B vs OCL: Comparing specification languages for Planning Domains

D. E. Kitchin, T. L. McCluskey and **M. M. West**
School of Computing and Engineering
The University of Huddersfield, Huddersfield HD1 3DH, UK
D.Kitchin,T.L.McCluskey,M.M.West@hud.ac.uk

Abstract

In this paper we examine the specification and validation of Artificial Intelligence Planning domain models using the B Abstract Machine Notation and its associated tool support. We compare this to the use of *OCL* (object-centred language) within its tool-supported environment, GIPO. We present encodings of two well-known AI planning domain models, the Blocks world and the Tyres world, with the aim of finding a correspondence between the B and the OCL languages. We also compare the tool-supported validation offered by their respective environments.

Introduction

As AI Planning and Scheduling systems become mature enough to be deployed in safety-related and safety critical systems, the reliability of the systems themselves, and the accuracy of the knowledge models that underlie the systems, need to be certified to a high level. Planning systems typically contain planning engines, plan execution architectures, plan generation heuristics and application domain models. In this paper we focus on techniques for the rigorous construction and validation of the application domain model. This typically contains a structural model of the objects and constraints in the planning world, and a model of the actions/events that affect objects in that world.

It is likely that any reasonably sized realistic domain model will continue to contain errors and inconsistencies for some time. A planner may manage to produce a solution despite the fact that the domain model is flawed. Alternatively no plan will be produced because of inconsistencies in the domain model. Whatever the case it is desirable to be able to validate the domain model *before* an attempt is made to generate a plan. One approach to this is to use model checking for validation, as in (Penix, Pecheur, & Havelund 1998), but this is limited by potential state space explosion. Another approach could be to assume **a priori** that the domain model will be incomplete as in the *SiN* algorithm (Munoz-Avila *et al.* 2001). SiN can generate plans given an incomplete domain theory by using cases to extend that domain theory, and can also reason with imperfect world-state information. This is a fruitful assumption in many ways, as philosophically no model can ever be 'proved' complete and correct. However, this approach neglects the issue of correctness -

the incomplete parts must still be validated and bugs identified and eliminated.

In this paper we investigate the use of a formal method from the area of software specification to capture planning domain models. These mathematically based methods are chiefly for use in applications where safety-critical software has to be produced, and where validation (of the specification) and verification (of software derived from the specification) are important considerations. The method we chose is B, in conjunction with its tool support the B-Toolkit (B-Core (UK) Ltd). B is an industrial-strength method which has been used in a wide variety of software applications.

To facilitate this, we compare it to a language and method specifically aimed at capturing planning domain models: the planning language OCL (object-centred language) and its platform GIPO (Graphical Interface for Planning with Objects) (Simpson *et al.* 2001)). GIPO is a GUI and tools environment for building AI planning domain models in OCL which supports some validation activities such as consistency and animation. GIPO provides both a graphical means of defining a planning domain model and a range of validation tools to perform syntactic and semantic checks of emerging domain models.

Superficially at least, formal specification languages and planning domain description languages are similar in that they share

1. the concept of a 'state'

2. the technique of using pre and post conditions in state transformation, via operations to specify state dynamics

3. the assumptions of closed world, default persistence and instantaneous operator execution

4. the presence of state invariants for validity and documentation purposes. State invariants are also used in OCL.

In the paper we use this correspondence to help in the comparison. In the next sections we apply both B and OCL methods to the acquisition of the two domains, and finish by making a comparison of their performance with respect to validation and consistency checking,

The Blocks World in OCL

As a case study that all planning researchers are aware of, we use a version of the well known 'Blocks world', show-

ing how this can be modelled in both languages, and what opportunities there are for validation. The version we use will consist of a table, on which there are a number of blocks. There are robot arms capable of gripping individual blocks and moving them from one location to another. The complete specification is in the *Resource* section at http://scom.hud.ac.uk/planform/.

The object-centred family of languages (OCL) and their associated development method (embedded in the GIPO tool) forms a rigorous approach to capture the functional requirements of classical planning domains. OCL derives from the work in reference (McCluskey & Porteous 1997). Originally designed for classical goal achievement planning, *OCLh* has been developed for HTN models and PDDL+ − like models (Simpson & McCluskey 2003). We will use the basic version of OCL and the tools available in GIPO to support this.

A specification of the model $\mathcal{M}$ of a domain of interest $\mathcal{D}$, is composed of sets of:

- sort names and object names: A sort (or object class) is a set of object identifiers representing objects that share a common set of characteristics and behaviours. A sort is *primitive* if it is not defined in terms of other sorts. *Sorts* in the Blocks world are *block* and *gripper* and are both primitive.

- predicate definitions: (*Prds*) A predicate from *Prds* represents a functional property of an object. Predicates can be static or dynamic - static predicates include built-in ones such as 'ne' (not equal). The *Prds* of the Blocks world are in Figure 1.

- invariant expressions on individual sorts *Exps*: this is a set of invariants which define all the possible "states" that an object of each sort can inhabit. These are called substates to distinguish them from a world state. An object description is specified by a tuple (s, i, ss), where s is a sort identifier, i is an object identifier, and ss an object's substate. For example, $(gripper, G, [free(G)])$ is an object description meaning that some object G of sort *gripper* is free. Substates operate under a *closed world* assumption local to this restricted set - thus in Figure 1 a block can either be gripped, stacked on another block and clear, stacked on another block and not clear, on the table and clear, or on the table and not clear. For object *block B*, substate $gripped(B, G)$ means that other predicates relating to *block* are *not* true: it is not on the table or clear or on another block.

- general domain invariants: Within OCL general constraints linking sorts can be stated and used in tools. A typical example in the Blocks world is the assertion "for any blocks B, B1, gripper G, $gripped(B, G), on\_block(B, B1)$ is inconsistent".

- operator schema: An action in a domain is represented by an *operator schema*. Actions or events change objects substates. An operator shows the set of object transitions for each object affected by the action. It is specified by a *name*, a set of *prevail* conditions, a set of *necessary* changes and a set of *conditional* changes.

```
predicates:
  on_block(block,block)
  on_table(block)
  clear(block)
  gripped(block,gripper)
  busy(gripper)
  free(gripper)
invariants of 'block': an object B must be
described by exactly one of the following
expressions:
  gripped(B,G)
  on_block(B,B1),clear(B),ne(B,B1)
  on_block(B,B1),ne(B,B1)
  on_table(B),clear(B)
  on_table(B)
```

Figure 1: The Predicates and Substates of Blocks World

```
name: grip_from_table
parameters: B - block, G - gripper
prevail - none
necessary transitions -
block, B:  [on_table(B),clear(B)]
           => [gripped(B,G)]
gripper,G: [free(G)]
           => [busy(G)])
conditional - none
```

Figure 2: Operator for Gripping a Block from a Table

Operators in the Blocks world contain only necessary transitions. These show the conditions on objects that must be true before an action can take place, and specify the new state of an object after the action has been executed. For example, the *grip_from_table* operator has a necessary transition:

```
For any block B
[on_table(B),clear(B)] => [gripped(B,G)]
```

meaning that block B has to be clear and on the table as a precondition and that its state after the transition is that it is gripped. An example *OCL* operator, *grip_from_table*, shows the state changes for two objects, a block and a gripper (see Figure 2 and Figure 3).

Transitions are only shown for objects which are changed. By default all other objects are assumed to remain unchanged. If an object is required to be in a particular state before the transition but does not itself change, it is included as a *prevail* condition. However it is not used in any of the actions of the Blocks world. The meaning of *conditional* change is that **if** a condition on an object is true before an action takes place, **then** the object changes to a new specified state. There are no conditional changes in the Blocks world domain model.

Validation and debugging in OCL

There are several built-in security checks in OCL. Firstly, the user has to capture the space of possible descriptions (substates) of an object of each sort within the sort invariants. Thus world states are formally defined as being a set of legal substates - one for each object declared. This gives an

grip_from_blocks(B, G):

```
[on_block(B,B1),clear(B),ne(B,B1)]
            => [gripped(B,G)]
[on_block(B1,B2),ne(B1,B2)]
            => [on_block(B1,B2),clear(B1)]
```

grip_from_one_block(B, G):

```
[on_block(B,B1),clear(B),ne(B,B1)]
            => [gripped(B,G)])
[on_table(B1)]
            =>[on_table(B1),clear(B1)]
```

Figure 3: Transitions for Blocks when being Gripped from a Block

explicit specification of all possible world states, allowing bugs in state expressions to be prevented. It also restricts the set of goal expressions to those that are feasible in the domain. Secondly, the object transitions in operator schema must conform to the invariants, and hence the transitions are restricted so that operator schema make objects transform to a legal state according to the invariant.

These checks are embedded in GIPO, the GUI and tools environment for building domain models. GIPO prevents errors being introduced (by restricting values in menus etc) and in some cases GIPO's validation checks reveals errors. Other kinds of validation supported by GIPO include: a *Stepper* which aids the user to interactively build up a plan, selecting and applying operator schema for a chosen task; and a *PDDL interface*, allowing third party planners to be bolted on, and their output returned back into a GIPO animator, so that the user can step through a complete plan.

The Blocks World in B

A specification in B will be constructed from one or more abstract machines, with the components of a machine being its variables, invariant, initialisation and operations. A typical abstract machine state comprises several variables which are constrained by the machine invariant and initialised. Operations on the state contain explicit preconditions; the postconditions are expressed as 'generalised substitutions'. Further information describing B can be found in (Schneider 2001).

Some of the sets and logic notation of B is 'standard'. There follows a brief explanation of other parts which may not be familiar to the reader.
If R is a relation from S to T and $A \subseteq S, B \subseteq T$:
$A \lhd R$ means 'restricting the domain of R to set A';
$A \lhd\!\!\!- R$ means 'restricting the domain of R to set $S - A$';
$R \rhd B$ means 'restricting the range of R to set B';
$R \rhd\!\!\!- B$ means 'restricting the range of R to set $S - B$';
If R_1, R_2 are relations from S to T:
$R_1 \lhd\!\!+ R_2$ means 'domain overriding of R_1 by R_2'. Hence on the domain of R_2, the value is given by R_2. Outside dom(R_2), the value is given by R_1.

In applying B to AI Planning, our strategy was to find a correspondence between B specifications of planning worlds and planning-specific languages, hence we used reverse en-

gineering on the OCL model. This gives the correspondence in Table 1.

OCL	B
primitive sorts	sets
predicate names	variable names
operator schema	operations
properties	boolean-valued functions
predicates(x,y)	relations between x and y

Table 1: OCL - B correspondence

Sets in B translate to 'primitive sorts' in OCL. For example *Block, Gripper* in B map to corresponding primitive sorts in OCL;

Boolean valued functions in B map to OCL predicates of arity one. For example function *On_Table(block)* maps to predicate *on_table(block)* in OCL;

Relations in B whose *domain* is the type of x and whose *range* is the type of y ($\in X \leftrightarrow Y$) map to OCL predicates of arity 2, *pred(x, y)*. For example *On_Block $\in$ Block $\rightarrowtail$ Block* becomes the predicate *on_block(block, block)* in OCL. (Note that we have restricted ourselves to domains capable of being modelled via predicates of arity two.)

The following comprises the B machine header, sets and variables clauses:

MACHINE *Blocks_World*
SETS *Block*; *Gripper*
VARIABLES
On_Block, On_Table, Clear, Gripped, Free

Note that the predicate 'Busy' from OCL is not represented for it is simply the negation of 'Free'. This exception was made to avoid unnecessary replication in the B model. A fragment is presented in the next subsection. The complete specification and that of the Tyres World is in http://scom.hud.ac.uk/scommmw/PlanningDomains/

Invariant and Initialisation of Blocks World in B

The types of the variables were reverse engineered - OCL predicates of arity one, *pred(x)* were modelled by B total functions whose *domain* is the type of x and whose *range* is the booleans. OCL predicates of arity 2, *pred(x, y)*, were modelled by B relations whose *domain* is the type of x and whose *range* is the type of y. Modelling in this manner allowed us to take advantage of the fact that in the Blocks world all the relations were functions and some were 1-1.

A simple initialisation condition (below) was specified - that each block is on the table, clear and not gripped. The '||' stands for *parallel substitution* where all variable substitutions are assumed to take place in parallel rather than in sequence. The idea of a fixed initialisation differs from OCL where the domain is initialised at the start of each plan.
INITIALISATION
On_Block := {} ||

INVARIANT

$On\_Block \in Block \nrightarrow Block \wedge$ (1)

$On\_Table \in Block \rightarrow BOOL \wedge$ (2)

$Clear \in Block \rightarrow BOOL \wedge$ (3)

$Gripped \in Block \nrightarrow Gripper \wedge$ (4)

$Free \in Gripper \rightarrow BOOL \wedge$ (5)

$\forall \, blk \, . \, (\, blk \in \mathrm{dom} \ On\_Block \Rightarrow On\_Block \, (\, blk \,) \neq blk \,) \wedge$ (6)

$\mathrm{ran} \ On\_Block \cup \mathrm{dom} \ Gripped = \mathrm{dom} \ (\, Clear \rhd \{ \, FALSE \, \} \,) \wedge$ (7)

$\mathrm{ran} \ Gripped \cap \mathrm{dom} \ (\, Free \rhd \{ \, TRUE \, \} \,) = \{\} \wedge$ (8)

$\mathrm{ran} \ Gripped \cup \mathrm{dom} \ (\, Free \rhd \{ \, TRUE \, \} \,) = Gripper \wedge$ (9)

$\mathrm{dom} \ On\_Block \cap \mathrm{dom} \ Gripped = \{\} \wedge$ (10)

$\mathrm{dom} \ Gripped \cap \mathrm{dom} \ (\, On\_Table \rhd \{ \, TRUE \, \} \,) = \{\} \wedge$ (11)

$\mathrm{dom} \ On\_Block \cap \mathrm{dom} \ (\, On\_Table \rhd \{ \, TRUE \, \} = \{\} \wedge$ (12)

$\mathrm{dom} \ Gripped \cap \mathrm{dom} \ (\, Clear \rhd \{ \, TRUE \, \} \,) = \{\} \wedge$ (13)

$\mathrm{dom} \ Gripped \cap \mathrm{dom} \ On\_Block = \{\} \wedge$ (14)

$\mathrm{dom} \ Gripped \cap \mathrm{ran} \ On\_Block = \{\} \wedge$ (15)

$\mathrm{dom} \ Gripped \cup \mathrm{dom} \ (\, On\_Table \rhd \{ \, TRUE \, \} \,)$

$\quad \cup \mathrm{dom} \ On\_Block = Block$ (16)

Figure 4: B Invariant for the Blocks World

$On\_Table := Block \times \{ \, TRUE \, \} \, \|$
$Clear := Block \times \{ \, TRUE \, \} \, \|$
$Gripped := \{\} \, \|$
$Free := Gripper \times \{ \, TRUE \, \}$

Blocks World Operations in B

It is only necessary to have one operation for gripping a block in B. However, *Grip_Block_On_Table* was represented plus one operation *Grip_Block_On_Block* to represent the two actions required by OCL. (See Figure 5). This was so that we could compare the representations more closely. For a similar reason two operations were also specified to release a block: *Put_Block_On_Table*, *Put_Block_On_Block*.

Validation

Reasoning about a formal specification and animation of a formal specification are both activities concerned with *validation*, and these are complementary activities.

A way of reasoning about a formal specification is via the generation and discharge of 'proof obligations'. A set of proof obligations involving *consistency properties* of a system can be automatically generated by the BTool. An example of two of these is (1) Consistency of initialisation: the initialisation *must* establish the invariant. (2) Consistency of operation: each operation must *preserve* the invariant. Other consistency properties involve the static parts of the machine (sets, constants, properties etc.). It is also possible to check the machine invariant during animation.

The B-Toolkit includes an animator to 'execute' operations, and a proof tool to check that proof obligations are met. The Blocks world in B was validated using the B-Toolkit. Each new version was animated to check for errors. The version was run for each operation with the invariant displayed and this provided a quick method for rooting out errors *before* the prover was used. The proof obligation generator and prover were then run - in all 72 proof obligations were generated with 40 undischarged by the prover -

these were subsequently hand-checked, which was a laborious process. During this procedure the invariant was frequently scanned and it was discovered that an unnecessary conjunct was present in the original versions:

$\mathrm{ran} \ (\, On\_Block \,) \cap \mathrm{dom} \ (\, Clear \rhd \{ \, TRUE \, \} \,) = \{\}$

was found to be already covered by (7) and (13).

As part of the checking process of the two models, we ran one of the planners in GIPO on a particular task, and then simulated this in the B-Toolkit using the animator. We used the well-known task (in AI planning literature), the Sussman Anomaly, as shown in Figure 6. This solution was obtained
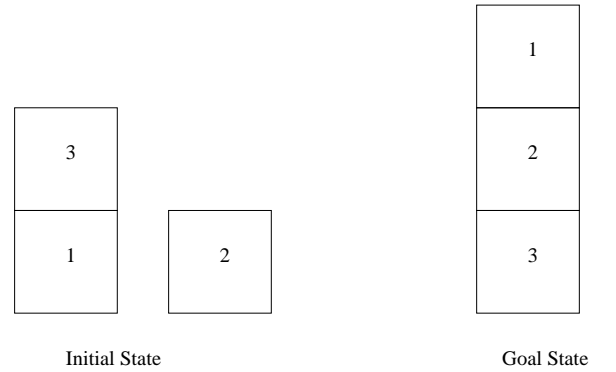
Figure 6: Initial and Goal States

from one of the planners:

```
grip_from_one_block(block3,block1,tom)
put_on_table(block3,tom)
grip_from_table(block2,tom)
put_on_one_block(block2,tom,block3)
grip_from_table(block1,tom)
put_on_blocks(block1,tom,block2,block3)
```

It was obviously not possible to generate an automatic sequence of operations using the B-Toolkit - as in the case of

Grip_Block_On_Table (*blk* , *grp*) $\widehat{=}$
   PRE
      grp $\in$ dom (*Free* $\triangleright$ { *TRUE* })$\wedge$
      blk $\in$ dom (*Clear* $\triangleright$ { *TRUE* }) $\wedge$
      On_Table (*blk*) = *TRUE*
   THEN
      Free (*grp*) := *FALSE* ||
      Clear (*blk*) := *FALSE* ||
      On_Table (*blk*) := *FALSE* ||
      Gripped (*blk*) := *grp*
   END

Grip_Block_On_Block (*grp* , *blk*) $\widehat{=}$
   PRE
      grp $\in$ dom (*Free* $\triangleright$ { *TRUE* }) $\wedge$
      blk $\in$ dom (*On_Block*) $\wedge$
      blk $\in$ dom (*Clear* $\triangleright$ { *TRUE* })
   THEN
      On_Block := { *blk* } $\lhd$ *On_Block* ||
      Free (*grp*) := *FALSE* ||
      Clear := *Clear* $\lhd$ { *blk* $\mapsto$ *FALSE* , *On_Block* (*blk*) $\mapsto$ *TRUE* } ||
      Gripped (*blk*) := *grp*
   END Put_Block_On_Block (*blk1* , *grp*) $\widehat{=}$
   PRE
      grp = *Tom* $\wedge$
      blk1 $\in$ *Block* $\wedge$
      Gripped (*blk1*) = *grp* $\wedge$
      Free (*grp*) = *FALSE*
   THEN
      Free (*grp*) := *TRUE* ||
      Gripped := {} ||

      ANY *blk2*
      WHERE
blk2 $ $ *in Block* $\wedge$
      Clear (*blk2*) = *TRUE*
      THEN
        On_Block (*blk1*) := *blk2* ||
        Clear := *Clear* $\lhd$ { *blk1* $\mapsto$ *TRUE* } $\lhd$ { *blk2* $\mapsto$ *FALSE* }
      END
      END

Figure 5: Operations in B

the planner. However the equivalent of the 'Sussman Anomaly' configurations was achieved by commencing from the initial state and placing block 3 on block 1, as shown in the following animation (where * means the variable has changed):

```
*On_Block {block3 |-> block1}
 On_Table {block3 |-> FALSE ,
           block1 |-> TRUE ,
           block2 |-> TRUE ,

           block4 |-> TRUE ,
           block5 |-> TRUE ,
           block6 |-> TRUE ,
           block7 |-> TRUE}
*Clear    {block1 |-> FALSE ,
           block3 |-> TRUE ,
           block2 |-> TRUE ,

           block4 |-> TRUE ,
           block5 |-> TRUE ,
           block6 |-> TRUE ,
           block7 |-> TRUE}
*Gripped  {}
*Free     {Tom |-> TRUE}
```

The desired final state was achieved using the operations

```
Grip_Block_On_Block ( block3, Tom )
Put_Block_On_Table ( block3 )
Grip_Block_On_Table ( block2 , Tom )
Put_Block_On_Block ( block2 )
   (Local Variable blk2 in 'ANY' set to block3)
Grip_Block_On_Table ( block1 , Tom )
Put_Block_On_Block ( block1 )
```

(see Figure 5) with end state:

```
*On_Block {block2 |-> block3 ,
           block1 |-> block2}
 On_Table {block1 |-> FALSE ,
           block2 |-> FALSE ,
           block3 |-> TRUE, .. }
*Clear    {block2 |-> FALSE ,
           block1 |-> TRUE ,
           block3 |-> FALSE, .. }
*Gripped  {}
*Free     {Tom |-> TRUE}
```

Tyres World Case Study

We used a similar strategy (i.e. reverse engineering in modelling variables) when we modelled the 'Tyres World' in B. The Tyres world involves the changing of a faulty wheel using a wrench and a jack, both of which are (usually) initially in the car boot. Wheel changing involves loosening and removing wheel nuts, then changing the wheel. The wrench, jack and spare wheel must be available when required and the actions must take place in the correct order. The objective of the case study was (first) as a preliminary investigation into the relationship between B and OCL and (second) to test the adequacy of the validation tools (fully described in (West & Kitchin 2003)). The 'Tyres World' domain (Russell 1992) was chosen because, in the field of Planning, it is a well-known and well-used model that is unlikely to have any hidden errors. In the case of OCL, two wheels, hubs and

their attached nuts were modelled plus a spare wheel in the car boot. The additional wheel (as compared with the 'usual' model in (Russell 1992)) was introduced so that extra validation checks could be introduced. In contrast, in the B model *four* wheels plus hubs and nuts were modelled, although as it turned out, two would have been sufficient. Actions in the OCL model include 'opening the car boot', 'loosening the nuts', 'removing the wheel' etc. and the B specification contained operations equivalent to these. Some exceptions were made where simplifications were possible in B; an example is the use of a single operation for 'fetching a tool'.

The approach was the deliberate introduction of equivalent errors into both the B and OCL models to see if the use of the stepper/animator and validation checks and proof tool, would identify these faults. Various tasks were tried out in GIPO using both the stepper and planning engines to see if errors and inconsistencies in the domain model were detected, and to compare its performance with that of the B-Toolkit. Validation checks in GIPO include checks on operators and checks on tasks. Thus operators must consist of legal expressions with respect to invariant expressions on the individual sorts; and initial states and goal expressions must likewise consist of a legal substate expressions.
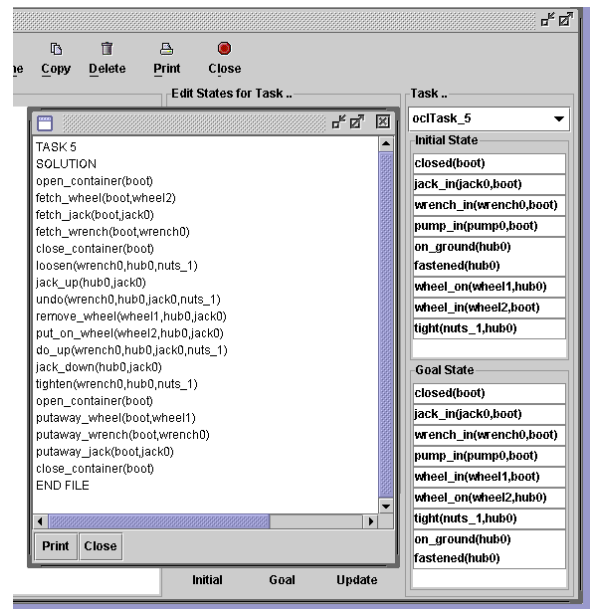

Figure 7: Task and Plan for changing a wheel

A screen-shot of GIPO (Figure 7) shows a plan generation for a task: initially all the tools and spare wheel (wheel2) are in the car boot and the goal is a change of wheel (wheel1). The next section describe the errors introduced and the results of validation for each of the two tools.

Challenging the Models

Errors in the initial state We introduced errors such as the obviously incorrect state of two wheels on a single hub. The result for the OCL model was that GIPO did not object to this inconsistent initial state - no errors were found

by the validation checks. When tried with a planner (FF (Hoffmann 2000)), it reported that the goal was impossible. Another initial state was created in which the nuts were on one of the hubs but no wheel was on that hub. The other hubs were in a correct state. This inconsistency is not found by the validation checks in GIPO. The user can find it by using the stepper or using a planner that incorporates such checks: FF for example reports that the goal is impossible. This uncovered a simple insecurity in the GIPO tool itself - although individual sort invariants were actively used in its tools, the general domain invariants were not being used to check state validity.

The errors were introduced in the B model by altering the initial state. Both of these errors introduced inconsistency in the B specification - of the initial state with respect to the invariant. They were discovered by the B-Toolkit most simply by using the animator. However the proof tool also provided a check.

Errors in setting tasks Errors were introduced in the OCL task description within GIPO. These involved an attempt to reach an illegal state (i.e. one which was inconsistent with the domain model). One example involved jacking up two hubs using the same jack. Again the validation checks of GIPO don't report an impossible task. When tested, the FF planner very quickly says the problem is proven unsolvable. It was subsequently discovered that the domain model in OCL did not contain the 'inconsistency constraint' in the prohibition of the jack on two different hubs.

Of course there is no equivalent function of 'setting a task' in B so in order to correspond with the GIPO task, a single operation, of attempting to jack up a wheel where the jack is already in use on a different wheel was tried using the animator. However the invariant of the Tyres World B specification was such that there is a 1-1 relationship between a wheel hub and a jack - and the operation 'JackUp-Car' supported this in its precondition that the jack should not be in use already and an error was generated.

Errors in pre- and postconditions Prevail conditions in OCL are pre-conditions that persist - that is, the object concerned does not change state during the operation. An example of this would be the prevail condition *have_wrench* of the *loosen* operator: in order to loosen the nuts we must have the wrench - and we will still have the wrench after the nuts have been loosened. We removed the *have_wrench* prevail condition from the *loosen* operator. This error, as expected, was not detected by validation checks, but became apparent when using GIPO's stepper. The operator was not able to be applied because the wrench was not available. This type of error does not affect overall consistency of the domain model, but is just concerned with a specific object being part of a particular operation.

In the case of the B model there was no problem with contravention of the pre-condition and no problem with the invariant. (Note that the OCL prevail condition can be modelled as a pre-condition in B as, by default, any variable for which there is no substitution does not change.)

The error is only demonstrated by the showing of a 'silly' result in that the wrench is still in the car boot. This is a 'domain-specific' error which could only have been demonstrated by animation.

Because of the manner in which actions are described in OCL - by the change in state of individual objects - it was similarly not possible for a 'post' condition to be removed on its own. For example: for the operator *fetch_jack* the jack object changes state from being in the car boot (precondition for the transition) to being available for use (post-condition for the transition). In an attempt to introduce this kind of error, we removed the transition for the 'jack' object from the 'fetch jack' action. The error became apparent when using the stepper.

The experiments also uncovered a previously unknown omission in the B model - a missing precondition for putting away the tyre.

Comparison of Operations

Two operations in B are compared with the equivalent actions in the OCL model.

Gripping a Block from the Table If we compare the operation *Grip_Block_On_ Table* in B with its equivalent in OCL (Figures 5 and 2) we see the same pre and postconditions. However they are structured differently in OCL where the precondition for each object is an appropriate substate from the substate classes:

1. The precondition that the block is on the table and clear becomes the left hand side of the necessary condition for the block.

2. The precondition that the gripper is free becomes the left hand side of the necessary condition for the gripper.

3. The substitution in B for the block, that it is gripped, becomes the right hand side of the necessary condition for the block. However in B it is stated explicitly that the gripped block is no longer on the table and not clear. In OCL this is assumed from the substate.

4. The substitution in B that gripper is in use becomes the right hand side of the necessary condition for the gripper.

Gripping a Block from a Block Comparing B and OCL versions of 'gripping block1 from block2', (Figures 5 and 3) we see that there is a change in the state of both blocks after the operation. For B, it is enough to state that block2 is now clear. However for OCL this is not enough and we must distinguish between 2 substates - where block2 is on the table and where block2 is on another block. Since we have made a change in the state of block2 we must be precise about the whole of its state. The different outcomes for block2 give rise to the two actions in OCL. As in the previous operation, what is implicit in OCL must be explicit in B. Thus we must say that the gripped block is no longer free.

A Comparison of the use of B and OCL to acquire planning domain knowledge

Here we summarise the results of our comparison:

1. The B language and toolkit is an industrial strength formal specification and development method, whereas OCL is a tool used in research and education. The exercise showed some problems with GIPO - in particular that its static validation checks should be extended to test the consistency of the initial state and goal expressions against the general domain invariants.

2. B allows the user to encode more precise details about the relations in the domain than GIPO - they can be be relations, 1-1 functions etc. This level of precision is certainly not available in most planning languages, and is attractive in safety-related applications.

3. As OCL is aimed specifically at planning, it has inbuilt structures and mechanisms that anticipate the entities that are to be represented. This makes the encoding rather more compact than the B specification. Encoding in B, as one would expect from a general language, one is left with more choices and decisions in the encoding process. The correspondence we used in our case reduced the encoding choices, but still the B encoding is rather 'flat' in that the invariant list contains all predicate and invariant information.

4. Both languages assume default persistence and a closed world. The differences in this respect are subtle, in that in B a variable involved in a precondition remains unaltered by default. However in OCL a 'prevail' is required if precondition variables are unchanged.

5. Regarding validation and debugging, both languages have effective, automated tool support which performs validation/consistency checks and identifies the presence of bugs. Not surprisingly, the B toolkit was more reliable at finding inconsistencies in some cases, as it demands a more detailed specification.

To further explore the comparison, we show how, in the Blocks world, the OCL specification of substates can be derived from the B invariant conjuncts (1–16) in Figure 4. We observe that the following three sets are disjoint and 'partition' Block:

$$\mathrm{dom}(Gripped),$$
$$\mathrm{dom}(On\_Table) \rhd \{TRUE\},$$
$$\mathrm{dom}(On\_Block)$$

Although the partition is not the same as the substate partition in OCL, it is possible to obtain an equivalent partition if we first note that:
$\mathrm{dom}(Clear \rhd \{TRUE\})$ and $\mathrm{dom}(Clear \rhd \{FALSE\})$ partition $Block$ and we have

$$\mathrm{dom}\, On\_Table \rhd \{TRUE\} = \mathrm{dom}\, On\_Table \rhd \{TRUE\}\ \cap$$
$$(\mathrm{dom}\, Clear \rhd \{TRUE\}) \cup (\mathrm{dom}\, Clear \rhd \{FALSE\})$$

and from (10) the partition now becomes:

$$\mathrm{dom}\, Gripped,$$
$$(\mathrm{dom}\, On\_Table \rhd \{TRUE\}) \cap (\mathrm{dom}\, Clear \rhd \{TRUE\}),$$
$$(\mathrm{dom}\, On\_Table \rhd \{TRUE\}) \cap (\mathrm{dom}\, Clear \rhd \{FALSE\}),$$
$$\mathrm{dom}\, On\_Block.$$

This can be made explicit and in a similar format to the OCL version using conjuncts from the invariant. For example from (2):

$$\mathrm{dom}\, On\_Block \cap (\mathrm{dom}(Clear \rhd \{FALSE\}) =$$
$$\mathrm{dom}\, On\_Block \cap (\mathrm{dom}(Clear \rhd \{FALSE\}) \cap Block =$$
$$\mathrm{dom}\, On\_Block \cap (\mathrm{dom}(Clear \rhd \{FALSE\}) \cap$$
$$(\mathrm{dom}\, On\_Table \rhd \{FALSE\} \cup \mathrm{dom}\, On\_Table \rhd \{TRUE\}) =$$
$$\mathrm{dom}\, On\_Block \cap (\mathrm{dom}(Clear \rhd \{FALSE\})$$
$$\cap(\mathrm{dom}\, On\_Table \rhd \{FALSE\}$$

In a similar manner using other conjuncts, this set can be equivalently expressed:

$$\mathrm{dom}\, On\_Block \cap (\mathrm{dom}(Clear \rhd \{FALSE\})$$
$$\cap(\mathrm{dom}\, On\_Table \rhd \{FALSE\}$$
$$\cap (Block - \mathrm{dom}\, Gripped)$$

This, with (6) can be expanded out:

$$\forall\, blk1 \in Block.(\exists\, blk2 \in Block.(On\_Block(blk1) = blk2$$
$$\wedge\, blk1 \neq blk2 \wedge Clear(blk1) = FALSE))$$
$$\cdots$$

which is equivalent to the substate
```
[on_block(B,B1),ne(B,B1)],
```
given the local closed world assumption.

Conclusions

In this paper we have investigated the use of a formal method to capture planning domain models. We have compared the method (together with its commercially-available tool support) with a planning-oriented method. The comparison shows a remarkable similarity between the two. The advantages in using a method such as B are that it is mathematically based so that formal reasoning can be used to deduce desirable (and potentially undesirable) properties. Support for the method is available via tools - such as the Toolkit. However, the disadvantages are that there are no special planning - oriented features, and that the B specification, once validated, would have to be translated into a more planner-friendly language in order to be used with current planning engines.

References

B-Core (UK) Ltd. http://www.b-core.com/.

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*.

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

Munoz-Avila, H. M.; Aha, D. W.; Nau, D.; Weber, R.; Breslow, L.; and Yaman, F. 2001. SiN: Integrating case-based reasoning with task decomposition. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 999–1004.

Penix, J.; Pecheur, C.; and Havelund, K. 1998. Using Model Checking to Validate AI Planner Domain Models. In *Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard*.

Russell, S. 1992. Efficient memory-bounded search algorithms. In *Proc. ECAI*.

Schneider, S. 2001. *The B-Method: An Introduction*. Palgrave.

Simpson, R. M., and McCluskey, T. L. 2003. Plan Authoring with Continuous Effects. In *Proceedings of the 22nd UK Planning and Scheduling Workshop (PLANSIG-2003), Glasgow, Scotland*.

Simpson, R. M.; McCluskey, T. L.; Zhao, W.; Aylett, R. S.; and Doniat, C. 2001. GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. In *Proceedings of the 6th European Conference on Planning*.

West, M. M., and Kitchin, D. E. 2003. Testing Domain Model Validation Tools. In *Proceedings of the 22nd Workshop of the UK Planning and Scheduling SIG, Glasgow*.