An Evaluation of Opmaker2

T.L.McCluskey, S.N.Cresswell, N.E.Richardson, R.M.Simpson and M.M.West
School of Computing and Engineering
The University of Huddersfield, Huddersfield HD1 3DH, UK

Abstract

Opmaker2 is a knowledge acquisition and formulation tool, which inputs a domain ontology and a training sequence, and outputs a set of PDDL operator schema. This paper evaluates Opmaker2 (a) by comparing it against GIPO's object life history editor tool (b) by analysis of its method and its robustness to choice of training sequence.

Introduction

The increasing number of applications embodying a planning function, and community events such as the international competitions in knowledge engineering for planning and scheduling ICKEPS (Bartak and McCluskey 2006), have led to a growing interest in the field of knowledge engineering for automated planning. Evaluation of knowledge engineering tools is problematic for many reasons, not least because the success of a tool often depends on the expertise of the user. The problems encountered in staging ICKEPS mirrors the problems of evaluating research contributions within the KE area. In this paper we attempt to analyse and evaluate a recently constructed tool for use in knowledge formulation (Opmaker2), by experiment and comparison with a rival tool.

Opmaker2 inputs a domain ontology and a solution to a problem, and automatically constructs operator schema for each distinct step in the solution. This is used instead of handcrafting operator definitions, relieving the domain encoder the task of constructing, debugging and maintaining them. In this paper we give an overview of Opmaker2, and we evaluate the use of the system by its application to the acquisition and maintenance of a new domain: the operator schema needed for automation of a role playing game (an RPG). We compare Opmaker2 with GIPO's OLHE (Simpson, Kitchin, and McCluskey 2007), by applying it to the same domain. The OLHE is a tool which in large part led to GIPO's success in ICKEPS-2005. We analyse Opmaker2's method, and explore the conditions under which the method will produce a unique set of operator schema.

An RPG Domain

The motivation behind the construction of a "role playing game" domain model was to show the potential of AI Planning to computer games students, and to illustrate a possible role for AI Planning in making games more believable. We want to be able to show how non-player characters (NPCs) can behave intelligently in a game scenario, by constructing and following rational plans. The game was designed as follows: there is one overall 'database' which can completely describe any state of the game. There are two opposing forces - one which controls a set of knights (NPCs) that attempt to bring about a set of goals (eg acquire treasure, kill monsters). There is another force which controls the monsters, with goals such as blocking doors, disabling knights etc. Both forces can use deliberative planning to try to achieve goals. The idea is that both forces execute their plans and replan when their plans are no longer executable. The player character could, for example, play the role of a 'wizard' that observed one force's plans and attempted to help achieve them.

A step towards this goal was to construct a planning domain model with operator schema representing the capabilities of the characters, and the effects of their actions. Two sets of operator schema, representing the two views of the overall domain model, have to be generated for this, one for each view of the opposing sides. These views would constitute two distinct operator sets, whereas the object structures (classes, legal states) would be common to both and constitute the database.

A snapshot (or state) of the world is illus-

trated in Figure 1. From the point of view of the force controlling the knights:

- knights can move between rooms and passages. Movement will take place via propositional steps such as "move-into-room" and "move-to-passage";
- passages can be blocked by monsters or locked doors;
- locked doors need to be opened by the correct key;
- knights can acquire keys if they are near them and exchange some treasure for them;
- monsters that are sleeping can be killed by a knight if it is near them and has a weapon (however the weapon is then spent);
- monsters that are awake are similar to sleeping monsters except the knight needs to pick up protection first (a shield or a spell) before fighting them;
- treasure can be picked up by a knight if it is in the same room as the treasure.

For example, consider the state in Figure 1. Assume that a goal to be achieved is for Aragorn to acquire the ring. Then the planner would have to generate a plan for him to move through rooms and passages to acquire a key to unlock one of the pasages leading to r2 (the only route to the ring in r1); and to acquire some treasure so that he can trade it for protection, so that he will be able to overcome the orc guarding room r1, when he eventually arrived there.

Opmaker2 Overview

The Opmaker2 algorithm was detailed in a paper in Plansig 2007 (McCluskey et al. 2007), and in a recent PhD thesis (Richardson 2008). It followed on from the original Opmaker idea (McCluskey, Richardson, and Simpson 2002). Its aims are similar to systems such as ARMS (Wu, Yang, and Jiang 2005) in that it supports the automated acquisition of a set of operator schema that can be used as input to an automated planning engine. It differs from ARMS in its method: while ARMS inputs many training examples and constraints, and is based on a propositional-style (PDDL) representation, Opmaker2 requires only one example of each operator schema that it learns, but also requires an ontology of objects and classes (called a partial domain model) as input. Opmaker's ontology is based on OCL (Liu and McCluskey 2000), where a state is defined as a set of object identifiers mapped to one of a set of well defined object-state values. After induction, Opmaker2's output operator schema can be translated to PDDL, as the description of states associated with object identifiers can be regarded as propositions.

Opmaker2 learns by example, and the form of a training sequence(s) input to it is a sequence of step names followed by a list of parameters. Example 1 and 4 below are examples of such training sequences. Each member of the parameter list is a reference to some object in the domain. They are the components necessary and sufficient to describe what happens at each step (no other object reference plays a role in the named step). We assume the world is structured in that objects have their own states, and the set of possible states can be enumerated. Additionally, within a training sequence:

- the same object is referred to by the same object identifier throughout the training sequence;
- objects that are named but do not change state are flagged (in the examples a "@" is used)
- objects are typical of their class

As introduced above, Opmaker2 has access to a "partial domain model" (PDM), and the initial and final state of each object referred in the sequence. The PDM contains the classes of the objects in the example, and the set of abstract states (abstract states are defined according to Simpson et al's state machine view (Simpson, Kitchin, and McCluskey 2007)), that each object may be in during plan execution. A two step procedure is used to induce operator schema as reported in detail in (McCluskey et al. 2007):

1. From the initial state of the training example, attempt to find enough details of each intermediate state description after the execution of each step, sufficient to derive a deterministic specification of what happens at each step.
2. Based on the "typical" nature of objects in a class, generalise the specifications to form an operator schema.

The training sequence used needs to contain at least one example of each required operator schema. Whereas our previous work also emphasised the benefit of using training sequences to encapsulate heuristics, here we concentrate on evaluating the domain construction and maintenance aspects of the method.

An Empirical Evaluation of Opmaker2 using the RPG domain

Up to now, published evaluations of both Opmaker (McCluskey, Richardson, and Simpson 2002) and OpMaker2 (McCluskey et al. 2007) were based on using domains that had been

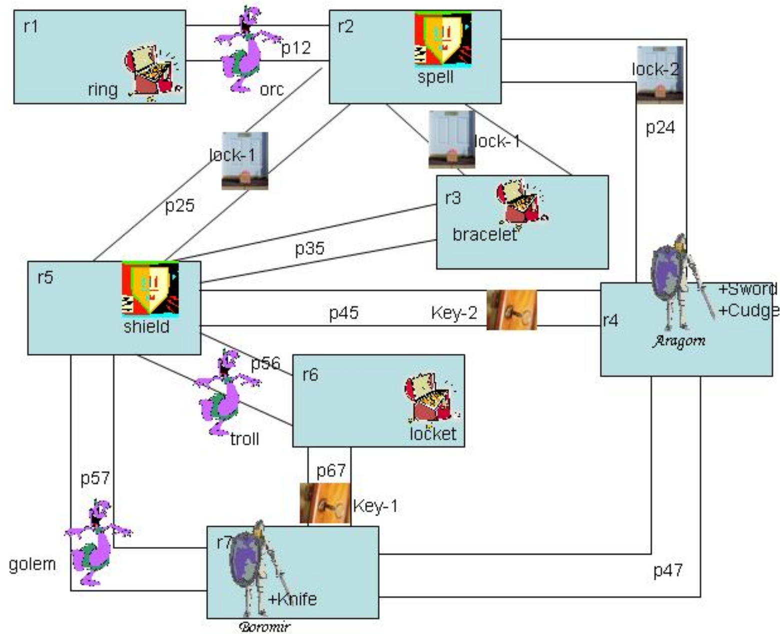

Figure 1: A state of the RPG

previously crafted. In this section we detail the initial construction of the RPG domain using Opmaker2. We then compare this with its construction using GIPO's OLHE, arguably the more powerful feature of GIPO version 3 (Simpson, Kitchin, and McCluskey 2007).

Comparative Criteria

We used Opmaker2 and GIPO's OLHE (Simpson, Kitchin, and McCluskey 2007) tools to acquire the RPG domain model from the point of view of the *force controlling the knights*. We will compare them by investigating:

- the amount and nature of effort required to formulate the domain to the point that planning could be enabled (and plans generated such as the one in Example 2)

- the amount of effort required to maintain the domain when a significant change is required.

With respect to the second point, we assume that after the initial version of the domain is created and operational, we would like to change the domain in the following respects.

- as a knight moves into a room it uses a unit of strength.
- rooms may contain food equivalent to a unit of strength which can be picked up from rooms
- knights start with a number of units of strength

Given the nature of the experiment, we cannot expect to use much more than an ordinal metric regarding the relative effort used in each, as the amount of effort is clearly dependent on the user.

Domain Acquisition using Opmaker2

The first step was to create a partial domain model using basic GIPO editors. This consisted of object names, their classes, the set of abstract object states for each class, and constraints specifying room and passage connectivity. To assist this process we used a sketch of a typical state (see Figure 1). This helped the user to decide on the kinds of states each object could occupy.

GIPO's basic validation tools helped to develop and refine the PDM, until we could use its random task generator to create random tasks (consisting of initial states and goal states). We then picked an initial state, and started a training sequence from this. As the training sequence is grown to include a new operator instance, at each step we create a new goal state for the affected objects, and run Opmaker2 to create a set of new operator schema. This was carried iteratively until reaching the 14 steps shown in Example 1 below. As mentioned above, objects annotated with "@" are deemed not to be affected by the step.

```
move-to-passage  @p47   knight  @r4
```

```
move-into-room  @p47   knight  @r7
pick-up-treasure  @knight  @r7  locket
move-to-passage  @p67   knight  @r7
exchange  locket  key1  @knight  @r7  @p67
move-into-room  @p67   knight  @r6
kill-sleeping-monster  @knight  sword  troll
                       p56  @r6
move-to-passage  @p56   knight  @r6
move-into-room  @p56   knight  @r5
get-protection  shield  @knight  @r5
fight-and-kill-monster  @knight  shield
                       knife  orc  p35  @r5
move-to-passage  @p35   knight  @r5
move-into-room  @p35   knight  @r3
unlock-passage  p23  @r3  @knight  @key1
```

Example 1: A training sequence for the RPG

Initial Domain Formulation: As Opmaker2 was run iteratively it generated new (deterministic) operators. A set of 8 operator schema was output by Opmaker2 after the full sequence in Example 1 was input (corresponding to the 8 distinct names in the sequence).

The generated operator set was tested with sample tasks and it was found that 4 operator schema were overgeneralised, as the connectivity constraint "contains(room,passage)" was omitted in each case, allowing operators to be executed without sufficient constraints (eg knights could kill monsters without being near them). After four constraint predicates were added, the domain functioned as expected under sustained testing. Example 2 below gives an example of a goal and solution (the initial state in illustrated in Figure 1), found using Hoffman's FF (Hoffmann 2000) in less than one second.

```
Goal: BOROMIR kills the TROLL and
ARAGORN acquires the RING
Solution found by FF:
 0: MOVE-TO-PASSAGE P67 BOROMIR R7
 1: MOVE-TO-PASSAGE P45 ARAGORN R4
 2: MOVE-INTO-ROOM P45 ARAGORN R4 R5
 3: GET-PROTECTION ARAGORN R5 SHIELD
 4: MOVE-INTO-ROOM P67 BOROMIR R7 R6
 5: KILL-SLEEPING-MONSTER BOROMIR
    R6 P56 KNIFE TROLL
 6: PICK-UP-TREASURE BOROMIR R6 LOCKET
 7: MOVE-TO-PASSAGE P67 BOROMIR R6
 8: MOVE-INTO-ROOM P67 BOROMIR R6 R7
 9: MOVE-TO-PASSAGE P67 BOROMIR R7
10: EXCHANGE BOROMIR R7 P67 LOCKET KEY1
11: MOVE-INTO-ROOM P67 BOROMIR R7 R6
12: MOVE-TO-PASSAGE P56 BOROMIR R6
13: MOVE-INTO-ROOM P56 BOROMIR R6 R5
14: UNLOCK-PASSAGE BOROMIR R5 P25 KEY1
15: MOVE-TO-PASSAGE P25 ARAGORN R5
16: MOVE-INTO-ROOM P25 ARAGORN R5 R2
17: FIGHT-AND-KILL-MONSTER ARAGORN
    R2 P12 SHIELD SWORD ORC
18: MOVE-TO-PASSAGE P12 ARAGORN R2
19: MOVE-INTO-ROOM P12 ARAGORN R2 R1
20: PICK-UP-TREASURE ARAGORN R1 RING
```

Example 2: An example goal and plan generated by FF for the RPG

The effort to create the domain model from conception to a functioning domain model outputting solutions as in Example 2, was equal to less than one day's effort. Half of this time was devoted to developing the PDM, while the other half was used in developing the training example and running Opmaker2.

Domain Maintenance: Following the new requirement above, a new class was created to represent "units of strength" in the PDM. Three abstract state classes were designed as follows: in use by a knight, stored in a room in the form of food, or spent. The initial state and goal states for the training sequence were updated to include the new objects. The sequence itself was changed as follows (where s1,s2,s3,s4 are units of strength):

```
move-to-passage  @p47   knight  @r4
move-into-room  @p47   knight  @r7 s1
...
move-into-room  @p67   knight  @r6 s2
acquire-food @knight @r6 s4
...
move-into-room  @p56   knight  @r5 s3
...
move-into-room  @p35   knight  @r3 s4
...
```

Example 3: Changes in the training sequence

A new set of 9 operator schema was created, with the new operator "acquire-food" and an updated "move-into-room". As previously, the extra four constraints had to be added to the four over-generalised operators, before the full set could be used operationally. The change took under one hour.

Domain Acquisition using GIPO's OLHE

GIPO's Object Life History Editor is a tool to create domain models graphically (Simpson, Kitchin, and McCluskey 2007). The user uses a kind of state-machine metaphor to represent classes of objects, and records interactions between each class. The tool is extensible in that one can create and store machine primitives (eg bi-state, tri-state, portable, stack etc), and reuse these to build future models. On request GIPO can translate the diagram constructed from the OHLE into a formal (textual) domain description, pass it through various validation checks, use it with an internal planner or export the domain as PDDL to a third party planner.

Initial Domain Formulation: After the first RPG version was created using Opmaker2,

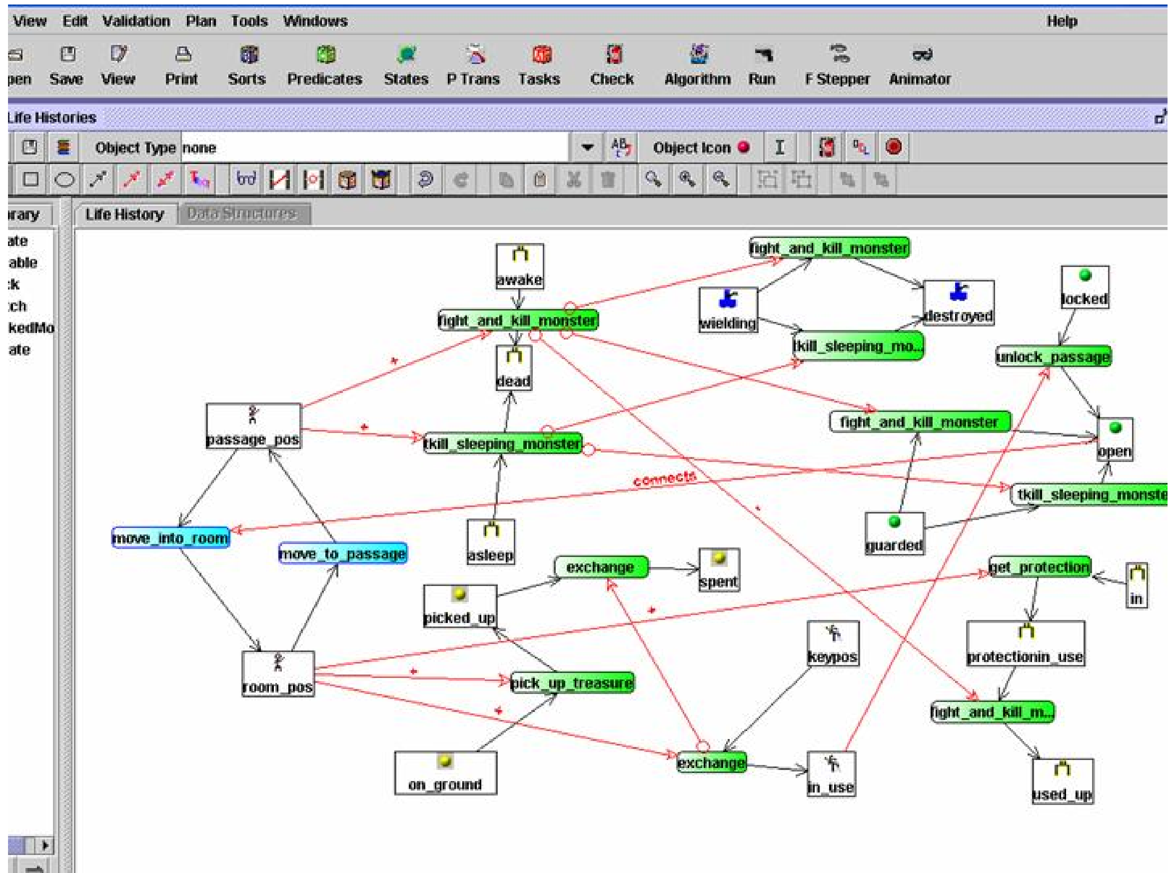

Figure 2: The Game captured within the OLHE

the OLHE was used to create a new version. This encoding took slightly more effort than with Opmakers, but still of the order of a day's effort, resulting eventually in the diagram shown in Figure 2 (the diagram is best viewed in colour as this is used to differentiate the various roles of arcs and nodes). The idea of the object states already developed in the PDM for Opmaler2 helped speed the choice from the library of machines from which to compose the domain model (4 tri-states and 3 bi-states). However, the process of domain construction is quite different from Opmaker2, as no partial domain model is explicitly constructed or no training examples are needed. Apart from editing and annotating the nodes and arcs shown in the figure, the user need only name objects for each class, and define any additional constraints required.

The exercise illuminated some interesting observations of the OLHE, in that several required features of the domain had to be encoded **after** the domain model was generated from the diagram:

- associations between two objects of distinct classes have to be produced by annotating a connection by a "+". However, this association might only occur at the initial state and hence the only way to specify it is manually. For example, the set of weapons held by knights are resourced from the initial state only;
- associations of >2 arity or adding multiple constraints regarding the same transition, have to be carried out after domain model generation (eg adding contains(roomX,passageZ) and contains(roomY,passageZ) to show that the same passage is contained in different rooms, is not possible unless the constraints are added as static constraints, rather that as properties of rooms or passages);
- letting some states have certain properties and some not is problematic. For example, we may want a protection to have a position property when it is doing nothing, but then this property does not exist after a knight has

picked it up. It is only possible to achieve this by artificially introducing null property values. For example to indicate that the position of a "ring" was no longer applicable it may be given the property position(ring,none).

The domain model resulting from the diagram shown therefore needed some textual changes along these lines, before becoming operational.

Domain Maintenance: Following the new requirement above, the original OLHE diagram was used and a new tri-state class was created to represent the unit of strength class. The arc connecting up this class with "move-into-room" was created, forming a transition for a unit of strength to be spent. The transition for acquiring food was connected with the precondition node that a knight had to be present.

This group of changes was relatively straightforward to construct. However, the main problem is that any change that has had to be made subsequently to the auto-generated domain model *needs to be redone*. Clearly, as the number of changes required *after* the OHLE-generation of the model grows, continued use of the OHLE graphics is less appealing. This is consistent with the observed behaviour of student use of GIPO. They enthusiastically use the OLHE as far as possible, but after a certain level of complexity they return to the use of GIPO's more "hands-on" tools. Once they have changed the model significantly, the OLHE diagram cannot be re-used.

Summary: To conclude, both methods supported the automated creation of operator schema for this domain successfully, so that the inevitable bugs introduced by hand-crafting operators did not appear. However, in both cases the generated models were incomplete. The OLHE appeared to be less useful after maintenance, as a number of hand-crafted changes to the domain tends to make the graphical model obsolete.

Analysis of Opmaker2

In this section we attempt to analyse why, and under what conditions, Opmaker2 works. One way of viewing the training problem is that the given PDM implicitly specifies a very large set of operator schema. What is not specified is which objects go through transitions together, which transitions are possible, and what are the constraints on each set of transitions. The training example is used to fill in these details.

The core of the induction problem identified in Opmaker2 is in deriving the details of intermediate states of objects involved in a training example. Once this is determined, the operator schema can be induced relatively easily. Normally training sequences involve several objects changing state. The object transitions are constrained by the known initial and goal states and whether or not there must be a state change. For example, an object that has 3 abstract states, starts at a different state than it ends, and is affected by 2 operators in sequence, has a unique path.

In general, the number of possible combinations of object transitions will grow combinatorially. Assume the number of abstract states of an object O^i is O^i_c, and assume the number of objects' states affected (changed) by an operator is N. Then an estimate of the number of possible operators at any step of the solution will be:

$(O^1_c - 1) * (O^2_c - 1) * ... * (O^n_c - 1)$

This assumes that objects are independent of each other, and none of the transitions are close to their initial or final state The number of potential paths (in the worst case) through the space of vectors of object values is thus the cross product of the formula above at each step in the training sequence, where each step represents a new operator. In general, however, where operators affect different groups of objects, the final state of an object might occur part way though a training sequence, if no operator after that changes that object. For example, "locket" in Example 1 reaches its final state after the fifth step.

As a concrete example, assume a propositional encoding of a tyre world, with a wheel being described as one of 4 states: wheel-in, have-wheel, wheel-on, wheel-fastened. The training sequence is "fetch-wheel, attach-wheel, fasten-wheel" The initial state contained fetch-in, and the goal state was wheel-fastened. In this case there are 7 possible paths through the object state space as follows:

```
wheel-in => have-wheel =>
            wheel-in => wheel-fastened
wheel-in => have-wheel =>
            wheel-on => wheel-fastened
wheel-in => wheel-on =>
            wheel-in => wheel-fastened
wheel-in => wheel-on =>
            have-wheel => wheel-fastened
wheel-in => wheel-fastened =>
            wheel-in   => wheel-fastened
wheel-in => wheel-fastened =>
            wheel-on => wheel-fastened
wheel-in => wheel-fastened =>
            have-wheel => wheel-fastened
```

In this case it would not be possible to induce a deterministic operator schema set from the training sequence. However, in general, objects are not independent of one another but re-

lated. For example, consider the following set of abstract states for a relational version of tyre-world (where wheel, boot and hub are object classes):

```
wheel in-a boot
wheel fastened-to-a hub
wheel hanging-on-a hub
wheel picked-up
```

Returning to the example in this new relational representation, assume the initial state contains "wheel1 in-a boot1", and the goal condition is "wheel1 fastened-to-a hub1". The training sequence would be

```
fetch-wheel wheel1 @boot1
attach-wheel wheel1 hub1
fasten-wheel wheel1 hub1
```

Given that the first step in this sequence does not refer to a hub, or two different boots, only the first of the four possible forms of transition below would be applicable:

```
wheel in-a boot => wheel picked-up
wheel in-a boot => wheel-fastened-to-a hub
wheel in-a boot => wheel hanging-on-a hub
wheel in-a boot => wheel in-a boot
         (-where the boots are different)
```

The next transition choice for attach-wheel:

```
wheel picked-up => wheel hanging-on-a hub
wheel picked-up => wheel in-a boot
wheel picked-up => wheel-fastened-to-a hub
wheel picked-up => wheel picked-up
        (-where the wheels are different)
```

again is determined as being the first of the possible transition forms. The second step in the training sequence contains a hub object and not a boot object, hence the second possibility is eliminated. The third option is eliminated by the constraint of the goal state, and the fourth is eliminated as the training step does not refer to two different wheels. Hence the presence of objects in the training sequence reduces the space of paths to one in this example. In general, where the set of abstract states for each class are distinguished by references to different object classes, Opmaker2 will obtain deterministic operator schema.

A further method to ensure Opmaker2 returns a unique set of operator schema is to add, within the PDM, a set of domain invariants which restrict the set of states. These invariants are used by Opmaker2 to provide constraints on the legal patterns of objects *from different classes*, when a set of operator schema are created from one instance in a training sequence, and the operators involved in the transition contain objects from more than one class. In this case the invariants are used to check that accumulated right hand sides of transitions in the candidate operators may form a consistent state. Whereas in the RPG model above no such invariants were needed, in tests reported with the tyre domain (McCluskey et al. 2007) eight invariants were required.

A systematic test for robustness

To investigate the robustness of Opmaker2's method with respect to training sequence choice and length, we decided to extend the testing reported in (McCluskey et al. 2007), and generate *random training sequences* within the tyre domain. This allowed us to explore the sensitivity of the method to choice and length of training sequence.

The tyre world PDM was input to GIPO, and a set of 25 random tasks were generated using GIPO's random task generator. The random task generator works by picking a state at random (using the abstract state descriptions in the PDM) for each of the N dynamic objects in the domain (dynamic objects are those that can have a changable state). These are all amalgamated into an initial state. Then it picks a random number M between 1 and N and generates M random goal conditions for the M distinct objects, again utilising each object's class's abstract state description. These are amalgamated into the goal of the task. For each of the tasks, we used GIPO's inbuilt planners and the hand-crafted domain model to find solutions. Tasks that could not be solved or had solutions that were less than 3 operators in length were ignored.

Results: The solutions to the remaining random tasks (10) were input to OpMaker2 as a set of training sequences. In 6 out of 10 cases, Opmaker2 produced a unique set of operators schema for each operator instance in the training set. In each case, the induced set of operator schema matched the hand-crafted set for one feature: there was an extra precondition "jack-in-use" placed on four of the operators. Although this precondition was implicit in the original domain model, it can be seen to tighten it (and arguably improve it).

Four of the 10 training sequences resulted in OpMaker2 outputting no operators. GIPO's planner using the original operators had found a solution, but opmaker2's invariants for this domain (see (McCluskey et al. 2007) for a list of the invariants) did not. We found that the set of invariants input to Opmaker2 were logically stronger than the hand crafted operators. For example, the invariants excluded the possibility of fastening up a hub without first attaching a

wheel, whereas the original set of operators allows this to happen.

This suggests that the task length or optimality of a training sequence does not have a bearing on whether a unique set of operator schema are induced. The example below is the solution to the full "change-wheel" flat tyre problem in Opmaker2 input ready form. It was formed from a (non-optimal) solution output from a standard planner using a hand crafted domain model. Inputting this into Opmaker2, 16 operator schema were output, identical to those produced by the random training sequences described above.

```
open_container boot
fetch_wheel @boot wheel1
fetch_jack @boot jack0
fetch_wrench @boot wrench0
close_container boot
loosen @wrench0 @hub0 nuts_1
jack_up hub0 jack0
undo @wrench0 hub0 @jack0 nuts_1
remove_wheel wheel2 hub0 @jack0
put_on_wheel wheel1 hub0 @jack0
do_up @wrench0 hub0 @jack0 nuts_1
jack_down hub0 jack0
tighten @wrench0 @hub0 nuts_1
open_container boot
putaway_wheel @boot wheel2
putaway_wrench @boot wrench0
putaway_jack @boot jack0
close_container boot
```

Example 4: A training sequence equivalent to the solution of the ''change-wheel" problem

These experiments add weight to the argument that OpMaker2's domain construction technique is not sensitive to the *order* or the $make - up$ of training examples. On the other hand, the utility of the heuristics acquired by OpMaker2 in the form of methods, are very much dependent on an expert choosing to supply "typical" problems and solutions, as argued previously (McCluskey et al. 2007).

Conclusions

Despite some fielded applications of planning, and two runs of the ICKEPS competition, there are few evaluations or critical analyses of tools for knowledge acquisition, formulation or engineering of planning domain models. Exceptions include the work of Grant (Grant 1996) and Wu (Wu, Yang, and Jiang 2005).

In this paper we evaluated the knowledge acquisition tool Opmaker2 against GIPO's OLHE using the formulation and maintenance of a new domain, the RPG. The generated domain models output from *both* tools had to undergo some "hand-crafting" before they were operational. This was less so with Opmaker2, as most problems arise from the underlying ontology: GIPO generates this, whereas Opmaker2 expects the user to create it.

We also analysed Opmaker2's method, and used random tasks to explore the robustness of the method to choice of training sequence. While the method formed the same operator schema regardless of training sequence, it also showed that the induction process could discover potential insecurities in the hand-crafted set.

For the future, we see the extension of KA tools to domains requiring more complex representations (eg hierarchical state decrptions, numerical attributes) as being desirable. Also, the extension of Opmaker2 to encompass incremental learning is desirable, as this may obviate the need to craft a strong set of invariants to include in the partial domian model.

References

Bartak, R., and McCluskey, T. L. 2006. The first competition on knowledge engineering for planning and scheduling. *AI Magazine*.

Grant, T. J. 1996. *Inductive Learning of Knowledge-Based Planning Operators*. Ph.D. Dissertation, de Rijksuniversiteit Limburg te Maastricht.

Hoffmann, J. 2000. A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm. In *Proceedings of the 14th Workshop on Planning and Configuration - New Results in Planning, Scheduling and Design*.

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.; Cresswell, S.; Richardson, N.; and West, M. 2007. Opmaker2: Efficient action schema acquisition. In *Proceedings of the 26th Workshop of the UK Planning and Scheduling Special Interest Group, Prague, Czech Republic, December 2007*.

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*.

Richardson, N. E. 2008. *An Operator Induction Tool Supporting Knowledge Engineering in Planning*. Ph.D. Dissertation, School of Computing and Engineering, University of Huddersfield, UK.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning Domain Definition Using GIPO. *The Knowledge Engineering Review* 22(1).

Wu, K.; Yang, Q.; and Jiang, Y. 2005. Arms: Action-relation modelling system for learning acquisition models. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning*.