ICAPS 2005
Monterey, California

ICAPS05

WS2

Workshop on the Role of Ontologies in Planning and Scheduling

Juan Fernández Olivares
University of Granada, SPAIN

Eva Onaindía
Polytechnical University of Valencia, SPAIN

ICAPS 2005
Monterey, California, USA
June 6-10, 2005

Workshop on the Role of Ontologies in Planning and Scheduling

Juan Fernández Olivares
University of Granada, SPAIN

Eva Onaindía
Polytechnical University of Valencia, SPAIN

Workshop on the Role of Ontologies in Planning and Scheduling

Table of contents

http://icaps05.icaps-conference.org/

ICAPS 2005
Monterey, California

Workshop on the Role of Ontologies in Planning and Scheduling

Preface

Ontologies are becoming increasingly important in several AI fields (such as knowledge management and integration, cooperative problem solving, knowledge acquisition and knowledge-based systems, e-commerce and the Semantic Web) and, at present, there is also an increasing interest about their use in Planning and Scheduling (P&S) systems.

In the field of P&S ontologies allow , on the one hand, knowledge exchange between intelligent processes (performed both by humans and other intelligent systems) in real world applications. On the other hand they allow to describe more complex domains and problems, since they are based on very rich representation languages (for example, semantic web languages as RDF, OWL,OWL-S). These languages are more expressive than those presently used in P&S, since they use the Open World Assumption rather than the planner-friendly Closer World Assumption. However , they are really "static" languages and do not include (generally) knowledge about states and state change, what prevents their "direct" application in current P&S systems. In any case, the way the use of ontologies and such languages impact the field of AI P&S needs to be investigated.

In summary, the integration between ontologies and P&S techniques is demanding more attention both from theorists and practitioners, and there are many different approaches in the literature on this issue. Therefore, one of the main topics of this workshop will be the study of the benefits that an ontology-based knowledge representation could bring into current P&S technologies.

The goal of the workshop consists in trying to understand and discuss different ways of integration between ontologies techniques and intelligent planning. This can be seen as a different, general approach to bridge the gap that currently exists between the very efficient P&S technology and its application to real world. Papers accepted in this workshop present theoretical / practical work and report experiences with applications on the following topics: definition of planning ontologies that overcome drawbacks detected on the real application of standard planning languages (such as PDDL), use of ontologies as the basis for knowledge exchange between different components of intelligent systems, where planning is a core technology, planning applications that use ontology concepts for their development and deriving planning domain models from existing ontological knowledge

Organizers

- *Eva Onaindía, Technical University of Valencia (Spain)*
- *Juan Fdez-Olivares, University of Granada (Spain)*

Programme Committee

- *Jose Luis Ambite, University of Southern California (USA)*
- *Jim Blythe, University of Southern California (USA)*
- *Luis Castillo, University of Granada (Spain)*
- *Lee McCluskey, University of Huddersfield (U.K.)*
- *Christoph Lenzen, German Space Operation Center (Germany)*
- *Falk Mrowka, German Space Operation Center (Germany)*
- *Angelo Oddi, ISTC-CNR (Italy)*
- *Bernd Schattenberg, University of Ulm (Germany)*
- *Evren Sirin, University of Maryland (USA)*
- *Laura Sebastiá, Technical University of Valencia (Spain)*
- *Paolo Traverso, ITC/IRST (Italy)*
- *Ioannis Vlahavas, Aristotle University of Thessaloniki (Greece)*

Importing Ontological Information into Planning Domain Models

T. L. McCluskey and S.N. Cresswell
School of Computing and Engineering
The University of Huddersfield, Huddersfield HD1 3DH, UK
lee,s.n.cresswell@hud.ac.uk

Abstract

We investigate an approach to alleviating the problems of knowledge engineering for AI Planning by importing object structures and object behaviours from shared knowledge structures. In this paper we describe our first steps at devising a method to import knowledge from an application ontology into a form usable within a planning domain model. We evaluate an implemented tool called OWL2OCL which assists in the translation of ontological information into a form usable by a planner.

Introduction

In the field of Knowledge Acquisition much attention has been given to the idea of re-using and sharing bodies of both general and application-specific knowledge in the form of ontologies. The benefits of this within fields such as eScience and Knowledge Management have led to the rapid development of ontologies in both science and commerce. Ontologies are (formal) vocabularies that relate terms together in the form of a precise specification. These can range from UML diagrams, to concept hierarchies, through to axioms in Description Logic (DL). The majority of published, formal, ontologies appear to be represented in a DL variant, although some are represented in First Order Logic using a standard syntax. There are several tools available for building ontologies such as Protege-2000 (Gennari *et al.* 2003). Superficially these are similar to GUIs such as GIPO (McCluskey, Liu, & Simpson 2003) which are used to acquire information about objects in AI Planning. However, the dominant focus of ontological engineering is to capture static knowledge whereas planning is more concerned with dynamic knowledge.

Our work in Knowledge Engineering in AI Planning is aimed at alleviating the task of deploying a planning engine in a particular application, and making planning technology in general more accessible. Experience in engineering planning applications indicate a range of problem areas related to the domain modelling phase. The initial choices of how to model the application in terms of relational predicates, classes and states is a major task. The problem of re-inventing models for every application is acute: given there are many existing domain models, how can one re-use previously captured planning domain knowledge?

In the area of AI Planning, there have been rapid development of general planning engines which input PDDL domain models. The problem with these planners is that for each application the user has to assemble an adequate knowledge base in the form peculiar to this type of planner. This requires a knowledge engineering task of translating and acquiring knowledge in an application into the input form of PDDL. One solution that we are exploring is to use an existing ontology if a suitable one exists. Rather than a knowledge engineer or expert crafting the domain model, they identify the relevant ontologies to be used in the application, and a translator assembles the knowledge in a planner-friendly representation. In other words, the translator would induce domain model structure and dynamics in an operational form suitable for input to a planning engine.

We speculate that the future of planning techniques is bound up with the semantic web: progress in the use of ontological information could also enhance the ability of a web agent to automatically assemble adequate knowledge in order to solve its own planning problems. Given an application ontology and a high level goal description, an agent will have to generate a plan to achieve this goal. In current technological terms, this amounts to generating a planning domain model, acquiring, tuning and executing a suitable planning engine.

In this paper we devise a process to import ontologies in a form that they can be used as the basis of a planning domain model. We evaluate its application to both contrived and previously published ontologies, and summarise the lessons learned from the experiments.

The Input Language

Developments in the www consortium have led to the development of a standard web ontology language called OWL (Patel-Schneider, Hayes, & Horrocks 2004). There are three versions of OWL: Lite, DL and Full. For our input language we have chosen the abstract syntax of OWL DL, which can be output from Protege-2000. As is common with ontologies in description logic, the presentation consists of a set of restricted first order axioms, with variables implicit. This has an advantage over graphical input - after axioms are changed, the best hierarchy can be automatically derived using a subsumption reasoning tool. This avoids the need for the user to try to engineer and re-engineer the application to fit into neat hierarchies.

We input domain models using Protege-2000 with the OWL plug-in, DL reasoner and graphics cability. This involves formulating the domain in terms of classes and subclasses of objects, properties between classes and other classes, property restrictions, property domains and ranges, and so on. Class hierarchies can be automatically generated and displayed for validation, and class definitions can be automatically checked for consistency.

Examples of OWL ontologies are shown in Appendix A and B. As a description logic, OWL is designed for capturing the class hierarchy of objects (classes are like unary predicates in FOL). Statements in OWL are about concepts/classes, which are defined extensionally via the objects they contain. Only binary predicates are allowed, and these are used to relate classes with other classes and with data type values.

The Target Language

According to the OWL literature (Patel-Schneider, Hayes, & Horrocks 2004), a class defines a group of individuals that belong together because they share some properties. We specialise this in AI planning - objects should share the same class if they share the same *dynamic behaviour*, i.e. they are related to each other in a more concrete way:

- The set of properties and relations which they can have is common.
- They share a common set of states (where a state is a collection of properties/relations that hold to be true).
- They share the same state-change behaviour, that is the same transitions between states.

The definition above is fundamental to an OCL (Liu & McCluskey 2000) domain model, and we will use OCL as the target language. In OCL, objects are grouped into $sorts$ if they share the same behaviour, and each part of a world state that describes one particular object is called a substate. A characterisation of the possible substates of an object in a sort is called a substate class description. OCL is the base language of the GIPO planning environment (McCluskey, Liu, & Simpson 2003). This contains a wide range of acquisition and validation tools for manipulating planning domain models. Once generated, the translated model can be fed into GIPO where it can be refined, and output as PDDL if necessary (see figure 1).

Ontology Translation: OWL2OCL

Our task is to import a pre-existing ontology in order to use it as the basis of a planning domain model as shown in figure 1. This is similar to the general problem of "Contextualising Ontologies" as discussed by Bouquet et al (Bouquet *et al.* 2004): changing the 'global' meaning of an ontology to a local, subjective view of a domain. Although the object-structure of OWL and OCL are superficially linked, there are many problems to be overcome. Two particular problems are as follows:

– OWL ontologies are composed under the assumption of open-world reasoning, whereas planning domain models tend to assume closed-world reasoning. As a simple example, two individuals in an OWL model are not necessarily distinct unless they can be proved so. In the planning world, we assume they are distinct unless it can be proved they are the same.

– there is no guarentee that an imported ontology is going to be complete or adequate for the planning task. In particular, ontologies written in description logic are typically *static* - they include no explicit information on the dynamics of the world.

Our initial solution is therefore to produce a *heuristic* transformation from an ontology language (OWL) to a planning domain modelling language (OCL) and let the user further validate and refine the model using the GIPO environment. As a starting point, this transformation should be able to re-create faithfully the static information of a planning domain model which has been hand-translated into an ontology. The example ontologies in Appendix A and B below are examples of this as they are encodings of familiar planning worlds.

Our overall method in this initial work assumes that a planning domain model is to be generated where all objects belong to a unique distinct sort. We will map the (not necessarily disjoint) description logic classes of the ontology into a set of disjoint classes which will be primitive sorts in the planning domain. Each class will have its set of attributes generated.

The Translation Process

The translation and subsequent domain acquisition process is in three stages. OWL2OCL is aimed at implementing the first two of these steps:

- Step 1: Build up a disjoint set of OCL object sorts such that for each sort, if an object belongs to it then the set of properties it can have are the same as any other object in that sort. Build up a set of predicates describing and relating the sorts.
- Step 2: Induce the individual state descriptions that objects of each sort can inhabit.
- Step 3: Build up a set of operator schema and refine the model, using GIPO. One effective way to complete the model with operator schema would be used to induce operator descriptions from training sequences as explained in (McCluskey, Richardson, & Simpson 2002).

As we may have a complex ontology, how do we choose at what level to group the OWL objects into OCL sorts? If we choose only the leaf classes of the ontology, we may over-discriminate. Instead, we choose to base the grouping on the domain and range declarations of properties. Objects which may take the same set of properties are grouped in the same sort. For example, if we have a property which is declared as:

```
ObjectProperty(
  drives domain(person) range(vehicle))
```

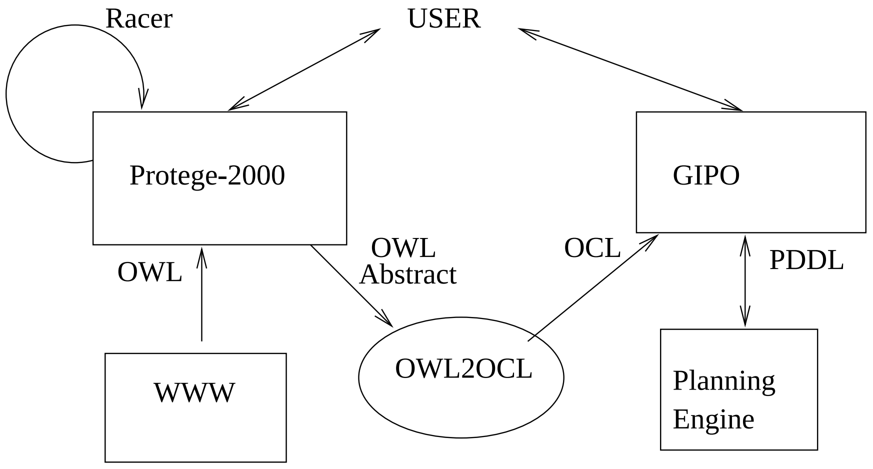


Figure 1: OWL2OCL in context

then we assume that this relation can exist between any `person` and any `vehicle`, so these classes will form useful groupings for the purpose of collecting substates.

In more detail, Step 1 and Step 2 are refined as follows:

1. Collect the set C' of OWL classes which form either the domain or range for some property.
2. Collect the set C'' of intersections of the classes in C' which are inhabited by at least one individual. In this way we create disjoint classes of objects each of which can have properties from a well-defined set - call this set S_p for each OCL class S.
3. For each class induce the substates class description, that is a characterisation of each of the states that an object of a class may inhabit, as follows. For each class $S \in C''$:
   (a) Collect the 'value' information for each individual (property) from O. That is collect all the instances of properties in S_p. We assume this an instance of a unique substate (that is all the information - the truth values of properties - is given).
   (b) For each object, generalise its set of property values to create a new class description. Form a set of new class descriptions (without duplicate equivalent classes).

What will be formed is a partial domain model which contains static and some (heuristic) dynamic knowledge.

Worked Examples

To give an idea of the transformation, we show two worked examples, using ontologies that have been written in OWL to reflect the original domain model.

(i) Appendix A: the Rocket World. The input OWL abstract syntax is shown together with our auto-generated domain model. For clarity, the namespace information has been removed.

Step 1: by examination of the domains and ranges of the declared properties, OWL2OCL establishes that the following classes are relevant: {Cargo, Location, Rocket, Level}. All of the classes are occupied with individuals and no individual occupies more than one of the classes. Hence the set of occupied combinations of classes is identical.

Step 2: The substates occupied within each sort are induced. In figure 2, each cargo object has either an 'at' or 'in' property, leading OWL2OCL to induce the substates of the class Cargo to be $\{at(Cargo, Location))\}$ and $\{in(Cargo, Rocket)\}$.

Object	Property	Value
`C1`	at	Paris
`C2`	at	London
`C3`	in	R1
`C4`	at	London

Figure 2: Object Properties in the OWL Rocket World

(ii) Appendix B: the Dockworkers World, taken from (Ghallab, Nau, & Traverso 2004). As with the Rocket World, this world was input into the GUI of Protege-2000 using the original planning world as a basis. The OCL output in Appendix B shows that OWL2OCL has succeeded in translating the object and property information into the more compact OCL formalism. Some dynamic, substate definitions have been induced successfully - for example Containers are in Piles and may or may not be at the top, and Piles, Cranes and Robots have dynamic location information. However, the quality and quantity of this is dependent on the amount and spread of 'value' information about individuals in the ontology.

In the OWL representation, unary predicates which are static are naturally representated as defined classes. As we process this information, it is used in creating the sorts in OCL. We assume that dynamic truth-valued data relating to individuals will be represented in the ontology using boolean data properties. For example, in the Dockworkers domain, we record whether a location is occupied, and this is dynamic data which can be changed by a plan operator.

```
DatatypeProperty(a:occupied Functional
  domain(a:Location)
  range(xsd:boolean))
```

In the OCL representation, the boolean data properties are translated into a pair of unary predicates `occupied(Location)` and `not_occupied(Location)`.

Experiments with generally available ontologies

OWL2OCL can translate information from any ontology output in OWL abstract form from Protege-2000. The discussion below draws from the use of Horrocks' 'people+pets' ('mad cows') ontology from Manchester, not because it forms the basis of any sensible planning domain, but because it at least includes a reasonably large ontology which uses many of the features of OWL and includes some 'value' information. A general result of the experiments is that OWL2OCL will only work well if the ontology contains information about the individuals in the domain (ABox information), and preferably has the individuals spread about their possible states.

Use of Description Logic Reasoners

In some cases it is possible to infer a more refined classification, e.g. by looking at domain and range restrictions for properties. An example is in the people+pets ontology: the concept 'dog' has no declared superclass, but a description logic reasoner can infer that 'dog' is a subclass of 'animal'. We have assumed that all implicit subclass relationships that can be found by a description logic reasoner have been made explicit, and we have used the RACER DL reasoner (Haarslev & Moeller 2001) via Protégé2000 (Knublauch *et al.* 2004) for this purpose (see figure 1).

Inverse roles

OWL allows for roles to have a declared inverse form, e.g. `has_pet` is the inverse of `is_pet_of`. There is a choice in the automatic handling of inverse roles in the translation. A useful factor in making this choice is whether the property is declared as functional in one of its forms. The functional form is more convenient to handle, as we are then able to constrain the number of occurrences of the property in a substate to 0 or 1. For example the `is_pet_of` property is functional (a pet has a single owner, but an owner can have many pets), so it is preferable to associate the property with the pet class rather than the owner.

Property hierarchies

OWL allows for the declaration of hierarchies of properties. For example, `has_father` is a subproperty of `has_parent`. We do not handle such properties in the conversion, as we cannot expect a planner to do inference using the role hierarchy. In principle, we could represent such cases by completing the descriptions by adding all implied super-properties for each instance. However, if the properties in question are to be considered dynamic in the planning domain, the planning operators would be required to adjust the properties appropriately at every level of the role hierarchy. The problems of representing *transitive* properties are similar to those presented by hierarchies of properties. Although we could compute transitive closure in advance, we cannot expect to define planning operators which can recompute it. [1]

Defining Planning Operators

In summary, the partial domain model created by OWL2OCL creates a useful starting point for domain acquisition. Within the current OWL standard it appears impossible to define change or action, and hence it has not been possible to explicitly extract this information from a domain model. To complete the planning domain we need to use GIPO to:

1. Refine or alter the structure of the substate class definitions which has been extracted from the ontology.
2. Fill in missing class definitions.
3. Define planning operators in terms of transitions between the defined class definitions.

Related Work

(McNeill, Bundy, & Walton 2004) describes a system for generating a PDDL planning domain from an ontology describing an agent system represented in the KIF formalism. The ontology is not restricted to static domain information, but already contains descriptions of actions. Hence the problem tackled is one of translating the planning problem between formalisms, as opposed to our aim of extracting state descriptions as a basis for defining actions.

PDDAML is a tool which translates between a Web-PDDL and DAML. As with the work of McNeill et al., it differs from OWL2OCL in that it assumes similar 'semantic content' in its target and source, and does not propose to induce or hypothesise any existing knowledge. Also, PDDAML was written with the benefit of a PDDL ontology definition (McDermott, Dou, & Qi 2004). In contrast we are trying to import and extract from ontologies as much planning related information as possible, rather than to create a web-planning language. We recognise that the current purpose of ontologies are generally to record the static information about a domain; this makes them unusable as a planning definition. To help the engineering of a dynamic domain, existing ontologies are in our work used as the starting point.

Conclusions

In this paper we have argued that knowledge available from online ontologies can be usefully imported into AI Planning domain models to act as the first step to creating a planning domain model. We have described a tool that translates 'OWL abstract' ontologies into the OCL plan language. The success of this tool depends on the amount and distribution of factual and individual knowledge (that is a

[1] although PDDL 2.2 language allows for axioms could be used for this.

full 'Abox'). Experiments with this tool illuminate other problems inherent in this approach: OWL ontologies tend to contain some richer information (such as property hierarchies) than planning domain models require, but also not sufficient dynamic information. It would be useful to know which data is static and which is dynamic in an OWL ontology. Unfortunately this is impossible to detect automatically from a static snapshot of the ontology.

Currently, there is no explicit way of representing time and resources in OWL, in such a way that it can be translated into planning operator schema. In this respect, other common languages for representing ontologies such as KIF are better suited (McNeill, Bundy, & Walton 2004). For future work we plan to further develop the technique to extract more information from OWL relevant to planning. For example, number restrictions recorded in OWL can usefully be used to infer constraints on what is a valid substate, and OWL's MinCardinality restrictions could be used to add extra properties to a substate.

References

Bouquet, P.; Giunchiglia, F.; van Harmelen, F.; Serafini1, L.; and Stuckenschmidt, H. 2004. Contextualizing ontologies. *Journal of Web Semantics* 1(4):325 – 343.

Gennari, J. H.; Musen, M. A.; Fergerson, R. W.; Grosso, W. E.; Crubezy, M.; Eriksson, H.; Noy, N. F.; and Tu, S. W. 2003. The evolution of Protege: an environment for knowledge-based systems development. *Int. J. Hum.-Comput. Stud.* 58.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann ISBN 1-55860-856-7.

Haarslev, V., and Moeller, R. 2001. RACER system description. In *International Joint Conference on Automated Reasoning, IJCAR'2001*.

Knublauch, H.; Fergerson, R. W.; Noy, N. F.; and Musen, M. A. 2004. The protégé OWL plugin: An open development environment for semantic web applications.

Liu, D., and McCluskey, T. L. 2000. The OCL Language Manual, Version 1.2. Technical report, Department of Computing and Mathematical Sciences, University of Huddersfield .

McCluskey, T. L.; Liu, D.; and Simpson, R. M. 2003. GIPO II: HTN Planning in a Tool-supported Knowledge Engineering Environment. In *The Thirteenth International Conference on Automated Planning and Scheduling*.

McCluskey, T. L.; Richardson, N. E.; and Simpson, R. M. 2002. An Interactive Method for Inducing Operator Descriptions. In *The Sixth International Conference on Artificial Intelligence Planning Systems*.

McDermott, D.; Dou, D.; and Qi, P. 2004. PDDAML: An automatic translator between PDDL and DAML. http://www.cs.yale.edu/homes/dvm /daml/pddl_daml_translator1.html.

McNeill, F.; Bundy, A.; and Walton, C. 2004. An automatic translator from KIF to PDDL. In *Workshop of the UK Planning and Scheduling Special Interest Group, PLANSIG 2004*.

Patel-Schneider, P. F.; Hayes, P.; and Horrocks, I. 2004. OWL web ontology language semantics and abstract syntax W3C recommendation 10 February 2004. http://www.w3.org/2004/OWL/.

Appendix A

The Rocket World:

```
Ontology(
 Class(Rocket partial)
 Class(Location partial)
 Class(Cargo partial)
 Class(Level partial)

 DisjointClasses(Rocket Location Cargo)

 ObjectProperty(at   domain(Cargo)
                     range(Location) )
 ObjectProperty(position
                     domain(Rocket)
                     range(Location) )
 ObjectProperty(fuel domain(Rocket)
                     range(Level) )
 ObjectProperty(in domain(Cargo)
                     range(Rocket) )

 Individual(R1 type(Rocket)
      value(position Paris)
      value(fuel full ) )
 Individual(R2 type(Rocket)
      value(position Paris)
      value(fuel full ) )
 Individual(C1 type(Cargo)
      value(at Paris) )
 Individual(C2 type(Cargo)
      value(at London) )
 Individual(C3 type(Cargo)
      value(in R1) )
 Individual(C4 type(Cargo)
      value(at London) )
 Individual(Paris type(Location))
 Individual(London type(Location))
 individual(full type(level))
 individual(empty type(level))
)

% Domain auto-generated from an OWL ontology

domain_name(owl).

% Objects

objects(cargo,[c1,c2,c3,c4]).
objects(level,[empty,full]).
objects(location,[london,paris]).
objects(rocket,[r1,r2]).

% Predicates

predicates(
   at(cargo,location),
   position(rocket,location),
   fuel(rocket,level),
   in(cargo,rocket)]).

% Object Class Definitions

substate_class(cargo,Cargo,[
   [at(Cargo,Location)],
   [in(Cargo,Rocket)]]).
substate_class(rocket,Rocket,[
   [position(Rocket,Location),
    fuel(Rocket,Level)]]).
```

Appendix B

The Dockworkers World:

```
Ontology(

 Class(a:Container partial)
 Class(a:Crane partial)
 Class(a:Location partial)
 Class(a:Pile partial)
 Class(a:Robot partial)

 ObjectProperty(a:adjacent Symmetric
  domain(a:Location)
  range(a:Location))
 ObjectProperty(a:at Functional
  domain(a:Robot)
  range(a:Location))
 ObjectProperty(a:attached Functional
  domain(a:Pile)
  range(a:Location))
 ObjectProperty(a:belong Functional
  domain(a:Crane)
  range(a:Location))
 ObjectProperty(a:holding
  domain(a:Crane)
  range(a:Container))
 ObjectProperty(a:in
  domain(a:Container)
  range(a:Pile))
 ObjectProperty(a:loaded
  domain(a:Robot)
  range(a:Container))
 ObjectProperty(a:on
  domain(a:Container)
  range(a:Container))
 ObjectProperty(a:top
  domain(a:Container)
  range(a:Pile))

 DatatypeProperty(a:empty Functional
  domain(a:Crane)
  range(xsd:boolean))
 DatatypeProperty(a:occupied Functional
  domain(a:Location)
  range(xsd:boolean))
 DatatypeProperty(a:unloaded Functional
  domain(a:Robot)
  range(xsd:boolean))

 Individual(a:a
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:pa))
 Individual(a:b
  type(a:Container)
  value(a:on a:a)
  value(a:in a:pa))
 Individual(a:c
  type(a:Container)
  value(a:on a:b)
  value(a:in a:pa)
  value(a:top a:pa))
 Individual(a:d
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:qa))
 Individual(a:e
  type(a:Container)
  value(a:on a:d)
  value(a:in a:qa))
 Individual(a:f
  type(a:Container)
  value(a:on a:e)
  value(a:in a:qa)
  value(a:top a:qa))
 Individual(a:g
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:pb))
 Individual(a:ga
  type(a:Crane)
  value(a:belong a:la)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:gb
  type(a:Crane)
  value(a:belong a:lb)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:gc
  type(a:Crane)
  value(a:belong a:lc)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:gd
  type(a:Crane)
  value(a:belong a:ld)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:ge
  type(a:Crane)
  value(a:belong a:le)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:gf
  type(a:Crane)
  value(a:belong a:lf)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:h
  type(a:Container)
  value(a:on a:g)
  value(a:in a:pb))
 Individual(a:i
  type(a:Container)
  value(a:on a:h)
  value(a:in a:pb)
  value(a:top a:pb))
 Individual(a:j
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:qb))
 Individual(a:k
  type(a:Container)
  value(a:on a:j)
  value(a:in a:qb))
 Individual(a:l
  type(a:Container)
  value(a:on a:k)
  value(a:in a:qb)
  value(a:top a:qb))
 Individual(a:la
  type(a:Location)
  value(a:adjacent a:li)
  value(a:occupied "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:lb
  type(a:Location)
  value(a:adjacent a:lj)
  value(a:occupied "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:lc
  type(a:Location)
  value(a:adjacent a:lj)
  value(a:occupied "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:ld
  type(a:Location)
  value(a:adjacent a:lj))
 Individual(a:le
  type(a:Location)
  value(a:adjacent a:li))
 Individual(a:lf
  type(a:Location)
  value(a:adjacent a:lj))
 Individual(a:li
  type(a:Location)
  value(a:adjacent a:lj))
 Individual(a:lj
  type(a:Location))
 Individual(a:m
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:pe))
 Individual(a:n
  type(a:Container)
  value(a:on a:m)
  value(a:in a:pe))
 Individual(a:o
  type(a:Container)
  value(a:on a:n)
  value(a:in a:pe)
  value(a:top a:pe))
 Individual(a:p
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:qe))
 Individual(a:pa
  type(a:Pile)
  value(a:attached a:la))
```

```
 Individual(a:pallet
  type(a:Container)
  value(a:top a:pf)
  value(a:top a:pc)
  value(a:top a:qf)
  value(a:top a:qc))
 Individual(a:pb
  type(a:Pile)
  value(a:attached a:lb))
 Individual(a:pc
  type(a:Pile)
  value(a:attached a:lc))
 Individual(a:pd
  type(a:Pile)
  value(a:attached a:ld))
 Individual(a:pe
  type(a:Pile)
  value(a:attached a:le))
 Individual(a:pf
  type(a:Pile)
  value(a:attached a:lf))
 Individual(a:q
  type(a:Container)
  value(a:on a:p)
  value(a:in a:qe))
 Individual(a:qa
  type(a:Pile)
  value(a:attached a:la))
 Individual(a:qb
  type(a:Pile)
  value(a:attached a:lb))
 Individual(a:qc
  type(a:Pile)
  value(a:attached a:lc))
 Individual(a:qd
   type(a:Pile)
  value(a:attached a:ld))
 Individual(a:qe
  type(a:Pile)
  value(a:attached a:le))
 Individual(a:qf
  type(a:Pile)
  value(a:attached a:lf))
 Individual(a:r
  type(a:Container)
  value(a:on a:q)
  value(a:in a:qe)
  value(a:top a:qe))
 Individual(a:ra
  type(a:Robot)
  value(a:at a:la)
  value(a:unloaded "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:rb
  type(a:Robot)
  value(a:at a:lb)
  value(a:unloaded "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:rc
  type(a:Robot)
  value(a:at a:lc)
  value(a:unloaded "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:s
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:pd))
 Individual(a:t
  type(a:Container)
  value(a:on a:s)
  value(a:in a:pd))
 Individual(a:u
  type(a:Container)
  value(a:on a:t)
  value(a:in a:pd)
  value(a:top a:pd))
 Individual(a:v
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:qd))
 Individual(a:w
  type(a:Container)
  value(a:on a:v)
  value(a:in a:qd))
 Individual(a:x
  type(a:Container)
  value(a:on a:w)
  value(a:in a:qd)
  value(a:top a:qd))

)


% Domain auto-generated from an OWL ontology

domain_name(owl).


% Objects

objects(container,[a,b,c,d,e,f,g,h,i,j,k,l,
      m,n,o,p,pallet,q,r,s,t,u,v,w,x]).
objects(crane,[ga,gb,gc,gd,ge,gf]).
objects(location,[la,lb,lc,ld,le,lf,li,lj]).
objects(pile,[pa,pb,pc,pd,pe,pf,qa,qb,qc,
      qd,qe,qf]).
objects(robot,[ra,rb,rc]).


% Predicates

predicates(
   adjacent(location,location),
   at(robot,location),
   attached(pile,location),
   belong(crane,location),
   holding(crane,container),
   in(container,pile),
   loaded(robot,container),
   on(container,container),
   top(container,pile),
   empty(crane),
   not_empty(crane),
   occupied(location),
   not_occupied(location),
   unloaded(robot),
   not_unloaded(robot)]).
```

```
% Object Class Definitions

substate_class(container,Container,[
   [on(Container,Container2),
                in(Container,Pile3)],
   [on(Container,Container2),
                in(Container,Pile3),
     top(Container,Pile4)],
   [top(Container,Pile2),
                top(Container,Pile3),
    top(Container,Pile4),
                top(Container,Pile5)]]).
substate_class(crane,Crane,[
   [belong(Crane,Location2),empty(Crane)]]).
substate_class(location,Location,[
   [],
   [adjacent(Location,Location2)],
   [adjacent(Location,Location2),
                occupied(Location)]]).
substate_class(pile,Pile,[
   [attached(Pile,Location2)]]).
substate_class(robot,Robot,[
   [at(Robot,Location2),unloaded(Robot)]]).
```

Planning from rich ontologies through translation between representations

Fiona McNeill, Alan Bundy, Chris Walton
Centre for Intelligent Systems and their Applications,
School of Informatics,
University of Edinburgh
{f.j.mcneill,a.bundy,c.d.walton}@ed.ac.uk

Abstract

The richness and expressivity of standard ontology representations and the limitations on expressivity required by modern planners have resulted in a situation where it is hard for an agent both to have a rich ontology and be capable of efficient planning. We discuss how translation between different kinds of representation can allow an agent to have different versions of the same ontology, so that it can simultaneously meet different demands of expressivity. We introduce our ontology refinement system (ORS), in which these ideas are implemented.

Using rich ontologies for planning

There is currently a disparity between the richness of ontological representation used in multi-agent systems, and ontologies used in Semantic Web like environments, and the richness of ontological representation used in planning. This disparity comes about due to the different requirements of each domain.

In a multi-agent system and environments such as the Semantic Web, rich, expressive ontologies are desirable. Such ontologies facilitate the encoding of detailed domain information: information about infinite domains, meta-information about ontological objects, complex class hierarchies which allow for slot information in classes, use of the open world assumption, and so on. Some common ontological representations for multi-agent systems, such as Knowledge Interchange Format (KIF) [3], are full first-order, and are thus extremely expressive. Other common ontological representations: for example, description logic based ontologies, such as RDF and OWL, are less expressive than full first-order logic, owing to the tractability problems associated with inference in full first-order logic, but, nevertheless, retain a high level of expressivity.

In popular planning representations, much of this expressivity is removed. Languages such as PDDL [2], resemble first-order languages; however, this is an illusion. Most planners that take domain information from PDDL files are propositional and thus, though PDDL provides a first-order window on to the propositional space, anything that is expressed in PDDL must be translatable into propositional logic. This places restrictions on what can be expressed; there are some ontological objects that are expressible in a first-order representation but not in a less expressive representation: for example, quantification over infinite domains. Additionally, the closed world assumption is normally used in planning.

One might argue that this disparity comes about partly due to the separation of the planning community and the ontology community: state-of-the-art planners are usually designed and assessed on how well they perform purely with respect to planning considerations; there is much less emphasis on how to balance good plan formation performance with consideration of other issues, such as dealing with richer ontologies. However, there is a more fundamental issue underlying this disparity. Automated planning is very difficult, largely because the search problems involved in finding even short plans are vast. The only feasible way to solve these problems is to reduce the search space. Thus the most important aspect of planning representations is that they are not difficult to search through; this inevitably leads to loss of expressivity.

There are two approaches to this problem. One approach is to attempt to balance the demands of an expressive ontological representation with those of a tractable planning representation. The resulting representation will be less expressive than a standard ontological representation and less efficient for producing plans than a standard planning representation; however, the advantage of combining both facets in a single representation may be thought to outweigh these problems. However, we believe that the best solution to the problem is provided by an alternative approach: allowing ontological knowledge to be represented in different ways, depending on the current required functionality, and translating between the different representations as necessary. Inevitably, information is lost through translation from a more expressive to a less expressive representation. However, if the most expressive representation is retained after it is translated to a less expressive representation, then the agent still has access to its complete ontology as well as to the less expressive representation that can be used, for example, for planning. We believe that the demands of a planning representation are incompatible with the demands of a standard ontological representation. The ability to form plans quickly and efficiently is vital to agents that are attempting to plan within multi-agent systems, but equally, the ability to represent complex information within their ontology is important. We believe that an attempt to combine the two needs in a single representation requires too great a loss to both domains, and therefore our approach is to develop translation processes between different kinds of ontological representations.

It should be noted that by ontology, we mean both the representation language of the domain and the knowledge base expressed in that language. Thus, in our terms, a PDDL ontology would consist of a domain file and one or more problem files; a KIF ontology would define the vocabulary but also contain the facts expressed in that vocabulary. Therefore, an ontology can be altered either by changing the representational language or by changing the facts expressed in that language (as occurs during plan execution).

In this paper, we describe the translation process that we have implemented, and explain its role in ORS. We describe the context that ORS is designed to work in: that of a multi-agent, service-based architecture, and mention how this context affects the kind of translation that is necessary in ORS. We discuss how our evaluation of ORS demonstrates that this translation process is successful, and show how, as a result, ORS can be used to dynamically refine ontologies in a planning environment.

Translating from KIF to PDDL

Currently, we have implemented one such translation process: translating from KIF ontologies to a PDDL representation. In translating from KIF, we have already tackled some of the most severe problems inherent in such an approach: KIF is full first-order, and thus the loss of expressivity in our existing translation process is at least as severe as the loss of expressivity in translating any ontological representation to PDDL, although our system does not currently deal with full KIF but only with a subset of it; thus not all these issues have been confronted. Certainly, there would be different implementation issues when translating from a language such as OWL to PDDL, but the theoretical problems surrounding loss of expressivity would be less. Full details of this translation process can be found in [8]; in this section we briefly discuss some of the chief issues involved in this process, which is illustrated in Figure 1.

We have implemented this translation process as part of our ontology refinement system (ORS), which is discussed in the following section. The translation process is important because the system deals with

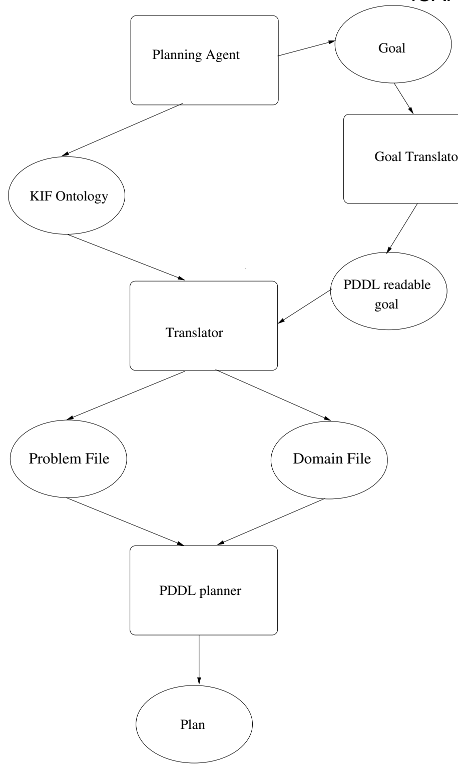


Figure 1: Architecture of Translation System

agents that are operating within a multi-agent system, but which also need to form plans. In our system, the KIF ontology is considered to be the definitive ontology: the agent's true understanding of the state of the world is represented in the KIF ontology. Other representations are used only when this is practically necessary for the agents, and any changes made to these other representations - for example, by actions being performed - must be made to the KIF ontology also, so that the KIF ontology is always up to date with respect to the agent's understanding of the world. In particular, the KIF ontology is translated to PDDL when the agent wishes to form a plan. ORS uses the planner Metric-FF [4], but any PDDL planner could be substituted in with minimal affect to the system. The difference in expressivity between KIF and PDDL mean that the PDDL representation is not completely equivalent to the KIF ontology. However, the agent has not lost information during this translation process, because it still has access to the original KIF ontology; it is simply that the agent's full ontology is not necessarily represented in PDDL. This loss of expressivity has a disadvantage in that it is possible that there are valid plans that could be formed from the KIF ontology that cannot be found from the more limited PDDL ontology; however, it has a strong advantage in that it is now possible to use this knowledge to efficiently form plans, and, moreover, this loss of information is concerned only with the formation of this particular plan and does not affect any other parts of the system.

In terms of expressivity, there are some ontological structures that are expressible in a full first-order representation but not in a representation that must be translatable into propositional logic; for example:

1. quantification over infinite domains,
2. uninstantiated variables

In our system, the first of these issues is not a concern, because it does not currently deal with KIF ontologies that contain quantification over infinite domains. A complete translation process that dealt with such KIF ontologies would need some way to represent this in PDDL, which might be through some kind of finite abstraction. However, this would create larger expressive differences between the original KIF and the resulting PDDL than we have currently experienced.

The second of these issues, however, is a concern. Not allowing uninstantiated variables in the representation constrains a plan to involve only individuals that are already present in the ontology. In many planning situations, that is quite acceptable; it is usually desirable for a plan to be fully instantiated before execution commences. However, there are some situations where this is not the case. Consider, for example, a plan which involved booking a flight, for which a flight reservation number was given, and then automatically checking in for the flight, using this reservation number. The fact that such a reservation number exists is important during plan formation; however, it is not only unnecessary, but impossible, to know the particular instantiation of the number before plan execution begins; this can only be instantiated during execution. In such situations, the propositional restrictions of PDDL present problems. We circumvent this problem through the use of *pseudo-variables*, which are declared as individuals when the PDDL files are produced, but are then uninstantiated by the agent when it interprets the plan produced by the planner. Thus, for example, an action rule that involved buying a ticket would force this reservation number to be the individual `PV1`. The translation process keeps track of how many pseudo-variables must be declared, which pseudo-variable refers to which uninstantiated object, and so on. Details of how this is done are given in [8].

There are many further implementational difficulties in the translation process. Many of these centre around the fact that PDDL-based planners keep track of the state whilst the plan is being formed, whereas such a concept has no meaning for a static ontology. Thus, for example, the value of numerical functions is automatically tracked in a PDDL planner and does not need to be stated explicitly, as it would in a static ontology. In PDDL, an initial declaration is made (if appropriate); for example:
`(= (Money ?Agent) 1000)`
and thereafter `(Money ?Agent)` can be referred to without explicit reference to this value; the planner keeps this information explicitly in its internal knowledge base, and thus this does not need to be represented explicitly in PDDL. If calculations are performed, these are again done without any explicit reference to the value; for example:
`(> (Money ?Agent) (Price ?Item))`
means that the value attached to `(Money ?Agent)` must be greater than the value of `(Price ?Item)`.

In a static representation such as KIF, such values have to be declared explicitly because there is no mechanism for keeping track of them, as there is in a planner. Thus instantiated numerical functions are declared in the same way as any other instantiated predicate; for example:
`(Money ?Agent 1000)`.
The function is always referred to in this way. Calculations such as the one described above must be represented using these explicit arguments; for example, the above calculation would be represented as:
`(Money ?Agent ?Amount)` ∧
`(Price ?Item ?Cost)` ∧
`(> ?Amount ?Cost)`
Such different requirements force extensive rewriting of the KIF ontological objects, particularly the action rules, in order to create valid PDDL ontological objects.

There are several other issues that need to be dealt with in such a translation, which are detailed in [8]; for example, dealing with arithmetic operators. However, most of these are merely a recoding of information, and do not affect the expressivity.

As mentioned above, our translation does not deal with full KIF, but with a constrained version that is sufficient to express the ontological information required in our system. A full KIF ontology would present further translation issues. We believe, however, that it is possible to produce a PDDL version even of full KIF that is correct, though not complete, with respect to the KIF ontology. This translation process allows us to take an abstraction of a rich ontology, so that it can be used in efficient planning.

The way in which ontologies are handled in our system requires translation only to be one way; we have not done any work on translating from PDDL back to KIF. Creating a valid KIF ontology from the PDDL ontology should not, on the whole, be much harder than translating from KIF to PDDL; for example, instead of folding a numerical predicate `(Money ?Agent ?Amount)` into `(Money ?Agent)`, this would be unfolded by adding a variable name. This variable name could be arbitrarily chosen, but would need to be consistently used: for example,
`(> (Money ?Agent) (Price ?Item))`
would first be converted to
`(> ?Var1 ?Var2)`
and then the meanings of these variables would need to be declared appropriately:
`(Money ?Agent ?Var1)` ∧ `(Price ?Item ?Var2)`

However, since there is usually some loss of expressivity in the translation process from KIF to PDDL, a retranslation of the PDDL ontology would result in a KIF ontology that was likely to be less expressive than the original ontology. It is thus better, if the situation allows, to always translate from a more expressive ontology to a less. Thus the most expressive version of the ontology can be considered to reflect the true understanding of the agent, and less expressive versions are produced when they are necessary, used to perform their role, such as planning, and then discarded. If any changes produced when using these less expressive ontologies are made directly to the most expressive ontology, rather than completely retranslating the less expressive ontology, then the most expressive ontology retains its full expressiveness, whilst also being kept up to date.

Example Translation

Consider the situation in which a planning agent (PA) is given a goal to purchase an on-line plane ticket. In order to achieve this goal, several steps must be carried out. For example, the agent must locate a ticket-selling agent, it must ensure it has sufficient funds, it must work out the correct origin and destination for the flight, and so on. Clearly, before the agent can act, it must have a plan for how to achieve the goal. Therefore, as soon as the agent identifies a goal, it sends the whole ontology, together with a suitable representation of this goal, to the translator. PDDL files for the ontology are produced, which can then be sent to the planner. Once the PA has the plan, it can then begin to execute the plan steps. In this short example, we have the following ontological objects in the original KIF ontology:

```
(Define-Frame PA
  :Own-Slots ((Instance-Of Agent))
  :Axioms ((Money PA 500)))

(Define-Frame Edinburgh
  :Own-Slots ((Instance-Of City))
  :Axioms ((Flight Edinburgh London 300)))

(Define-Individual London (City))

(Define-Function Flight
  (?Place-0 ?Place-1) :-> ?Value
  :Def (And (Place ?Place-0)
            (Place ?Place-1)
            (Number ?Value)))

(Define-Function Money
  (?Agent-0) :-> ?Value
  :Def (And (Agent ?Agent-0)
            (Number ?Value)))

(Define-Class Agent (?X)
  :Def (And (Thing ?X)))

(Define-Class City (?X)
  :Def (And (Place ?X)))

(Define-Class Place (?X)
  :Def (And (Thing ?X)))
```

```
(Define-Axiom Book-Flight :=
 (=>
  (And (Flight ?Agent-Loc ?Conf-Loc
                             ?Price)
       (Money ?Agent ?Amount)
       (< ?Price ?Amount))
  (And (Has-Ticket ?Agent)
       (= ?Newamount
            (- ?Amount ?Price))
       (Money ?Agent ?Newamount)
       (Not (Money ?Agent ?Amount)))))
```

There are objects referred to in the axiom that are not defined in the ontology section above: these are omitted for brevity.

Our translation would produce the following PDDL domain file from the above KIF ontology:

```
(define (domain domain Ont)
  (:requirements :strips :fluents :typing)

  (:predicates
    (Agent ?Agent)
    (Place ?Place)
    (City ?City)
  )

  (:functions
   (Money ?Agent)
   (Flight ?Place1 ?Place2)
  )

  (:action Book-Flight
    :parameters (?Agent ?City1 ?City2)
    :preconditions (And
                     (City ?City1)
                     (City ?City2)
                     (< (Flight
                          ?City1 ?City2)
                     (Money ?Agent))
                     (Agent ?Agent))
    :effects (And
               (Has-Ticket ?Agent)
               (decrease
                 (Money ?Agent)
                 (Flight ?City1 ?City2)))
  ))
```

and the following PDDL problem file:

```
(define (problem problemOnt)
 (:domain domainOnt)
 (:objects London Edinburgh PA)
 (:init
  (Agent PA)
  (City London)
  (City Edinburgh)
  (= (Money PA) 500)
  (= (Flight Edinburgh London) 300)
 )
 (:goal
   (Has-Ticket PA)))
```

Ontology refinement in a planning context

In this section, we briefly introduce our ORS system, to illustrate the role of the translation process in the system, and the role of the two different ontologies. More detailed information about ORS can be found in [1, 7].

The central function of ORS is to allow agents to refine their ontologies when they discover that they are incompatible with the ontologies of other agents. It is common in current multi-agent systems that an assumption is made that agents have the same ontology; ontological incompatibility leads to failure. However, this is often not a reasonable assumption. Off-the-shelf ontologies are frequently updated, as they are found, during use, to be too limited for the task required, or to be encoding excessive information that is found to be unnecessary, or the ontology moderators decide it would be useful to extend or restrict the domain encoded in the ontology. This results in off-the-shelf ontologies that exist in several different versions. Additionally, individual users might take an off-the-shelf ontology and alter it for their own ends. Thus it is not uncommon for agents to have ontologies that are broadly similar, but that are different in certain respects, and for these differences to result in failure of the agents to interact successfully.

If two agents have vastly different ontologies, it is difficult to see how they could interact in a fully automated manner, since they would have no basis for understanding one another. However, if two agents have ontologies that are, for the most part, the same, but that differ in some respects, then there is potential for fully automated interaction between these agents, even when they need to deal with the parts of their ontologies that are mismatched, because the parts of their ontologies that they share gives them a basis for understanding one another, and terms in which to discuss what their ontological mismatches might be. The purpose of ORS is to allow agents to use this shared portion of their knowledge to diagnose how their ontologies are mismatched, and to patch their ontologies so that successful interaction becomes possible.

The way in which two different versions of an ontology may differ depends on their representation. A common difference may be the removal or addition of complete ontological objects. However, a more interesting difference is when existing ontological objects are modified in some way. In a first-order ontology, such as KIF, this may happen in, for example, the following ways:

- changing the arity of a predicate so that it could encode more or less information;
- changing the name of a predicate: this is most interesting and easier to detect if the name is changed to a related name, perhaps a subclass or superclass of the existing name; for example, the predicate `Money` may be changed to the predicate `Dollars`;
- changing the class requirement for an argument of a predicate: this, again, may involve changing the class to a sub- or super- or otherwise related class; a similar alteration is a change in the order of arguments in a predicate;
- an action rule may be altered by adding or removing a precondition, or by adding, removing or otherwise altering an effect of an action.

Ontological mismatch is not inherently related to planning; this could occur in any situation where ontologies are used by inference. However, we have been investigating this problem in a planning context. ORS is used within a multi-agent system, in which the agents are either planning agents (PAs), or service-providing agents. Although the system is compatible with the existence of more than one PA, we make the assumption that only one plan is being executed at one time, and thus consider that there can only be a single active PA in the system at any one time. This assumption is not compatible with complex multi-agent systems, and future versions of the system will work on relaxing this assumption. In ORS, plan steps are always tasks that can be performed by other agents: for example, a `buy-ticket` action might be performed by a *ticket-selling agent*.

The most significant way in which our system differs from standard methods of ontology mapping and merging [5] is that we do not assume that we have access to all of both ontologies. Instead, we assume that we have access to all of one of the ontologies (the PA's), but that the other ontology, that of the service-providing agent, which is not usually owned by the same user as the PA, is only revealed through direct questions that are put to the service-providing agent by the PA, and by information gleaned from plan execution failure. We believe that this is a more realistic approach in situations such as agent interaction in large multi-agent systems, as not only is it impractical to completely map two ontologies where only a small change may be necessary, but also there may be concerns about secure and commercially sensitive information that mean agents would be unprepared to reveal their entire ontologies. Additionally, we assume that any agent operating in such an environment would be able to answer direct questions and, if it is a service-providing agent, perform tasks; it is not a normal part of agent interaction to reveal large sections of ontology, and we therefore cannot expect it of agents. Two agents may have very different underlying representations but still be able to interact via an agent protocol; thus each agent must glean sufficient information via this protocol; fully mapping the two ontologies is not helpful.

Exploring ontological mismatch in a planning context is facilitated by the fact that a planning context provides a clear indication that some kind of ontological mismatch has occurred: a service-providing agent refuses to perform an action that the PA believed to be performable in the current situation, thus indicating that the service-providing agent and the PA are not using identical ontologies; and a clear indication that the patching performed has removed the particular mismatch: the previous point of failure no longer causes a problem.

The architecture of ORS is illustrated in Figure 2, and is as follows:

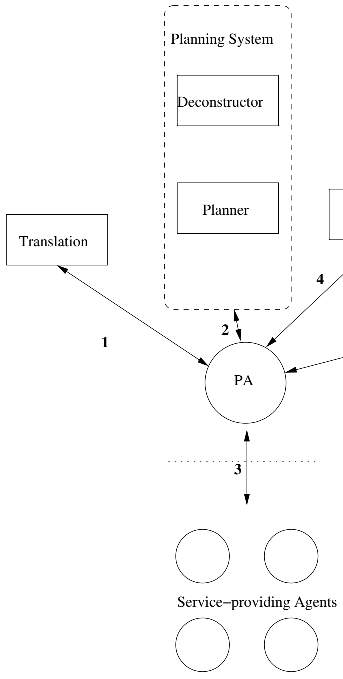


Figure 2: Architecture and interaction of the dynamic ontology refinement system.

1. The PA sends its KIF ontology to the translation system. The trigger for this is the PA receiving a goal that it is required to fulfil. The translation system returns appropriate PDDL files, with the goal correctly inserted, and also returns a version of the ontology in Prolog readable syntax, which is required for direct interpretation of the ontology by the PA, and also in the plan deconstructor (see step 2).
2. The PA sends the PDDL files to the planner, which produces a plan for achieving the goal. This plan is interpreted by the agent and translated into a Prolog readable version, which is then sent, together with the Prolog readable version of the ontology, to a *plan-deconstructor*, whose role it is to link the plan steps to the underlying ontology. One drawback of using propositional planners, which we do not discuss in detail here, is that they cannot provide first-order level information about how the plan produced is related to the underlying ontology: what action-rule was used to perform each action, why the preconditions of these rules were thought to be valid in the particular situations, and so on. This information is sometimes important in linking plan execution failure to mismatches in the underlying ontology. The purpose of the plan deconstructor is to provide this information. The deconstructor steps through the produced plan, meta-interpreting it with respect to the ontology, and returns this information as a *justification* of the produced plan. It thus acts in a similar way to a first-order planner, but circumvents the massive search problem faced by such planners by using a plan that has already been produced by an efficient planner. Further information about this process can be found in [9].
3. The plan, annotated with a justification for each step, is returned to the PA and execution begins. This execution occurs in an agent communication system, where the PA can locate the agents with which it needs to interact.
4. If failure occurs, information about the communication thus far, together with the relevant parts of the justification, are sent to the diagnosis system. Further agent communication is usually required to pinpoint the exact source of the problem. In some situations, it is impossible to accurately diagnose the source of the mismatch.
5. If an exact, or at least plausible, diagnosis can be made, this diagnosis is passed to the refinement system, which implements the relevant change to the KIF ontology of the agent.
6. The process is repeated, with the updated KIF ontology being retranslated into PDDL, and a new plan formed. This is repeated until the goal is achieved, until the diagnosis system fails to return an applicable diagnosis or until the refinements to the ontology determined by the diagnostic process result in an ontology from which it is not possible to reach the goal.

Example Mismatches

Before a service-providing agent can perform a service for a planning agent, it must ensure that all the preconditions for performing the service are fulfilled. The value of some of these preconditions it can ascertain for itself, some must be checked with other agents, and some must be checked with the PA. For example, the service-providing agent may need to check how much money PA has, so that it can ensure this is enough for providing the service. It might thus put the following question to PA:
SPA: `(Money PA Dollars ?Amount)`
By consulting the example ontology in the previous section, we can see that this does not correspond to PA's ontology, where money is represented as a binary predicate and does not include the `Currency` argument. Thus PA cannot appropriately respond to the service-providing agent's question, and must re-

ply:
PA: no.

The service-providing agent will then refuse to perform the service for PA, because it can see that the preconditions are not met. PA will then use the diagnostic system to analyse why failure occurred, which in this case is fairly clear: there is a mismatch between the service-providing agent's representation of `Money` and the PA's. The PA must then refine its ontology (described above), so that the following changes are made:

```
(Define-Frame PA
  :Own-Slots ((Instance-Of Agent))
  :Axioms ((Money PA MetaVar 500)))

(Define-Function Money
  (?Agent-0 ?Currency-0) :-> ?Value
  :Def (And (Agent ?Agent-0)
            (Currency ?Currency-0)
            (Number ?Value)))

(Define-Axiom Book-Flight :=
 (=>
  (And (Flight ?Agent-Loc ?Conf-Loc
                              ?Price)
       (Money ?Agent ?Currency
                              ?Amount)
       (< ?Price ?Amount))
  (And (Has-Ticket ?Agent)
       (= ?Newamount
            (- ?Amount ?Price))
       (Money ?Agent ?Currency
                           ?Newamount)
       (Not (Money ?Agent ?Currency
                              ?Amount))
      )))
```

Note that all the ontological objects mentioned in the first example but not listed here are those that are not affected by the change.

Once these refinements have been made, the process is repeated. After translation this time, the ontology will become:

```
 (define (domain domain Ont)
   (:requirements :strips :fluents :typing)

   (:predicates
     (Agent ?Agent)
     (Place ?Place)
     (City ?City)
   )

   (:functions
    (Money ?Agent ?Currency)
    (Flight ?Place1 ?Place2)
   )

   (:action Book-Flight
     :parameters (?Agent ?City1 ?City2)
     :preconditions (And
                      (City ?City1)
                      (City ?City2)
                      (< (Flight
                           ?City1 ?City2)
                         (Money ?Agent
                                  ?Currency))
                                (Agent ?Agent))
     :effects (And
                (Has-Ticket ?Agent)
                (decrease
                  (Money ?Agent ?Currency)
                  (Flight ?City1 ?City2)))
   ))
```

and the following PDDL problem file:

```
 (define (problem problemOnt)
  (:domain domainOnt)
  (:objects London Edinburgh PA)
  (:init
   (Agent PA)
   (City London)
   (City Edinburgh)
   (= (Money PA Dollars) 500)
   (= (Flight Edinburgh London) 300)
  )
  (:goal
    (Has-Ticket PA)))
```

Note that during refinement of the KIF ontology, the `Money` fact changes from `(Money PA 500)` to `(Money PA MetaVar 500)`. This `MetaVar` is used because we cannot know the correct way of instantiating this new variable. The class of `MetaVar` is restricted to being `Currency` because of the function definition of `Money`. However, when this ontology is used for planning, this variable must be instantiated. An appropriate instantiation is therefore chosen, and the fact becomes `(= (Money PA Dollars) 500)`. This creates some risk of error, as we cannot be sure that `Dollars` is the correct instantiation.

We may deduce from this refinement that a currency argument is also relevant to modify the `Price` argument of the `flight` function: this may become `(Flight ?Origin ?Destination ?Price ?Currency)`
However, ORS does not make any such deductions, refining only those ontological objects for which it has direct evidence of mismatch. If this refinement were necessary, so would only come to light through further plan failure.

Evaluation of ORS

The implementation of ORS described above has been completed, and has been evaluated with respect to off-the-shelf ontologies for which we have different versions. We have been somewhat hampered in this endeavour by the fact that is not easy to find existing ontologies for situations such as the ones we are investigating. Because the system is designed primarily for an environment that is still in development, the Semantic Web, there are not many existing ontologies for such situations. Additionally, the planning community tends not to keep large bodies of ontologies that have been updated over time, since the ontologies themselves have not historically been of much interest to the planning community; they are built to provide a basis for testing planners. The continual update and alteration of ontologies that is found in the more traditional ontology communities has not been so important in planning. The very problem that the system is attempting to aid: that of closer interlinking between standard ontologies and planning; has made it difficult to find suitable ontologies with which to evaluate the system. The most important requirement for our evaluation, which is the most difficult to fulfil, is that different versions of the ontology should be available, so that we can test our system against mismatches that agents would really encounter if they were using different versions of the same ontology.

Our solution has been to take existing ontologies, designed for encoding complex information but not for planning, and overlay a planning scenario on top of these. Such ontologies are not immediately appropriate for planning, not solely because of the expressivity issues, which ORS is specifically designed to deal with, but also because they are static, and not designed for use in a dynamic situation. There are very few, or no, action rules that tell us how to alter the ontology, as would be found in an ontology designed for planning. We have therefore taken these ontologies and added such information to them, so that we can use them in a planning domain.

We have tested ORS using six different ontologies. Three of these are off-the-shelf ontologies: PSL (Process Specification Language) [11], SUMO (Suggested Upper Merged Ontology) [12] and AKT (Advanced Knowledge Technologies) [10]. The other three are planning ontologies for which we have developed plausible ontological mismatches: a blocks world ontology, a lift scheduling ontology and a conference booking ontology.

We have demonstrated that the system can be successfully used to translate these ontologies, form plans from the translated ontology, execute these plans until an ontological mismatch causes plan execution failure, diagnose the root of the mismatch, patch the original ontology accordingly and retranslate and replan from this updated ontology. Many of the mismatches we have encoded between the PA and the service-providing agents are genuine mismatches that would occur if they were using different versions of these ontologies.

Another aspect of ontologies that makes evaluation of the system difficult is that ontologies are currently updated on the assumption that these updates will be read and interpreted by humans. Thus, although many of these changes can be detected automatically by our system, many cannot. For example, there is little attempt to describe new ontological objects directly in terms of existing ontological objects so that an agent can interpret how they should fit in to its ontology; instead, devices such as using similar names are used, and commenting is used to describe what has been done, so that it is immediately obvious to a human user how these new objects fit into the ontology, but it is difficult to deduce this in a fully automated manner. We believe that if systems such as ORS become more widely used, more effort will be made to update ontologies in a way that would facilitate the automated patching of mismatches, and thus the process of ontology refinement will become easier.

Conclusions

The integration of different fields of AI is extremely important to the development of the subject. The techniques developed in planning and scheduling could be very useful to many other fields, such as the Semantic Web, e-commerce, multi-agent systems, and so on. The application of planning to these domains is hampered by the different representational requirements. We believe that forcing representational constraints on users to make these fields more compatible will not be successful: ontologists will not be willing to lose the current richness of ontologies; planners will not be willing to make do with less efficient planners. We suggest, therefore, that the most appropriate solution to this problem is to allow many different representations of the same, or similar, knowledge to exist simultaneously, and that work must be done on developing translation processes between these representations.

In this paper, we have described how we have done this for two representations: KIF and PDDL; and how this has allowed us to develop a working system where agents have extremely expressive ontologies, but are also capable of efficient planning. We have described ORS, which is fully implemented and can successfully refine ontologies, both in plausible planning situations and with genuine ontological mismatches gleaned from off-the-shelf ontologies that are available in different versions. An important aspect of this system is the ability to handle and translate between different ontological representations, so that an agent can use different levels of expressivity, depending on the task at hand.

This approach could be extended to many different ontological representations. A translator between DAML and PDDL has already been developed [6]. There are different implementation issues in the different translation processes, and different degrees of expressivity loss, but there is no reason why a correct, though not necessarily complete, version of any ontological representation cannot be rendered in any other. As well as creating further translation processes, we are also interested in using these with ORS to allow ontology refinement for different ontological representations. The potential mismatches that would occur would vary: a representation such as OWL could not be altered in the same way as a full first-order representation, and thus some effort would be required to adapt ORS to each representation. Nevertheless, the framework provided by the system is not representation dependent, and could be used in many different circumstances.

References

[1] F.McNeill, A. Bundy, and C. Walton. Facilitating agent communication through detecting, diagnosing and refining ontological mismatch. In *Proceedings of the KR2004 Doctoral Consortium*. AAAI Technical Report, in press.

[2] Maria Fox and David Long. An extension to PDDL for expressing temporal planning domains. Available from Durham Planning Group webpage:. http://www.dur.ac.uk/computer.science/research/ stanstuff/planpage.html.

[3] M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Stanford, CA, USA, 1992.

[4] Jorg Hoffmann. FF. http://www.mpi-sb.mpg.de/ hoffmann/ff.html.

[5] Yannis Kalfoglou and Marco Schorlemmer. Ontology mapping: the state of the art. *The Knowledge Engineering Review*, 18:1:1–31, 2003.

[6] Drew V. McDermott, Dejing Dou, and Peishen Qi. An automatic translator between pddl and daml. http://www.cs.yale.edu/homes/dvm/daml/ pddl_daml_translator1.html.

[7] F. McNeill, A. Bundy, and C. Walton. Diagnosing and repairing ontological mismatches. In *Proceedings of the second starting AI Researchers' symposium*, Valencia, Spain, August 2004.

[8] Fiona McNeill, Alan Bundy, and Chris Walton. An automatic translator from KIF to PDDL, December 2004. http://planning.cis.strath.ac.uk/plansig/index.php ?page=past22,Cork.

[9] Fiona McNeill, Alan Bundy, Chris Walton, and Marco Schorlemmer. Plan execution failure analysis using plan deconstruction, December 2003. http://planning.cis.strath.ac.uk/plansig/index.php ?page=past22.

[10] AKT Project. http://www.aktors.org, 2002.

[11] PSL. http://www.mel.nist.gov/psl/.

[12] SUMO. http://ontology.teknowledge.com/.

Planning and the Process Specification Language

Michael Grüninger
Institute for Systems Research
University of Maryland
College Park, MD 20742
gruning@cme.nist.gov

Joseph B. Kopena
Geometric and Intelligent Computing Laboratory
Dept. of Computer Science
Drexel University
3141 Chestnut Street
Philadelphia, Pa 19104
tjkopena@cs.drexel.edu

Abstract

The ontology of the Process Specification Language (PSL) has been desgined to address several challenges posed by planning problems – the integration of planning systems with other software systems, the formalization of the intuitions underlying existing planning systems, and the need for a rich language to represent planning domains. This paper provides an overview of the theories within the PSL Ontology and illustrate how it can be used as a framework for the analysis of planning systems and the specification of planning problems in complex domains.

Introduction

Representing plans, activities, and the constraints on their occurrences is an integral aspect of commonsense reasoning. To facilitate the sharability and reusability of plans, such a representation is often based on an ontology, which explicitly specifies the intended meanings of the terms being used, either by the planning algorithm or in the representation of the planning problem itself. This approach to reasoning about plans poses three challenges in the development of ontologies for planning systems and domains.

The first challenge is the need to integrate planning systems with other software applications. However, interoperability is hindered because the applications use different terminology and representations of the domain. These problems arise most acutely for systems that must manage the heterogeneity inherent in various domains and integrate models of different domains into coherent frameworks. Even when applications use the same terminology, they often associate different semantics with the terms. This clash over the meaning of the terms prevents the seamless exchange of information among the applications. Any ontologies for planning must be capable of explicitly specifying the terminology of the applications in an unambiguous fashion.

The second challenge is the need to formalize the intuitions that underly existing planning systems. Any such formalization would allow the designer to explicitly reason about the capabilities and limitations of different representations and techniques. An axiomatic theory would provide the foundations for reasoning about plans that can define the capabilities of planning systems and explain why certain techniques fail when extended to other domains. Many systems have invisible, buried assumptions about their domain, not explicitly documented in publications, which must be rendered explicit if we are to identify principles on which planning systems are based.

The third challenge is the role played by knowledge. System designers need to specify what knowledge is used in planning algorithms, and how knowledge of the world influences the generation and analysis of plans.

The Process Specification Language (PSL) ((Grüninger 2003), (Grüninger & Menzel 2003)) has been designed address these challenges. Its primary function is as a neutral interchange ontology ((Ciocoiu, Grüninger, & Nau 2001)) to facilitate correct and complete exchange of process information among manufacturing systems such as scheduling, process modeling, process planning, production planning, simulation, project management, workflow, and business process reengineering software. More recently, there has been interest in using PSL directly, as a rich and expressive representation language and as a logical framework for specifying the semantics of terminology in informal ontologies.

This paper will provide an overview of the theories within the PSL Ontology, as well as a survey of the concepts that are axiomatized in these theories. It will also illustrate how the PSL Ontology can be used as a framework for the analysis of planning systems and for extensions to the specification of planning problems. The paper concludes with a short example of an application of PSL to communication services and planning on mobile devices.

Process Specification Language

PSL consists of a core ontology which outlines basic objects that exist in the domain, a partially ordered set of extensions that axiomatize additional primitive process concepts, and a multitude of definitional extensions that provide a rich terminology for describing process knowledge. This section will survey both the core theories and definitional extensions within PSL, illustrating the breadth of concepts that are definable within the ontology.

The PSL ontology is a set of theories in the language of first order logic. Theories that introduce new primitive concepts are referred to as core theories, while theories containing only conservative definitions are referred to as definitional extensions[1].

All core theories within the ontology are consistent extensions of PSL-Core, (T_{psl_core}), which axiomatizes a set of intuitive semantic primitives that is adequate for describing the fundamental concepts of manufacturing processes. Specifically, PSL-Core introduces four disjoint classes: activities, activity occurrences, timepoints, and objects. Activities may have zero or more occurrences, activity occurrences begin and end at timepoints, and timepoints constitute a linearly ordered set with endpoints at infinity. Objects are simply those elements that are not activities, occurrences, or timepoints.

PSL-Core alone is not strong enough to provide definitions of the many auxiliary notions that become necessary to describe all intuitions about manufacturing processes. To supplement the concepts of PSL-Core, the ontology includes a set of extensions that introduce new terminology. Extensions to PSL-Core defining the core theories include axiomatizations of occurrence trees, discrete states, subactivities, atomic activities, and complex activities. An overview of these theories can be found in the appendix.

Interoperability and Planning Systems

The development of ontologies has been proposed as a key technology to support semantic integration—two software systems can be semantically integrated through a shared understanding of the terminology in their respective ontologies.

A semantics-preserving exchange of information between two software applications requires mappings between logically equivalent concepts in the ontology of each application. The challenge of semantic integration is therefore equivalent to the problem of generating such mappings, determining that they are correct, and providing a vehicle for executing the mappings, thus translating terms from one ontology into another.

Generating Semantic Mappings

Translation definitions specify the semantic mappings between the interlingua ontology and application ontologies. Within applications of PSL, translation definitions have a special syntactic form—they are biconditionals in which the antecedent is a class in the application ontology and the consequent is a formula that uses only the lexicon of the interlingua ontology. For example, the concept of *AtomicProcess* in the OWL-S Ontology [2] (McIlraith, Son, & Zeng 2001) has the following translation definition with respect to the PSL Ontology (Grüninger & Kopena 2005):

$$(\forall a)\ AtomicProcess(a) \equiv$$
$$primitive(a) \land markov\_precond(a) \land$$
$$(markov\_effects(a) \lor context\_free(a))$$

Translation definitions can be semiautomatically generated by using the organization of the PSL Ontology. Many ontologies are specified as taxonomies or class hierarchies, yet few ever provide any justification for the classification. When classifying the models of axiomatic theories of mathematical structures, logicians use properties of models, known as invariants, that are preserved by isomorphism. Classes are defined to consist of models with the same value of the invariant. For some classes of structures, such as vector spaces, invariants can be used to classify the structures up to isomorphism; for example, vector spaces can be classified up to isomorphism by their dimension. For other classes of structures, such as graphs, it is not possible to formulate a complete set of invariants. However, even without a complete set, invariants can still be used to provide a classification of the models of a theory.

Following this methodology, the set of models for the core theories of PSL are partitioned into equivalence classes defined with respect to the set of invariants of the models. Each equivalence class in the classification of PSL models is axiomatized using a definitional extension of PSL. In particular, each definitional extension in the PSL Ontology is associated with a unique invariant; the different classes of activities or objects that are defined in an extension correspond to different properties of the invariant. In this way, the terminology of the PSL Ontology arises from the classification of the models of the core theories with respect to sets of invariants.

Each definitional extension in the PSL Ontology corresponds to a different invariant used in the classification of the models of the ontology. Every class of activity, activity occurrence, or fluent in an extension corresponds to a different value for the invariant. The consequent of a translation definition is equivalent to the list of invariant values for members of the application ontology class.

[1]The complete set of axioms for the PSL Ontology (written in the Knowledge Interchange Format) can be found at `http://www.mel.nist.gov/psl/psl-ontology/`. Core theories are indicated by a .th suffix and definitional extensions by a .def suffix. As of March 2005, Parts 1, 11, and 12 of PSL have been accepted as an International Standard through project ISO 18629 within the International Organisation of Standardisation.

[2]OWL-S is an OWL (Ontology Web Language) ontology for describing Web services, created by a coalition of researchers through the support of the DARPA Agent Markup Language (DAML) program. OWL-S supplies Web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in unambiguous, computer-interpretable form.

For example, the above translation definition from OWL-S, is based on intuitions about preconditions and effects. Within the PSL Ontology, these arise from the classification of occurrence trees, which characterize all sequences of activity occurrences. Not all of these sequences will intuitively be physically possible within the domain; consequently, we need to characterize the subtree of an occurrence tree that consists only of possible sequences of activity occurrences; such a subtree is referred to as a legal occurrence tree, and elements of this subtree are referred to as *legal activity occurrences.*

Preconditions specify the constraints under which activities can legally occur in some domain. The most prevalent class of occurrence constraints is that of markovian activities, whose preconditions depend only on the state prior to their occurrences (e.g., to withdraw money from a bank account, there must be sufficient funds in the account). However, PSL also contains the definitions of classes of activities that do not have markovian preconditions. In such cases, the legal occurrences may depend on the time (e.g. the activity can only occur during office hours), or they may depend on the occurrences of other activities.

Effects characterize the constraints on how activity occurrences change fluents. Analogous to the notion of preconditions, the most prevalent class of activities is that of markovian activities, activities whose effects depend only on the state prior to their occurrences (e.g., the balance in account after a withdrawal depends only on the amount withdrawn and the account balance prior to the withdrawal).

Using Semantic Mappings

The set of translation definitions for all concepts in a software application's ontology defines a *semantic integration profile* for that application.

For example, suppose Alice's ontology contains a class of activities $C_1^{alice}(a)$ which has unconstrained preconditions (i.e., they are always possible) and whose effects are either context-free or they depend only on the state prior to occurrences of the activities. Suppose that Bob's ontology contains a class of activities $C_1^{bob}(a)$ whose preconditions are either unconstrained or markovian and whose effects are context-free. Using the invariants for the PSL Ontology, the following translation definitions can be generated:

$$(\forall a)\, C_1^{alice}(a) \equiv$$
$$unconstrained(a)$$
$$\wedge(markov\_effects(a) \vee context\_free(a)).$$
$$(\forall a)\, C_1^{bob}(a) \equiv$$
$$(unconstrained(a) \vee markov\_precond(a))$$
$$\wedge context\_free(a).$$

Translation between integration targets may be accomplished by applying deduction to the axioms of the interlingua, the semantic mappings, and the input to be translated. Given the above example mappings from the two application ontologies of Alice and Bob into PSL, the following mappings between the two concepts may be inferred:

$$T_{psl} \models (\forall a)\, context\_free(a) \supset (C_1^{alice}(a) \supset C_1^{bob}(a)).$$
$$T_{psl} \models (\forall a)\, unconstrained(a) \supset (C_1^{bob}(a) \supset C_1^{alice}(a)).$$

The antecedents of these sentences can be considered to be guard conditions that determine which activities can be shared between the two ontologies. This can either be used to support direct exchange, or simply as a comparison between the application ontologies. In this example, the *alice* can export any *unconstrained* activity description to *bob* and *bob* can export any $context\_free$ activity description to *alice*; however, *alice* cannot import $markov\_precond$ activity descriptions from *bob* and *bob* cannot import any $markov\_effects$ activity descriptions from *alice*.

The objective of this approach is the integration of planners with other software, such as scheduling and simulation engines. Translation definitions have been specified for the composite process control constructs of OWL-S (Grüninger & Kopena 2005). Based on a set of translation definitions, syntactic translators have been implemented that can export IDEF-3 process models and import them into Ilog Schedule (Ciocoiu, Grüninger, & Nau 2001). Future work will include the specification of translation definitions for planning formalisms such as PDDL (Ghallab 1998) and O-Plan (Tate, Drabble, & Kirby 1994), thus providing a first-order axiomatization of the intended semantics of their constructs.

PSL as a Semantic Foundation

Recent work has extended the translation definition approach (in which PSL serves as a mediating ontology between source and target ontologies) to the problem of providing a rigorous model-theoretic foundation for informal ontologies. One of the most promising efforts in this direction is in applications of ontologies to the Semantic Web.

The Semantic Web Services Language (SWSL) is intended to support richer semantic specifications of Web services, based on a comprehensive representational framework that spans the full range of service-related concepts. Such a framework will enable fuller, more flexible automation of service provision and use, support the construction of more powerful tools and methodologies, and promote the use of semantically well-founded reasoning about services. For example, richer semantics can support greater automation of service selection and invocation; automated translation of message content between heterogeneous interoperating services; automated or semi-automated approaches to service composition; more comprehensive approaches to service monitoring and recovery from failure; and fuller automation of verification, simulation, configuration, supply chain management, contracting, and negotiation of services.

SWSL includes an axiomatized ontology of service concepts (known as FLOWS, First-order Logic Ontology for Web Services), which provides the conceptual framework for describing and reasoning about services. FLOWS is built upon OWL-S and PSL as primary starting points. From the perspective of OWL-S, FLOWS represents a more comprehensive ontology, expressed using a more expressive language, and builds on more mature conceptual models. From the perspective of PSL, FLOWS may be viewed as a collection of extensions that situate the description of processes within a larger context of message-based communications across networks. Each concept in FLOWS has a first-order definition using PSL terminology; in this sense, the semantics of the process ontology within FLOWS is explicitly axiomatized using PSL.

Finally, PSL may also be used as the semantic foundation for past efforts to formalize intuitions about plans in projects such as the Shared Plan and Activity Representation (SPAR) (Tate 1998).

New Planning Problems in PSL

Typically, a deductive planning problem is specified as follows:

- a set of atomic activities
- precondition and effect axioms
- initial state
- goal state

A plan is then a sequence of atomic activities whose effects achieve the goal.

In this section, we will show how this approach can be generalized by incorporating the classes of activities and constraints from the PSL Ontology.

Planning with complex activities Rather than restrict the set of activities in a plan to be atomic, we can include complex activities. The specification of the planning problem must then incorporate the process descriptions for the complex activities, which specify the subactivities and the constraints on the occurrences of the subactivities. The plan will itself be a complex activity with the property that the effects of the atomic subactivities on each branch achieve the goal state (recall that each branch of an activity is a sequence of atomic activity occurrences). The notions of precondition and effect axioms can also be generalized to complex activities. For example, some fluent is an effect of a complex activity if it is achieved by some atomic subactivity, and not falsified by a later atomic subactivity.

More generally, the specification of the planning problem can incorporate the distinction between preconditions, conditional activities and triggered activities. In the first case, the fluents determine if an activity can possibly occur, although the activity does not necessarily occur. In the second case, the fluents determine which subactivities occur within an occurrence of the complex activity. However, with triggered activities we want to capture the intuition that an activity must occur if the fluents hold, regardless of whatever other activities are occurring (e.g. if the alarm is ringing, evacuate the building). Other classes of activities can also be defined with respect to different kinds of constraints; for example, there may be activities that are launched at specific times (e.g. send status reports every hour) and which are not subactivities of any other complex activity.

Planning with additional constraints In addition to achieving the goal state, we can generalize the problem so that plans must also satisfy additional constraints. An example of this is planning in the presence of external activity occurrences; such plans may either depend on the occurrence of external activities or prevent the occurrence of external activities that interfere with the plan. An example of a necessary external activity is the arrival of FedEx for package pickup, whereas an example of a forbidden external activity is the occurrence of an accident that prevents the FedEx truck from arriving. Within the PSL Ontology, these intuitions are captured by the property that not all occurrences of subactivities need be subactivity occurrences that are elements of an activity tree; complex activities can therefore be classified by the constraints on the subactivity occurrences.

There may also be temporal constraints on subactivity occurrences, so that the complex activity could not occur on branches of the occurrence tree that violated such constraints. For example, problems in logistics planning often require reasoning about the spoilage of perishable products – given products with a given shelf life, find a production and delivery plan that prevents spoilage from occurring. Similarly, a complex activity may be defined by requiring that state constraints be satisfied after all occurrences; branches of the occurrence tree in which external activities interfered with the effects of subactivities could not be branches of the activity tree of such an activity.

Complex activity synthesis Finally, we can generalize the specification of the planning problem so that we generate a complex activity in some class, rather than a plan that is only a sequence of activities. In practice, this arises most prominently in the context of automated Web service composition; recent work on the development of SWSL has considered reasoning problems for web service specifications such as determining the consistency of a composite web service and determining whether a set of atomic services are composable.

Restricted versions of this problem can also be considered. For example, given a partial specification of a complex activity, generate a complex activity that is a consistent extension. In this case, a partial specification is a process description whose models include multiple activity trees, each of which contain multiple branches.

Each activity tree of the synthesized complex activity is a subtree of some activity tree of the initial complex activity.

Reasoning about Plans

The above generalizations of the planning problem lead to an approach in which plans are complex activities. Since complex activities are objects within the PSL Ontology, we can explicitly reason about plans within the language of the ontology, and therefore explicitly define classes of plans.

The PSL Ontology can be used as a formal framework for the characterization of planning algorithms. As shown above, PSL can be used to identify which classes of constraints are included in the specification of the planning algorithm, as well as to characterize the properties of plans that are generated by the planning algorithm.

This supports the comparison of different planning algorithms through the explicit characterization of which algorithms share the same assumptions about constraints and plans. For example, HTN methods may be characterized by process descriptions that axiomatize the relationship between complex activities and their subactivities; the methods used by different algorithms can then be compared by reasoning over the process descriptions and their corresponding classes of activities.

Case Study: Communication Services and Planning on Mobile Devices

Mobile devices such as laptops, PDAs, smart phones, and other small, network-capable computing platforms are an increasingly pervasive element of society. Ubiquitous and mobile computing environments as enabled by these devices are well suited to leverage service-based computing:

- Software in ubiquitous computing settings will have to interact with a wide array of services and applications. Just a few applications include calendar agents for arranging meetings, web and email gateways, and kiosks for retrieving menus, historical data, or other information. These tasks will require discovering and utilizing available services developed by any number of organizations, run on different platforms, and following a variety of protocols.
- Mobile computing features a large and growing body of hardware platforms, accessories, and operating systems. Current software is generally limited to the specific product families and systems for which it is developed. To increase the cost-efficiency and usefulness of mobile software development, it must be able to operate on the variety of platforms on which it may be installed and adapt to the capabilities of the hardware present on the device.
- Networking in mobile computing possesses a number of properties not found in traditional, wired communications. These include substantial latency, low bandwidth, unreliable links, and poor connectivity. Even with optimal network layer support, these environments necessitate new approaches in creating effective applications. Software and agents that can discover and utilize application and infrastructure services are uniquely suited to address these challenges (Kopena *et al.* 2005).

Example: Calendar Agent. Consider a calendar on a PDA that synchronizes with a base computer as events are posted, which in turn relays appropriate updates to co-workers, family, and friends. Cellular and wireless ethernet capabilities exist on the PDA. Wireless ethernet can only connect to the base computer when in range of an access point. The cellular module can always contact the base computer via a satellite Internet connection. However, it consumes substantial amounts of power and charges a fee.

To contact the base computer in all situations, the calendar must discover and utilize both interfaces. However, use of the cellular interface should be minimized: it should not be used if wireless is available or for minor or far-off updates. Instead, updates should be collected until several may be transmitted or ethernet can be used. However, if an imminent change is posted, the cellular connection should be used so that the base computer may be notified and begin synchronizing calendars of affected friends, family, or co-workers. Further, the calendar must determine and execute the process for communicating with the synchronization program, such as an authentication step followed by an update.

It is easy to see that this scenario requires the correct exchange of service descriptions, and the subsequent planning and scheduling on those descriptions to achieve the goals and satisfy other domain constraints. Furthermore, since the network may include many planners of varying abilities, we need to characterize their capabilities with respect to the expressiveness of the constraints that they can represent.

Formalization

The structure of the different theories involved in this is shown in Figure 1. System-wide components allow agents to exchange descriptions, while agent-specific theories map that into the agent's behaviors and reasoning.

System-wide Theories. Although an ontology such as OWL-S offers a more expressive language and formal semantics in comparison to workflow and Web Services languages such as BPEL4WS and WSDL, the underlying theory of OWL-S is unclear and there are ambiguities present in the language because of its reliance on natural language definitions. These deficiencies limit the role of OWL-S as the foundation for the system-wide theories. With PSL, on the other hand, we have a

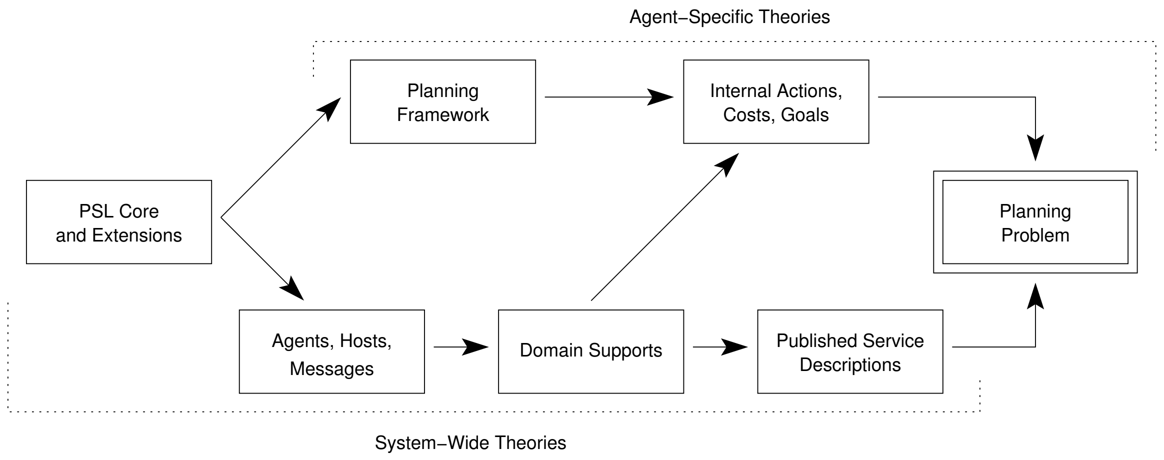


Figure 1: Relationships between theories involved in sharing descriptions and utilizing services. System-wide theories facillitate knowledge exchange. Agent-specific theories map that knowledge into an agent's behaviors and reasoning mechanisms.

rigorously defined language which is strong enough to define a wide variety of services, and which is able to explicitly characterize the range of planning capabilities within realistic software environments.

A basic theory of agents, hosts, and messages that extends PSL provides a basis for the terminology about the entities in this domain. also incorporated on top of PSL. This provides a shared ontology by which an agent may communicate about, recognize, and utilize communication resources. A first order theory of belief based on k-accessibility between occurrences is also included in the theory. It defines *KD* (see e.g. (Fagin *et al.* 1995)), the logic of consistent belief, within the fluent space of PSL. This enables the expression of agent interfaces in the form of doxastic activity effects, as well as an agent's communication goals. Belief, and specifically *KD*, was chosen due to its relatively weak commitments on agent reasoning abilities and trust.

Domain supports provide shared ontologies with which agents can talk about preconditions and effects—constraints that are fairly general and well known, such as power.

Agent-Specific Theories. The planning framework provides a characterization and mapping into a planner or other reasoner, e.g. it might characterize its reasoning as a classical plan, HTN, conformant planning, or even map into a probabilistic method such as an MDP.

The agent's internal actions, costs, goals, etc. specify its behaviors, preferences, and possible courses of action. Together, all of these things define a planning and scheduling problem which may be fed into some method or tool to solve and then given to the agent to execute. This may happen as part of a registry or be part of the agent itself (on mobile devices, it's probably part of the registry, which is part fo why it's important to characterize its abilities, so an agent knows the power of the host its on).

Calendar Agent Axiomatized

Using all this, agents and communication resources may then describe their interfaces and abilities, e.g. the base computer in the Calendar Agent scenario.

$$\begin{gathered}\forall o, a, s, m \cdot \texttt{occurrence\_of}(o, a) \wedge \texttt{legal}(o) \\ \wedge\, \texttt{message-activity}(a, s, \texttt{SYNC}, m) \supset \\ [\forall c \cdot \texttt{holds}(\texttt{contents}(m, c), o) \supset \\ \texttt{holds}(\texttt{believes}(\texttt{SYNC}, c), o)]\,.\end{gathered}$$

The cellular and wireless interfaces may be described as:

$$\begin{aligned}&\forall o, a, s, d, m \cdot \texttt{occurrence\_of}(o, a) \wedge \texttt{legal}(o) \wedge \\ &\quad \texttt{sat-msg}(a, s, d, m) \supset \texttt{message-activity}(a, s, d, m) \wedge \\ &\quad\; \texttt{occ-util-summ}(o, -0.5). \\ &\forall o, a, s, d, m \cdot \texttt{occurrence\_of}(o, a) \wedge \texttt{legal}(o) \wedge \\ &\quad \texttt{radio-msg}(a, s, d, m) \supset \texttt{prior}(\texttt{in-range}(s, d), o) \wedge \\ &\quad\; \texttt{message-activity}(a, s, d, m) \wedge \texttt{occ-util-summ}(o, -0.1).\end{aligned}$$

The agent's internal communication activities would be:

$$\begin{aligned}&\texttt{activity}(\texttt{defer}). \\ &\forall o, m, d, n \cdot \texttt{occurrence\_of}(o, \texttt{attach}(m, \texttt{update}(d, n))) \wedge \\ &\quad \texttt{legal}(o) \supset \wedge \texttt{holds}(\texttt{contents}(m, \texttt{update}(d, n)), o).\end{aligned}$$

The agent's behavior may then be captured as the time-dependent utility of delivering updates to the base computer:

$$\begin{gathered}\forall o, c, d, n \cdot \texttt{holds}(\texttt{current-day}(c), o) \wedge \\ \texttt{holds}(\texttt{believes}(\texttt{SYNC}, \texttt{update}(d, n)), o) \supset \\ \texttt{occ-util-summ}(o, 1 - ((d - c)/7)).\end{gathered}$$

With the addition of an episode delimiter defined as occurrences of `defer` or `message-activity`, supporting axioms not listed above such as the definition of `update`, and closure axioms, a plan based on these definitions will:

- Deliver any update if a wireless connection exists.
- Use the satellite interface if warranted by an update or some collection of updates. For example, any update concerning the next 3.5 days will be posted immediately.
- Otherwise defer until new updates are posted or conditions change, e.g. wireless contact is made.

A simple agent may use the PSL Core, Occurrence Trees, and Discrete States theories. A deductive planning problem $(A, I, \psi, C) \rightarrow \rho$ is then defined on top of that as follows[3]:

- A are the *activity axioms*, preconditions and effects.
- I defines *initial fluent state*: $\forall \delta_i \cdot \texttt{initial}(\delta_i) \supset \delta_i \models I$.
- A *chain* δ is a sequence of successive legal occurrences such that $\texttt{initial}(\delta_1) \wedge \delta_n \models$ the episode delimiter ψ.
- C defines *utility summands* μ over occurrences, related by $\texttt{occ-util-summ}(\delta_i, \mu)$.
- *Occurrence utility*, $\texttt{occ-util}(\delta_i)$, is defined as $\sum \mu$ over $\{\mu | \texttt{occ-util-summ}(\delta_i, \mu)\}$.
- *Utility of a chain*, $\texttt{utility}(\delta)$, equals $\sum \texttt{occ-util}(\delta_i)$.
- A *rational chain* δ is a chain such that there does not exist a δ' such that $\texttt{utility}(\delta') > \texttt{utility}(\delta)$.
- A *rational plan* ρ is then the sequence of activities associated with the occurrences in a rational chain.

Summary

Within the increasingly complex environments of enterprise integration, electronic commerce, and the Semantic Web, where process models are maintained in different software applications, standards for the exchange of this information must address not only the syntax but also the semantics of planning concepts. These environments are also distinguished by the expressiveness required to adequately capture the semantics of the concepts. The Process Specification Language draws upon well-known techniques from mathematical logic to provide a robust semantic foundation for the representation of plan information. In this way, PSL can serve as a framework for the analysis of existing planning systems, as well as providing the basis for extensions of the classical planning problem in future planning systems.

References

Bock, C., and Gruninger, M. 2005. Psl: A semantic domain for flow models. *Software and Systems Modeling* to appear.

Ciocoiu, M.; Grüninger, M.; and Nau, D. 2001. Ontologies for integrating engineering applications. *Journal of Computing and Information Science in Engineering* 1:45–60.

Fagin, R.; Halpern, J. Y.; Moses, Y.; and Vardi, M. Y. 1995. *Reasoning About Knowledge.* Cambridge, Mass.: MIT Press.

Ghallab, M. e. a. 1998. *PDDL: The Planning Domain Definition Language v.2. Technical Report CVC TR-98-003.* Yale Center for Computational Vision and Control.

Grüninger, M., and Kopena, J. 2005. Semantic integration through invariants. *AI Magazine* to appear.

Grüninger, M., and Menzel, C. 2003. The Process Specification Language (PSL) theory and applications. *AAAI AI Magazine* 24(3):63–74.

Grüninger, M. 2003. Ontology of the process specification language. In Staab, S., and Studer, R., eds., *Handbook of Ontologies and Information Systems*, 599–618. Springer-Verlag.

Kopena, J. B.; Cicirello, V. A.; Peysakhov, M.; Malfettone, K.; Mroczkowski, A.; Naik, G.; Sultanik, E.; Kam, M.; and Regli, W. C. 2005. Network awareness and the Philadelphia-Area Urban Wireless Network Testbed. In *AAAI Spring Symposia on AI for Homeland Security (to appear).*

McIlraith, S.; Son, T.; and Zeng, H. 2001. Semantic web services. *IEEE Intelligent Systems, Special Issue on the Semantic Web* 16:46–53.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-plan: An open architecture for command, planning, and control. In Fox, M., and Zweben, M., eds., *Intelligent Scheduling.* Morgan Kaufmann.

Tate, A. 1998. Roots of spar - shared planning and activity representation. *Knowledge Engineering Review* 13(1):121–128.

Appendix: Overview of PSL Theories

PSL-Core alone is not strong enough to provide definitions of the many auxiliary notions that become necessary to describe all intuitions about manufacturing processes. To supplement the concepts of PSL-Core, the ontology includes a set of extensions that introduce new terminology. Extensions to PSL-Core defining the core theories include axiomatizations of occurrence trees, discrete states, subactivities, atomic activities, and complex activities.

Occurrence Trees The occurrence trees that are axiomatized in the core theory $T_{occtree}$ are partially ordered sets of activity occurrences—for a given set of activities, all discrete sequences of their occurrences are branches of a tree. An occurrence tree contains all occurrences of *all* activities, not simply the set of occurrences of a particular (possibly complex) activity. As

[3]Axioms and grammars are omitted for brevity but are avail. at `http://edge.cs.drexel.edu/services/`; fundamental and closure/frame axioms are omitted from the problem for simplicity.

each tree is discrete, every activity occurrence in the tree has a unique successor occurrence of each activity.

There are constraints on which activities can possibly occur in some domain. This intuition is the cornerstone for characterizing the semantics of classes of activities and process descriptions. Although occurrence trees characterize all sequences of activity occurrences, not all of these sequences will intuitively be physically possible within the domain. We will therefore want to consider the subtrees of the occurrence trees that consist only of *possible* sequences of activity occurrences; such a subtree is referred to as a legal occurrence tree.

Discrete States The core theory T_{disc_state} introduces the notion of fluents (state). Fluents are changed only by the occurrence of activities, and fluents do not change during the occurrence of primitive activities. In addition, activities have preconditions (fluents that must hold before an occurrence) and effects (fluents that always hold after an occurrence).

Subactivities The PSL Ontology uses the *subactivity* relation to capture the basic intuitions for the composition of activities. This relation is a discrete partial ordering in which primitive activities are the minimal elements.

Atomic Activities The core theory T_{atomic} axiomatizes intuitions about the concurrent aggregation of primitive activities. This is represented by the occurrence of concurrent activities, rather than concurrent activity occurrences.

Complex Activities The core theory $T_{complex}$ characterizes the relationship between the occurrence of a complex activity and occurrences of its subactivities. Occurrences of complex activities correspond to sets of occurrences of subactivities; in particular, they form subtrees of the occurrence tree. An activity tree consists of all possible sequences of atomic subactivity occurrences beginning from a root subactivity occurrence. In a sense, activity trees are a microcosm of an occurrence tree, in which we consider all of the ways in which the world unfolds *in the context of an occurrence of the complex activity.*

Each activity tree is composed of a set of isomorphic copies of a unique minimal activity tree consisting only of subactivity occurrences. Not every occurrence of a subactivity is a subactivity occurrence; there may be other external activities that occur during an occurrence of an activity, or subactivity occurrences may need to satisfy temporal constraints. Within models of $T_{complex}$, these constraints on subactivity occurrences are captured by different ways of embedding the minimal activity tree into the activity tree.

Subactivity Occurrence Ordering This extension axiomatizes different partial orderings over subactivity occurrences within a complex activity. In general, ordered activities have the property that any branch of the activity tree of an ordered activity satisfies the ordering constraints, although there may be sequences of subactivity occurrences that satisfy the ordering constraints yet which do not correspond to branches of the activity tree. The Subactivity Occurrence Ordering theory captures a stronger notion of ordering that is typically captured by flow diagrams ((Bock & Gruninger 2005)) and workflow control constructs ((McIlraith, Son, & Zeng 2001)). For example, the subactivity occurrences of a Split activity in OWL-S are partially ordered, such that every linear extension of the ordering corresponds to a branch of the activity tree. Similarly, the subactivity occurrences of a Unordered activity in OWL-S are partially ordered, such that all subactivities are incomparable and every linear extension of the ordering corresponds to a branch of the activity tree.

Duration The timeline axiomatized within PSL-Core is simply an infinite linear ordering. The Duration core theory augments PSL-Core with a metric over the timeline, and the ability to identify the timepoint that is some duration after some other timepoint. Since each activity occurrence has a unique beginof timepoint and endof timepoint, the notion of duration can be extended to activity occurrences. Furthermore, it become possible to specify new classes of duration-based constraints, such as duration-dependent effects and notions such as spoilage (in which it is not possible for an activity occur within some duration of another activity occurrence).

Resource Requirements This core theory specifies the conditions that must be satisfied by any object that is a resource for an activity. The axiomatization of resources centers on the notion of interacting activities. Two activities are interacting if there exist properties of activities that are no preserved by their composition into a more complex activity. For example, the effects of two activities that occur in isolation may not be preserved if they are performed concurrently. Two activities may possibly occur separately, but the complex activity that contains both as subactivities may not possibly occur because the effects of one falsify the preconditions of the other. The axioms of the Resource Requirement core theory state that for any set of interacting activities there exists a resource that is shared by the activities. This supports a higher level of abstraction for reasoning about interactions between subactivities.

A Portable Process Language

Peter E. Clark[1], David Morley[2], Vinay K. Chaudhri[2], Karen L. Myers[2]

[1]M&CT, Boeing Phantom Works, PO Box 3707, Seattle, WA 98124
[2]Artificial Intelligence Center, SRI International, Menlo Park, CA 94025
peter.e.clark@boeing.com, {morley,chaudhri,myers}@ai.sri.com

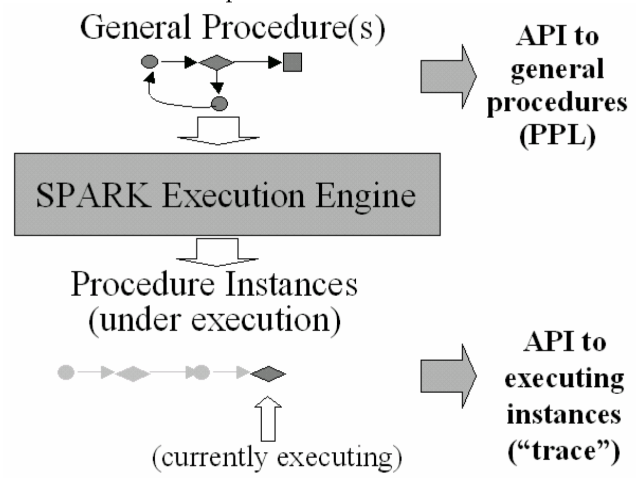
Abstract

Process representation languages designed to support execution have evolved to support specialized reasoning capabilities like action selection and task decomposition, but do not readily support inferences that one might need for explanation or question answering. In this paper, we report on a process language, PPL, that we have designed to serve as a bridge between a representation designed for execution and a representation designed for applications such as question answering and explanation generation. Through its use of a propositional-style representation of process structure, PPL can enable the use of generalized reasoning methods for those purposes. PPL is novel in that it directly encodes the process "flow chart" in a neutral, KIF-like syntax, allowing other modules to introspect on the process structure.

Introduction

SPARK (SRI Procedural Agent Realization Toolkit) [Morley and Myers 04] is an agent framework that builds on a Belief-Desire-Intention (BDI) model of rationality. SPARK provides a flexible plan execution mechanism that interleaves goal-directed activity and reactivity to changes in its execution environment. SPARK's procedural language has a clear, well-defined formal semantics that can support reasoning techniques for procedure validation, synthesis, and repair. The SPARK representation language (called SPARK-L) is an extended form of Hierarchical Task Network (HTN) representation [Erol et al. 04]. Extensions beyond standard HTN include iteration, conditional branching, and certain runtime-specialized constructs (WAIT, TRY).

SPARK includes a comprehensive API for monitoring executing procedures—called *procedure instances*—allowing external modules to view the "trace" of a procedure's execution. For example, by using this API an external module can find the specific task(s) currently being executed, and trace back to find specific previously executed tasks and their details. However, while allowing access to how a general procedure is playing out, this API does not allow access to the general "flow chart" of the procedure itself, for example, to find possible future tasks, choice points, cycles, and alternative ways the procedure may play out. However, many tasks require that a system be able to introspect on its own general procedures, in particular language understanding, question answering, and explanation. This has been particularly true in the context of CALO, an integrated cognitive assistant.[1] As a result, we have enhanced SPARK with a second API allowing it to export (an abstraction of) its general procedures also. These exported procedures are expressed in logic in Portable Process Language (PPL), a language we have designed for this purpose, complementing SPARK-L, the native process language of SPARK. The goals of PPL are to make explicit the steps, their arguments, and their relationships in the procedures, and to abstract away some of the details critical for execution but not needed for introspection.


General Procedure(s)
API to general procedures (PPL)
SPARK Execution Engine
Procedure Instances (under execution)
API to executing instances ("trace")
(currently executing)

As a close analogy, consider a system wanting to introspect on a piece of software, but only having an API to the trace of that software executing. The system could, of course, literally try to parse the source code representation of the software and try to work out what it did (in fact, some software analysis tools do exactly this), but this requires the system to have a complete internal model of the programming language, both syntax and semantics. Instead, it would be nice if the software could export a high-level summary of its behavior—literally, a flow-chart-like data structure—that other systems could

[1] See http://www.ai.sri.com/project/CALO

then manipulate, expressed in some relatively implementation-independent language. Our goal is that PPL be such a language, in the specific context of SPARK-like planning systems.

In this paper, we describe a simplified version of the SPARK language and how it is executed, the reasoning requirements, PPL, and experiences in using the language in the context of an agent system. We conclude with directions for future work.

Description of SPARK Process Language

The SPARK process language provides a hierarchical representation of processes. A library of *procedures* provides declarative representations of activities for responding to events and for decomposing complex tasks into simpler tasks. Each procedure has a *precondition* stating conditions under which it can be applied, and a *task network expression* describing how to respond to an event or to decompose a non-primitive action. The hierarchical decomposition of tasks bottoms out in primitive actions that bring about some change in the outside world or the internal state of the agent. A SPARK agent specification includes declarations of predicate and task symbols, facts about the initial state, and a library of procedures. The key syntactic structures include term expressions, logical expressions, actions, task expressions, and procedures.

A *term expression* represents a value. Atomic term expressions are constants such as `42` and `"Hi"`, and variables of the form `$x`. These are combined to form compound term expressions, including lists such as `[1 2 3]`.

Logical expressions are constructed from predicate (fluent) symbols applied to term expressions, e.g., `(= 1 $x)`, `(True)`, logical operators, and quantifiers. These expressions can be used to alter the flow of execution or to bind variables for use later in the procedure.

An *action* is constructed from an action symbol and term expressions as parameters, for example, `(laptop_query $criteria $quotes)`. The action may be primitive, which is performed by executing some procedural attachment, or nonprimitive, which is hierarchically expanded into subtasks using procedures. The parameters of an action may include free variables that are not bound at the time the action is attempted. These variables are bound by the successful execution of the action and provide a means of returning values from executing the action.

A *task network expression* is a pattern of activity that when attempted may either succeed or fail. Task network expressions include such constructs as

`[set:` *V T*`]` - Set variable *V* to the value *T*.
`[do:` *A* `]` – Perform action *A*.
`[seq:` *N1 N2 …* `]` – Attempt task networks *N1*, *N2*, … sequentially.
`[parallel:` *N1 N2 …* `]` – Attempt task networks *N1*, *N2*, … in parallel.
`[select:` *P1 N1 P2 N2 …* `]` – Execute the task expression *Ni* corresponding to the first logical expression *Pi* that has a solution. Fail if none has a solution.
`[wait:` *P1 N1 P2 N2 …* `]` – As select, but wait instead of failing.
`[while:` *P N*`]` – While *P* has a solution execute *N*.

At its simplest, a SPARK *procedure* has the form `{defprocedure` *name* `cue:` *trigger* `precondition:` *P* `body:` *N*`}`

This indicates that if *P* is true when *trigger* occurs, then executing *N* is a valid way of responding. The cue may be of the form `[newfact:` *P*`]` to respond to the fact *P* being added to the KB, or `[do:` *A*`]` to expand the action *A*.

For example, the following procedure, `Get_Bid`, describes one way of expanding the task of performing action find_bids. It is applicable if the condition `(Online "DM4QR")` holds. It performs a `laptop_query` action that binds variable `$temp_quotes` and then depending upon whether or not this list is empty, either performs `relax_and_redo_query`, binding `$quotes`, or binds `$quotes` to `$temp_quotes`.

```
{defprocedure Get_Bid
  cue:
   [do: (find_bids $item $criteria $quotes)]
  precondition: (Online "DM4QR")
  body:
  [seq:
    [do: (laptop_query $criteria
                            $temp_quotes)]
    [select:
      (=  $temp_quotes [])
        [do: (relax_and_redo_query $criteria
                                     $quotes)]
      (True)
        [set: $quotes $temp_quotes]]]}
```

SPARK Process Execution

Figure 1 shows the architecture of each SPARK agent. Each agent is embedded in the world and interacts with the world though sensors and effectors. Each agent has its own knowledge base of beliefs and library of procedures. The initial state of the beliefs and procedure library are initially loaded from files written in the SPARK language. The beliefs are updated by the agent's sensors and through the agent executing procedures. The set of procedure instances that the agent is executing at a given time form the intentions of the agent. At any time an agent may be executing multiple intentions. At SPARK's core is the executor whose role is to manage the execution of intentions. The executor can post procedure subtasks as actions. Primitive actions cause effects through the effectors. Non-primitive actions are expanded by the executor according to the procedures the agent has available.

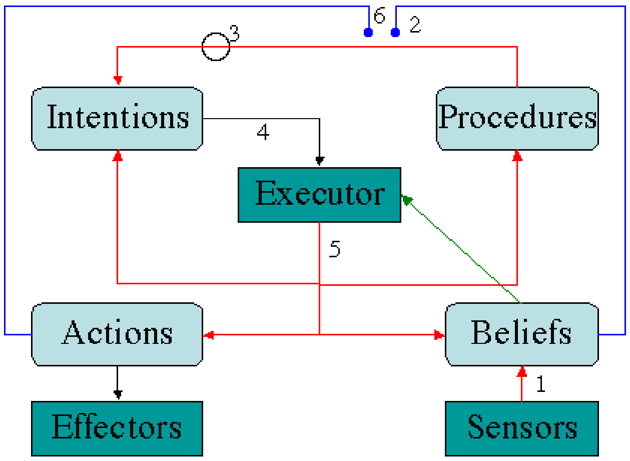


Figure 1: The Architecture of each SPARK agent.

SPARK process execution proceeds as follows: (1) Sensors update the agents beliefs. (2) These belief changes trigger the instantiation of procedures - those with a cue that matches the event and a precondition that is satisfied with respect to the agent's beliefs. (3) A subset of these applicable procedures is selected to be added to the intentions. (4) The executor selects one intended procedure instance to progress by executing a single step. (5) This may cause updates to the beliefs or the posting of new subtasks. (6,2) This in turn triggers the instantiation of procedures, and so on.

The step by step execution of each procedure instance results in the binding of values to variables in the procedure instance. Each predicate that is tested or action that is executed results in all free variables becoming bound. Usually when a logical expression (such as may appear in a context: or select: task network expression) is tested, SPARK needs to commit to a single solution without the possibility of backtracking.

The approach that SPARK and many other BDI agent systems take to the hierarchical expansion of tasks is based on an expectation that the agent is working in a highly dynamic environment: the expansion of tasks is performed *at the time the task is to be executed* and the choice of the procedure to use is based on the *current state of the agents beliefs*. BDI agent systems have been integrated with planning systems (e.g., [Myers 99, Lemai and Ingrande 04]) however SPARK currently has no planning component.

Reasoning Requirements

We consider here the reasoning requirements for three applications, and how introspection on the process model itself is necessary to support them: (1) dialog interpretation, (2) question answering, (3) explanation generation.

Reasoning Requirements for Dialog Interpretation

Robust language processing is challenging in that what the user says can be ambiguous, incomplete, and erroneous. In the context of the CALO system that we are developing, one class of dialogs (between the user and computer) that we have been studying is "purchase dialogs", where the user is directing CALO to purchase an item on the user's behalf. In these dialogs, these problems can all occur, and resolution of them requires the system to have strong expectations, hence requiring knowledge of the processes themselves.

For example, while instructing CALO to buy a computer, if the user says "Find me an appropriate machine", the user actually means "You have authorization to start retrieving quotes from vendors for the laptop I want to purchase" (as opposed to, for example, start physically searching the building to find machines). To find the correct interpretation, CALO needs to have strong expectations about what sort of activities may occur in the future, and then match the user's utterance with those expectations. In this case, CALO contains a process model of how to purchase items, and as this process model is currently active (triggered by earlier statements by the user, such as "I want to purchase a laptop"), CALO should be able to look at future steps that could be construed as "finding a machine" to identify what the user is referring to. In this case, a subsequent step in the process is to retrieve quotes for the to-be-purchased computer, and a matching algorithm can identify this as the most likely thing to which the user is referring [Yeh et al. 03]. To do this, the system needs to introspect on its general purchase plan (procedure), to identify that a future step that can be construed as "finding" will occur.

Reasoning Requirements for Question Answering

As a general-purpose assistant, CALO is expected to field answers to a wide variety of questions from a user, including about its (CALO's) own knowledge of how to do things. While SPARK's execution trace API allows CALO to answer questions about specific things it has done (e.g., "When you purchased the laptop, did you request authorization?"), CALO also needs knowledge of the procedural flow chart itself for more general questions about procedures, for example,

1. How do you purchase a laptop?

2. Will you need to access to the Web during the purchase?
3. Is authorization required [i.e., is there an authorization step] to purchase a laptop over $2000?
4. What will happen if the authorizing manager is unavailable?
5. Email the quotes you find to my home email address.
6. Get authorization from Joe, not Steve.

The first four of these questions are independent of any specific execution of the procedure, and necessarily require access to the general procedure itself to answer the questions. The last two questions are in the context of a partially executed procedure, in which the user is making reference to a to-be-executed future step. Again with these two questions, the system needs a representation of the general procedure to identify the future step to which the user is referring (these are not in the execution trace, as they have not yet happened).

Requirements for Supporting Explanation

In a related piece of work reported elsewhere, we are working on explaining processes. Answering "why" questions particularly requires introspecting not just on what happened, but the specific tests and conditions that caused those things to happen, a particular form of meta-reasoning. Again, knowledge of the structure of the process flow chart is often required to answer these questions, including details of tests directing flow of execution at branch points, and details of how earlier steps support later steps. (In its current form PPL does not capture all this knowledge, but it is a step toward this.) Example questions from the user to CALO include

1. Why didn't you ask for authorization?
2. Why did you send the purchase request to Dallas?
3. Why haven't you started searching yet?

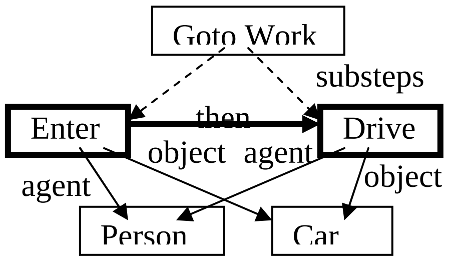
Portable Process Language (PPL)

Overview

To meet the reasoning requirements already identified, we must be able to introspect SPARK processes to obtain information as follows: What tasks are involved in task X? What task comes after task Y? Might task Z occur in process W? The goal of the Portable Process Language (PPL) is to represent just this kind of "flow chart" information. PPL captures a filtered view of the SPARK process models to include tasks, subtasks, task parameters, and task ordering. It does not currently capture the semantics of conditional tests, context, and other logical evaluations. Such extensions can be added in the future.

We now describe PPL, and then describe how it complements PSL (Process Specification Language), a language that captures the semantics but not the explicit structure of a process flow chart.

Consider a toy example of a two-step process—namely, going to work—with a trivial two-step flow chart consisting of (i) entering a car and then (ii) driving the car to work:



The PPL for a skolem instance process is follows:

```
;;; Steps in the flow chart
(instance-of goto-work1 goto-work-step)
(instance-of enter1 enter-step)
(instance-of drive1 drive-step)

(possibleTask goto-work1 enter1)
(possibleTask goto-work1 drive1)
(followedBy enter1 drive1)

;;; Parameters to those steps
(instance-of person1 person-var)
(instance-of car1 car-var)

(agent goto-work1 person1)
(object goto-work1 car1)

(agent enter1 person1)
(object enter1 car1)

(agent enter1 person1)
(object drive1 car1)
```

In PPL, each of the steps in the flow chart is denoted by a *specific individual*. Note that these individuals are *not* instances of events in the world (e.g., the specific event of entering the car at a certain time); rather, they are instances of steps in a flow chart. Similarly, each parameter (e.g., the person, the car) for a flow chart step is denoted by a *specific individual*. Again, note that these individuals are *not* instances of objects in the world (e.g., a specific person or car); rather, they are instances of parameters (variables) in the flow chart. In other words, PPL is a somewhat literal logical depiction of flow chart steps, rather than of event sequences. We can relate flow chart steps to event types (classes), and flow chart parameters to object types (classes), with some simple correspondence statements:

```
(event-type-for goto-work-step goto-work)
(event-type-for enter-step     enter)
(event-type-for drive-step     drive)
(object-type-for person-var    person)
(object-type-for car-var       car)
```

where goto-work, enter, drive are types (classes) of actual events in the world, and person, car are types (classes) of actual objects in the world.

Given a PPL flow chart, in principle one could "execute" it to perform the plan it denotes. However, the AI planning community has long matured beyond these kinds of "toy" executions—in the real world, plan execution also involves plan monitoring, recovery in the case of failure, consideration of time and resource constraints, and so on. To adequately represent such executable processes, much richer languages are needed, and this is precisely the role of SPARK. Rather, one should view PPL as capturing a simple abstraction of a complex executable process, in a language-neutral form suitable for introspection, to support the kinds of tasks discussed earlier.

For contrast, a simple SPARK representation of the above procedure would be

```
{defprocedure Goto_Work_by_Car
  cue: [do: (goto_work $person $car)]
  precondition: (True)
  body:
  [seq: [do: (enter $person $car)]
        [do: (drive $person $car)]]}
```

While suitable for execution, this syntactic structure is more difficult to manipulate for introspection. Note also that some information in this SPARK representation is implicit: PPL makes explicit the step ordering (implicit in the ordering of lines of SPARK-L representation) and step-substep relations (implicit in the grouping of tasks within a single procedure in SPARK-L). This allows other tools easy access to the process itself.

Conditionals

PPL denotes conditional branches in a flow chart using a predicate

```
(conditionalFollowedBy <step> <test> <next-step>)
```

meaning that if execution is at <step>, and <test> evaluates to true, then the next step will be <next-step>. At present, <test> is the opaque (quoted) logical expression copied from the SPARK-L test itself, but our plan is to replace this with a transparent logical expression, whose variables include the parameter instances in the rest of the PPL. For example, from the SPARK procedure for purchasing a laptop, the step "relax-and-redo" follows "laptop-query" only if no quotes were found. This appears in PPL as

```
(conditionalFollowedBy laptop-query1
    (and "(= $temp_quotes [])")
    relax-and-redo1)
```

This allows the external systems to see that relax-and-redo1 is a possible next step in the plan, but not at present to understand details of conditions under which it will be executed (unless it was to parse the quoted logical expressions).

Additional Representational Properties

While it is not our intention that PPL capture the full details of the original SPARK process models, it is clear that there are additional details that should be captured. These include preconditions, "cue" conditions, and better handling of logical assertions and tests in the original SPARK. These are all items for future work.

How PPL is Generated

The PPL is generated by introducing specific individual names for each variable, for the cue task, and for each atomic step, such as [do: A] in the procedure. SPARK action symbols, such as enter, become event types. Type and role declarations for the actions are translated into object type statements and role statements. Thus, for an action enter with parameter roles agent of type person and object of type car, we translate

```
[do: (enter $person $car)]
```

into

```
(instance-of enter1 enter-step)
(event-type-for enter-step enter)
(agent enter1 person1)
(instance-of person1 person-var)
(object-type-for person-var person)
(instance-of car1 car-var)
(object-type-for car-var car)
(object enter1 car1)
```

Each of the specific individuals corresponding to the atomic steps is related to the individual corresponding to the cue by possibleTask. These tasks are considered only “possible” because conditional branching or unexpected failures may prevent the tasks from being attempted:

```
(possibleTask goto-work1 enter1)
```

Of more interest is the ordering relationship between the atomic steps, represented by the followedBy and conditionalFollowedBy predicates. Determining this relationship requires walking over the body task network expression, and keeping track of all the possible prior atomic steps and the sequences of conditions that must be satisfied for the current step to follow each of these. We have to consider multiple prior atomic steps when considering a step following a parallel or select construct. An atomic step following a context construct or within a select or wait will be executed only if the appropriate conditions holds, and there may be multiple conditions between the execution of one atomic step and another. To translate iterative constructs, we need to introduce "null" tasks to link the start and end of the loop.

For simplicity, we have ignored the possibility of alternative procedures for the same action. However, this can be represented by making each procedure a subtype of the event type for the action, and the making the cue an instance of a step of that subtype.

PPL and other Languages

PPL and PSL

A well-known standard for process representations is the Process Specification Language (PSL) [Gruninger 04], a logic-based standard for capturing the semantics of processes.
PPL is intended as a complement to, rather than competitor of, PSL, as it captures process knowledge in a different way. The most significant difference between PPL and PSL is that PSL's representation of the ordering of steps is in terms of actual events ("activity occurrences"), while PPL orders the abstract flow chart steps ("activities") themselves. For example, in PSL the above toy plan for going to work would be

```
% enter is a subactivity of going to work
(forall (?person ?car)
   (subactivity (goto-work ?person ?car)
               (enter ?person ?car)))

% driving is a subactivity of going to work
(forall (?person ?car)
   (subactivity (goto-work ?person ?car)
               (drive ?person ?car)))

% In all occurrences of going to work, a driving
occurrence
% follows a entering occurrence.
(forall (?occ ?person ?car)
  (implies (occurrence_of ?occ (goto-work ?person ?car))
     (exists (?occ1 ?occ2)
        (and (occurrence_of ?occ1 (enter ?person ?car))
           (occurrence_of ?occ2 (drive ?person ?car))
           (subactivity_occurrence_of ?occ1 ?occ)
           (subactivity_occurrence_of ?occ2 ?occ)
           (successor ?occ1 ?occ2)))))
```

The last axiom states that for all occurrences of going to work, there will be an occurrence of entering followed by an occurrence of driving. While this makes the semantics of the original flow chart explicit, the actual structure of the flow chart has been lost (the simple relationship "enter $\rightarrow$ drive" is expressed as a complex quantified logical expression).

In principle one could perhaps recover the original flow chart by reverse-engineering it from these PSL axioms, either by parsing the axioms themselves[1] or by theorem proving the general relationships (e.g., proving that driving always follows entering in goto-work). However, neither of these options is particularly easy. In contrast, our goal with PPL is to preserve the original flow chart structure so that it is directly accessible for other agents. One could imagine extending PSL to include some predicate "macros" that would allow the general flow chart relationships to be made explicit, and which would also expand to the traditional PSL axioms such as those shown above. Conversely, PSL makes explicit the actual semantics of the flow chart, and PSL could be generated from PPL if one wanted to make these semantics explicit (indeed, PPL could be a "straw man" candidate for such PSL "macros").

PPL and OWL-S

OWL-S is a OWL-based Web service ontology, which supplies Web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in an unambiguous, computer-interpretable form. Like PPL, OWL-S represents generic procedures themselves, and similarly uses individuals to denote process objects and parameters used by those processes (In this sense, PPL is more similar to OWL-S in approach than to PSL). Generally speaking, OWL-S process is not a program to be executed. It is a specification of the ways a client may interact with a service. A process can generate and return some new information based on information it is given and the world state, or it can produce a change in the world. This transition is described by the preconditions and effects of the process. Processes can be atomic or composite. The composite processes may have control structure such as sequence, parallel, split, join, if-then-else, etc.

Clearly, the scope and the applicability of OWL-S is much different from either SPARK-L or PPL. In terms of expressiveness, OWL-S is comparable to SPARK-L as both are expressive process description languages. OWL-S, PPL, and SPARK-L all use similar representations for

[1] Mike Gruninger reports that a group has done this for a database application, but that this relies on an assumed syntactic regularity in the axioms in order to make parsing feasible [personal communication].

inputs, outputs, and results. Over and above this shared core, PPL provides a representation for the control structure in a process by using the followedBy relation. The current design of PPL is limited to just that. PPL is a subset of SPARK-L designed to support the requirements of reasoning applications. If one were to perform a similar reasoning over OWL-S processes, a PPL-like subset will need to be identified in order to avoid the potential overkill of using OWL-S in its entirety. In a similar vein, PPL has a simple KIF-like syntax, intended to be easily generated and processed by sending/receiving applications, both abstracting out some details and making some implicit semantics of SPARK-L explicit (e.g., event ordering). In contrast, OWL-S is specifically designed to support Web-based services, and hence uses an RDF-based syntax, clearly appropriate for Web-based applications but possibly more cumbersome to deal with in the wider context of process communication. Conversely, OWL-S (and similarly BPEL4WS, the Business Process Execution Language for Web Services) has gone further than PPL in defining different types of process ordering and parallelism. Some of these constructs would be useful to incorporate in PPL as it matures.

Experience Using PPL

PPL directly represents the steps in a procedure, the parameters of those steps, and their ordering. This representation allows a user's ambiguous utterances (e.g., ”find me”) to be matched against expected tasks in the procedure (e.g., ‘‘find_computer”) to identify the user's intent.

We have implemented a translator from the SPARK representation language to PPL. Using this translator, we exported SPARK process models. Using the exported process model, we ran a suite of tests on a dialog interpretation module, and for answering questions. In both cases, the system was able to resolve the user's utterances correctly in a simple, scripted dialog, illustrating proof of concept. Obviously, this is only a first step, but the demonstrated feasibility of the mechanism is encouraging.

Future Plans

The work we have reported here is just an initial attempt at developing a representation that will bridge the requirements of executing a process, and being able to answer questions about it. Clearly, more work needs to be done. First, the language needs to be extended to capture a larger subset of SPARK-L representation. The current PPL ignores many of the details of a process model, for example, the conditions. Second, we would also like to use PPL for specifying and/or modifying existing processes, e.g., through interaction with the user. This latter goal requires reversing the information flow so that PPL is used to generate SPARK-L, rather than the reverse. Currently PPL is too impoverished to support this, but with some small extensions this should be possible, so that either the PPL contains enough information to generate/modify a SPARK process, or suitable software can be written to “fill in the gaps” appropriately (and perhaps interactively) when passing information back to the execution environment.

Although PPL is still preliminary, the general idea of distinguishing representations for execution and representations for introspective reasoning has proved fruitful in our work, and one which we believe will continue to have value as our project progresses further.

Acknowledgement

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), or the Department of Interior-National Business Center (DOI-NBC).

References

Myers, K. L. CPEF: *A Continuous Planning and Execution Framework*. AI Magazine, vol. 20, no. 4, 1999.

Lemai, S., Ingrande, F. Interleaving Temporal Planning and Execution in Robotics Domains, *AAAI 2004*, July 25-29, 2004, San Jose, California, USA.

Morley, D. and Myers, K. The SPARK Agent Framework, in *Proc. of the Third Int. Joint Conf. on Autonomous Agents and Multi Agent Systems (AAMAS-04)*, New York, NY, pp. 712-719, July 2004.

Yeh, P., Porter, B., and Barker. K. Using Transformations to Improve Semantic Matching. *Second International Conference on Knowledge Capture*, October 23-25, 2003.

Gruninger, Michael. Ontology of the Process Specification Language, in *Handbook of Ontologies*, pp575-592, Ed: S. Staab, R. Dtuder, Berlin: Springer (2004).

Erol, K., Hendler, J., and Nau, D. *Semantics for Hierarchical Task-Network Planning.* Technical Report CS-TR-3239, Computer Science Department, University of Maryland, 2004.

Semantic Support for Visualisation in Collaborative AI Planning

Natasha Queiroz Lino, Austin Tate and Yun-Heh (Jessica) Chen-Burger
Centre for Intelligent Systems and their Applications
School of Informatics - The University of Edinburgh
Appleton Tower - Room 4.12, Crichton Street, Edinburgh, EH8 9LE, UK
{natasha.queiroz, a.tate, jessicac}@ed.ac.uk

Abstract

In the last decades, many advances have been made in intelligent planning systems. Significant improvements related to core problems, providing faster search algorithms and shortest plans have been proposed. However, there is a lack in researches allowing a better support for a proper use and interaction with planners, where, for instance, visualization can play an important role.
This work proposes a general framework for visualisation of planning information using an approach based on semantic modelling. It intends to enhance the notion of knowledge-based planning applying it to other aspects of planning, such as visualisation. The approach consists in an integrated ontology set and reasoning mechanism for multi-modality visualisation destined to collaborative planning environments. This framework will permit organizing and modelling the domain from the visualisation perspective, and give a tailored support for presentation of information.

1 Introduction

The need for a broader use of knowledge-based planning has been discussed in recent years. In [Wilkins and desJardins, 2001] it is advocated that the use of knowledge-based planning will bring many advantages to the area, mainly when focusing in solving realistic planning problems. Complex domains can benefit from methods for using rich knowledge models. In this perspective, among the existing planning paradigms, hierarchical task network (HTN) [Erol et al., 1994] is the one more appropriate to this proposition, in contrast to methods that use a minimal knowledge approach, such as the ones using a simple knowledge representation such as these based on STRIPS [Fikes and Nilsson, 1971]. However, despite the HTN paradigm having many advantages, it also has limitations. So, there are many researches opportunities in order to improve and permit a broader use of knowledge models in real world planning problems.

According to [Wilkins and desJardins, 2001] and based on their experience in planning for military and oil spill domains, the following capabilities are needed to solve realistic planning problems: (1) numerical reasoning, (2) concurrent actions, (3) context –dependent effects, (4) interaction with users, (5) execution monitoring, (6) replanning, and (7) scalability. However, the main challenges in real-world domains are that they cannot be complete modelled, and consequently they raise issues about planner validation and correctness. So, in order to make AI planning technology useful for realistic and complex problems there is a need of improvement of the use of knowledge models in several aspects related to planning; and the development of methods and techniques able to process and understand these rich knowledge models.

Three types of planning knowledge are identified by [Kautz and Selman, 1998]: (1) knowledge about the domain; (2) knowledge about good plans; and (3) explicit search-control knowledge. [Wilkins and desJardins, 2001] extended this list about planning knowledge mentioning that knowledge-based planners also deal with: (4) knowledge about interacting with the user; (5) knowledge about user's preferences; and (6) knowledge about plan repair during execution.

Recent researches are following these principles to develop more expressive knowledge models and techniques for planning. For instance [McCluskey and Simpson, 2004] is proposes work in this perspective of knowledge formulation for AI planning, in a sense that it provides support to knowledge acquisition and domain modelling. GIPO (Graphical Interface for Planning with Objects) consists of a GUI and tools environment to support knowledge acquisition for planning. GIPO permits knowledge formulation of domains and description of planning problems within these domains. It can be used with a range of planning engines, since that the planners can input a domain model written in GIPO and translate into the planner's input language. GIPO uses an internal representation that is a structured formal language for the capture of classical and hierarchical HTN-like domains. Consequently it is aimed at the classical and hierarchical domain model type. The advantages of GIPO are that it permits opportunities to identify and remove inconsistencies and inaccuracies in the developing domain model, and guarantees that the domains are syntactically correct. It also uses predefined "design patterns", that are called Generic Types, that gives a higher level of abstraction for domain modelling. To permit a successful use of AI planning paradigms GIPO has an operator induction process, called opmaker aimed at the knowledge engineer that doesn't have a good background in AI planning technology. The GIPO plan visualiser tool allows engineers to graphically view the output of successful plans generated by integrated planners. However it assumes knowledge about the domain.

Based on these ideas of a knowledge enrichment need in AI planning, in this paper we argue that this vision should be even more augmented in other aspects of planning. Our claim is that knowledge enhancement can bring benefits to other areas, and we highlight the planning information visualisation area. Knowledge models developed from the information visualisation perspective will permit modelling and reasoning about the problem, and in this paper we contribute present our approach of semantic support for visualisation in planning systems

The remainder of this document is organised as follows. Section 2 presents the approach overview and architecture. Section 3 details the knowledge models in which our approach is based. Section 4 discusses an information visualisation reasoning motivation in the I-Rescue domain. Finally, we draw some conclusions on Section 5.

2 Framework Approach Overview and Architecture

This work proposes a way to address the problem of visualisation in intelligent planning systems via a more general approach. It consists in the development of several semantic models which when used together permit the construction of a reasoning mechanism for multi-modality visualisation destined for collaborative planning environments. This framework will permit organizing and modelling the domain from the visualization perspective, and give a tailored support of information presentation.

The framework is divided in two main parts: a knowledge representation aspect and a reasoning mechanism. In the knowledge representation aspect of this work, a set of ontologies permits organising and modelling the complex problem domain from the visualisation perspective. The reasoning mechanism will give support to reasoning about the visualisation problem based on the knowledge base available and designed for realistic collaborative planning environments.

The main aspects considered in the semantic modelling include: the nature of planning information and the appropriate tailored delivery and visualisation approaches for different situations; collaborative agents that are playing different roles when participating in the planning process; and the use of mobile computing and its devices diversity. This needs a powerful approach with great expressive power and flexibility. The semantic model is composed by the following (sub) models: Visualisation Modalities, Planning Information, Devices, Agents, and Environment.

Section 3 will be presenting these models in more details, but here we give an introductory explanation:

- **Visualisation Modalities**: Permits the expression of the different modalities of visualisation considered in the approach;
- **Planning Information**: Representation of planning information at a higher level of abstraction, and it is partially based on the I-X <I-N-C-A> (Issues-Nodes-Constraints-Annotations) ontology [Tate 2001];
- **Devices**: Permits description of features of the mobile devices types being targeted, such as, cell phones, PDAs, pocket computers, etc;
- **Agents**: Allows the representation of agents' organisations, including different aspects, such as agents' relationships (superiors, subordinates, peers, contacts, etc.), agents' capabilities and authorities for performing activities, and also, agents' mental states;
- **Environment**: This model allows the representation of information about the general scenario. For instance, position of agents in terms of global positioning (GPS), etc.

Figure 1 illustrates the framework architecture. Using semantic modelling techniques (ontologies), several knowledge models complement each other to structure a planning visualisation information knowledge model. This knowledge model permits modelling and organising collaborative environments of planning from an information visualisation

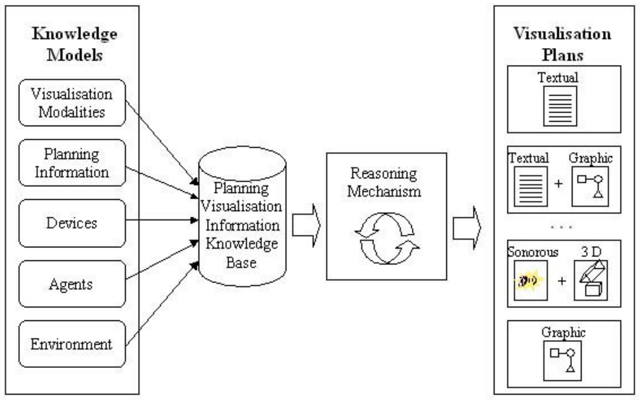


Figure 1- Framework Architecture

perspective. Then, a reasoning mechanism based on the knowledge available, outputs visualisation plans tailored for each situation.

The following sections explain the framework in more details; where Section 3 is concerned with the semantic modelling aspect, while Section 4 exemplify how the reasoning mechanism would work in a search and rescue scenario (I-Rescue domain).

3 Semantic Modelling

In the proposed approach, the definition of the *Planning Visualisation Framework* [Lino and Tate, 2004] is expressed through five different models that define the main aspects of the problem. The next subsections will explain each of them in detail.

3.1 Multi Modal Information Visualisation Ontology

Information Visualisation (IV) is defined by [Card et al., 1999] as the use of computer-supported interactive visual representation of abstract data to amplify cognition. Many classifications of visual representation exist on the literature. [Shneiderman 2004] classifies data types of information visualisation in: 1-Dimensional, 2-Dimensional, 3-Dimensional, Multi-Dimensional (more then 3 dimensions), Temporal, Tree, and Network data. [Lohse et al., 1994] propose a structural classification of visual representations. It makes classification of visual representations into hierarchically structured categories. This classification is divided in six groups: graphs, tables, maps, diagrams, networks and icons. Another classification of visualisation types is proposed in [Burkhard 2004] from a perspective of architects. The visualisation types described are: sketch, diagram, image, object, and interactive visualisation.

These classifications are relevant in many aspects, including help to construct the framework categorisation, to understand how different types of visualisation communicate knowledge, and help identifying research need. Furthermore, the existing development of prototypes for each category offers design guidance.

However, despite the power of information visualisation, in certain circumstances it is not sufficient to transmit knowledge to users. People assimilate information in different manners, and have distinct limitations and requirements. For instance, deaf or hearing impaired people have different needs related to information acquisition. Therefore, different modalities of visualisation and interaction are needed for different users. For this reason, to permit broad possibilities of planning information delivery, it has been included in the framework not only visual representations but also others forms of user interaction, such as natural language interfacing, sonification and use of sounds, etc., as other forms for communicating knowledge. These concepts are modelled in the 'Multi Modal Information Visualisation and Communication Ontology'.

Therefore, this model and ontology definition is derived from previous work as classifications of information visualisation, and furthermore, in requirements for planning information visualisation to real problems [Wilkins and desJardins, 2001], which is representative of the type of scenarios that is being targeted. Then, the core of the semantic definition of this model is based on multi modal visualisation and interaction definitions and also on user tasks that can be performed upon the visualisation modalities.

The ontology includes the following main categories and concepts:

- **1-D Textual:** This category is based on textual representation of information. This modality is appropriated for simple devices that doesn't have many computational resources to present elaborated visual representations;
- **2-D Tabular/GUI/Map:** In this category, it is considered abstractions of information that are represented in two dimensions. For instance, tabular, GUI and map representation. Tabular defines a more structural way to present text (but not only) information, and together with GUI and map based, these representations requires devices with more computational capabilities to present information then text based ones;
- **3-D World:** This modality considers three-dimensional representations of the world for information presentation. Due to the more sophisticated nature of information structure, this category is suitable for more powerful devices;
- **Complex Structures:** In this category it is included complex abstractions of data representation for information visualisation, such as: Multi Dimensional, Tree and Network representations. Multi-Dimensional concerns about representations considering more then 3 dimensions. One example of abstractions of this type is the use of parallel coordinates [Macrofocus, 2004] that represent several dimensions, via a vertical bar for each dimension. Tree and Network visualisation are also included in this category of complex structures. In the literature there are many approaches to address these structures, and the nature of some data types can benefit from these forms of representation;
- **Temporal:** Many solutions for temporal data visualisation is proposed on the literature. Temporal data needs a special treatment. For instance, works such as LifeLines [Alonso at al., 1998] addresses the problem. In the ontology, this modality abstracts the concepts involved in the presentation of temporal data.
- **Sonore (Audio/voice):** In this category audio and voice solutions are incorporated in the ontology. Audio and voice aid can be very useful in certain situations, where the user agent is incapacitated of making use of visual information;

- **Natural Language:** Finally, natural language concepts are also considered in the semantic modelling. Although it is claimed that natural language cannot completely substitute graphical interfaces [Shneiderman, 2000], it is suitable for many situations as it is going to be discussed on Section 4 of this paper.

Other aspects also included the conceptual modelling of this ontology, for instance the user tasks that can be performed. The user tasks are classified as follows:

- **Obtain Details;**
- **Extract;**
- **Filter;**
- **Obtain History;**
- **Overview;**
- **Relate; and**
- **Zoom.**

Depending on the information visualisation and communication modality, the same user task can involve different mechanisms and components to be accomplished.

3.2 Planning Information Ontology

The 'Planning Information Ontology' categorises, at a high level, planning information of the following nature:

- **Domain Modelling:** In this category it is included concepts of planning information related to domain modelling;
- **Plan Generation:** Here, the semantic modelling is concerned with plan generation information concepts and abstractions;
- **Plan Execution:** In this category the ontology includes vocabulary regarding plan execution;
- **Plan Simulation:** Finally, this category models abstractions regarding plan simulation information.

Initially, the main focus of this ontology is the conceptualisation of plan generation information, however the conceptualisation is generic.

Apart from the core planning information definition of this ontology, another important aspect modelled is the aspect of planning for which the information is going to be manipulated. These concepts permit the understanding of planning information from a visualisation perspective. It helps, for instance, in defining strategies for information delivery, based on the aim.

In this way, for the modelling of this idea, the following concepts are considered in the ontology:

- **Planning Information Aim:** Here it is considered that planning information can be used for different aims, which can be domain modelling, plan generation, plan execution and plan simulation. According to the literature and existing planning systems, depending on the aim, planning information is approached in different ways. So, delivering information for domain modelling is not the same to delivering for plan generation.
- **Planning Information:** The conceptual definition of planning information for the purpose of the visualisation framework is based on the I-X <I-N-C-A> [Tate, 2001] model for collaborative planning processes.
- **Planning Information Delivery Strategies:** Based on the literature and existing planning systems it is possible identify that each one of the planning information aim categories (domain modelling, plan generation, plan execution and plan simulation), in general, they deal with different types of information. So for each one can be identified different delivery strategies, because there are different requirements of data presentation, summarisation, etc.

Therefore the main aim of this ontology is to abstract and model these concepts regarding planning information regarding the framework objective of information visualisation.

3.3 Devices Ontology

In the 'Devices Ontology' [Lino at al., 2004] we investigated an approach of knowledge representation of devices capabilities and preferences concepts that will integrate the framework proposed.

CC/PP [W3 Consortium, 2004a] is an existing W3C standard for devices profiling. The approach of CC/PP has many positive aspects. First, it can serve as a basis to guide adaptation and content presentation. Second, from the knowledge representation point of view, since it is based on RDF, it is a real standard and permits to be integrated with the concepts of the Semantic Web construction. For our work, the Semantic Web concepts will also be considered. We envisage a Semantic Web extension and application of the framework that will be addressed in future publications. Third, another advantage of CC/PP is the resources for vocabulary extension, although extensibility is restricted.

On the other hand, CC/PP has some limitations when considering aplying it to the realistic collabortative planning environment we are envisaging. It has a limited expressive power, that doesn't permit a more broaden semantic expressiveness. Consequently it restricts reasoning possibilities. For example, using CC/PP it is possible to express that a particular device is Java enabled. However this knowledge only means that it is possible to run Java 2 Micro Edition (J2ME) on that device. But, it can have a more broaden meaning, for example, when considering '*what really means be Java enabled*?' or '*what is J2ME supporting?*'. Having the answers for questions like these will permit a more powerful reasoning mechanism based on the knowledge available for the domain. For instance, if a device is Java enable, and if J2ME is supporting an API (Application Program

Interface) for Java 3D, it is possible consider delivering information in a 3D model.

For that there is a need to develop a more complex model for devices profiling that will be semantically more powerful. It is necessary to incorporate in the model other elements that will permit enhance knowledge representation and semantic.

The 'Devices Ontology' proposes a new model approach that intends to enhance semantics and expressiveness of existing profiling methods for mobile and ubiquitous computing. Consequently, reasoning capabilities will also be enhanced. But, how will semantics be improved? In many ways, as we will categorise and discuss below.

Semantic improvement can be categorised as follow in the new model being proposed:

- **Java Technology Semantic Enhancement:** In this category is intended to enhance semantic related to the Java world. It is not sufficient to know that a mobile device is Java (J2ME) enabled. On the other hand, providing more and detailed information about it can improve device's usability when reasoning about information presentation and visualisation on devices. For that, in this new model proposed is included semantic of information about features supported by J2ME, such as support to 3D graphics; J2ME APIs (Application Program Interface), for instance, the Location API, that intends to enable the development of location-based applications; and also J2ME plug-inns, such as any available Jabber [5] plug in that will add functionalities of instant messaging, exchange of presence or any other structured information based on XML.
- **Display x Sound x Navigation Semantic Enhancement:** One of the most crucial things in development of mobile devices interfaces is the limited screen space to present information that makes it a difficult task. Two resources most used to by pass this problem are sound and navigation approaches. Sound has been used instead of text or graphic to present information; for example, give sound alerts that indicate a specific message to the user. Indeed, it can be very useful in situation where the user is on the move and not able to use hands and/or eyes depending on the task he is executing. In relation to navigation, this resource can be used sometimes to improve user interface usability, if well designed. However, good navigation design has some complexity due to: devices diversity and because in some devices navigation is closely attached to the devices characteristics (special buttons, for example). So, this category intends to enhance semantic related to these aspects, that will permit a good coordination and reasoning through these resources when presenting planning information to mobile device's users participating in collaborative processes.
- **Open Future New Technologies Semantic Enhancement:** This category of semantic enhancement is the more challenging one in this new model proposition. Mobile computing is an area that is developing very intensely. New devices and technologies are been created every day. In this way it's easy to create technologies that will be obsolete in few years time. Trying to overpass this problem, we envisage that will be possible to provide semantic to future new technologies in mobile computing via a general classes and vocabulary in the model and framework proposed.

3.4 Agents Ontology

This ontology is used to model and organise agents (software and human) regarding their mental states, capabilities, authorities, and preferences when participating in a collaborative process of planning.

The development of this ontology is based on BDI [Rao and Georgeff, 1995] concepts, and also on the I-X ideas. I-Space [Tate et al., 2004] is the I-X concepts for modelling collaborative agents' organisations. Techniques such as agent profiling are being developed to permit adaptation of planning information presentation, since it permits to adapt the type of information delivery to the agent requirements.

3.5 Environment Ontology

The environment ontology is responsible for permitting expression of environment awareness. In particular, location based awareness is being considered, where this type of information is based on GPS (Global Positioning System). Dealing with location-based information will allow the guidance of presentation of information.

4 Motivating Scenario: Reasoning on the I-Rescue Domain

In this section an application of the framework will be motivated. The domain used for that is the I-Rescue [Siebra and Tate, 2003] domain.

The reasoning component of the framework will permit do adjustment of the visualisation and interfacing modalities to agents, devices, environment conditions and type of planning information requirements. In this way, planning information will be delivered in a tailored way.

The kind of reasoning that is performed is based on some principles designed from a study about information visualisation in existing AI planning systems. These principles are based on:

(1) The identification of the type of plan representation that differs depending on the planning approach adopted by the planners;
(2) Understanding of which kind of information is need to be presented and interacted with users;
(3) Classification of the different types of users involved in the planning process;

(4) Identification of most common visual structures (graphical and non-graphical) used in AI planning systems to present information, and;
(5) To which nature of planning information these structures are used to in the planners approaches of information visualisation; and
(6) Finally, in the attempts reported in the literature of adding new forms of interaction with the user, for instance, via natural language processing techniques.

Based on these principles described above and in addition in new requirements desired in collaborative planning information visualisation, rules are being created, in which the reasoning will be based on. For instance, an example of such new requirements is the need of a feedback of human agents that are collaborating on the move in the planning process. Regarding planning information visualisation, this feedback concerns the human agent setting his/hers preferences about change of current conditions while on the move (making use of mobile devices) that will affect the desired planning information visualisation modality for him/her. For example, if the human agent is engaged in an activity that requires extreme visual attention, a visualisation modality based only on graphical representation will not be useful for him/her, because can cause distraction from the main activity being performed. On the contrary, modalities that don't need only visual interaction can suit the situation requirements; such as the ones based on natural language processing and that are sound supported.

The framework is aimed at realist domains of collaborative planning, and the I-Rescue domain fits the requirements of such domains. On I-Rescue scenarios, human and software agents work together and share knowledge and capabilities to solve mutual goals in a coalition support systems fashion. An important feature in systems like that is their ability to support collaborative activities of planning and execution. During planning processes, joint agents share knowledge so that a plan can be built in accordance with the perspectives of each agent. Then the activities in the execution are assigned to specific agents, which will use their individual capabilities to perform the allocated tasks. I-Rescue scenarios consist of relief situations in natural disasters or adversities caused by humans. Situations like that need an immediate response of joint forces with the main objective of saving people lives and minimising suffering. The Kobe Earthquake of January 1995 is an example of how disasters have a tragic effect in urban areas. Most recently the tragedy of The Indian Ocean Tsunami in December 2004 shows the unseen proportions of effects. Situations like that need an immediate response to relief human loss and suffering, and the use of AI techniques and applications can help provide assistance.

5 Conclusions

In this paper it is proposed an integration of ontologies and reasoning mechanism for multi-modality visualisation in collaborative planning environments. The set of ontologies and its integration will permit the expressiveness of several aspects related to real world applications in environments of mixed initiative planning. The reasoning mechanism will allow a tailored delivery and visualisation of planning information. The main contributions of the framework are: (1) it consists in a general framework; (2) the ontology set will permit organising and modelling the domain from the visualization perspective; (3) the reasoning mechanism will give support to presentation of information tailored for each situation; (4) the framework will serve as base for implementations, and (5) the framework is based on real standards (W3C) that will ease communication and interoperability with other systems and services, such as web services.

In addition, we would like to highlight the originality aspect of this work. A semantic modelling approach has not yet been applied to planning visualisation as far as we are aware. The use of ontologies is becoming a trend in the information visualisation field, where an increasing number of works related to this subject have appeared in recent international conferences on the topic. However its use in an intelligent planning context has not been explored yet. This work is an attempt to apply semantic modelling techniques, more specifically via ontologies to a complex collaborative environment of planning.

Furthermore the framework discussed in this paper consists in a high level abstract model that is based, on an implementation level, on W3C standards, which permits the possibility of easy extension and application on the Semantic Web [W3 Consortium, 2004b].

Acknowledgments

The first author is sponsored by CAPES Foundation under Process No.: BEX1944/00-2. The University of Edinburgh and research sponsors are authorised to reproduce and distribute reprints and on-line copies for their purposes not withstanding any copyright annotation here on. The views and conclusions contained here in are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of other parties.

References

[Alonso at al., 1998] D. Alonso, A. Rose, C. Plaisant, and K.Norman.Viewing Personal History Records: A Comparison of Tabular Format and Graphical Presentation Using LifeLines. In Behavior and Information Technology, pages 249-262, 17, 5, 1998.

[Burkhard, 2004] R. A Burkhard. Learning from Architects: The Difference between Knowledge Visualisation and Information Visualisation. In *Proceedings of the Eight International Conference on Information Visualisation* (IV-04). London, England, UK, 2004.

[Card et al., 1999] S. K. Card, J. D. Mackinlay, and B. Shneiderman. Readings in Information Visualization;

Using Vision to Think. Los Altos, CA, Morgan Kaufmann, 1999.

[Erol et al., 1994] K. Erol, J. Hendler and D. S. Nau. HTN Planning: Complexity and Expressivity. In Proceedings of the Twelfth National Conference on Artificial Intelligence AAAI-94, Seattle, Washington, USA, 1994.

[Fikes and Nilsson, 1971] R. E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. In Artificial Intelligence, 2(3-4):189-208, 1971.

[Kautz and Selman, 1998] H. Kautz and B. Selman. The Role of Domain-Specific Knowledge in the Planning as Satisfiability Framework. In *Proceedings in the 4th International Conference on Artificial Intelligence Planning Systems (AIPS 1998)*, Pittsburgh, PA, USA, 1998.

[Lino and Tate, 2004] Lino, N. and Tate, A. A Visualisation Approach for Collaborative Planning Systems Based on Ontologies, in Proceedings of the 8th International Conference on Information Visualisation (IV 2004), London, UK, 14-16 July 2004, IEEE Computer Society Press, 2004.

[Lino at al., 2004] Lino, N., Tate, A., and Chen-Burger, Y. (2004) Improving Semantics in Mobile Devices Profiling: A Model Conceptual Formalisation and Ontology Specification, Workshop on Semantic Web Technology for Mobile and Ubiquitous Applications at the 3rd International Semantic Web Conference, Hiroshima, Japan, 7-11 November 2004.

[Lohse et al., 1994] G. Lohse, K. Biolsi, N. Walker, H. Rueter. A Classification of Visual Representations. In *Communications of the ACM*, 37 (12), pp. 36-49.1994.

[Macrofocus, 2005] Macrofocus. SurveyVisualizer. Url: *http://www.macrofocus.com/solutions/market_research.html*, 2005.

[McCluskey and Simpson, 2004] T. L. McCluskey and R. Simpson. Knowledge Formulation for AI Planning. In *Proceedings of 4th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2004),* pages 459-465, Whittlebury Hall, Northamptonshire, UK, 2004.

[Rao and Georgeff, 1995] Rao, A. S., Georgeff, M. P. (1995). BDI Agents: From Theory to Practice. In Proceedings of the First International Conference on Multiagent Systems}, San Francisco, EUA.

[Shneiderman, 2000] Shneiderman, B. The Limits of Speech Recognition. In Communications of the ACM, Vol. 43, No. 9, pp. 63-65, 2000.

[Shneiderman, 2004] B. Shneiderman. Information Visualisation: Research Frontiers and Business Opportunities. *Course Notes in the 8th International Conference in Information Visualisation*, London, England, 13 July 2004.

[Siebra and Tate, 2003] Siebra, C. and Tate, A. I-Rescue: A Coalition Based System to Support Disaster Relief Operations. Proceedings of the Third International Conference in Artificial Intelligence and Applications}, Benalmadena, Spain, 2003.

[Tate, 2001] Tate, A. I-X and <I-N-C-A>: an Architecture and Related Ontology for Mixed-Initiative Synthesis Tasks, In Proceedings of the Workshop on Planning/ Scheduling and Configuration/Design, Vienna, Austria.

[Tate et al., 2004] A. Tate, J. Dalton, J. M. Bradshaw, and A. Uszok. Agent Systems for Coalition Search and Rescue Task Support, in Knowledge Systems for Coalition Operations (KSCO-2004), Czech Technical University Press, Prague, Czech Republic, 2004.

[Wilkins and desJardins, 2001] D. Wilkins and M. desJardins. A call for knowledge-based planning. AI Magazine, vol. 22, no. 1, Winter 2001.

[W3 Consortium, 2004a] W3 Consortium. (2004). CC/PP Information Page. In http://www.w3.org/Mobile/CCPP.

[W3 Consortium, 2004b] W3 Consortium. (2004). Semantic Web Information Page. In http://www.w3.org/2001/sw/.

The modeling language of GSOC's mission planning team

Christoph Lenzen, Falk Mrowka

DLR - German Space Operation Center (GSOC)
Christoph.Lenzen@dlr.de
Falk.mrowka@dlr.de

Abstract

This paper deals with the classical scheduling problem, assigning certain tasks to a timeline. The tools Pinta and Plato are developed by the mission planning team of the GSOC to support both, automated scheduling as well as interactive scheduling by the operator. Our aim is to provide a flexible and expressive modeling language, which on the one hand allows reusing the software in future missions and on the other hand enables the operator to provide relevant information for the scheduling engine, which can improve the computer generated result.
The resulting model has become quite descriptive. Nevertheless an operator needs to know all details of the model and the given scheduling problem. No effort has been done to implicitly define constraints to 'make life easier', as proposed by [6].

The Modeling Language

A scheduling problem consists of *tasks*, which shall be assigned to the timeline, the *timeline* itself and the *environment*, which allows expressing *constraints* on the assignments. Besides the algorithm can be supported by supplying a certain structure:

The Structure

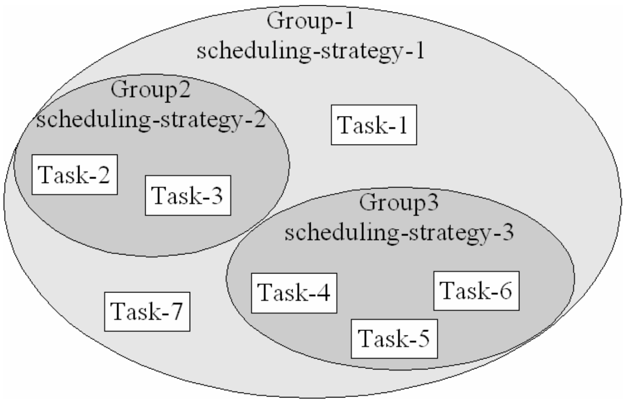
Tasks
The most basic entity of a scheduling problem is the task. It can be given an assignment on the timeline. The main properties of a task are:

- Minimum-duration
  a real number, the minimum duration of each assignment of this task
- Maximum-duration
  a real number or plus infinity, the maximum duration of each assignment of this task
- Scheduled-when
  a real number which indicates, how big the sum of the durations of all assignments of this task must be until this task is considered to be scheduled
- Desired-duration
  a real number which indicates how big the sum of the durations of all assignments of this task shall become. (It is not considered to be a conflict in case this time is exceeded, but the algorithm won't do any effort to schedule the task when this number is reached.)

Groups
Some scheduling engines consider all tasks equal. They may supply functionalities, which analyze the tasks with respect to their properties and constraints and use algorithms, which take these analyses into account. But usually a lot more information on a scheduling problem is known than the mere information on certain tasks and its constraints. The simplest one is that some tasks 'belong together': for example a certain activity can't be represented by a single task, so several tasks have to be defined. One might think of taking a satellite photo, where one has to schedule the uplink (tell the satellite what to do) the data take itself and the downlink (get the photo to the earth). Obviously it may be an advantage to schedule these tasks as a unity. Furthermore it might need a certain algorithm to schedule these tasks (in our example this might be the simple approach 'first schedule the data take, next schedule the uplink as late as possible, at last schedule the downlink as early as possible'). Therefore we allow the tasks to be grouped in the following way:

- Each group can contain tasks and groups, but no cycles are allowed. This means, a group mustn't contain itself as a direct or an indirect element. The direct and indirect elements of a group G are all elements of G and all direct and indirect elements of all groups, which are element of G (i.e. no group G may contain itself and G mustn't contain any group H, which contains G, and so on).
- Tasks and groups may belong to more than one group – in this case the algorithm will usually consider them multiple times.
- Each group may be given a special planning strategy.
- Each group may be given the values
  - Scheduled-when
    a non-negative integer which indicates, how many elements of this group must be scheduled until this group is considered to be scheduled.

- Desired-number-to-schedule
  a non-negative integer which indicates, how many elements of this group shall be scheduled by the algorithm



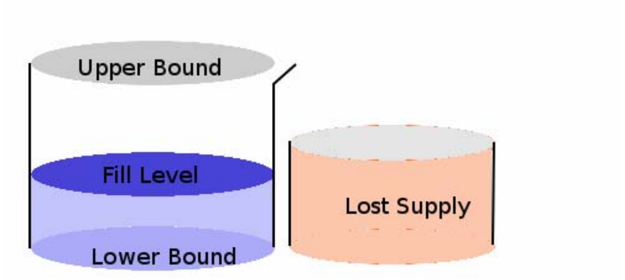
Picture 1: Grouping of Tasks – each group can be given a special planning strategy, which is applied when scheduling the group's elements.

Resources

A resource is a collection of functions in time, which describes a part of the environment, e.g. the amount of energy which is stored in a battery. A resource has the following properties:

- fill-level : a function in time, which describes the state of the resource, in our example the amount of mAh left in the battery
- upper-bound : a function in time, which constitutes an upper bound for the fill-level (may be plus infinity), in our example the capacity of the battery
- lower-bound : a function in time, which constitutes a lower bound for the fill-level (may be minus infinity), in our example 0
- upper-bound-causes-conflict-p : either true or false, in our example false
- lower-bound-causes-conflict-p : either true or false, in our example true

In case upper-bound-causes-conflict-p is set to true, the fill-level may exceed the upper-bound, but this is considered to be a conflict. Otherwise the fill-level is forced to stay below the upper-bound (which means that no conflict exists) and the surplus increase is lost. In our example, this value is false, because in case we are able to supply more energy than we can store in the battery, this surplus energy can be discarded. [Of course this 'lost supply' mustn't be forgotten, otherwise we would forget information: In case the fill-level has reached its upper bound and in case we schedule another supplier (i.e. we try to increase the fill level once more), we actually don't modify the fill-level function. Yet this further supply can be consumed when scheduling another consumer at the same time. For example, the battery is full and the solar panels are switched on. The fill-level of the battery is not modified by the solar panels, but in case one switches on any consumer, this consumer will first use the former lost energy from the solar panels instead of lowering the fill-level. To cope with this lost supply, the scheduling engine will have to keep another function in time rather than only the Boolean value upper-bound-causes-conflict-p.]
Lower-bound-causes-conflict-p is the respective value for the lower bound. In our example, this value is set to true, because any attempt, to lower fill-level below the lower-bound, indicates that we try to use more energy than the battery can supply, which is a conflict. [In this case we need no further function in time to store the surplus consumption, because the fill-level may exceed the bound and thus reflects all consumptions of this resource.]

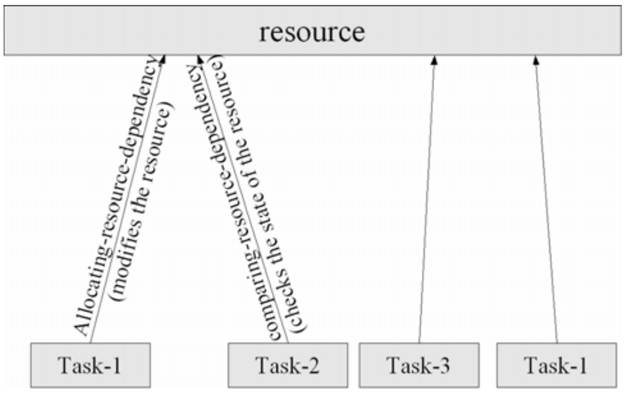


Picture 2: A resource with conflict causing lower bound and non-conflict-causing upper bound

Remark: As you can see, there are different types of resources, but these types are not distinguished by the way the tasks interact with them, but they are distinguished by the way they react, when the fill-level reaches one of the bounds. So one does not have 'equipment', 'opportunity' or 'depletable' resources any more. This distinction will be supplied by the different resource constraints and allows a resource to have multiple roles. In our example there might be a task which does not consume any battery power, but for some reason (e.g. to be able to switch into a safe mode in case of some failure) the battery power must exceed a certain amount. In this case the battery power would act as an opportunity to this task, nevertheless with respect to all other tasks it is a replenishable resource.

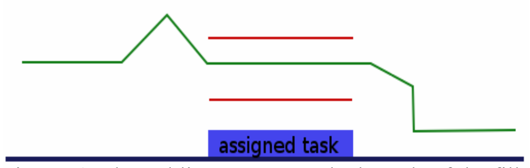
Constraints

Resource Constraints


resource
Allocating-resource-dependency (modifies the resource)
comparing-resource-dependency (checks the state of the resource)
Task-1
Task-2
Task-3
Task-1
Picture 3: A resource as medium for interactions between multiple tasks, where each task can use the resource in its own way (as opportunity, equipment, consumable, etc)

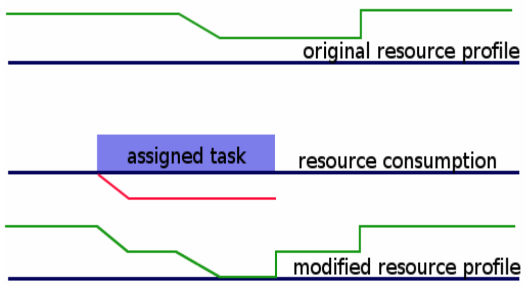
Until now we didn't distinguish between *equipment*, *opportunity* and *depletable resources*. We won't do this now either, because the resource itself shall not determine how tasks may interact with it. Instead we introduce the following constraints C, which all concern one resource R and one task T:

- Comparing Resource Dependency:
  This constraint is mainly given by the value of its properties *upper-bound* and *lower-bound*. These are functions in time periods, which transform to functions in time when the start-time of an assignment is given. The time-period '0sec' will represent the start-time of the assignment. The constraint is that T may only be scheduled at times where the fill-level of R takes values in between upper-bound and lower-bound during the assignment of T. It is considered to be a conflict in case this rule is violated.



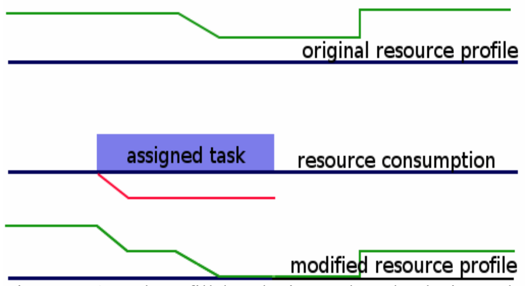
Picture 4: The red lines represent the bounds of the fill-level (green line) caused by an assignment, which has a comparing resource dependency.

- Allocating Resource Dependency:
  This constraint is mainly given by the value of its property *change-function*, which again is a function in time-period, transforming to a function in time by identifying time-period '0sec' with the start-time of an assignment of T. During the assignment of T, the fill-level of R is modified by adding this change-function.
  In case the resource has a conflict causing bound, this constraint is considered to cause a conflict in case there exists a time *t* where the fill-level of R exceeds this bound by a bigger extent than it would without the given assignment.
  In case the resource has a bound which can't cause a conflict, any exceeding of the fill-level is flattened by reducing the value of the respective bound. The lost increase or decrease of the fill-level must be remembered in a certain way.
  This constraint may model the usage of an equipment resource, e.g. crew member allocation.



Picture 5: The red line represents the consumption profile of the assigned task. The original fill-level of the resource (the green line in the upper part of the picture) is lowered during the assignment. Result is the fill-level as given in the lower part of the picture.

- Accumulating Resource Dependency:
  Similar to allocating resource dependency, but the value of the change function at the end-time of the assignment is taken to modify the fill-level in all future times after the end-time of the assignment.
  This constraint may model the usage of a (non)depletable resource, e.g. the usage of battery energy.



Picture 6: The fill-level is reduced during the assignment and after it.

- Dynamic Resource Dependency
  Sometimes the value of a resource must be modified depending on the fill-level of some other resource or of the resource itself. The problem here is that this results in very complex problems to find suitable places where the assignment can start. Therefore we restrict to the following functionality:
  The bounds of the resource must be constant. A *change-formula* is given. This formula takes a real number and returns a real number. When an

assignment of T is made, the fill-level of R at the start-time of the assignment is handed over to this formula. The resulting value then is added to the whole fill-level function, starting with the start-time of the assignment.

To be able to find suitable start times for the assignments, one needs a further formula, namely the *inverse-formula*, which must be the inverse to the change-formula.

Remark: This construct will be implemented, mainly because we can't handle non affine-linear functions (for details see section 'How to Handle Functions in Time').

For example to model the temperature of a machine, which increases punctually when starting some task and which falls exponentially all the time, one has to model the logarithm of the temperature, which falls linear. The punctual raise can be calculated, depending on the current value of the log-temperature, via the change-formula as soon as an assignment is given. To find such an assignment, one will need the reverse-formula, the bounds and the fill-level of the resource.

This construct shall serve as a first analytic approach into more complex dependencies and it is not yet elaborated thoroughly. I therefore expect that there exists a better (i.e. more general) possibility to handle special non-linear functions

The comparison resp. change of the fill level of R need not start exactly at the assignment's start time, but within all resource dependencies one may specify offsets and one may also specify whether the start-time or the end-time of the assignment shall serve as reference to this offset.

Interactions

Of course one must be able to formulate constraints like 'It is a conflict to schedule A without scheduling B' or 'at least 5, but not more than 10 tasks out of a certain group of tasks must be scheduled'. These constraints are called *Interactions* and should be kept separate from other constraints like time-dependencies. Some scheduling systems mix up the terms time-dependency and interaction and therefore they try to compensate the lack of model flexibility by supplying loads of time dependency types.

The two types of interactions are:

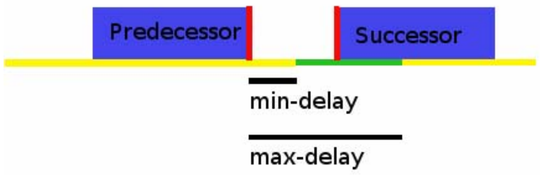
- Demand
  a task, called 'dependant' mustn't be scheduled unless one of a given set of tasks and/or groups, called 'destinations', is scheduled
- Interacting Group
  Let G be a group. One can define an interaction on this group by supplying
  - a minimum number to schedule and
  - a maximum number to schedule.

  In case one of the elements of G is scheduled, it is considered to be a conflict in case the number of scheduled elements of G is smaller than the minimum number to schedule or bigger than the maximum number to schedule.

Time Dependencies

A time dependency concerns two tasks A and B and constitutes a relation between the temporal occurrences of the assignments of A with the assignments of B. In case only one of the tasks is assigned, no conflict may occur. To include the constraint 'Task A mustn't be scheduled unless Task B is scheduled', additionally use the interaction constraints above.

1. Ordered-time-dependencies



Picture 7: Time dependency 'end-before-start'. The successor may start at any time in the green interval.

Task A is called predecessor and task B is called successor. One may specify whether to compare the start or end time of task A with either the start or end time of task B and one may specify the minimum delay and the maximum delay, which must lie between the predecessor A and the successor B. Observe that when using negative delays or when using 'start-before-end', the predecessor might be scheduled after the successor.

This time dependency type might be modeled using resource dependencies:

A resource *AMayStartP* initially is one. An accumulating resource dependency on task B is defined such that adding an assignment of B results in decreasing AMayStartP by 1, beginning with the assignment of B. Task A mustn't be scheduled at places where AMayStartP is <=0. Nevertheless we introduce this time-dependency type, because it is one of the most common constraints and the explicit definition allows the implementation of much faster scheduling methods and time-dependency analyzing algorithms. Besides the time dependency is a 'symmetric' constraint, but the representation via resource dependencies results in two constraints, which treat the two tasks completely different – one of them uses the resource as opportunity and the other one refers to it as if it was a renewable. I therefore expect that any good algorithm needs to recalculate the original time dependencies from the resource dependencies to be able to handle these constraints appropriately.

In case of multiple assignments of individual tasks, we restrict to the interpretation 'all assignments of A and all assignments of B must satisfy the given time-dependency'. Any other interpretation may be modeled similar to the one mentioned above anyway.

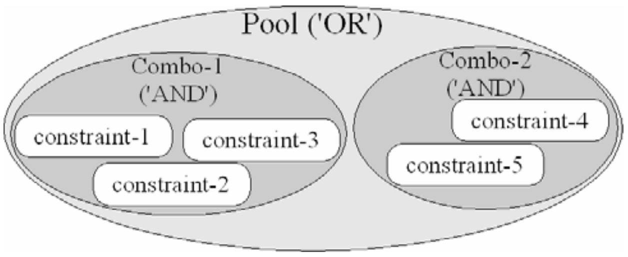
2. Overlap-constraint
   This type of constraint shall once model time-dependencies of the type 'In case A and B are scheduled, their assignments must overlap at least 5 minutes' or 'In case Task A and Task B are assigned, all assignments of A must be included in the assignments of B'. We didn't implement these constraints yet, because we didn't need them, therefore we didn't elaborate the modeling of them either.

Pools and Combos

A combo is a set of constraints. These constraints initially are ignored, i.e. no resource modifying resource dependency is applied and it does not matter in case there exists a conflict on any of them. To consider a combo means to unignore its constraints (i.e. resource modifying constraints are applied and conflicts on constraints are considered to be conflicts of the timeline).

A pool is a set of combos together with a number 'n', which indicates, how many combos of the pool must be considered. Switching between the selected combos is allowed at any time (but this can result in significant calculation effort). Although the combo selection must be done from the moment the pool is defined, the combo selection of a pool won't make any difference unless one of the following happens:

- The dependant of a demand of a combo of P is scheduled
- An element of a group G is scheduled, and there exists an interaction on G, which is part of a combo of P
- A task is scheduled, which has a resource-dependency D, which is part of a combo of P
- Both tasks of a time-dependency D are scheduled and D is part of a combo of P



Picture 8: Only one combo of constraints will be selected. The constraints of this combo will be considered, the constraints of the other combo will be ignored.

Usually a combo is defined to model something like alternative equipment usage.

Example 1: Task 'check timeline for inconsistency' can either be performed by an operator or by a computer. Which option to choose will depend on the availability of the operator and of the computer. In this case the constraints of one pool all refer to one specific task.

Nevertheless no restriction is made on what constraints may be combined.

Example 2: One has to buy two items from a person, which is willing to give away one item for free. One might model this by adding two constraints 'reduce amount of money', one for each task 'buy A' and 'buy B'. These constraints are wrapped by a combo each and put into the same pool. Then the model reflects that one can choose which item has to be paid.

The feature 'pool and combo' may be considered redundant, since one can model alternatives in constraints by creating clones of tasks, each having the constraints of one specific combo. When collecting these clones in a group, where only one element needs to be scheduled, the algorithm can choose between the different constraint combos by selecting the respective task. In example 1, this would result in the tasks 'check timeline by operator' and by task 'check timeline by computer', which are gathered in a group with maximum number to schedule = 1.

More general, the alternative modelling introduces multiple tasks, groups and interactions as follows:

Let P denote a pool. The set M shall contain

- All tasks and groups, which are part of an interacting group, which is part of a combo of P
- All tasks, which are dependants of demands, which are part of a combo of P
- All tasks, which have a resource dependency, which is part of a combo of P
- For each time-dependency, which is part of a combo of P:
  Either the predecessor or the successor

The pool P and its combos can be removed as follows:

- Remove all tasks T of M from the planning problem and remove all groups G of M from the planning problem, including G's direct and indirect elements (i.e. remove all elements of G and in case the element is a group H, do the same with all elements of H)
- For each combo C of P and for each element E of M, add a clone E(C) of E to the planning problem, where all constraints of E are copied to E(M), except for the pool P. Instead of P, the constraints of C are added directly to E(C).
- For each combo C, collect all clones E(C) in a new group G(C)
- Collect all groups G(C) inside a new group Q with the interaction 'not more than one element of Q shall be scheduled'.

In this way, Q replaces the pool P. In case there exist multiple pools, this procedure has to be iterated. Nevertheless some information gets lost, which needs to be recovered by an algorithm, which compares the different choices to find the best one, e.g. with respect to balanced resource usages. This information is the set of all constraints, which the different groups G(C) have in common. For example, the analysis of resource usages may ignore the constraints which are outside the pool P,

but since P has been replaced by Q, this set of common constraints is not known any more. Besides this replacement of pools can increase the number of tasks significantly, especially when there are more than one pools defined on the same task: As soon as one has replaced the first pool P, one single task has been cloned to n tasks, which all have all remaining pools as constraints. Each of these n clone-tasks must be multiplied when replacing the remaining pools and so on.
I therefore suppose that it should be a desired feature to implement pools and combos not only as a 'user interface', but also as a feature of the scheduling engine.

The Timeline

The timeline contains all assignments of the tasks of the planning problem. An assignment consists of a start time and a duration and can either be defined by the operator or by the algorithm. The start times and the durations can be restricted to a certain resolution, but this restriction is optional. By default, the timeline can take any rational value.

How to Include Advanced Scheduling Techniques

The previously described modeling language has been designed to follow a scheduling approach of the type 'Fill up a timeline', where one can define sophisticated variable (= task) and value (= assignment) choice strategies (such as in [2]) and include further move-mechanisms (as done in [5]). 'Repair an existing timeline', as described in [4], can be done as well. This approach can also be used with techniques like network analysis (as done in [1]). In this case the main algorithm would be the process of mapping the pre-calculated network of tasks to the timeline.
Furthermore consider the use of a set of agents, each trying to reach its own goal and all following the rules of a common environment (as described in [3]). This environment (and therefore the internal modeling of each agent) may be described by this modeling language. This approach sounds especially interesting when using multi-processor systems.

How to Handle Functions in Time

The most difficult task of our scheduling engine is to cope with functions in time and functions in time periods. Consider the following situation:
Let TPF denote a function in time periods, which is defined on the interval [0 tp]. Let TF denote a function in time. When selecting a time t, TPF can be transformed to a function in time TPF(t) by identifying a point of time pot with the time period (pot – t). TPF(t) shall be defined on the interval [t (t + tp)]. The restriction of TPF(t) to the interval [t (t + dur)] with dur < tp shall be denoted by TPF(t , dur).
In order to find suitable assignments for the tasks, the algorithm must at least be able to answer the following questions:

1. What times t result in TPF(t) <= TF on the interval [t , (t + tp)] ?
2. What times t and time periods dur result in TPF(t dur) <= TF on the interval [t , (t + dur)] ?

Remarks:

- The problem is not only to avoid rounding, but to calculate the result for all rational times simultaneously.
- When working with resources, which have one non-conflicting and one conflicting bound, one has to take into account the lost-supply resp. the lost-consumption. The questions above therefore actually have to be extended to include these cases, too.

When starting the work on our new scheduling engine, we tried to find some tool, which can calculate the answer to these questions, but we didn't find anyone. (In case you know one, please let us know. ☺) Therefore we created our own functions mechanism:

Our solution: Affine Linear Functions

To be able to cope with these questions, we restrict to affine-linear functions. (It should be possible to extend this to polynomial functions of degree 2, but as soon as the sum of the degrees of TPF and TF is > 4, our approach can't work any more for certain theoretic reasons.)
A function F in time resp. in time period is represented by a list of triples, which we call function pieces:

(start-time start-value interval-values)

Here start-time denotes a point of time (may be infinite), start-value is a real number or +∞ or -∞ and interval-values is either a polynomial of degree 1 or +∞ or -∞.
This list of triples is ordered increasingly in the start-times, no two start-times may be equal and the first start time must be -∞ and the last start-time must be +∞.
To determine the value of F at a given time or time-period X, find the function piece FP with greatest start-time ST, which is not bigger than X.
In case ST=X: F(X) = start-value of F
Otherwise: F(X) = interval-values(X)
To answer the questions above, one only needs to answer the questions, when restricting to each of the function pieces of TPF, compared to each of the function pieces of TF (each restricted to their respective interval – wherever they don't overlap, no restriction takes place). The intersection of these answers is the result.
The extension to the cases with lost supply resp. lost consumption needs some effort, especially with respect to question 2.

Unsolved Problems and Missing Features

1. The following situation can not be handled:
   Task A modifies Resource R1. Task B modifies Resource R2 in a way, which depends on the state of R1.
   The dynamic resource dependency is a first step in this direction, but a lot has to be done to generalize this. For example the following situation can't be handled directly:
   A cross country vehicle can switch between 4 by 4 and normal drive (Resource R1 models these two states by interpreting values 1 as 'vehicle is in 4 by 4 gear' and 0 as 'vehicle is in normal gear'). The fuel consumption is bigger when using 4 by 4, so switching between these modes affects the fuel consumption of task 'drive'. (R2 is the amount of fuel left in the tank, A is the task 'switch to the other gear' and B is the task 'drive'.)
2. As mentioned above, our time profiles (= functions in time resp. functions in time period) are restricted to piecewise affine linear functions. Although they can be used to approach any practically relevant function, this would result in a deficit in the scheduling performance, since the time needed for calculation on time functions raises with the complexity (i.e. the number of function pieces) of the time functions.
   A workaround for one specific non-linear type of resource dependency has been sketched above (again the dynamic resource dependency), but I can't see any general solution to this problem.
3. A timeline is completely deterministic. An event which will be fired during execution of the timeline (see for example [1]) can not yet be modelled.

Summary

The following remarks on our features shall outline the benefits and lacks of our modelling language.

- The structure
  Activities, which are too complex to be modelled by a simple task, must be represented by multiple tasks (which not always have time dependencies, demands etc.). To face the algorithmic problems, which result in this splitting, we allow the tasks to be grouped together (which usually means that they will be scheduled together) and to be scheduled using a special algorithm. Nevertheless a group specific scheduling algorithm must be determined separately and an optimization by varying the algorithm should also include these group specific planning algorithms.
  Besides one can define alternative tasks by supplying the values 'desired-number-to-schedule' and 'scheduled-when' or by adding an interaction on the group.
- Resources
  Resources are not divided into different types, according on how tasks can interact with them. They only differ in how they react when the bounds are reached. This does not only allow different tasks to interact with one resource in different ways, it also saves a lot of resource types and thus a lot of coding.
- Constraints
  - Resource dependencies
    the most descriptive property of a resource is how tasks can interact with them. But not always do all tasks interact with a given resource in the same way, therefore this distinction has been moved to the constraint. In this way, the resource type of a resource R is mainly specified by the resource dependencies which point to R.
  - Time dependencies
    the time dependencies do not involve any constraint of the type 'A can only be scheduled when B is scheduled' any more. This safes a lot of time dependency types and thus a lot of coding. To add such a constraint, use demands.
  - Interactions
    - Demands
      see ‚time dependencies' above
    - Group interactions
      this constraint allows to model alternative sets of tasks
- Timeline
  By default, the timeline has no resolution. This means on the one hand that the scheduling problem has an infinite value domain, which makes it impossible to use certain algorithms, such as brute force. In case one wants to apply such an algorithm, one first has to find a suitable resolution.
  On the other hand, no time profiles should be handled as arrays of constant length, where one element represents the value of the ith time section on the timeline, because this would restrict the application of the scheduling engine to cases where the duration of all tasks is comparable with the time horizon. For example in our satellite mission 'Terra-Sar-X', we have to cope with data takes of a few seconds length (the difference in lengths is even lower) and a time horizon of years. The resulting opportunity profile of a single data take for example would be represented by loads of 0s with a few interrupting 1s. So why restrict to a certain resolution when

the time profiles are represented dynamically anyway?

Not all features described in this document have already been implemented in our tools Pinta and Plato and some details may differ. But all main features have already been implemented and we found that the power of a common today's personal computer can handle this modelling in an acceptable time. (A strict performance test has not yet been done, mainly because of lack of time, but also because some features to speed up the algorithm have not yet been implemented.)

References

[1]A. Barrett, R. Knight, R. Morris, R. Rasmussen: *Mission Planning and Execution within the Mission Data System*, Proceedings of the 4th International Workshop on Planning and Scheduling for Space (IWPSS-04), page 13
WPP-228, Noordwijk, Netherlands: ESTEC - ESA Publications Division

[2]H.M. Calvani, A.F. Berman, W.P. Blair, J.R. Caplinger, M.N. England, B.A. Roberts, R. Hawkins, N. Ferdous, T. Krueger: *The Evolution of the FUSE Spike Long Range Planning System,* Proceedings of IWPSS-04, page 25
WPP-228, Noordwijk, Netherlands: ESTEC - ESA Publications Division

[3]B. Clement, A. Barrett S. Schaffer: *Argumentation for Coordinating Shared Activities,* Proceedings of IWPSS-04, page 44
WPP-228, Noordwijk, Netherlands: ESTEC - ESA Publications Division

[4]A.S. Fukunaga, G. Rabideau, S. Chien: *Robust Local Search for Spacecraft Operations using Adaptive Noise*, Proceedings of IWPSS-04, page 78
WPP-228, Noordwijk, Netherlands: ESTEC - ESA Publications Division

[5]L.A. Kramer and S.F. Smith: *Task Swapping: Making Space in Schedules for Space,* Proceedings of IWPSS-04, page 107
WPP-228, Noordwijk, Netherlands: ESTEC - ESA Publications Division

[6]J. Jaap, E. Davis, L. Richardson: *Maximally Expressive Model*, Proceedings of IWPSS-04, page 86
WPP-228, Noordwijk, Netherlands: ESTEC - ESA Publications Division

Using ontologies for planning tourist visits

Juan David Arias
Departamento de Informática
Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
jdarias@inf.uc3m.es

Laura Sebastiá
Univ. Politecnica de Valencia
Valencia, Spain
lstarin@dsic.upv.es

Daniel Borrajo
Departamento de Informática
Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
dborrajo@ia.uc3m.es

Abstract

In order to let software applications use planning technology, one of the needs consists on the knowledge exchange between the planner component and the rest of components. Currently, one of the most widely used means of supporting this exchange in a declarative way is through the use of common ontologies. In this paper, we present our on-going work on trying to build one such systems, SAMAP. [1] The aim of this project is to build a software tool to help people visit different cities. This tool integrates modules that dinamically capture user models, determine lists of activities that can provide more utility to a user given past experience of the system with similar users, and generates plans that can be executed by the user. This system is intended to work in portable devices (mobile phones, PDAs, etc,) with internet connection. In this paper, we focus on the common knowledge representation, which is modeled by means of an ontology, and the planning component of the project.

Introduction

During the last years, global networks, such as the Internet, have experimented an important growth. This is causing a social and economic impact in many aspects. For example, from the commercial point of view, we can buy a different range of products through the Internet, i.e., books and films, and we can even buy a plane ticket or book a hotel. The success of these new commercial activities is related with the fact that we can ubiquitously use hardware devices with connection capabilities, such as Personal Digital Assistants (PDAs) or mobile phones with access to the Internet, so that we can compare prices, and buy products we are interested in.

Electronic tourism is one of the activities that have enjoyed of an important success in the Internet, not only from the commercial point of view but also from a social perspective. Many sites provide information about hotels, plane tickets, etc. There are also recommender systems that tell us which destination is more suitable according to our preferences (Delgado & Davidson 2002; Fesenmaier *et al.* 2003). Once selected a destination, we can find many sites which give us information about places to visit in a city, activities to do during the travel, restaurants, etc. But this information is static in most cases; that is, it is presented to all users in the same way. Also, the quantity of available information can be large and, therefore, the user must select which pieces of information are interesting for him/her. Finally, and most importantly with respect to the goals of this paper, they do not automatically provide plans and schedules, according to user needs and sites schedules. So, from the user point of view, it is useful to have a recommender system that tells him which places may be interesting to visit in a certain city taking into account his/her profile, computing a tourist daily plan, with indications about which places to visit in the given timeframe, and also how to go from one place to another.

We have partially developed a system which provides this functionality. It dynamically captures and updates a user model about different city visits, analyses past planning behavior of the user and similar users in the same type of visit, and selects a list of places that have a high probability to be interesting for the user. Then, taking into account distances, places timetables, etc., it computes a plan, and also shows how to go from one place to the next in the plan. In order to store all the information the system needs, we have defined an ontology.

Nowadays, the use of ontologies in the internet is increasing, specially motivated by the Semantic Web efforts. Ontologies are being also used when building multiagent systems, in order to share their common knowledge in a declarative way. SAMAP has been built as a multiagent system, consisting of three main agents: user modelling and interface agent, case-based agent, and planning agent. In (Heflin & Muñoz-Avila 2002), Heflin and Muñoz-Avila present another example of this kind of system that integrates HTN planning and Semantic Web ontologies for defining agents capable of solving complex information integration tasks. Other related work are the trip planning systems for booking flights or hotels, renting cars, etc. though most of

[1] This work has been partially supported by the Spanish MCyT under project TIC2002-04146-C05-05

this work does not report in-city planning/scheduling and/or the use of ontologies (Ambite *et al.* 2002; Knoblock & Minton 1998; Camacho *et al.*). There are other proposals of tourism ontologies, like for example Harmonise[2]. The goal of Harmonise is that participating tourism organisations keep their proprietary data format while cooperating with each other. Harmonise focuses on obtaining the interoperability between different organizations with different standards so that they can share information with others.

In contrast, SAMAP does not solve the problem of travelling to a specific place. Therefore, it does not need to know anything about flights, travel agencies or other elements related to travelling from a city to another one. SAMAP focuses on facilitating the activities that a tourist can perform in a city, also considering the transport means that can be used to move within the city that is visited.

The following section details the ontology defined in SAMAP. Next section introduces the architecture of our system, in particular, it details the planning agent whereas the next section describes how this agent uses the information stored in the ontology. We finish with some conclusions.

SAMAP **ontology**

The system relies on an ontology that is shared by all agents, where each agent uses part of it. The main classes and their relationship appear in Figure 1.

In order to define this ontology we have used PROTÉGÉ which is an ontology editor developed at Stanford University.[3] This editor helps on the creation of ontologies and exports the information about created ontologies in many different ontology formats, such as CLIPS, which is the format used SAMAP because it is more suitable for our system. However, other languages, such as OWL (Ontology Web Language), which is very powerful and considered the standard in Web services, could be used in the future as PROTÉGÉ has a set of extensions or plugins that allow to load, to edit and to save ontologies in OWL format.

The SAMAP ontology consists on three main classes: user, activities and city information. The **User class** has attributes about the personal information of the user like name, profession, mobility, language, sex, etc, but it also references other classes where other user information is considered (see Figure 1). These attributes as well as the captured user model are used for determining which visits the system should recommend to the user. For example, if SAMAP knows that a user has a reduced mobility (s/he needs a wheelchair), then the planner will not offer plans with places that are not prepared to receive visitors in a wheelchair. The same applies to other attributes. The user model with his/her preferences are also used for proposing the user plans with activities that were performed by people similar to the user on that same city.

The User class is related to the **User-Context class**. This class has context-dependent information, such as user localization, if the user has car on that visited city, money available, residence, etc. The user localization is useful for planning the itinerary whereas knowing how much money is available is used for computing price-dependent plans. There are other attributes in this class as the ones referring to the type of connection device that help on performing device dependent interaction (but we do not focus on this aspect in this paper). An important attribute in this class is *current-visit* that is a reference to one instance of the Visit class. The **Visit** class has the information on the city that is being visited, the reasons of the visit, the free time the user has, as well as the activities (plans) that the user performed in the past, the activities (plans) that s/he rejected, etc. The ontology also accepts storing multiple past visits of the same or different users, so it can deduce new user preferences by performing machine learning or case-base (CB) reasoning on past visits.

A city is represented by the **City class** together with classes that describe transport means, places that can be visited, as well as the streets that compose the map of a city. For instance, streets are represented in terms of intersections and street sections, and contain information about the district they belong to, type of street, or traffic in that section (in case we can connect to on-line traffic information resources). Using these classes, we can define the graph that represents the map of a city, so SAMAP can perform path planning to know how to go optimally from one place to another, according to a given user defined quality metric.

The **Place class** defines specific sites within a city that can be visited, such as restaurants, museums, generic buildings, bars, open-spaces, or theaters. It contains information related to the user, such as accessibility, price, or ways of payment, as well as information needed for planning, such as timetables, which will be used to schedule the plan. The location in a street section is also specified for each place.

The transport means are represented by the **Transport class** and its subclasses, which denote specific transportation means: underground, railway, bus, taxi and walking. Some of them contain information on the graphs related to their itineraries.

Activities, which are represented in the **Activity class**, can be of two types: generic and specific. A generic activity is, for example, going to **a** museum, without specifying which one. This is useful when a user likes visiting museums in general. A specific activity relative to museums would be to visit the Hermitage museum. Therefore the system can recommend to visit several museums if “going to a museum” appears in the list of generic activities, or to visit an specific museum if it is in the list of specific activities. Planning goals will be automatically generated from the set of activities that the user has selected to perform within the

[2]www.harmonise.org
[3]http://protege.stanford.edu

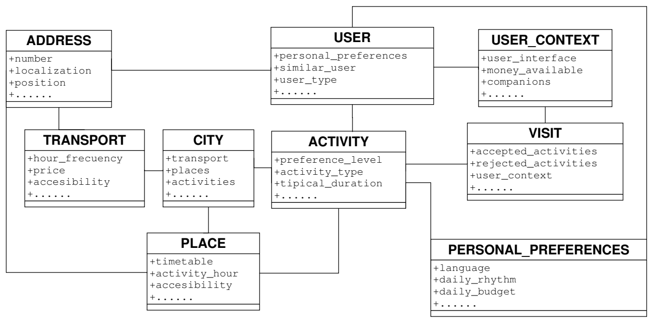


Figure 1: High level view of the ontology developed for SAMAP.

city, as well as the ones that SAMAP has selected automatically from the user preferences and from previous visits of similar users to the same city. Examples of the activities that can be performed in a city are to visit a museum, a church, eat in a restaurant, go to a concert, attend a show, go to the cinema or theatre, etc. Each one of these types of activities is defined in a separate subclass.

While some of the attributes of the Activity class are useful for the planning process, others are important to select which places will be recommended to the user. For example, the attribute that determines the type of food helps the system to recommend a restaurant according to the user preferences. On the other hand, the attribute that indicates the typical duration of an activity helps to decide which activities could be performed taking into account the available time of the user.

The **Movement class** is a special subclass of Activity, given that it represents the action of going from one place to another. It provides a uniform way of representing the output of the planning agent, plans that contain performing activities of visiting places, as well as activities for going from one place to another. The system admits several types of transport means, including walking, and represents and computes several quality metrics related to them, as price, duration, distance, or utility for the user (related to preferences).

One of the advantages of having an explicit knowledge model such as the ontology we are using is that some elements of the ontology are recovered from Internet by means of wrappers. Information relative to cinemas, theaters, restaurants or museums is updated from Internet addresses, which can improve the robustness of the plans. We plan to study in the future the relationship between the continuous access to updated information and reliability of plans.

SAMAP

We have developed a prototype called SAMAP, whose goal is to compute a tourist plan for a user with Internet access via a ubiquitous device, such as a PDA or a mobile phone. This system needs to have access to the information represented in the ontology:

- Information about the city, that is, the context that surrounds the user (i.e. monuments and interesting places in a city, etc.)
- Personal data, interests and preferences of the user, that is, type of activities that this user likes to do when s/he visits other cities (user model)
- Places that other people similar to our current user liked when they visited the same city (plans of other users)
- Basic services that this user requires to perform his/her activities (i.e. payment with credit card, use only taxi for movements, etc.)

Once we have all this information, the application computes a plan that contains the following elements:

- a selection of the most interesting places for this user according to his/her model
- indications about which transportation means s/he should take to move between different places including walking
- recommendations about where to have lunch or dinner (restaurants, bars, etc.)
- proposals of places of leisure such as cinemas or theatres

The following subsections detail the architecture of this system, specially with respect to its architecture.

SAMAP **Architecture**

Figure 2 shows a high level view of the architecture of SAMAP. The first step consists of **building the user model**. This requires the user to enter information about him/herself, that is, personal data, interests and preferences about, i.e., art, monuments, meals, etc. Also, the user should specify which city is s/he is going to visit, under which schedule, etc. This information can be gathered by using any device with Internet connection. In order to obtain more interesting data about the user, the system (by means of learning techniques) can use past information about the same user (provided s/he has used the application before). This information will be stored in the User and Visit related classes.

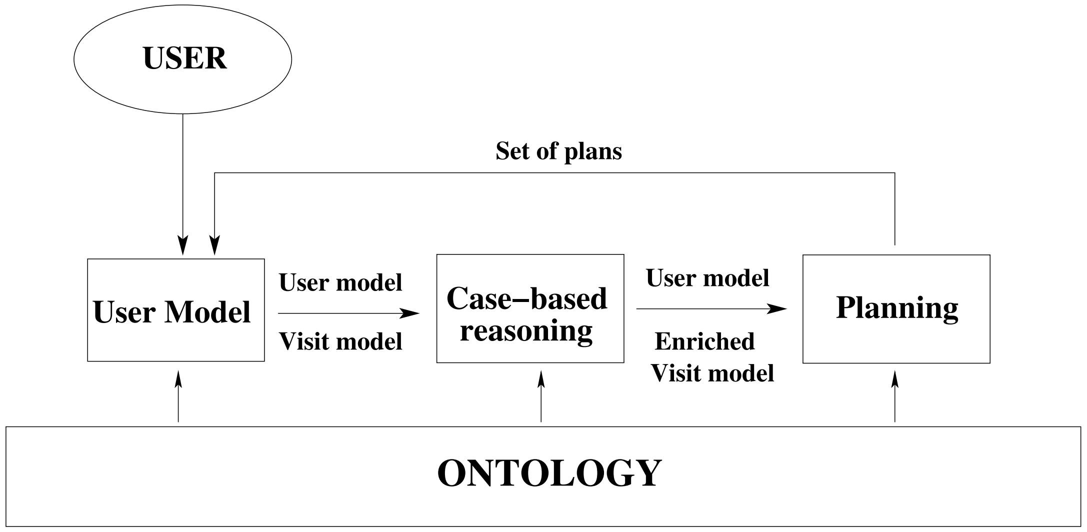


Figure 2: High level view of the architecture of SAMAP.

The second step consists of the generation of a list of activities that s/he might like to perform in the current visit according to the preferences. As explained before, the activities might come directly from the user, or automatically generated by CB reasoning from similar users plans in the city or similar cities. Each activity will also be described with the expected utility that performing this activity might have for the user. This utility can be directly specified by the user, or computed by SAMAP from knowledge about similar users.

The last step is the computation of the tourist plan by taking into account the previously computed list of activities. One of the questions that arises here is why not use a CB reasoning mechanism throughout the system. We prefer to use a planning approach with "classical" operators because (a) there is not a CB planning tool that allows to develop appplications quickly and (b) in big cities where there are a number of different goals and ways to go from one place to another, we would need a very rich adaptation agent. Therefore, we think we have adopted the easiest solution: the planner takes this information as input together with information about the city (streets and intersections, situation of each place, etc.) to compute the plan for the user. The next section describes the architecture of the planning system and Section *Using the ontology within the planning agent* details how this information is used in the planning system.

Planning agent Architecture

One of the inputs of the planning agent is the list of activities selected by the CB reasoning agent. This list is not directly the goal of the planning problem, as it can contain more places than the user will be able to visit. Therefore, the planner must select which of them should be included in the plan. Moreover, the planner must schedule each visit according to the place timetable, deal with the city map, etc. This planning task has several features that make it hard for current planners:

- **Time management:** each visit should be scheduled according to the opening hours of each place and the expected duration of the visit according to the user model. Also, it should consider the time to go from a place to another.
- **Management of numerical values:** the prices of the visits, meals and transports must not exceed the available budget.
- **Locations:** the plan indicates how to go from one place to another, that is, which transport the user should take. In case it is preferable to walk, the plan indicates which route the user should follow.
- **Goals:** we can specify three types of goals:
  – totally instantiated goals, i.e., visit a specific museum
  – partially instantiated goals, i,e., generic goals like visiting any museum
  – a metric indicating that the plan must maximize its utility

  Moreover, not all the available places must be visited, that is, not all goals will be achievable, because of scheduling constrains related to timetables (one cannot enter Prado's museum at night) or to the available time of the user (s/he cannot visit five places if s/he only has time to visit two).

Most of these features can be handled by most temporal planners. However, we will focus on the ones that are more specific:

1. **Each visit is scheduled according to the opening hours of each place.** This implies that we need

an explicit management of the current time. In addition to the restrictions that existing temporal planners take into account when scheduling an action, our system must consider the current time because a place cannot be visited if it is closed. This also causes that there can be empty time points, that is, time points when no action is executed (from the point of view of the user, an empty time point is free time). This free time is not handled by temporal planners; every time point has an associated action.

2. **The plan indicates how to go from one place to another.** This task can be performed (theoretically) by any planner. But our system has to deal with all the transport means (subway, bus, taxi, ...) in a city and the routes one can take to walk from one place to another.
3. **The objective of the problem contains partially instantiated goals.** This means that the planner must select only one specific place whose type is equal to the partially instantiated goal.
4. **The plan is computed in order to maximize the utility of the visits.** In a general planning problem, even when we deal with durative actions and numerical values, we specify the goal as a set of literals that must be true at the end. However, in our case, there is not such a set of literals; our goal is to maximize the utility of the plan. This kind of problems were introduced in the International Planning Competition held in 2003 in the hard numeric track, but only few planners (MIPS (Edelkamp 2002), SHOP2 (Nau *et al.* 2003) and TLPlan (Bacchus & Ady 2001)) were able to solve them, two of them require manual codification of the domain.

As it is difficult to use a simple planner for solving the planning part of the application, we propose a hybrid system composed by a planner and other modules. As we said above, the input of the system is an original list of activities that represents the places of interest of the user (including eating and leisure places) together with a number indicating their utility, i.e the *satisfaction degree* that visiting such site provides to the user (remember that this satisfaction degree is calculated by the CB reasoning agent). Each activity can be totally or partially instantiated, as in `visit museum of Modern Art` or `visit a museum`. The output is one or more tourist plans which contain a list of scheduled visits along with the indications about how to move from one place to another. The system also computes the cost of the plans trying to maximize its quality according to the established metric. By default, the quality metric is to maximize the total utility, but it could be to diminish the cost, a combination of both or any other one. The ontology stores all the information:

- about the city: streets and intersections, public transport networks, etc.
- about the places the user can visit: situation, timetable, price, etc.
- about the user model: a list of activities together with the corresponding utility

The final step of the planning system is to store the generated plans into the ontology. Figure 3 shows an schematic view of the proposed architecture. It is composed of five modules:

- **Translator** module: it transforms the original list of activities into the predicates required by the planner. The ontology stores all necessary information. The output is the initial state of the problem, the goals (list of activities) and the transport graph, i.e, all the information related with the transportation (buses, underground, users preferences . . .). It also transforms the generated plans into the corresponding instances of the ontology. This module will be deeply studied in Section *Using the ontology in the planning module.*
- **Control** module: it coordinates the rest of modules providing the input that they need in the suitable format and gathering their output. It can invoke the Transport module or let the Planner module do it
- **Selector of activities** module: the aim of the system is to maximize the utility of the plan and/or to satisfy some specific or generic goals, but not to fulfill all the available activities. This module selects the most appropriate actions to be solved by the planner each time. This module has been designed to have a similar behaviour to a tourist. For example, it can select first those activities which are near places with the highest utility.
- **Planner** module: it generates the plans. Its input is a domain theory (set of operators and types hierarchy), planning problem (initial state and goals, list of activities), a quality metric, a cost bound and a time bound. It tries to generate one or more plans that solve the problem during the time bound. The cost of the solutions cannot be greater than the cost bound, according to the quality metric. There can be more than one Planner module each with a different planning technique. The Control module compiles all the results. This module can directly communicate with the Transport module.
- **Transport** module: it receives an origin and a destination point and returns the transport subplan for moving a person from the origin to the destination. It also returns the cost and time of the itinerary.

Using the ontology in the planning agent

Once we have defined the architecture of the planning component of SAMAP, we describe here the interfaces needed to translate the knowledge contained in the ontology that is relevant for planning, and translate back the resulting set of plans into the ontology, for further use by other components. Figure 4 shows how the translation is performed in SAMAP.

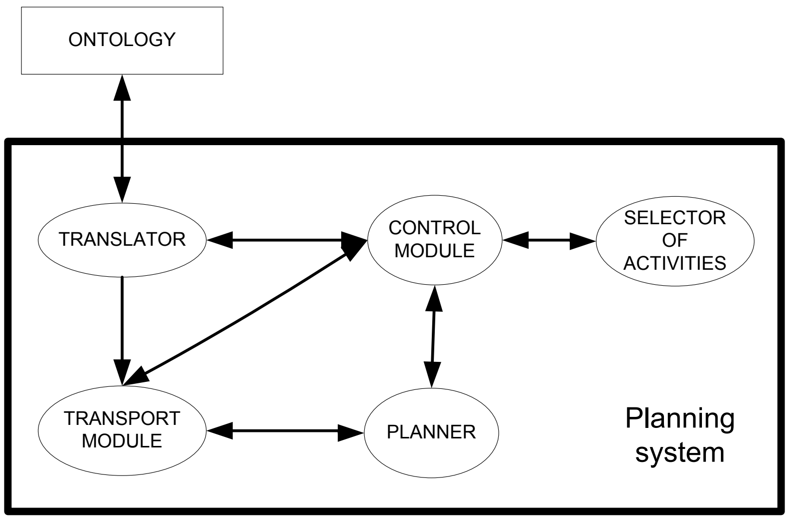


Figure 3: Planning System Architecture.

The input knowledge to the planner component is stored in the ontology spread in different classes as it was described earlier. In particular, from the point of view of planning, the most relevant knowledge contained in the ontology refers to the city and the user. Every specific city contains instances of city related classes in a file with the `pins` extension (CLIPS extension for knowledge about instances of classes). As we said above, the planning agent is interested in those classes that refer to the places the user can visit, to the list of activities recommended by the CB reasoning agents and to the topology of the city and transports, that is, streets, intersections, buses, underground, places, etc. The knowledge about streets and transports is translated into a path planning problem (how to go from one place of the city into another one). The path planning problem can be solved by the planner at the same time as solving the planning problem (what activities to perform and when), or solved by the transport module described in the previous section.

The first solution is adopted if we use a planner such as FF (Hoffmann & Nebel 2001) given that it does not easily allow to plug-in other problem solving components. So the whole transport (or path planning problem) gets translated into PDDL (Fox & Long 2002) as part of the initial state. In this case the path planning graph (nodes are intersections, and addresses and arcs are transport means that move one person from one node or another) is represented with two predicates: `coordinates` that represent where nodes in the graph are; and `arc` that contains information about the type of arc (walk, bus, underground, ...), origin and destination, utility (for some people walking is better than taking the bus, but worse than taking the underground), distance, time, price, and cost (we can represent and reason about any other type of cost).

The second solution (a separate transport problem) is the one we use with PRODIGY, given that it is quite easy to plug-in new execution components to this planner. In this case, the translator generates before planning a path planning graph from the knowledge about the city in the ontology. Then, at planning time, every time PRODIGY tries to apply a transport operator (such as walk from A to B), it calls a path planner (for instance, an IDA* algorithm) with the origin and destination. And the path planner will use the planning graph to compute a solution, which returns to the planner.

On the other side, the translator is also in charge of building the initial state and goal of the planning problem. The initial state contains all those predicates that refer to the situation, schedule, price, ..., of each place together with the available schedule and the current location of the user, etc. The goal of the problem represents, in our context, the list of activities that the previous two modules have selected as relevant (useful) for the user. However, as we explained above, the existing planners try to satisfy all the predicates specified

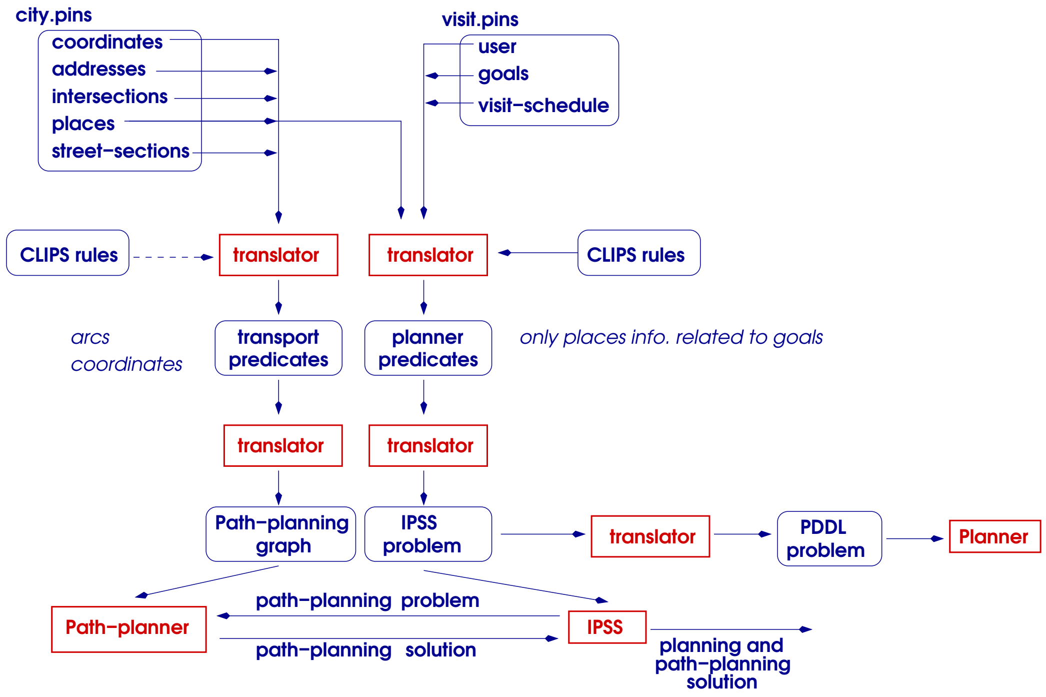


Figure 4: Translation between ontology instances and the planners within SAMAP.

in the goal. For this reason, the selector of activities first shrinks the list of recommended activities to those most appropriate according to some criteria (proximity, etc.).

We have used CLIPS for this translation given that it provides a declarative and easy way of performing the translation. We also used CLIPS in a previous version of the transport problem translator, but due to the size of path planning graphs for medium size cities such as Valencia (Spain), it took way too long to translate it. Consider that our current map of Valencia has over 7600 nodes and over 22000 arcs (only for walking, without considering buses or underground for now). So, this part of the translator has been re-programmed in CommonLisp, although it could have been easily done with more appropriate tools, such as YACC and BISON.

Once the transport and planning translators generate the initial state and goals of the planning problem (possibly also a path planning problem), planning can start. We are using now two planners, PRODIGY and FF (we are also working on adapting a HTN planner, but this requires more effort in the domain definition). The translators generate PDL (PRODIGY Description Language) problems, but we have also built a PDL to PDDL translator, so we can also use FF for solving planning problems. After the planning process, the planners can either return no solution (when all goals cannot be scheduled in the given time frame), or a list of plans (in the case of FF only one). If no solution was found, then the control module calls the activities selector module to return a new sublist of goals and the planner is called again with the same initial state and the new subset of goals. If a set of solutions was found, it is translated back to ontology instances in a solution file. Every planning solution is a list of instantiated operators. These operators can be either visit some place, attend some event (cinema, theatre, concert, ...) or perform a move operator (walk from one place to another, take a bus from one stop to another, ...). The solutions file is sent to the user module for further processing (selection of a plan by the user, modification of conditions and replanning, or execution of the plan).

Conclusions

The application of AI Planning&Scheduling to real world problems is a growing area that attracts more researches each day. Also, the development of global networks such as Internet, together with new hardware devices with good connection capabilities, is causing a social and economic impact in many aspects. Here, we have described our on-going work on building an application for assisting any tourist for planning the visit to a city, using a PDA or a third generation mobile phone. These plans are adapted to the user preferences proposing only a list of activities that could be really interesting and achievable for him/her.

SAMAP has been built as a multiagent system, consisting of three main agents: user modelling and interface agent, CB reasoning agent, and planning agent. In order to facilite the information exchange among these agents, we have developed an ontology. It stores information about the user and his/her preferences, the activities that can be performed in a city and the city itself. In this paper, we have focused on how this information is used in the planning agent.

References

Ambite, J. L.; Barish, G.; Knoblock, C. A.; Muslea, M.; Oh, J.; and Minton, S. 2002. Getting from here to there: Interactive planning and agent execution for optimizing travel. In *Fourteenth Innovative Applications of Artificial Intelligence Conference (IAAI)*. Edmonton, Alberta, Canada: AAAI.

Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency: A forward chaining approach. In *International Joint Conference on Artificial Intelligence (IJCAI-2001)*, 417–424.

Camacho, D.; Aler, R.; Borrajo, D.; and Molina, J. M. A multi-agent architecture for intelligent gathering systems. *AI Communications*. In Press.

Delgado, J., and Davidson, R. 2002. Knowledge bases and user profiling in travel and hospitality recommender systems. In *Proceedings of the ENTER 2002 Conference*, 1–16. Innsbruck, Austria: Springer Verlag.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *AIPS'02*. AAAI Press.

Fesenmaier, D. R.; Ricci, F.; Schaumlechner, E.; Wöber, K.; and Zanellai, C. 2003. DIETORECS: Travel advisory for multiple decision styles. In *Proceedings of the ENTER 2003 Conference*, 29–31. Helsinki, Finland: Springer Verlag.

Fox, M., and Long, D. 2002. *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains*. University of Durham, Durham (UK).

Heflin, J., and Muñoz-Avila, H. 2002. LCW-based agent planning for the semantic web. In *Ontologies and the Semantic Web. Papers from the 2002 AAAI Workshop WS-02-11*, 63–70. Menlo Park, CA,: AAAI Press.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Knoblock, C. A., and Minton, S. 1998. The ariadne approach to web-based information integration. *IEEE Intelligent Systems* 13(5).

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Mur, J. W.; and Wu, D. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.