Knowledge Representation in Planning: A PDDL to OCL_h Translation

R. M. Simpson, T. L. McCluskey, D.Liu, D.E.Kitchin
Department of Computing Science
University of Huddersfield, UK
email ron@zeus.hud.ac.uk

March 16, 2000

Abstract:

Recent successful applications of AI planning technology have highlighted the knowledge engineering of planning domain models as an important research area. We describe a prototype implementation of a translation algorithm between two languages used in planning representation: PDDL, a language used for communication of example domains between research groups, and OCL_h, a language developed specifically for planning domain modelling. The translation algorithm has been used as part of OCL_h's tool support to import models expressed in PDDL to OCL_h's environment. In this paper we detail the translation algorithm between the two languages, and discuss the issues that it uncovers. The tool performs well when its output is measured against hand-crafted OCL_h models, but more importantly, we show how it has helped uncover insecurities in PDDL encodings.

1 Introduction

Despite many years of research into AI Planning and Scheduling, knowledge engineering for applications of AI Planning technology is very much in its infancy. Recent successful AI planning applications [12, 16, 2] have nonetheless highlighted the problems facing knowledge engineering in planning. Questions include how to choose appropriate planner technology for a given application, and how to encode knowledge into domain models for use with planning algorithms. The engineering of knowledge-based planners has resulted in a set of workshops and initiatives, including [3, 14].

Currently in the AI planning community an accepted syntax for exchange of models is PDDL, a *planning domain definition language*. PDDL is a convention for representations of actions, and many established planners can be obtained via the internet with a set of domains encoded in this syntax. PDDL emerged from the need to construct a common language for the biannual AIPS competitions (for details of PDDL and domain examples consult reference [4]). Language conventions such as PDDL help the research community to some extent in the problems of exchanging research information, and in the independent validation of research results.

Domain definition languages such as PDDL, however, are not designed with the same criteria in mind as a *domain modelling language*. The latter would be associated with a

domain building methodology, be structured to allow the expeditious capture of knowledge, and have the benefit of a tools environment for knowledge engineering. OCL_h is a family of fairly simple planning-oriented domain modelling languages stemming from reference [11]. The benefit in using OCL_h is seen as twofold: to improve the planning knowledge acquisition and validation process; and to improve and clarify the plan generation process in planning systems. A range of planners have been implemented for use with OCL_h [17, 9, 7], and the language is being used as a prototype for a collaborative UK project to create a knowledge engineering platform for planning [15]. OCL_h is structured to allow the capture of object and state-centred knowledge, as well as action-centred knowledge [1], and it is encased in a tools environment.

In this paper we discuss the issues raised in the construction of one of the tools in OCL_h's environment: a translator, to help import models written in PDDL into the OCL_h environment. The translation is feasible because PDDL and OCL_h share similar underlying assumptions about worlds - they are assumed closed, actions are deterministic and instantaneous. A major difference, however, is that whereas PDDL action specifications are based on default persistence, this assumption is limited in OCL_h. The paper starts by briefly describing the languages PDDL and OCL_h and an association between the two which forms the basis for the translation. This highlights some of the differences between the two languages. We then outline the translation algorithm, and the results in applying it to example domain models. The tool's output is used to both make comparisons with hand-crafted models, and to identify omissions and insecurities in the PDDL encodings.

2 The Planning Domain Definition Language

PDDL was created by the AIPS-98 Competition Committee to enable competitors to have a common language for defining domains, and to aid the development of a set of problems written in PDDL on which the different planners could be tested [4]. PDDL provides a wide range of syntactic features, although not all planners are able to utilise all of them. Planners can just deal with particular subsets of the features that the language offers by declaring those language features required when the domain is defined. Here we only mention those features relevant to the paper.

PDDL's basic level of representation is the literal, and a model's central element is a set of operator schemae representing generalised domain actions (very much in the style of 'classical planning' literature with its roots in STRIPS [5]). Each operator is defined with a precondition and effect, where the semantics are interpreted under the STRIPS assumptions. Below are two examples of simple PDDL operator definitions which use typed parameters. They belong to an encoding of an example domain called the Tyre World which was taken from the distribution examples associated with reference [4]. A planner using the Tyre World should be able to output sequences of ground operators to solve goals involving changing a flat tyre. We will use this domain as a "running example".

```
(:action loosen
   :parameters (?n - nut ?h - hub)
   :precondition (and (have wrench)(tight ?n ?h)(on-ground ?h))
   :effect (and (loose ?n ?h) (not (tight?n ?h))))
```

[1] traditionally, models of planning domains were equated with a set of action specifications, and were therefore only 'action-centred'

```
(:action fetch
      :parameters (?x - (either tool wheel) ?c - container)
      :precondition (and  (in ?x ?c) (open ?c))
      :effect (and (have ?x)
                   (not (in ?x ?c))))
```

The *loosen* operator models the action of undoing (but not removing) the nuts that fasten a wheel onto a hub. The *fetch* operator models the action of removing a tool or a wheel from a container in which it was stored (such as a car's trunk).

Problems for a planner to solve are posed as an initial state (a set of ground literals) and a goal condition. Although the current PDDL version includes many other features (hierarchically-defined operators, domain axioms, safety constraints, quantification over parameter domains etc) the majority of the planners competing in the AIPS-98 competition input the simple form of PDDL similar to that described above.

3 The Object-Centred Language OCL_h

OCL_h was designed to be a kind of 'lifted' STRIPS-language, aimed to keep the generality of classical planning but to incorporate a model-building method and be structured to help the validation and operationalisation of domain models. For more information on the OCL_h family and its development method, examples and tools, consult references [14, 8, 11]. An OCL_h world is populated with dynamic/static objects grouped into *sorts*[2]. Each dynamic object exists in one of a well defined set of states (called 'substates'), where these substates are characterised by predicates. On this view the application of an operator will result in some of the objects in the domain moving from one substate to another. In addition to describing the actions in the problem domain, OCL_h provides information on the objects, their sort hierarchy and the permissible states that the objects may be in. Relations and propositions are not fully independent entities – rather they now belong to collections that can be manipulated as a whole. So instead of dealing with literals planning algorithms reason with objects. Similarly to a typed PDDL specification, the objects and the sorts they belong to are predefined, as is the sort of each argument of each predicate in the OCL_h model.

An object description in a planning world is specified by a tuple (s, i, ss), where s is a sort identifier, i is an object identifier of sort s, and ss is its *substate*. A substate is a set of predicates which all *describe* i. For example, $(nut, nut0, [loose(nut0, hub1)])$ is an object description meaning that $nut0$ of sort *nut* is loosely done up on hub1. Or again, $(container, trunk1, [closed(trunk1), locked(trunk1)])$ is an object description meaning that container trunk1 is closed and locked. Only a restricted set of predicates are allowed to describe an object and appear in its substate. Substates operate under a *closed world* assumption local to this restricted set - thus in the last example, the predicate $open(trunk1)$ is false because (a) it is used to describe objects of this sort (b) it does not appear in the substate.

The domain modeller defines the predicates used to describe objects, and the form of each substate, using **substate class definitions**. The predicate expressions in such definitions are constructed to form a complete, disjoint covering of the space of substates for objects of

[2] we use the name 'sorts' rather than 'object classes' to emphasis that OCL_h is an abstract object-centred modelling language - in contrast to an OO implementation language

each sort. When fully ground, an expression from a substate class definition forms a legal substate. For example, the substate class definitions for the sorts container, nut and hub are[3]:

```
substate_classes(container,C,[[closed(C)], [open(C)], [closed(C),locked(C)] ])
substate_classes(nut,N,[[loose(N,H)],[tight(N,H)],[off_hub(N)]])
substate_classes(hub,H,[[on_ground(H), fastened(H)], [jacked_up(H,J), fastened(H)],
       [free(H),jacked_up(H,J),unfastened(H)] , [unfastened(H),jacked_up(H,J)] ])
```

The first example means that objects of sort container can be either closed, not open and not locked, *or* open, not closed and not locked, *or* closed, locked and not open (*or* here is exclusive). Thus, in OCL_h *negation* is implicit: if it is the case that $\neg$ $open(trunk1)$, then this means that $trunk1$ must be in one of two substates, its object description being $(container, trunk1, [closed(trunk1)])$ or $(container, trunk1, [closed(trunk1), locked(trunk1)])$. One can see that the examples of object descriptions given above contain valid substates according to this definition.

A domain model is built up in OCL_h by creating the operator set at the same time as creating the substate class definitions. We define:
– an **object expression** to be a tuple (s, i, se) such that the expression part se is a generalisation of one or more substates (se is normally a set of predicates containing variables).
– an **object transition** to be an expression of the form $(s, i, se \Rightarrow ssc)$ where i is an object identifier or a variable of sort s, (s, i, se) forms a valid object expression, and ssc is taken from one of the substate class definitions. Thus when ssc is ground it will always be a valid substate.

An action in a domain is represented by operator schema O with an identifier $O.id$, a prevail condition $O.prev$, and a list of transitions. Each expression in $O.prev$ must be true before execution of O, and will remain true throughout operator execution.

Two OCL_h operators hand-crafted to (loosely) correspond to the PDDL operators above are as follows:

```
operator(loosen(N,H,W),
     [ (wrench,W,[have(W)]), (hub,H,[on_ground(H),fastened(H)]) ],
     [(nut,N,[tight(N,H)]=>[loose(N,H)])] )
operator(fetch(T,C),
     [(container,C,[open(C)])],
     [(tool_or_wheel,T,[in(T,C)]=>[have(T)])])
```

As with PDDL, OCL_h has many other features such as conditional operators, hierarchical operators, atomic and general invariants, but due to lack of space we refer the reader to the literature for these details.

4 Lifting PDDL To OCL_h

4.1 The general framework

We base the translation on two main assumptions: (1) the input to the translator will be any model written in the subset of PDDL that includes STRIPS-like operators with literals having typed arguments. The feasibility of using other variants of PDDL as input is briefly discussed in section 6 below. (2) the translation should keep, as far as possible, the names and

[3]whereas in PDDL we write a variable as an identifier beginning with '?', in OCL_h variables are identifiers with leading capitals

structure of the input model. This leads us to the following general framework for translation:

PDDL parameter type *name* $\Rightarrow$ OCL_h sort *name*
PDDL predicate $\Rightarrow$ OCL_h predicate
PDDL operator *name* $\Rightarrow$ OCL_h operator *name*

The first association preserves the type hierarchy and translates it to an equivalent OCL_h sort hierarchy. The consequential allocation of predicates to sorts, however, turns out to be the fundamental problem faced in extracting OCL_h information from PDDL. The requirement to be able to identify all legal states of each primitive sort within the domain entails the identification of a complete set of descriptions that can characterise any object of the sort at any instance in time. Once this is done, re-writing the PDDL operators by extracting the *object transitions* and the *prevail clauses* from the raw STRIPS operator is relatively straightforward.

4.2 Inducing Substate Classes

Steps in the OCL_h method that are used to derive substate class definitions are as follows:

1. Identify the sorts that are dynamic and those that are static
2. For each dynamic sort, identify those predicates that are to be included in defining its substate classes
3. For each dynamic sort, define its substate classes

For step 1, a sufficient condition for a sort s to be dynamic is that PDDL type s is described by a property which can be changed by a PDDL operator. Those types that have no changeable properties, but are referred to within a changing relation may or may not be mapped to a dynamic sort – this choice will become clear after our discussion of step 2.

In step 2, a difficult issue arises in that given a predicate $p(s1, s2, .., sn)$, what subset of the OCL_h sorts $s1, s2, .., sn$ should it be associated with, to describe that sorts' substates classes? In the method associated with OCL_h it is proposed that normally each predicate describe a single sort (although if the sort were not primitive the predicate would be used in distinct primitive sorts). To illustrate this problem consider the PDDL predicate *in*, with two arguments of type *tool* and *container* respectively. Both types are mapped over to OCL_h dynamic sorts, and the question arises: should the predicate "in" be used to describe the state of an object of sort *tool*, the state of an object of sort *container*, or both? Though from a logical point of view there is no more reason to say that the predicate *in* characterises a *tool* than there is to say it characterises a *container* there are strong pragmatic reasons to classify the predicate as belonging to only one of the objects referenced. If we allow predicates to describe all its sorts' states then there is a clear redundancy in our representation, in that we record the same information twice. More serious than this, allowing a relational predicate to characterise all referenced sorts introduces the *frame problem* in a particularly acute manner. Recall that the right hand sides of OCL_h transitions must fully characterise the resulting substate of the dynamic object participating in the transition, without default persistence but with a closed world assumption local to the predicates describing that sort. Then to record the possible substates of the *container* we would have to consider the possible combinations of the container being open, closed and locked along with all possible combinations of objects

such as the tools and wheels being either in or not in the container; this would lead to a proliferation of object transitions and operators.

The discussion above shows that it is not practicable to let a predicate be used in the substate descriptions of objects in every one of its argument sorts. Our solution to this frame problem is to try to follow the intuition in building an OCL_h model manually: let the algorithm choose *one* single sort. This distinguished sort is said to *own* the predicate. Though from a logical point of view this may seem arbitrary it coincides with intuition in the sense that we would not naturally think of the action of opening the trunk as having a different result depending on the trunk's contents. In English an action verb is typically thought of as characterising the subject of the sentence rather than the object. In this spirit we say the the predicate *in(wrench,trunk)* describes the state of the *wrench* but not that of the *trunk*.

Given that we will only allow a predicate to characterise a single sort the choice of sort could be made in a number of competing ways. We could try to allocate predicates to sorts in a way to try and minimise the frame problem or to minimise the number of sorts that change state in the actions concerned, or we could simply allocate them to the first mentioned object in each predicate. Up to now our experiments have shown the third strategy gives satisfactory results when the auto-generated OCL_h model has been compared to a hand crafted version.

Returning to step 1, this analysis determines the split of static and dynamic sorts: if some dynamic predicate has the property that its first argument can contain object identifiers of sort s, then s is a dynamic sort; otherwise, s will not be described by any dynamic predicates and hence will be static.

4.3 Dealing with Negation

Defining substate classes (i.e. step 3) when manually creating an OCL_h model involves finding a set of adequate substate class definitions. The aim is that for any expression in a substate class definition, any instantiantion will be a valid substate, and any valid description of an object is an instantiation of some expression in a substate class definition.

These classes can only be *induced* from the PDDL operators by examining the operator's effects, and, to some extent, using problem examples. For example, from the opening and closing actions, we arrive at the following sketch of the substate classes of container:

¬ open(container),locked(container)
¬ open(container),¬ locked(container)
open(container),¬ locked(container)

The fourth possibility *open(container),locked(container)* is not achievable using the PDDL operator set.

Negation in OCL_h is not represented explicitly, because of the local closed world assumption used in substates. It may be the case, however, that a negative form (or opposite) is required. We deal with this by potentially creating for each predicate a negative form, identified by prepending the predicate with *not_*. Though we start with the availability of all such possible negations, not all are used in the final translation.

5 The Translation

5.1 Description of the Algorithm

To simplify the expression of the algorithm we use abbreviation and the dot notation to access elements of PDDL or OCL_h Specifications. For example PDDL.actions refers to the actions of the specification, OCL_h.sscd[o.sort], refers to the OCL_h *substate class definition* for the *sort* of object *o* (page 4) and ot.rhs without further prefix refers to the right hand side predicates of an object transition of an OCL_h action. In general we will also equate conjunctions with sets.

algorithm toOCL-First-Pass
In Pt : PDDL-types, Pp : PDDL-predicates, Pa : PDDL-actions
Out OCL : OCL_h-domain-specification

1. OCL.types := PDDL.types
2. OCL.predicates := PDDL.predicates
3. ∀ a ∈ PDDL.actions
4. oa := new OCL-action
5. ∀o ∈ controlling_objects(a.effect)
6. ot := new OCL-state-change-clause
7. ot.rhs = owned_predicates(o,a.effect)
8. ot.lhs = owned_predicates(o,a.precondition)
9. ot.rhs := ot.rhs ∪ {p : ot.lhs | ¬ (p ∈ ot.rhs) ∧ ¬ (¬ p ∈ ot.rhs)}
10. oa.nec := oa.nec ∪ {ot}
11. OCL.sscd[o.sort] := OCL.sscd[o.sort] ∪ {ot.rhs}
12. end_for
13. ∀o ∈ owning_objects(a.precondition) − owning_objects(a.effect)
14. se := new OCL-state-expression
15. se.exp := owned_predicates(o.precondition)
16. oa.prev := oa.prev ∪ {se}
17. end_for
18. end_for
19. end.

Figure 1: First Pass of the PDDL to OCL_h Translation Algorithm

The first pass of the translation algorithm deals with the initial identification of object sorts with their PDDL counterparts the argument types. Candidate predicates are similarly identified at this stage (lines 1,2 in Figure 1) Negations are recorded at this stage for predicates if they are used in an action and are not generated automatically for all predicates.

The main part of the algorithm begins with the processing of the PDDL actions (lines 3 - 19). For each PDDL action we create an empty OCL_h action (line 4) and identify the variables and types for each owning object (i.e first argument of a predicate) in the actions *effect* section, and create a *object transition* (line 6). The right hand side of the transition clause is composed of the conjunction of predicates owned by that object in the PDDL action's *effect* (line 7). The lefthand side is composed of the owned predicates in the actions *precondition* (line 8). For the PDDL *fetch* action (page 2) there is only a single owning object variable `?x` in the effect clause therefore we produce the transition

```
(tool_wheel,X,[in(X,C)] => [have(X),not_in(X,C)])
```

In cases where a predicate of the owning object appears in the precondition of an operator

but its negation does not appear in the effect then we reason that the predicate must persist and accordingly we add that predicate to the right hand side of the transition (line 9) We now check the recorded *substate class definition* set for the owning object sort and add if not already present the right-hand side of the transition as a new candidate substate (line 11). In the fetch example we add the state `[have(X),not_in(X,C)]` to the substate list of the `tool - wheel` sort. The remaining processing of the action is to identify owning objects referenced in the actions precondition but not the effect. For each such object variable we create a substate expression clause from the conjunction of owned predicates to form part of the resulting OCL_h operator's *Prevail* section(13-17), which for the *fetch* operator produces the expression:

```
(container,C,[open(C)])
```

To complete the first pass of the algorithm the domain problems given in the PDDL specification are translated. The strategy is similar to that adopted for operators but with the problem initial state forming the basis for identifying more candidate object substate class expressions (this part of the algorithm is not shown in the figure).

5.2 Translation Results

The first pass translation results are encouraging in that they are close to those produced by hand translation from the same PDDL source. See samples for the Tyre World and the "Gripper World" in the resources section of the web site in reference [15]. Of the thirteen actions in the Tyre World two of the actions contained anomalies flagged up by the translation. Eight of the translated actions contained unnecessary, though correct, negations on their right hand sides and two actions had incomplete object transitions.

The translation at this stage not only forms a basis for hand completion but also has the power to flag up potential problems and insecurities with the PDDL domain specification. The most interesting of the anomalies uncovered in the Tyre World domain occurs with the *jack-down* action which is translated as follows:

```
  (:action jack-down
      :parameters (?h - hub)
      :precondition (not (on-ground ?h))
      :effect (and (on-ground ?h) (have jack)))
operator(jack_down(H),
  [],
  [ (hub,H,[not_on_ground(H)]=>[on_ground(H)]),
    (tool,jack,[]=>[have(jack)])] )
```

The transition for the jack indicates that it may be in any state prior to being possessed as a result of jacking down the wheel. This is not adequate as the mechanic only possesses the jack after execution of the action because it was used to jack-up the wheel in the first place. The PDDL formulation works (operationally) because in the domain there is no alternative way of getting the wheel off the ground (although we might have an alternative jack-up action, such as use a block and tackle).

A second anomoly which arises with the encoding of the *jack_down* action is that we treat `[on_ground(H)]` as a complete substate of the *hub*. From the auto-generated substate class definition for the *hub* we see that either the predicate *fastened(H) or unfastened(H)* must also apply to the the hub and this raises the following question: should it not be the case that we

should make it a precondition of the action that the hub has the wheel fastened to it prior to jacking down the hub? In effect, in OCL_h terms the transition should be

```
(hub,H,[not_on_ground(H),fastened(H)]=>[on_ground(H),fastened(H)]),
```

5.3 Work Required to Complete the Translation

The primary issues to be addressed in a second pass of the translation algorithm, still to be fully implemented are as follows:

Incomplete Object States

As OCL_h requires the right hand side of a transition to completely characterise the resulting state we must inspect each such clause constructed so far to determine if there are missing predicates that in the PDDL representation would simply persist. An extension to the algorithm could identify such omisions after the first pass translation. Consider two candidate substate class expressions x and y. If x contained a subset of the predicates of y, and the negation of the predicates in $y - -x$ cannot be inferred, then it would be reasonable to conclude that x was an incomplete version of y.

Implicit agents

A problem with agents of actions being implicit in PDDL domain specifications arises when translating to OCL_h. The problem is amply illustrated by the following first pass translation of the *move* rule from the Gripper domain where we have a robot that can move from a named location to another named location.

```
operator(move(TO, FROM),
  [],
  [ (room,TO,[]=>[at_robby(TO)]),
    (room,FROM,[at_robby(FROM)]=>[not_at_robby(FROM)])] )
```

The rooms *TO* and *FROM* are being classed as dynamic objects subject to change, but we would more naturally want to say that it is the robot that has changed. In OCL_h parlance locations should be treated as static. To solve this problem we need to recode the *at_robby* predicate and introduce an agent i.e. the robot. *at(Agent,Location)*. If the agent has, as in this case been implicitly encoded into the predicate then there will only be one such agent and we can effectively introduce a new constant *agent0* and a new type *Agent*.

Poorly Chosen Argument Ordering

Despite the decision to select the first place argument to denote the *owning* object of a predicate there are situations where this should be rejected. The problem may arise if the first argument of a relational predicate describes a property of an object referenced in a later position. For example if the wheel status in the Tyre World was encoded in a predicate *status(condition,wheel)* the translation of the *inflate* action would contain the following body:

```
(wheel,Wheel,[have(Wheel)])
(condition,intact,[status(intact,Wheel)] => [])
(condition,inflated,[] => [status(inflated,Wheel)])
```

The symmetry of the empty right and left hand clauses is the indicator that we have a malcoded predicate which should be reordered. The result will eliminate the prevail clause and result in the transition

```
(wheel,Wheel,[have(Wheel),condition(Wheel,intact)]=>
                [have(Wheel),condition(Wheel,inflated)])
```

6 Discussion

It has been acknowledged since the modern inception of AI that the representation of knowledge has a critical bearing on the performance of a problem solver. In planning especially, there have been relatively few insights or research projects in this area - instead the planning literature has tended to concentrate on the efficiency issues of planners, or the adequacy of expression of their domain model languages. We see our ongoing work on the translation from PDDL to OCL_h as promoting the debate on the relative merits of planning domain encodings, and, in time, the matching up of appropriate planner technology to application domain.

Working with a domain modelling language such as OCL_h gives opportunities for higher level domain validation with rich tool support that eases domain modelling. Our 'first pass' translator from PDDL to OCL_h has already given us access to a rich source of research examples written in PDDL to test OCL_h tools. More importantly, it has highlighted those issues in representation such as use of negation, completeness/security of models, and construction of object hierarchies that are fundamental to the creation of a planning domain model.

6.1 Related Work

The basic strategies of re-casting domain knowledge from a predicate base into an object-centred base are not new and have been discussed in the literature for some period. An early general discussion of the issues is to be found in reference [13]. OCL_h contrasts with previous domain modelling languages for planning such as [18, 1] in its simplicity and clarity. On the other hand, OCL_h is far less sophisticated (for a comparison of O-Plan's TF and OCL_h see reference [10]).

Fox and Long in reference [6] show that the limitation of requiring arguments to be typed in the PDDL specification is not fundamental to the translation. They demonstrate that type information can be extracted from a set of PDDL operator schema only. Fox and Long's TIM uses the operator schemae to analyse the domain and produce types such that objects belonging to them are identical up to naming. It therefore appears to produce a type structure more appropriate to OCL_h. Our future work will involve merging the translation algorithm with the TIM engine into a tool that should produce a more adequate OCL_h model.

References

[1] B. Drabble A. Tate and J. Dalton. O-Plan: a Knowledged-Based Planner and its Application to Logistics. AIAI, University of Edinburgh, 1996.

[2] A. Tate (editor). *Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative.* IOS Press, 1996.

[3] Benjamins, Nunes de Barros, Shahar, Tate and Valente (eds). *Workshop on Knowledge Engineering and Acquisition for Planning: Bridging Theory and Practice.* Procceedings of AIPS, 1998.

[4] AIPS-98 Planning Competition Committee. PDDL - The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[5] R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2, 1971.

[6] M. Fox and D. Long. The Automatic Inference of State Invariants in TIM. *JAIR vol. 9*, pages 367–421, 1997.

[7] D. E. Kitchin. *Object-Centred Generative Planning.* PhD thesis, School of Computing and Mathematics, University of Huddersfield, forthcoming,2000.

[8] D. Liu and T.L.McCluskey. The OCL Language Manual, Version 1.2. Technical report, Department of Computing Science, University of Huddersfield , 2000.

[9] T. L. McCluskey. Object Transition Sequences: A New Form of Abstraction for HTN Planners. In *To appear in Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (aips-2000)* , 2000.

[10] T. L. McCluskey, P. Jarvis, and D. E. Kitchin. OCL_h: a sound and supportive planning domain modelling language. Technical report, Department of Computer Science, The University of Huddersfield, 1999.

[11] T. L. McCluskey and J. M. Porteous. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. *Artificial Intelligence*, 95:1–65, 1997.

[12] B. Pell N. Muscettola, P. P. Nayak and B. C. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, 1998.

[13] N. J. Nilsson. *Principles of Artificial Intelligence.* Springer-Verlag, 1982.

[14] PLANET. *First Workshop of the PLANET Knowledge Acquistion Technical Coordination Unit.* http://helios.hud.ac.uk/planet, 1999.

[15] Planform. *An Open Environment for Building Planners* . http://helios.hud.ac.uk/planform, 2000.

[16] S.Chien (editor). *Proceedings, 1st NASA Workshop on Planning and Scheduling in Space Applications.* NASA, Oxnard CA, 1997.

[17] R. M. Simpson and T. L. McCluskey. A Object-Graph Planning Algorithm. In *Proceedings of the 18th Workshop of the UK Planning and Scheduling SIG*, 1999.

[18] D. Wilkins. Using the SIPE-2 Planning System: A Manual for SIPE-2, Version5.0. SRI International, Artificial Intelligence Center, 1999.