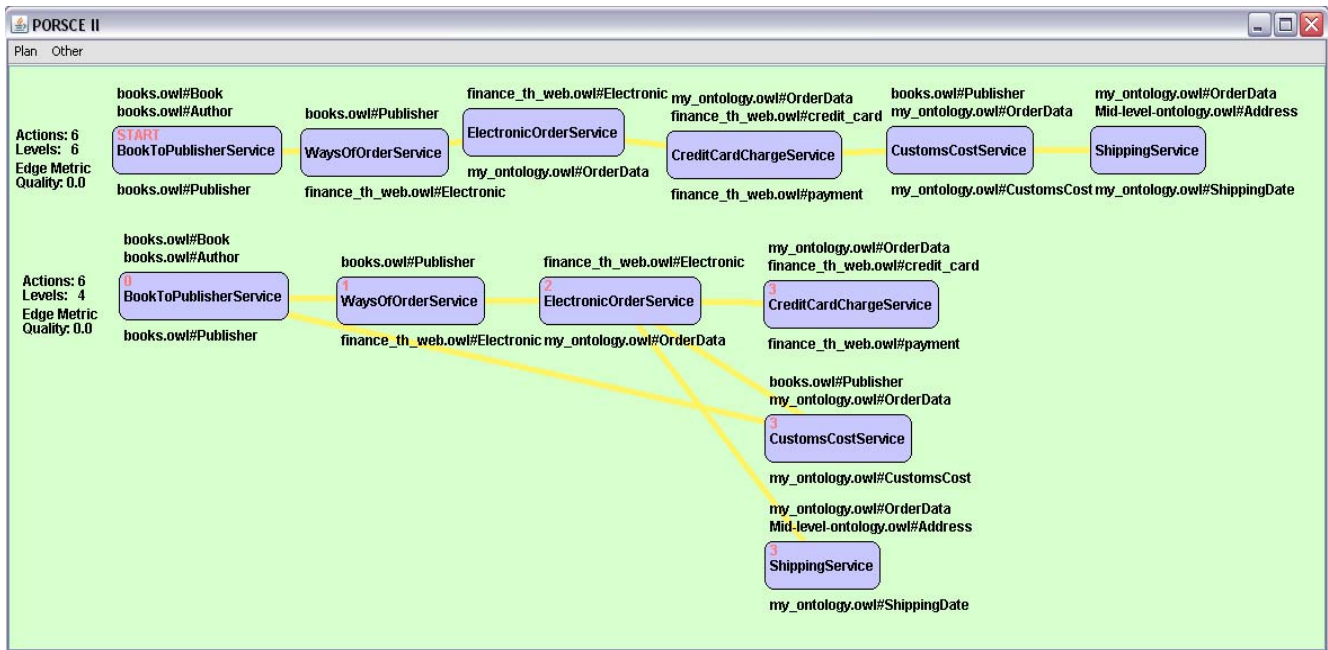


Fig. 7. The plans from JPlan (top) and LPG-td (bottom) for the specific case study.

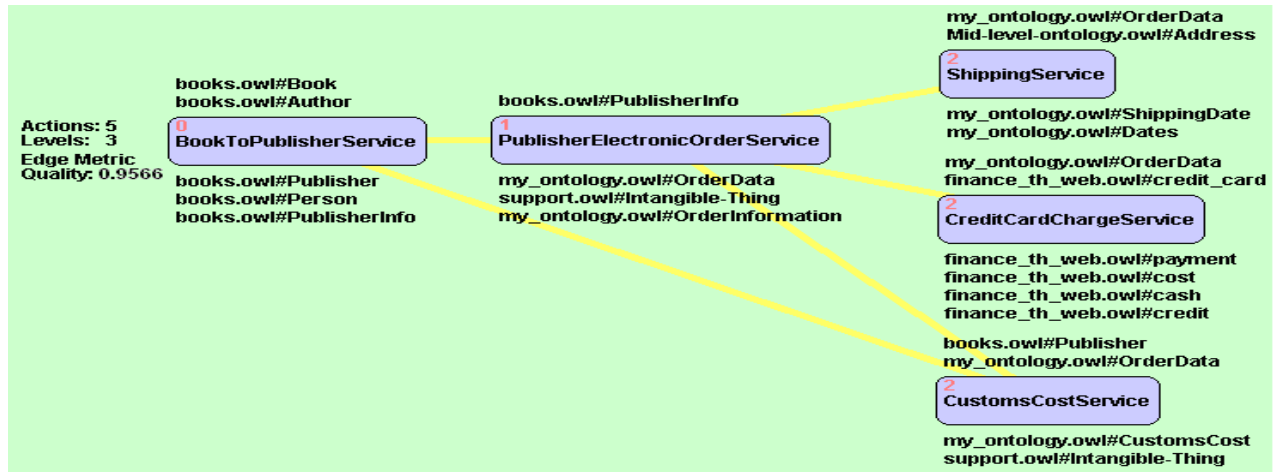


Fig. 8. Approximate complex service.

In order to study the behavior of the system as the number of available web services increases, web service profiles were added to the domain progressively in batches. The time performance results presented in Table 2 were obtained from a number of runs of the system on a machine with Dual-Core AMD Opteron Processor at 2.20GHz with 1GB of RAM memory and concern times for preprocessing, transformation of the OWL-S service profiles to PDDL actions and planning using LPG-td.

Measurements took place for domains of different sizes, namely 10, 100, 500 and 1000 OWL-S profiles. Some of the experiments were performed without semantic relaxation (X), while others were performed with semantic relaxation using either the edge-counting distance metric (E) or the upwards cotopic metric (C). The preprocessing time did not show significant fluctuation, as it depends only on the number and structure of the processed ontologies and not on the number of available web services. The total transformation time evidently increased as the number of available web services increased, however the average transformation time per web service profile converged to approximately 0.8 seconds for the exact matching and the edge-counting distance metric cases. In the upwards cotopic metric distance, the increase in the average transformation time is significant as available web services increase, due to the higher complexity of the algorithm used for the calculation of the upwards cotopic relevance between two concepts. As far as average planning time is concerned, LPG-td shows an increase in planning time as the number of actions increases, however it is still proved remarkably fast.

Table 2. Time measurements in milliseconds.

Number of web services		10	100	500	1000
Preprocessing time		5857	6104	5875	5703
Total transformation time	X	4594	70062	350836	792109
	E	4531	75725	335477	796797
	C	4585	74688	728633	3901141
Transformation time per web service	X	459	700	702	792
	E	453	671	757	797
	C	459	746	1457	3901
Planning time (LPGtd)	X	1	13	16	17
	E	4	6	15	16
	C	3	5	16	16

8. Conclusions and Future Work

This paper presented PORSCIE II, which combines planning with semantic object relevance in order to approach the semantic web service composition problem. Each web service composition problem is translated into a planning problem, possibly enhanced with semantically relevant concepts and exported to PDDL. The system integrates external planning system which perform planning with the desired degree of semantic relaxation. The obtained plans, which represent descriptions of the

desired complex web service, are evaluated, visualized and modified.

Future goals include the extension of the system in order to translate the plan describing the complex service into OWL-S so the complex web service can be invoked and provide feedback. In addition, another goal concerns incorporating the quality distance metric with plan statistics in a common metric. Furthermore, integration with the VLEPPO system [9] is a promising future direction, in order to accommodate design and solving of the web service composition problems. Finally, it lies in our immediate plans to study ways to enhance the services representation and explore the ability to produce various complex services according to non-functional properties.

References

- [1] JPlan: Java Graphplan Implementation, <http://sourceforge.net/projects/jplan>
- [2] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz, Pellet: A Practical OWL DL Reasoner, J. Web Semantics, 2007
- [3] OWLS-TC version 2.2 revision 1, <http://projects.semwebcentral.org/projects/owls-tc/>
- [4] R. Fikes, N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving", Artificial Intelligence, Vol 2 (1971), 189-208.
- [5] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, D. Wilkins, "PDDL -- the Planning Domain Definition Language". Technical report, Yale University, New Haven, CT (1998).
- [6] OWL-S 1.1. <http://www.daml.org/services/owl-s/1.1/>
- [7] O. Hatzi, G. Meditskos, D. Vrakas, N. Bassiliades, D. Anagnostopoulos, I. Vlahavas, A Synergy of Planning and Ontology Concept Ranking for Semantic Web Service Composition, IBERAMIA 2008, 11th Ibero-American Conference on AI, Lisbon, Portugal, October 14-17, 2008. Proceedings. Lecture Notes in Computer Science 5290 Springer 2008, pp 42-51.
- [8] A. Gerevini, A. Saetti, I. Serina, LPG-TD: a Fully Automated Planner for PDDL2.2 Domains" (short paper), in International Planning Competition, 14th Int. Conference on Automated Planning and Scheduling (ICAPS-04).
- [9] O. Hatzi, D. Vrakas, N. Bassiliades, D. Anagnostopoulos, I. Vlahavas, VLEPPO: A Visual Language for Problem Representation, 26th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2007), Roman Bartak (Ed.), pp. 60 – 66.
- [10] A. Maedche and V. Zacharias, Clustering Ontology-Based Metadata in the Semantic Web, European Conf. Principles of Data Mining and Knowledge Discovery, 2002.
- [11] WSMO, <http://www.wsmo.org/>
- [12] WSDL-S, <http://www.w3.org/Submission/WSDL-S/>
- [13] SAWSDL, <http://www.w3.org/2002/ws/sawSDL/>
- [14] OWL, <http://www.w3.org/TR/owl-ref/>
- [15] E. Sirin, B. Parsia, D. Wu, J. Hendler and D. Nau, 2004. HTN planning for web service composition using SHOP2. Journal of Web Semantics, 1(4) 377–396.
- [16] M. Klusch, A. Gerber, M. Schmidt: Semantic Web Service Composition Planning with OWLS-XPlan. AAAI Fall Symposium on Semantic Web and Agents, USA, 2005.

Augmenting Instructable Computing with Planning Technology

Clayton T. Morrison

University of Arizona
Department of Computer Science
1040 E. 4th Street
Tucson, Arizona 85721

Daniel Bryce

Utah State University
Department of Computer Science
Old Main 414
Logan, Utah 84322

Ian R. Fasel

Antons Rebguns
University of Arizona
Department of Computer Science
1040 E. 4th Street
Tucson, Arizona 85721

Abstract

Advances in human-instructable computing are contributing to a new breed of computer systems that can be taught by natural instruction rather than requiring direct programming. The current approach in the MABLE “electronic student” emphasizes the interface that maps different modes of instruction to machine learning algorithms that can learn the concepts and task knowledge being taught. While the interface provides more natural interaction with the system, there are still many constraints put on how the teacher teaches, in particular in what the teacher can assume about MABLE’s ability to compose previously learned concepts. We present a method for automatically translating MABLE’s learned task knowledge into a STRIPS planning domain, and planner-generated plans back into MABLE’s knowledge representation. In this way, existing planning technology is used to augment MABLE’s problem solving ability. This allows us to relax the requirement that the teacher explicitly teach *every* composite procedure and also provides a role for planning to contribute directly to learning in a more capable student.

Introduction

Human-instructable computing aims to build computational systems that can be taught through natural instruction rather than requiring programming by specially-trained engineers. We are currently working in the DARPA Bootstrapped Learning Project to build a human-instructable “electronic student” called MABLE, the Modular Architecture for Bootstrapped Learning Experiments (Mailler et al. 2009). MABLE consists of a set of learning *strategies* designed to interpret different instruction methods naturally used by humans. These instruction methods include giving declarative definitions and descriptions, providing examples and demonstrations, and giving feedback based on student actions. MABLE interacts with a teacher that provides instruction by using these methods to teach concepts that build on one another – *bootstrap* – to more complex concepts and skills. These concepts, in turn, are represented in MABLE’s knowledge representation language, *Interlingua*.

However, there are still a number of unnatural requirements placed on the teacher of MABLE. In the current system, lessons teach self-contained concepts, such as the actions corresponding to steps in a procedure. The lessons, however, tend to be specific to what is involved in executing the action and do not always include information about the action’s effects and preconditions. For MABLE to understand how to compose its previously learned actions into a composite procedure, the teacher *must* give an explicit later lesson that wraps the actions as steps in an HTN-like procedure.

We have developed a learning strategy called *Learning by Noticing* (LbN) whose job is to identify patterns in teaching and concept use that might be otherwise implicit – i.e., not directly included in the teacher’s utterances or part of the target concept being taught in the lesson. By observing the lessons that teach atomic steps in a procedure, LbN constructs action models that fill in action effects and in some cases preconditions of actions. These action models provide the raw materials for defining planning operators.

To make full use of these nascent planning operators, we have developed a process that translates these action models into the PDDL planning language (McDermott and the AIPS’98 Planning Competition Committee 1998) so that compound procedures using the component actions as steps can be identified through planning. We target planning for the STRIPS domain and use an implementation of the Graphplan planning algorithm as our planner (Blum and Furst 1997). When planning is successful, the plan produced by the planner is translated back into Interlingua and incorporated into MABLE’s knowledge base. In this way, LbN action model construction combined with planning technology *relaxes* the need for the teacher to necessarily teach every compound procedure through an additional explicit lesson.

For the purposes of the ICKEPS competition, our method is best viewed as a service translating concepts taught in MABLE’s Interlingua (IL) knowledge representation to PDDL and PDDL-expressed plans back to IL. A key contribution of our method is that it is not merely translating IL to and from PDDL but also performs inference to fill in the action models for the learned Interlingua task knowledge. From a broader perspective, although we are translating from IL to PDDL, the vision is for translating from human interaction to formal concept representation and pro-

cess languages like PDDL, in a human-instructable computing framework. We believe our tool demonstrates the benefits of both action model learning as well as the benefits of light weight planning for instructable computing, and the potential for instructable computing to provide a natural human interface for teaching planning knowledge.

The following sections present a brief overview of the MABLE architecture and its learning environment, followed by a description of IL, the language underlying MABLE's knowledge representation. We then present how LbN constructs action models, followed by a discussion of our method for translating from IL to PDDL, and planner-generated plans back into IL for MABLE's use. We conclude with a discussion of future directions for exploiting the synergy between instructable computing and planning.

MABLE and the Learning Environment

MABLE is currently being developed within a larger interaction framework that includes a simulated Environment and Teacher. All three components interact with each other on a message *Timeline*.

The *Environment* simulates a perceivable world, keeping track of its state and any changes produced by actions from the Teacher or MABLE. The Environment posts perceptual update messages representing the current world state to the Timeline. A variety of simulators are currently available for use as the Environment. These include a version of the classic "blocks world", the 2-dimensional robocup soccer simulator (Kitano et al. 1997), a simulation of an unmanned aerial vehicle in a 3-dimensional world, a 2-dimensional tactics-level wargame simulation, and a simulation of the system control for the International Space Station. For exposition, we take the bulk of our examples from the more familiar blocks world simulator.

A set of structured *curricula* have been authored by humans to teach various concepts in each of the different simulator domains. Some concepts depend on others, so the curricula are decomposed into sets of lessons that depend on one another. Each lesson aims to teach one concept. These lessons form "rungs" in a partially-ordered curriculum "ladder", with some lessons making use of the concepts taught in earlier lesson rungs. Lessons themselves also have structure. They begin with a set of *teaching epochs*, in which the Teacher teaches a concept according to one of the natural instruction methods. These are followed by a *testing epoch*, in which the Teacher sends messages including an imperative for MABLE to answer questions or perform actions. A grade is assessed according to MABLE's performance in a test.

The simulated *Teacher* itself executes teaching and testing epochs as a script, with some allowance for interaction based on MABLE's possible response messages. The scripts manage initializing the Environment state and the generation of Teacher messages. Teacher messages include utterances and imperatives, as well as action messages that induce changes in the Environment world state.

MABLE's task is to observe the messages that come across the Timeline and use them to build a model of the current state of teaching (and testing), using the messages from the Teacher and percept updates from the Environment to

learn. The Mable architecture itself consists of a set of modules that work together to incrementally learn concepts from the Timeline messages. A set of learning *strategies* provide component interfaces to the different methods the teacher uses for teaching concepts. For example, the *ProcedureByTelling* strategy interprets declarative teacher utterances to construct the component steps in a procedure; on the other hand, the *ConditionByExample* strategy identifies how the teacher is providing examples of a rule. In all cases, learning strategies extract, interpret and repackage data from the Teacher and Environment messages to construct concepts. Learned concepts are represented in IL and stored in MABLE's knowledge repository. When appropriate, strategies invoke learning *services*, dedicated machine learning algorithms that can help with concept learning. For example, (i) a predicate learning service is driven by inductive logic programming, (ii) regression algorithms can learn numeric functions, and (iii) reinforcement learning is used to learn policies by feedback. All module activities are coordinated by a *control* module, which has the job of identifying learning targets, ensuring the appropriate learning strategies and services are invoked, and ensures that progress is being made toward learning the target concept in order to perform well in the testing epoch. Finally, MABLE's *execution engine* is used to execute IL and monitor procedure execution, for example when a learning strategy requests to evaluate an IL expression, or in order to answer an imperative messages sent by the Teacher. We will return to the execution engine in the next section.

Interlingua

The *Interlingua* (IL) language has been designed to accommodate a broad spectrum of duties (Oblinger 2008). IL provides the building blocks necessary to construct the variety of concepts MABLE will need to represent as background knowledge and as the result of learning. These include rules, a type hierarchy of classes, functions, and procedures. IL is also expressive enough to define other languages as extensions within IL. In particular, the *Interaction Language* (ITL) is a specialized IL extension that is used to represent all messages that appear on the Timeline between MABLE, the Teacher and the Environment. Here we present the components of IL and ITL that we need as background to explain how models of actions are constructed and how we will translate the models and problem instances into PDDL.

Core IL consists of three component languages. The first of these is the *syntax language* and is used to define types, using the **is** construct, and properties, using the **arg** construct. For example, the expression

```
is Dog Animal;
```

defines a type called *Dog* that is a sub-type of *Animal*. Properties associate values with instances of a type (called the property's *host* type). They are named and also impose a constraint on the type of values that may be bound to them. For example, the expression

```
arg Dog age Integer;
```

defines a property of a *Dog* called *age*, and only *Integers* can be associated with a *Dog*'s *age*. All IL types that have

properties associated with them are called *composites*; `Dog` is therefore a composite. IL also defines a set of *atomic* types that have no properties associated with them (and therefore no property-based composite structure); IL provides familiar atomic types such as Numbers, Integers, Floats, Booleans, Strings and Symbols. All property values can also be bound to a special atomic `Null` value, irrespective of the type constraints placed on the property value; this is a feature of IL that we will return to, below.

Like many other typed object-oriented programming languages (e.g., Java), properties associated with inherited types are also inherited and associated with the new type. A new type may have new properties associated with it, which are then added to the list of properties inherited, or a property of a new type may *restrict* inherited properties by giving them new, compatible type restrictions. In the case of restriction, the new type restriction must be a subtype of the type restriction of the inherited property.

Unlike many languages, however, IL also allows for multiple inheritance. That is, a type *A* might inherit from types *B* and *C*. More formally, the *is* type assertion is reflexive and transitive, but not symmetric, and combined with the possibility of multiple inheritance, this means the overall type hierarchy structure is that of a directed acyclic graph (DAG). Figure 1 shows a portion of the type hierarchy for the blocks world domain. Here, the `Object` type inherits from both the `Composite` and `Percept` types. Multiple inheritance poses a critical challenge to translating IL concepts into typical planning domain representations such as PDDL; we will address this in our translation, below.

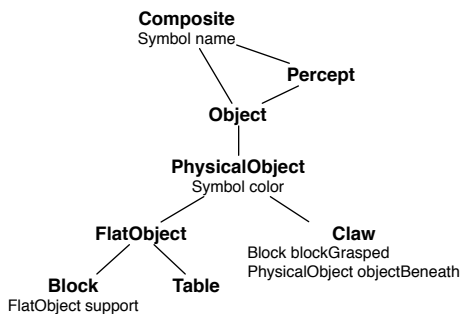


Figure 1: Portion of the blocks world type hierarchy; property definitions are displayed below their host type, with the property value type constraint followed by the property name.

For convenience, IL defines a “macro” called `defSyntax` that combines type inheritance and property associations of a type in one expression. For example,

```
defSyntax Dog extends Animal (
  Integer age
  Symbol color );
```

defines our `Dog` with its `age` property, and also includes the property `color`, which is a `Symbol`; a `defSyntax` may include multiple extensions for multiple inheritance.

The second core IL language, the *instance encoding language*, is used to express ground instances and their property

value bindings, as they exist in real or simulated worlds at a particular time. For example

```
Dog(name=Rover, color=black, age=3)
```

describes a `Dog` named `Rover` that is age 3 and is black in color.

Finally, the third core language of IL is the *code body language*. This language specifies functions and procedures that can be executed. An expression defining a code body includes specification of a code body *interpreter* which handles executing the code body. IL provides several code interpreters, including one for evaluating functions, another for running recursive procedures that are formed with combinations of control statements such as “if” and “while”, and an interpreter for evaluating predicates defined in first-order predicate logic.

The following example defines a function to add 1 to an input integer:

```
defSyntax Add1 extends Function
  (Integer arg);
defCode Add1 FunctionEngine
  FunctionBody(Return(Plus(arg,1)));
```

The `defSyntax` defines a new type of `Function` called `Add1`, and it has the property `arg` that is an `Integer`. The corresponding `defCode` specifies that the body of the code will be interpreted and executed by the `FunctionEngine`. The `FunctionEngine` knows how to execute the `Plus` function, which takes two argument; the reference to `arg` is interpreted as whatever value is bound to the `arg` property in an instance of the `Add1` type. In this sense, properties and their values are treated by the code interpreters as arguments for the code body. The return value of `Plus` is then returned as the value of executing `Add1`.

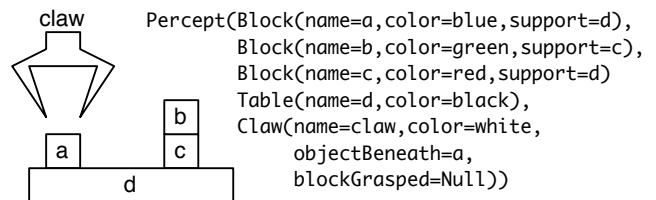


Figure 2: Example blocks world state and its IL `Percept` representation.

Interaction Language

We finish this summary of IL by presenting components of the Interaction Language (ITL) we will use in our discussion below. ITL is a specialized language built within IL that defines the composite terms used for interaction between the Teacher, Environment and MABLE. `Percept` expressions are generated by the Environment simulator and describe the current state of the environment. `Percepts` wrap ground IL instance expressions that describe the state of the world. Figure 2 depicts an example blocks world state and the corresponding `Percept` message describing it.¹

¹In order to make our examples of IL instance expressions more concise, we will often drop properties/value pairs that would other-

Observations of the execution of actions that affect the Environment simulator are presented in a different form. First, in addition to the specialized executable language terms (such as `Plus`), MABLE is also provided with a set of primitive simulator commands. These are defined in the same way as other executables (with `defSyntax` and `defCode`), but their execution interpreter specifies the Environment simulator and they are interpreted as direct commands to the simulator. While the `defSyntax` tells MABLE what arguments the action takes and their general type constraints, it does not say how the actions are to be used, i.e., what values should be bound to arguments; these bindings are taught by the Teacher or acquired by observation (by the LbN learning service).

MABLE observes the Teacher executing an action when a Timeline message contains a special ITL term, like the following:

```
TeacherAction(action=Grasp());
```

This indicates that the `Grasp` action has been executed. Only primitive simulator actions will show up in these messages. In order for the Teacher to specify that a non-primitive action is being executed (e.g., one that the Teacher is currently teaching MABLE), an utterance like the following is used:

```
Utter(utterance=WatchMe(DoWith(
  MoveOnto(Block(name=a),
            Block(name=b))));
```

Finally, ITL *imperatives* are used to direct an agent to execute some command or code body. The Teacher and MABLE can both send messages with imperatives to the Environment in order to execute simulator commands. When an imperative is directed from the Teacher to MABLE, it is a request for MABLE to execute a code body. In most cases the imperative request directly names the action the Teacher expects MABLE to have learned a code body for and to execute. However, the Teacher can also use the special imperative `MakeSo` to specify a world state the Teacher would like MABLE to achieve.

The Limits of Execution

We can now state precisely what MABLE is missing that the LbN learning service partially provides and that our planning language translation service completes. MABLE's execution engine executes IL code bodies either when prompted by a learning strategy or service, or when the Teacher requests MABLE to do so. In general, these calls specify, in the instance encoding language, the name of the form to be executed along with any values bound to properties (in this case, treated as arguments). The execution engine does not itself know how to select which action to execute without this information.² The procedure interpretation language

wise be present in the expression. For example, the contents of the `Percept` in Figure 2 would also include the `Percept` instance name, and the list of objects would be enclosed as a list bound to the `perceptsGained` property).

²Internally, MABLE's knowledge repository keeps track of multiple hypothesized learned code definitions, and a process guided by control selects the "best" current hypothesis.

does include control flow constructs and recursive procedure and function calling, but this machinery only works once the procedure has been selected for execution. The execution engine itself does not have a mechanism for determining that one or more executable code bodies could be used to achieve some specified state or result, such as that specified in a `MakeSo` imperative. In the next section, we describe how the LbN learning strategy learns action models that can answer Teacher `MakeSo` imperatives.

Learning by Noticing

In general, MABLE's learning strategies respond to and do their work based on specific classes of messages and percepts from the Teacher and world. A notable exception is the *Learning by Noticing* (LbN) learning strategy. LbN's primary job is to analyze incoming messages and mine them for patterns that may indicate information relevant to learning that is not otherwise explicitly expressed in teacher utterances or world percepts. When LbN identifies such patterns, it makes them explicit by posting the findings in special `Relevant` messages and as hypothesized new concepts, both made available to the other learning strategies in MABLE.

LbN cannot look for all patterns, so it relies on a set of heuristics to decide what kinds of patterns to search for. One of LbN's core heuristics is to look for changes in the world state that result from observed actions. This requires building a transition model of world states. For the purpose of building models of actions, this transition model is based on `Percept` messages describing world state changes. (Other heuristics also use Teacher utterances and imperatives.)

LbN treats the sequence of `Percept` messages that come over the Timeline as a time series of structured objects. To construct pattern identifiers, LbN decomposes this set of structured objects into a set of atomic events over which random variables are defined. For our presentation here, we focus on representing instances of properties and their value binding. Each atomic event is defined as a pattern consisting of four pieces of information: the type of the host composite (the composite the property is associated with), the composite's unique identifier (its name), the property name, and the value currently bound to the property.³ If the property value is itself another composite object (as opposed to an atomic value), then the value is represented by the name of the composite. For example, the following instance

```
Block(name=block-47,
      support=Table(name=table-3)),
```

is represented as the property-binding event

```
[Block, block-47, support] <= table-3
```

These events are also used to build generalized patterns, for example by "wild-carding" the name of the object instance:

```
[Block, *, support] <= table-3
```

³We ignore here two other classes of random variables: representing how many instances of a type exist at a given time, and the values of derived relations resulting from other functions and predicates, such as *Near* and *Distance*.

This pattern matches any instance of a `Block` whose `support` property is bound to table-3. These event patterns are used to define random variables whose values can be tracked over time as well as used to define probabilities.

The final representation of random variables resembles a discrete-time sensor readout, where the “sensors” are random variables determined by the event patterns that exist in the world at each time they are sampled.

For building models of the effects of actions, LbN keeps track of how the random variables change when actions are executed. A minor complication here is that there may be several `Percept` message updates about the world state between each action. To handle this, LbN divides the `Timeline` of `Percept` messages into chunks based on the boundaries of lessons and the time ticks at which actions are executed: the first chunk consists of the `percepts` between the start of the lesson and the first action; the second chunk between that first action and the second, and so on, with the effects of the last action being the world state updates between the last action execution and end of the lesson. Although multiple `Percept` updates may happen within the chunk, LbN only pays attention to the state of the world by the end of the chunk.⁴ In this work, all primitive actions are atomic and their effects are assumed to occur within the chunk directly after the action. An action a taken at time t is denoted a_t . The world state just prior to the action is represented by the world state changes of the previous chunk of `Percept` updates up to time t , and is denoted s_t . The world state after executing a_t is s_{t+1} and is represented by the world state at the end of the `Percept` update chunk after t , but before action a_{t+1} (or the end of the lesson).

The random variables defined for s_t and their changes from s_t to s_{t+1} provide for a variety of models of how actions effect changes in the world. LbN makes use of several heuristics to identify whether actions may be relevant to some aspect of the unfolding world state. However, in order to meet the demands of constructing action models that conform to the semantics of STRIPS planning operators, LbN uses the following two-stage construction process.

First, LbN keeps track of the values of each property-binding event before and after an action is taken and then looks for cases where taking an action *consistently* causes the properties to change from one value to another. LbN collects statistics for such transitions, and when the probability of *change*, $P(X_{t+1} \neq u | a_t, X_t = u)$, exceeds a threshold, the random variable involved in the transition is treated as a candidate effect of the action. These candidate random variables are collected for each action.

In the second phase, LbN then analyzes for each action the set of random variables identified as candidate effects. The random variables denote properties that are believed to be consistently affected by the execution of the action.

⁴Because a number of `Percept` updates might have occurred since the start of the chunk, this means LbN may be ignoring important information about the dynamics of world changes between any two actions. This is a topic of ongoing research; in the work we present here, this information loss has not affected LbN’s overall ability to construct action models, but we expect it may in the future.

For these candidate effects to be considered as a model of a STRIPS operator, the properties themselves must be “hosted” by (i.e., belong to) a composite instance that satisfies one of the following *identifiability* criteria:

1. The host instance is always unique in every situation in which the action is taken (e.g., there is always one and only one `Claw` in the blocks world simulation). LbN identifies this condition by counting the number of instances of a type it observes; if there is always one and only one, then the host instance of the property satisfies this criterion.
2. The host instance is the value of one of the arguments to the action.
3. The host instance appears as one of the values bound to another property in the set of candidate effects.

The purpose of this test is to ensure we can always identify the relevant properties that are changing as a result of the action, even if their host composite changes from one action execution to another (as in criteria 2 and 3).

If this test is satisfied for each candidate effects property, then LbN can *lift* the representation of the candidate effects, replacing the (non-Null) ground values in the specific action execution instances with variables (leaving Null values in place); host composites are also consistently replaced with variables. The *same* variables are used within the set of candidate effects properties when the following condition holds: Across all of the instances of the action being taken, the (non-Null) value of candidate effect property **A** in state s_t always appears bound in state s_{t+1} to candidate effect property **B**. All other ground values are assigned different random variables.

As an example, suppose LbN identifies that the `Release` action affects the random variables representing the property bindings for the `support` property of a `Block` and the `blockGrasped` and `objectBeneath` properties of the `Claw`, so that for a particular instance, the changes to the properties are as follows:

```
Block(name=a) support :
  Null -> Table(name=d)
Claw blockGrasped :
  Block(name=a) -> Null
Claw objectBeneath :
  Table(name=d) -> Block(name=a)
```

Each of the properties satisfies one of the identifiability criteria. *If* this same pattern of properties and value changes consistently occurs for all instances of the `Release` action, then the following lifted representation of the action model is produced:

```
?a support :
  Null -> ?b
?c blockGrasped :
  ?a -> Null
?c objectBeneath :
  ?b -> ?a
```

The variables are also typed. Variables that replace host composites of properties are given the type of the host composite, and variables replacing property values are given the

type of the property’s value type constraint. In the above example, this means that `?a` is of type `Block` and `c` is of type `Claw`, according to the type hierarchy and property assignments in Figure 1. `?b` poses a conflict because as the outcome of the change in the `support` property, it is constrained to be a `Block`, but as the prior value of `objectBeneath`, it is a `PhysicalObject`. As long as the IL type definitions are consistent, then one type will always be a subtype of another. In such cases, we always choose the more restrictive type, so `?b` is constrained to be of type `Block`.

Identifying the effects of `MoveClaw` provides another illustration. In this case, `MoveClaw` takes a single argument, constrained to be a `PhysicalObject`, which according to the type hierarchy in Figure 1 includes `Blocks`, `Tables` and even the `Claw` itself. `LbN`, however, has already identified that the argument always ends up being the value bound to the `Claw`’s `objectBeneath` property after the action

```
MoveClaw arg1 = Block(name=a)
Claw objectBeneath :
  Table(name=d) -> Block(name=a)
```

Again, the property satisfies one of the identifiability criteria. In this case, `Block(name=a)` is replaced in two places by the same variable, but the `Table(name=d)` gets a unique variable:

```
MoveClaw arg1 = ?a
?b objectBeneath :
  ?c -> ?a
```

Again, the variables are typed according to the roles they play in the property definition. And as with variables `?b` of `Release`, the conflict between possible type constraints for variable `?a` is chosen to be the more restrictive `Block`.

`LbN` can identify and construct these models only when effects are consistently produced by an action, and only when enough data are presented in the form of action executions with observed prior and outcome world states. But when the data is available and `LbN` does identify these transitions, then `LbN` can provide lifted action models that can be used for STRIPS planning.

A PDDL Translation Service for Interlingua

After teaching a set of actions through a series of lessons, the Teacher may request that `MABLE MakeSo` some world state. IL provides a variety of terms that can be used to describe a world state, but here we focus on two: `SameAs` and `Of`. `SameAs` is a predicate that asserts that two arguments are the same. `Of` also takes two arguments, where the first gives the symbol name of a property, and the second gives the name of a composite instance. In this way, `Of` is used to refer to the value of a property of a composite instance. The following example combines these terms:

```
MakeSo(SameAs(Of(support, a), b))
```

This imperative requests that `MABLE` transform the current state (which for the following example we will assume is that depicted in Figure 2) into the state where the support of `a`, which happens to be a `Block`, is `b`, which also happens to be a `Block`.

This triggers `MABLE`’s control module to execute the IL-to-PDDL translation service. The first task this service engages is to create the domain model, translating from IL concepts and action models into a PDDL domain model. The translation service executes the following steps:

STEP 1: Identify action models and associated types and properties. We do not need to translate the entire `MABLE` knowledge repository; nor do we need to translate all of the elements currently observed within the world state. Instead, what we translate will be driven by the current set of action models provided by `LbN`. For this working example, we assume that in addition to the action models defined for `Release` and `MoveClaw`, `LbN` has also formed a model for `Grasp`:

```
?a support :
  ?b -> Null
?c blockGrasped :
  Null -> ?b
?c objectBeneath :
  ?b -> ?a
```

The translation service takes these definitions and collects all of the different types and properties referenced. Only these types and properties need to be translated to the PDDL domain definition, and only current world objects that are members of these types need to be translated to the PDDL problem definition. Any other objects in the world and any other types in the `MABLE` knowledge base are irrelevant – they play no role in the current action effects model, so won’t help in planning.

STEP 2: Translate IL types into PDDL. The IL multiple inheritance type system is not directly compatible with most standard planning domain representations, such as PDDL, because they are restricted to single inheritance. There is an active strand of research that is looking at various methods for translating multiple inheritance into single inheritance systems (Dao et al. 2004; Crespo, Marquès, and Rodriguez 2002). However, for our purposes here, rather than using the PDDL `:types` features, instead we treat types as a property of objects in the PDDL domain, and therefore translate types as PDDL domain predicates. In order to ensure there are no accidental name clashes in the translation from existing IL type and object names, the following naming convention is used: each IL type is translated to a PDDL domain predicate by appending `_t-` to the IL type name to create the predicate name. For example, the IL type `Block` is translated as: `(_t-Block ?t)` (The variable name doesn’t matter here).

In the PDDL problem domain, types must then be asserted for each object, in the `:init` clause. Each object is not just an instance of its base type, but also every ancestor type the base type inherits from. A predicate assertion is added for each ancestor. Thus, a `Block` object named `b` will have the following list of type assertions: `(_t-Block b)`, `(_t-FlatObject b)`, `(_t-PhysicalObject b)`, `(_t-Object b)`, `(_t-Percept b)`, and `(_t-Composite b)`.

STEP 3: Translate IL properties into PDDL. IL Properties naturally translate to PDDL predicates in the `:predicates` domain definition. For example, the prop-

erty support associated with `Block` is translated as (again using a special naming convention): `(_p-support ?b ?f)`. However, this definition does not itself put constraints on the two variables. Type checking will now be moved into the action precondition clause, and it is up to the translation process to keep track of the appropriate type constraints. For example, if the support predicate is asserted in an action precondition as `(_p-support ?b ?f)`, then the translation process must also include the type constraint assertions on `?b` and `?f`: `(_t-Block ?b)`, `(_t-FlatObject ?f)`.

All properties can be bound to the special `Null` value. Rather than reify `Null` as a special object, instead we treat it as a predicate, one for each property. For example, the condition of the support property being set to `Null` is translated as: `(_isNull-support ?s)`.

STEP 4: Translate Action Models into PDDL. The basic building blocks for translating IL action models to PDDL have already been defined. The lifted action models constructed by LbN specify the properties that change. Each component property change model specifies four things: the property that changes, the host composite the property is associated with, and the value of property before the action (at s_t) and after (s_{t+1}). For example, the support property in the `Release` action model has been lifted so that the host composite is represented by variable `?a`, which is constrained to be of type `Block`, and in s_t it is `Null`, and then in s_{t+1} becomes set to the value of variable `?b`, which is constrained to be type `Block`. To represent this change, the prior value of support is asserted in the `:precondition` clause of `Release` as an instance of the `_p-support` predicate with its associated action-model-assigned value. If the value assignment was a variable, that would be asserted along with the host composite variable. But in this case, the support is `Null`, so the `_isNull-support` predicate is asserted: `(_isNull-support ?a)`. For the effect outcome, the resulting value of support at s_{t+1} is asserted as `(_p-support ?a ?b)` in the `:effect` clause for `Release`. We also need to negate the `isNull` predicate: `(not (_isNull-support ?a))`. Once these precondition and effect predicate assertions are made, we assert the type constraint predicates for any variables mentioned in the precondition clause, in this case for `?a` and `?b`. Finally, any variables we have mentioned so far are added to the `:parameters` clause. We repeat the above translation for each of the property effects components of `Release`, in this case for `blockGrasped` and `objectBeneath`. The final translated action definitions is as follows:

```
(:action Release
:parameters (?a ?b ?c)
:precondition
  (and (_t-Block ?a)
        (_t-Block ?b)
        (_t-Block ?c)
        (_isNull-support ?a)
        (_p-blockGrasped ?c ?a)
        (_p-objectBeneath ?c ?b))
:effects
  (and (not (_isNull-support ?a))
```

```
(_p-support ?a ?b)
(not (_p-blockGrasped ?c ?a))
(_isNull-blockGrasped ?c)
(not (_p-objectBeneath ?c ?b))
(_p-objectBeneath ?c ?a) ))
```

The above steps are repeated for each action model provided by LbN. This, along with the prior steps, completes the domain definition.

STEP 5: Complete Problem Definition. The final step in the translation process is to complete the problem definition. First, all of the objects that are of the types identified in *STEP 1* are added as to the `:objects` clause; objects are added according to the name. For the world state represented in Figure 2, this would include: `a`, `b`, `c`, `d` and `claw`. As described in *STEP 2*, ground type predicates are asserted for all of the types these objects inherit.

Finally, the `MakeSo` request is translated as described at the beginning of the section and asserted as the ground formula in the `:goal` clause.

This completes the the translation to PDDL. The PDDL form is then provided to a planner (in our case, an implementation of Graphplan), and a plan is produced. The plan produced by the planner consists of a sequence of ground actions. For the example we have constructed throughout this section, the plan is:

```
((MoveClaw C Claw Table))
((Grasp C Claw Table))
((MoveClaw A Claw Table))
((Release Claw C A))
```

The translation schemes between the IL and PDDL versions of the actions make the translation back to IL simple, producing:

```
MoveClaw(C)
Grasp()
MoveClaw(A)
Release()
```

This ground plan may solve the particular problem instance defined in the problem. However, we generalize this solution using the same lifting technique we used above.

This new capability also opens the possibility for the Teacher to make use of MABLE's planning ability to teach compound actions whose defcode procedure consists of the steps in the plan. To do this, the Teacher can provides an explicit name and syntax for the action to be learned, along with arguments corresponding to the arguments the teacher expects the action would take to achieve the `MakeSo` request. For example:

```
DoWith(MoveOnto(a,b))
```

Combined with the lifted solution plan, the next composite procedure learned by MABLE is as follows:

```
MoveOnto(?x, ?y)
Do(InSequence(
  MoveClaw(?x),
  Grasp(),
  MoveClaw(?y),
  Release() ));
```

Of course, this learned procedure is not necessarily general enough. For example, there are a number of preconditions not represented here:

1. The Claw may already be above the block to be moved, so the initial move isn't needed.
2. The Claw may currently grasp an object, so need to first release the currently held block, and do so *not* above the block that is to be moved.
3. The Claw may already have the target block grasped.

These conditions still need to be learned (either through trial and error or further instruction)

Conclusion

Our ability to successfully create a plan, execute it and have it successfully achieve the target goal given the current state, is based entirely on how *complete* and *accurate* our current action models are.

There are (at least) two places where our action models may fail us: (1) An *incomplete model for planning*: our action models may not be complete in the sense that they may not provide enough information for the planner to identify a sequence of actions that will transform the world state to the goal state; and (2) an *incomplete or incorrect model of the world*: our action models may also be inaccurate with respect to the world: we may think an action will bring about some change in the world when in fact it either doesn't, or the effect is conditional on other world states being true or other actions taken, or our actions may have other effects we haven't represented. Also, successfully constructing a plan and achieving the world state does not mean our action models are correct, as pointed out at the end of the last section.

All of these are possibilities and we won't know without trying to form a plan and executing it. In this sense, forming a planning model for a given problem description in IL, attempting to build the plan, and then attempting to execute the plan, are all components of an experiment. Failure at any step of the process from translation, to planning, to execution can be very informative: (1) Failure during planning may provide information about deficits in the completeness of action models, and could lead to questions and other tests or explorations to fill out our action models. This is a topic for future work, likely including looking a plan-generation traces and analyzing the partial plan graph; and (2) Successfully generating a plan but then failing during execution provides information about where things might go wrong. Here, we will want to look at the execution trace and analyze where the world appears to have diverged from the expected plan execution model.

The translation framework we have presented here (in detail!) still leaves quite a number of questions unanswered and many directions for improvement. The following are directions for future work:

1. Augment the planner to analyze a partial plan graph or plan trace in cases of planning failure.
2. Handle conditional plans: take different actions depend on results of a test

3. Handle planning with objects that may be "created" and "destroyed" by actions. For example, the action of "cutting" a piece of paper in half in a sense "destroys" the original paper and now produces two separate objects. This is a deep issue in planning domain knowledge engineering, but one that will have to be addressed in full human-instructable computing in the real world.
4. Handle numeric values; MABLE learns numeric functions and many of the domains involve numeric property values that are affected by actions.
5. Finally, handling IL lists. They will likely be handled similar to how other composite objects are manipulated, but a special set of list manipulation actions need to be defined and appropriately represented.

Acknowledgments

This work was supported in part by contract HR0011-07-C-0060 with the Defense Advanced Research Projects Agency (DARPA) as part of the DARPA Bootstrapped Learning Program.

References

- Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.
- Crespo, Y.; Marquès, J.; and Rodriguez, J. 2002. On the translation of multiple inheritance hierarchies into single inheritance hierarchies. In Black; Ernst; Grogono; and Sakkinen., eds., *Proceedings of th Inheritance Workshop at ECOOP 2002*, 30–37.
- Dao, M.; Huchard, M.; Libourel, T.; Pons, A.; and Villerd, J. 2004. Proposals for multiple to single inheritance transformation. In *Proceedings of MASPEGHI'04: 3rd Workshop on Managing SPECIALIZATION/Generalization Hierarchies*.
- Kitano, H.; Asada, M.; Kuniyoshi, Y.; Noda, I.; and Osawa, E. 1997. Robocup: The robot world cup initiative. In *Proceedings of the first international conference on autonomous agents*, 340–347. New York, NY: ACM.
- Mailler, R.; Bryce, D.; Shen, J.; and O'Reilly, C. 2009. MABLE: A modular architecture for bootstrapped learning. In *The Eighth International Conference on Autonomous Agents and Multiagent Systems (AAMAS09)*.
- McDermott, D., and the AIPS'98 Planning Competition Committee. 1998. PDDL - the planning domain definition language. Technical report, Yale University, Available at: www.cs.yale.edu/homes/dvm.
- Oblinger, D. 2008. Bootstrapped learning – external materials. Technical report, <http://www.sainc.com/bl-extmat/>.

From Requirements and Analysis to PDDL in itSIMPLE_{3.0}

Tiago Stegun Vaquero^{1,2} and José Reinaldo Silva¹ and Marcelo Ferreira³
Flavio Tonidandel³ and J. Christopher Beck²

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 paper 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 au-

tomatically provided to be read by AI planners. The planning solutions given by these solvers are then simulated and evaluated in the tool.

This paper 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.

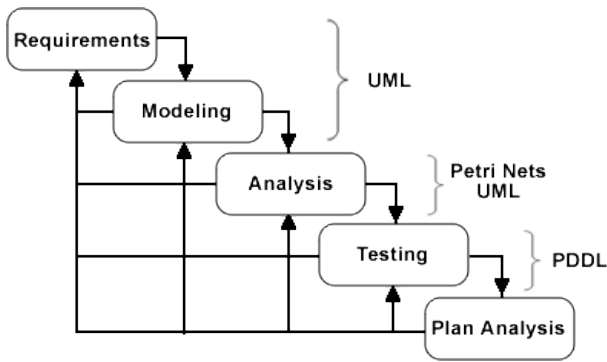


Figure 1: Design process in itSIMPLE_{3.0}

In the proposed process, requirements gathering and modeling are performed 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.

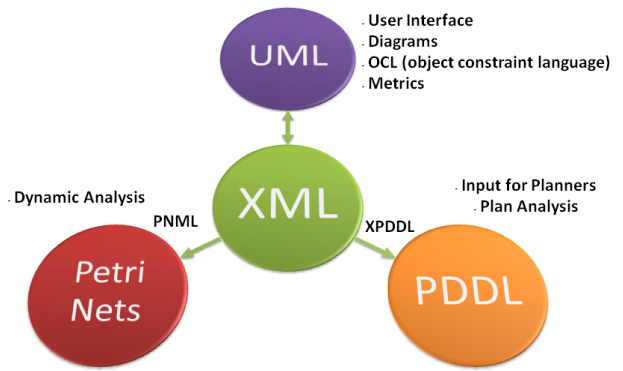


Figure 2: The architecture of the integrated languages

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.

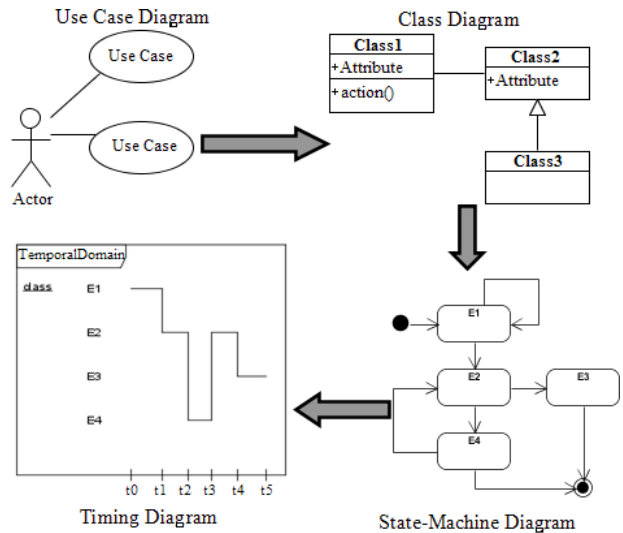


Figure 3: UML diagrams in itSIMPLE_{3.0}

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 s_1 and goes to the state s_2 . In this example, an 'initial state' points to s_1 . 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

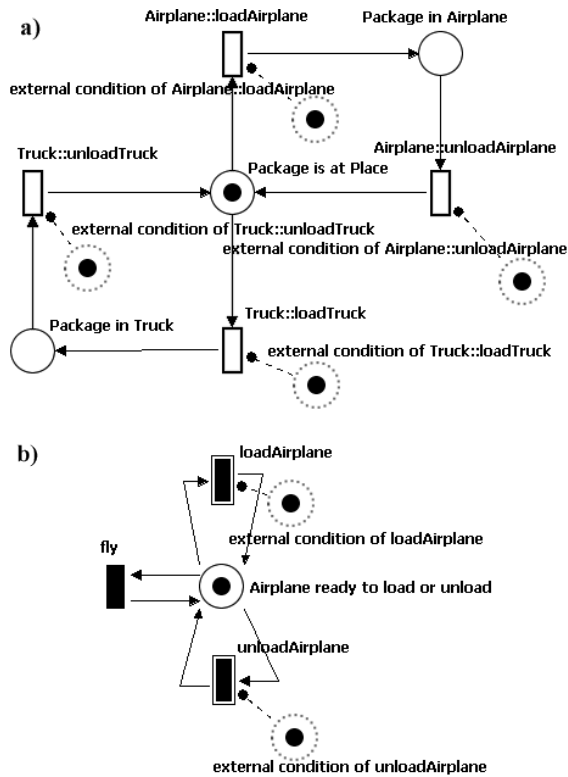


Figure 4: Modular Analysis of (a) Package and (b) Airplane

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 flu-

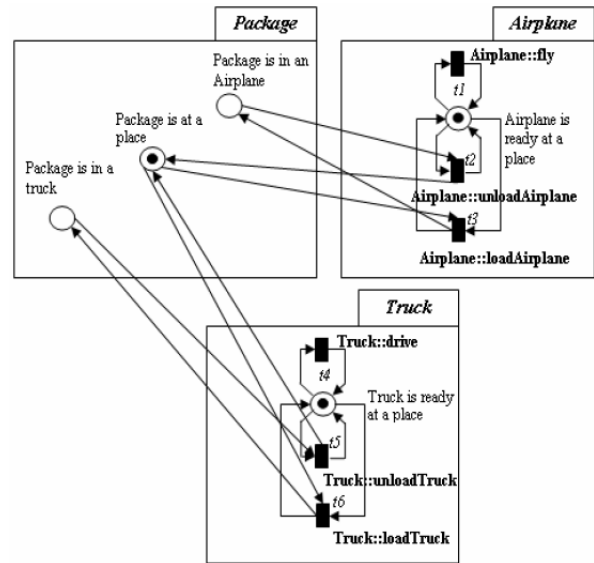


Figure 5: Interface Analysis of the classes Package, Truck and Airplane

UML	PDDL
	<pre>(:types MyClass - object ...)</pre>
	<pre>(:types Class0 - MyClass Class1 - MyClass MyClass - object ...)</pre>

Figure 6: Mapping PDDL types from class diagrams

ents in `:functions` of PDDL.

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 3.1¹, 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

¹PDDL3.1 <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, let's 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
	<pre>(:predicates (attr ?cla - Class0))</pre>
	<pre>(:functions (attr ?cla - Class0))</pre>
	<pre>(:predicates (attr ?cla0 - Class0 ?cla1 - Class1)) OR (:functions (attr ?cla0 - Class0 - Class1))</pre>
	<pre>(:predicates (attr ?p0 - ClassP0 ... ?pn - ClassPN))</pre>
	<pre>(:predicates (role ?cla0 - Class0 ?cla1 - Class1))</pre>
	<pre>(:predicates (role1 ?cla0 - Class0 ?cla1 - Class1) (role0 ?cla1 - Class1 ?cla0 - Class0))</pre>

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
	<pre>(:action act (:parameters ?p0 - Class0 ?p1 - Class1 ... ?pn - ClassN) (:precondition ...) (:effect ...))</pre>

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

²User documentation. <http://dlab.poli.usp.br>

OCL Expression	PDDL
p0 = p1 where: p0 and p1 are parameters of the operator. The case $p0 \triangleleft p1$ is identical to $\text{not}(p0 = p1)$.	$(= ?p0 ?p1)$
p0.attr = true where: p0 is a parameter of the operator and attr is an attribute of p0 that has a Boolean type.	$(\text{attr } ?p0)$ A false value would be represented as: $(\text{not}(\text{attr } ?p0))$
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.	$(\text{attr } ?p0 ?o1 \dots ?on)$
p0.role = p1 where: p0 and p1 are parameters of the operator; and role is a <i>rolename</i> , with multiplicity "1" or "0..1", for any association between two classes of p0 and p1. If " \triangleleft " is used instead of "=", the mapping is $\text{not}([\text{expression}])$.	$(\text{attr } ?p0 ?p1)$
p0.role->exists(p p = p1) where: p0 and p1 are parameters of the operator; and role is a <i>rolename</i> , with multiplicity greater than 2 or "*", of any association between classes of p0 and p1. If " \triangleleft " is used instead of "=", the mapping is $\text{not}([\text{expression}])$.	$(\text{attr } ?p0 ?p1)$
attr(o1,...,on) = true where: attr is a global Boolean parameterized attribute. If " \triangleleft " is used instead of "=", the mapping is $\text{not}([\text{expression}])$.	$(\text{attr } ?o1 \dots ?on)$

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. Accordingly 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

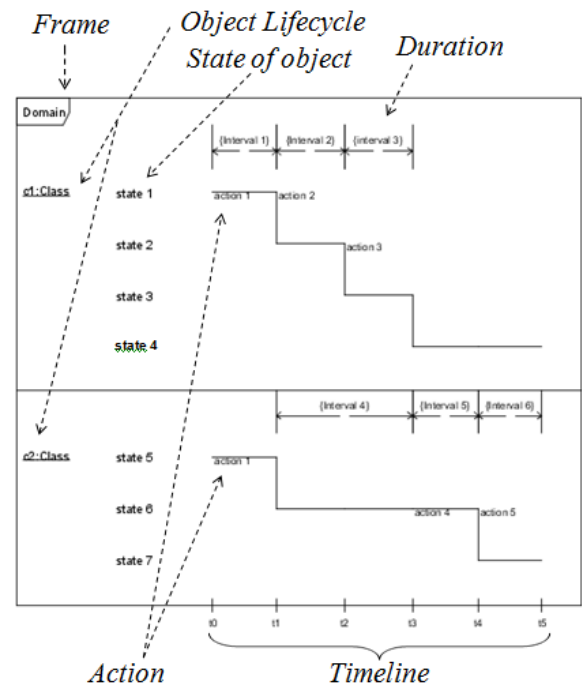


Figure 10: Timing diagram in itSIMPLE_{3.0}

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.

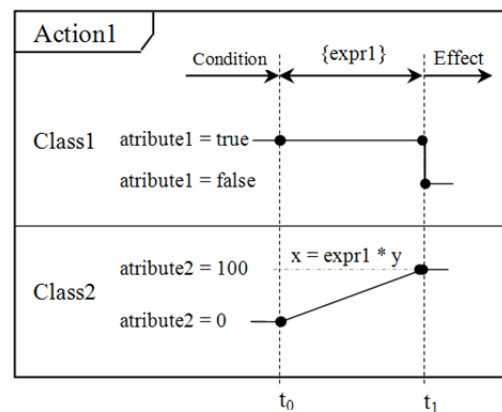


Figure 11: Timing diagram of an action in itSIMPLE_{3.0}

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.

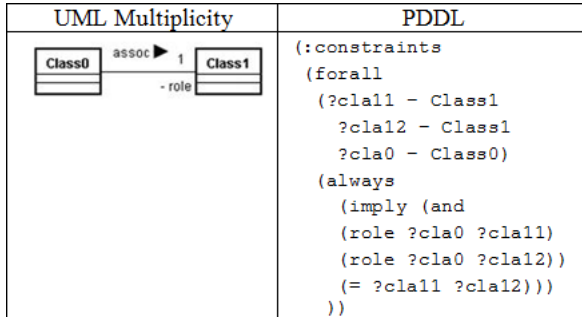


Figure 12: Mapping association multiplicity to PDDL constraints

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 - Robot) (<(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.

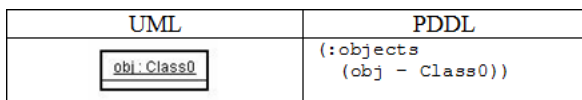


Figure 13: Mapping PDDL objects from object repository

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*.

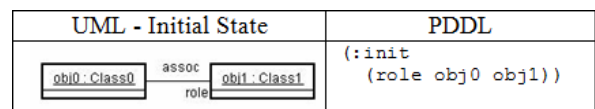


Figure 14: Mapping PDDL init from object diagram

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-xx1, LPG-TD, LPG, hspss, 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* (SysML³) to the itSIMPLE’s framework for model verification and validation.

³www.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*.