Creating Planning Domain Models in KEWI

Gerhard Wickler
Artificial Intelligence Applications Institute
School of Informatics
University of Edinburgh
g.wickler@ed.ac.uk

Lukáš Chrpa and **Thomas Leo McCluskey**
PARK Research Group
School of Computing and Engineering
University of Huddersfield
{l.chrpa, t.l.mccluskey}@hud.ac.uk

Abstract

This paper reports on progress towards a tool for the representation of shared, procedural and declarative knowledge whose aim is to be used for various functions to do with the automation of a complex process control application - primarily to guide the response phase during an emergency situation, but also for supporting normal automated operation.

The tool is a Knowledge Engineering Web Interface called KEWI. The focus of the paper is on the conceptual model used to represent the declarative and procedural knowledge. The model consists of three layers: an ontology, a model of basic actions, and more complex methods. It is this structured conceptual model that facilitates knowledge engineering. We are aiming to evaluate the use of a *central knowledge model* for a range of planning-related functions, where the parts of the model are automatically assembled e.g. into PDDL for operational use.

The current application area that drives the development of the tool is well control.

Introduction

Domain-independent planning has grown significantly in recent years mainly thanks to the International Planning Competition (IPC). Besides many advanced planning engines, PDDL, a de-facto standard language family for describing planning domain and problem models, has been developed. However, encoding domain and problem models in PDDL requires a lot of specific expertise and thus it is very challenging for a non-expert to use planning engines in applications.

This paper concerns the use of AI planning technology in an organisation where (i) non-planning experts are required to encode knowledge (ii) the knowledge base is to be used for more than one planning and scheduling task (iii) it is maintained by several personnel over a long period of time, and (iv) it may have a range of potentially unanticipated uses in the future. The first concern has been a major obstacle to using AI-based, formal representations, in that the expertise required to produce such representations has normally been acquired and encoded by planning experts (e.g. as in NASA's applications (Ai-Chang et al. 2004)). The other concerns are often not covered in the planning literature: in real applications the knowledge encoding is a valuable, general asset, and one that requires a much richer conceptual representation than, for example is accorded by planner-input languages such as PDDL.

We present here a Knowledge Engineering method using a Web Interface aimed at AI Planning, called KEWI. The primary idea behind KEWI then is to ease this formalization of procedural knowledge, allowing domain experts to encode their knowledge themselves, rather than knowledge engineers having to elicit the knowledge before they formalize it into a representation. A number of frameworks exist that support the formalization of planning knowledge in shared web-based systems. Usually, such frameworks build on existing Web 2.0 technologies such as a wiki. A wiki that supports procedural knowledge is available at wikihow.com, but the knowledge remains essentially informal. A system that uses a similar approach, namely, representing procedural knowledge in a wiki is CoScripter (Leshed et al. 2008). However, their representation is not based on AI planning and thus does not support the automated composition of procedures. More recently, an AI-based representation has been used in OpenVCE (Wickler, Tate, and Hansberger 2013).

As far as we are aware, very few collaborative, domain-expert-usable, knowledge acquisition interfaces are available that are aimed at supporting the harvesting of planning knowledge within a rich language for use in a number of planning-related applications. After initial acquisition, the validation, verification, maintenance and evolution of such such knowledge is of prime importance, as the knowledge base is a valuable asset to an organisation.

Related Work

There have been several attempts to create general, user-friendly development environments for planning domain models, but they tend to be limited in the expressiveness of their underlying formalism. The Graphical Interface for Planning with Objects (GIPO) (Simpson, Kitchin, and McCluskey 2007) is based on object-centred languages *OCL* and OCL_h. These formal languages exploit the idea that a set of possible states of objects are defined first, before action (operator) definition (McCluskey and Kitchin 1998). This gives the concept of a *world state* consisting of a set of states of objects, satisfying given constraints. GIPO uses a number of consistency checks such as if the object's class

hierarchy is consistent, object state descriptions satisfy invariants, predicate structures and action schema are mutually consistent and task specifications are consistent with the domain model. Such consistency checking guarantees that some types of errors can be prevented, in contrast to ad-hoc methods such as hand crafting.

itSIMPLE (Vaquero et al. 2012) provides a graphical environment that enables knowledge engineers to model planning domain models by using the Unified Modelling Language (UML). Object classes, predicates, action schema are modelled by UML diagrams allowing users to 'visualize' domain models which makes the modelling process easier. itSimple incorporates a model checking tool based on Petri Nets that are used to check invariants or analyze dynamic aspects of the domain models such as deadlocks.

The Extensible Universal Remote Operations Planning Architecture (EUROPA) (Barreiro et al. 2012), is an integrated platform for AI planning and scheduling, constraint programming and optimisation. This platform is designed to handle complex real-world problems, and the platform has been used in some of NASA's missions. EUROPA supports two representation languages, NDDL and ANML (Smith, Frank, and Cushing 2008), however, PDDL is not supported.

Besides these tools, it is also good to mention VIZ (Vodrážka and Chrpa 2010), a simplistic tool inspired by itSimple, and PDDL Studio (Plch et al. 2012), an editor which provides users a support by, for instance, identifying syntax errors or highlighting components of PDDL.

In the field of Knowledge Engineering, methodologies have been developed which centre on the creation of a precise, declarative and detailed model of the area of knowledge to be engineered, in contrast to earlier expert systems approaches which appeared to focus on the "transfer" expertise at a more superficial level. This "expertise model" contains a mix of knowledge about the "problem solving method" needed within the application and the declarative knowledge about the application. Often a key rationale for knowledge engineering is to create declarative representations of an area to act as a formalised part of some requirements, making explicit what hitherto has been implicit in code, or explicit but in documents. Knowledge Engineering modelling frameworks arose out of this, such as CommonKads (Schreiber et al. 1999), which were based on a deep modelling of an area of expertise, and emphasising a lifecycle of this model. The "knowledge model" within CommonKADS, which contains a formal encoding of task knowledge, such as problem statement(s), as well as domain knowledge, is similar to the kind of knowledge captured in KEWI. Unlike KEWI however, this model was expected to be created by knowledge engineers rather than domain experts and users.

Conceptual Model of KEWI

KEWI is a tool for encoding domain knowledge mainly by experts in the application area rather than AI planning experts. The key idea behind KEWI is to provide a user-friendly environment as well as a language which is easier to follow, especially for users who are not AI planning experts. A high-level architecture of KEWI is depicted in Figure 1.

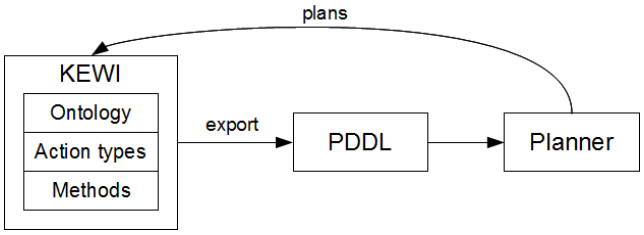

Figure 1: An architecture of KEWI.

Encoded knowledge can be exported into the domain and problem description in PDDL on which standard planning engines can be applied, and retrieved plans can be imported back to KEWI. Hence, the user does not have to understand, or even be aware, of any PDDL encodings.

A language in which domain knowledge is encoded in KEWI has three parts, which are explained in the following subsections. First, a domain ontology is defined. The domain ontology consists of definition of classes of objects, hierarchies of classes and relations between objects. Second, action types, concretely action name, preconditions and effects, are defined. Third, methods which introduce additional ordering constraints between actions, are defined.

Ontology: Concepts, Relations and Properties

Ontological elements are usually divided into concepts and instances. Typically, the concepts are defined in a planning domain whereas the instances are defined in a planning problem. Since our focus for KEWI is on planning domains we shall mostly deal with concepts here.

Concepts A concept is represented by a unique symbol in KEWI. The formal definition of a concept is given by its super-class symbol and by a set of role constraints that define how instances of the concept may be related to other concepts. In KEWI, the definition of a concept also includes other, informal elements that are not used for formal reasoning. However, the knowledge engineering value of such informal elements must not be underestimated, much like the comments in programming often are vital for code to be understandable.

Definition 1 (KEWI Concept)**.** *A concept C in KEWI is a pair $\langle C^{sup}, R\rangle$, where:*

- *C^{sup} is the direct super-concept of C and*
- *R is a set of role constraints of the form $\langle r, n, C'\rangle$ where r is a symbolic role name, C' is a concept (denoting the role filler type), and n is a range $[n_{min}, n_{max}]$ constraining the number of different instances to play that role.*

We assume that there exists a unique root concept often referred to as *object* or *thing* that acts as the implicit super-concept for those concepts that do not have an explicit super-concept defined in the same planing domain. Thus, a concept C may be defined as $\langle \triangle, R\rangle$, meaning its super concept is implicit. This implicit super-concept has no role constraints attached.

For example, in the Dock Worker Robot (DWR) domain, the concepts `container` and `pallet` could be defined

with the super-concept `stackable`, whereas the concept `crane` could be defined as a root concept with no super-concept (implicitly: $\triangle$). A role constraint can be used to define that a crane can hold at most one container as follows: $\langle$`holds`$, [0, 1],$ `container`$\rangle$.

Since super-concepts are also concepts, we can write a concept C as $\langle\langle\langle\triangle, R_n\rangle, \ldots, R_2\rangle, R_1\rangle$. Then we can refer to all the role constraints associated with C as $R^* = R_n \cup \ldots \cup R_2 \cup R_1$, that is, the role constraints that appear in the definition of C, the role constraints in its direct super-concept, the role constraints in its super-concepts super-concept, etc.

The reason for introducing this simple ontology of concepts is that we can now constrain the set of possible world states based on the role constraints. States are defined as sets of ground, first-order atoms over some function-free language $\mathcal{L}$. This language shall contain symbols to denote each instance of a concept defined in the ontology $(c_1, \ldots, c_L)$ where the type function τ maps each instance c_i to its type C, a concept in the ontology. The relation symbols of $\mathcal{L}$ are defined through the role constraints.

Definition 2 (Relations in $\mathcal{L}$)**.** *Let $\langle r, n, C'\rangle$ be a role constraint of some concept C. Then the first-order language $\mathcal{L}$ that can be used to write ground atoms in a state contains a binary relation $C.r \subseteq C \times C'$.*

In what follows we shall extend the language to include further relation symbols, but for now these relations defined by the ontology are all the relations that may occur in a state. The reason why the relation name is a combination of the concept and the role is simply to disambiguate between roles of the same name but defined in different concepts. Where all role names are unique the concept may be omitted.

We can now define what it means for a state to be valid with respect to an ontology defined as a set of KEWI concepts. Essentially, for a state to be valid, every instance mentioned in the state must respect all the role constraints associated with the concepts to which the instance belongs. Since role constraints are constraints on the number of possible role fillers we need to be able to count these.

Definition 3 (Role Fillers)**.** *Let s be a state, that is, a set of ground atoms over objects $c_1, \ldots, c_L$ using the relations in $\mathcal{L}$. Let $\langle r, n, C'\rangle$ be a role constraint of some concept C. Then we define $vals_s(C.r, c_i) = \{c_f | C.r(c_i, c_f) \in s\}, c_i \in C, c_f \in C'$, that is, the set of all constants that play role r for c_i in s.*

Definition 4 (Valid State)**.** *Let C be a KEWI concept. Then a state s is valid if, for any instance c_i of C and any role constraint $\langle r, n, C'\rangle$ of C or one of its (direct and indirect) super-concepts, the number of ground atoms $a = C.r(c_i, *)$ must be in the range $[n_{min}, n_{max}]$, i.e. $n_{min} \leq |vals_s(C.r, c_i)| \leq n_{max}$.*

Thus, a concept definition defines a set of role constraints which can be interpreted as relations in a world state. The numeric range defines how many ground instances we may find in a valid state. This is the core of the ontological model used in KEWI.

For example, let `k1` be a crane and `ca` be a container. Then a state may contain a ground atom `crane.holds(k1,ca)`. If a state contains this atom, it may not contain another one using the same relation and `k1` as the first argument.

Relations While the relations defined through the concepts in KEWI provide a strong ontological underpinning for the representation, there are often situations where other relations are more natural, e.g. to relate more than two concepts to each other, or where a relation does not belong to a concept. In this case relations can be defined by declaring number and types (concepts) of the expected arguments.

Definition 5 (Relations in $\mathcal{L}$)**.** *A relation may be defined by a role constraint as described above, or it may be a relation symbol followed by an appropriate number of constants. The signature of a relation R is defined as $C_1 \times \ldots \times C_R$ where C_i defines the type of the ith argument.*

A valid state may contain any number of ground instances of these relations. As long as the types of the constants in the ground atoms agree with the signature of the relation, the state that contains this atom may be valid.

Properties In reality, we distinguish three different types of role constraints: *related classes* for defining arbitrary relations between concepts, *related parts* which can be used to define a "part-of" hierarchy between concepts, and *properties* which relate instances to property values.

The first two are equivalent in the sense that they relate objects to each other. However, properties usually relate values to objects, e.g. an object may be of a given colour. While it often makes sense to distinguish all individual instances of a concept, this is not true for properties. While the paint that covers one container may not be the same paint that covers another, the colour may be the same. To allow for the representations of properties in KEWI, we allow for the definition of properties with enumerated values.

Definition 6 (Properties)**.** *A property P is defined as a set of constant values $\{p_1, \ldots, p_P\}$.*

It is easy to see that the above definitions relating to role constraints and other relations can be extended to allow properties in place of concepts and property values in place of instances. A minor caveat is that property values are usually defined as part of a planning domain, whereas instances are usually given in a planning problem.

Action Types

Action types in KEWI are specified using an operator name with typed arguments, a set of preconditions, and a set of effects. This high-level conceptualization of action types is of course very common in AI planning formalisms. KEWI's representation is closely linked with the ontology, however. This will enable a number of features that allow for a more concise representation, allowing to reduce the redundancy contained in many PDDL planning domains.

Object References In many action representations it is necessary to introduce one variable for each object that is somehow involved in the execution of an action. This variable is declared as one of the typed arguments of the action type. The variable can then be used in the preconditions and

effects to consistently refer to the same objects and express conditions on this object.

Sometimes, an action type may need to refer to specific constants in its preconditions or effects. In this case, the unique symbol can be used to identify a specific instance. In the example above, `k1` was used to refer to a crane and `ca` to refer to a container. In most planning domains, operator definitions do not refer to specific objects, but constants may be used as values of properties.

In addition to variables and constants, KEWI also allows a limited set of function terms to be used to refer to objects in an action type's preconditions and effects. Not surprisingly, this is closely linked with the ontology, specifically with the role constraints that specify a maximum of one in their range.

Definition 7 (Function Terms)**.** *Let $\langle r, n, C' \rangle$ be a role constraint of some concept C where $n_{max} = 1$. Then we shall permit the use of function terms of the form $C.r(t)$ in preconditions and effects, where t can again be an arbitrary term (constant, variable, or function term) of type C'.*

Let s be a valid state, that is, a set of ground atoms over objects $c_1, \ldots, c_L$ using the relations in $\mathcal{L}$. Then the constant represented by the function term $C.r(c_i)$ is:

- *c_j if $vals_s(C.r, c_i) = \{c_j\}$, or*
- *`nothing` ($\bot$) if $vals_s(C.r, c_i) = \emptyset$.*

Note that the set $vals_s(C.r, c_i)$ can contain at most one element in any valid state. If it contains an element, this element is the value of the function term. Otherwise a new symbol that must not be one of the constants $c_1, \ldots, c_L$ will be used to denote that the function term has no value. This new constant `nothing` may also be used in preconditions as described below.

The basic idea behind function terms is that they allow the knowledge representation to be more concise; it is no longer necessary to introduce a variable for each object. Also, this style of representation may be more natural, e.g. to refer to the container held by a crane as `crane.holds(k1)` meaning "whatever crane `k1` holds", where the role constraint tells us this must be a container. As a side effect, the generation of a fully ground planning problem could be simpler,given the potentially reduced number of action parameters.

Interestingly, a step in this direction was already proposed in PDDL 1, in which some variables were declared as parameters and others as "local" variables inside an operator. However, with no numeric constraints on role fillers or any other type of relation, it is difficult to make use of such variables in a consistent way. Similarly, state-variable representations exploit the uniqueness of a value. However, this was restricted to the case where n_{min} and n_{max} both must be one.

Condition Types The atomic expressions that can be used in preconditions and effects can be divided into two categories. Firstly, there are the explicitly defined relations. These are identical in meaning and use to PDDL and thus, there is no need to discuss these further. Secondly, there are the relations based on role constraints which have the same form as such atoms in states, except that they need not be ground.

Definition 8 (Satisfied Atoms)**.** *Let s be a valid state over objects $c_1, \ldots, c_L$. Then a ground atom a is satisfied in s (denoted $s \models a$) if and only if:*

- *a is of the form $C.r(c_i, c_j)$ and $a \in s$, or*
- *a is of the form $R(c_{i_1}, \ldots, c_{i_R})$ and $a \in s$, or*
- *a is of the form $C.r(c_i, \bot)$ and $vals_s(C.r, c_i) = \emptyset$.*

The first two cases are in line with the standard semantics, whereas the the last case is new and lets us express that no role filler for a given instance exists in a given state. Note that the semantics of atoms that use the symbol `nothing` in any other place than as a role filler are never satisfied in any state.

The above definition can now be used to define when an action is applicable in a state.

Definition 9 (Action Applicability)**.** *Let s be a valid state and act be an action, i.e. a ground instance of an action type with atomic preconditions $p_1, \ldots, p_a$. Then act is applicable in s if and only if every precondition is satisfied in s: $\forall p \in p_1, \ldots, p_a : s \models p$.*

This concludes the semantics of atoms used in preconditions. Atoms used in effects describe how the state of the world changes when an action is applied. This is usually described by the state transition function $\gamma : S \times A \to S$, i.e. it maps a state and an applicable action to a new state. Essentially, γ modifies the given state by deleting some atoms and adding some others. Which atoms are deleted and which are added depends on the effects of the action. If the action is not applicable the function is undefined.

Definition 10 (Effect Atoms)**.** *Let s be a valid state and act be an action that is applicable in s. Then the successor state $\gamma(s, a)$ is computed by:*

1. *deleting all the atoms that are declared as negative effects of the action,*
2. *for every positive effect $C.r(c_i, c_j)$ for role constraint $\langle r, n, C' \rangle$ with $n = [n_{min}, 1]$, if $C.r(c_i, c_k) \in s$ delete this atom, and*
3. *add all the atoms that are declared as positive effects of the action.*

Following this definition allows for a declaration of actions using arbitrary relations and state-variables that may have at most one value. The ontology, more specifically the numeric role constraints can be used to distinguish the two cases.

The symbol `nothing` is not allowed in effects in KEWI. Of course, it would be easy to define the semantics of such a construct as one that retracts all such atoms from the state. However, we have chosen not to go this way in KEWI for two reasons. Firstly, this construct would severely restrict the number of planners that can handle this mass retraction, although it may be possible to express this as a universally quantified effect. Secondly, it is not clear what an example of such an action would be in practise.

Methods

The definition of methods in KEWI is not yet finished. As the framework is at least partially application-driven, we may need to further refine the conceptual framework outlined (but not fully defined) below.

The approach adopted in KEWI follows standard HTN planning concepts: a method describes how a larger task can be broken down in into smaller tasks which, together, accomplish the larger task.

A method is defined by a method name with some parameters. The name usually suggests how something is to be done and the parameters have the same function as in action types; they are the objects that are used or manipulated during the instantiation of a method. Next, a method must declare the task that is accomplished by the method. This is defined by a task name usually describing what is to be done, and again some parameters. For primitive tasks, the task name will be equal to the name of an action type, in which case no further refinement is required. For non-primitive tasks, a method also includes a set of subtasks. In KEWI, the ordering constraints between subtasks are declared with the subtask, rather than as a separate component of the method. This is simply to aid readablility and does not change the expressiveness.

In addition, to these standard components, KEWI allows the specification of high-level effects and subgoal-subtasks. The aim here is to allow for a representation that supports flat, PDDL-like planning domains as well as hierarchical planning domains.

When a method declares that it achieves a high-level effect, then every decomposition of this method must result in an action sequence which will achieve the high-level effect after the last action of the sequence has been completed. This could allow a planner to use a method as if it was an action in a backward search. An alternative view is that such a method functions as a macro action type in the domain.

A method may also include subtasks that are effectively subgoals. For example, the subtask "achieve $C.r(c_i, c_j)$" may be used to state that at the corresponding point in the subtask the condition $C.r(c_i, c_j)$ must hold in the state. The idea being that a planner may revert to flat planning (such as state-space search) to find actions to be inserted into the plan at this point, until the subgoal is achieved.

This mixed approach is not new and has been used in practical planners like O-Plan (Currie and Tate 1991). However, the semantics has not been formally defined for this approach, something we shall attempt in future work.

Export to PDDL

Given that most modern planners accept planning domains and problems in PDDL syntax as their input, one of the goals for KEWI was to provide a mechanism that exports the knowledge in KEWI to PDDL. Of course, this will not include the HTN methods as PDDL does not support hierarchical planning formalisms.

Function Terms The first construct that must be removed from KEWI's representation are the function terms that may be used to refer to objects. In PDDL's preconditions and effects only variables (or symbols) may be used to refer to objects. The following function can be used to eliminate a function term of the form $C.r(t)$ that occurs in an action type O's preconditions or effects.

function eliminate-fterms($C.r(t), O$)
  if is-fterm(t) **then**
    eliminate-fterms(t, O)
  $v \leftarrow$ get-variable($C.r(t), O$)
  replace every $C.r(t)$ in O by v

The function first tests whether the argument to the given function term is itself a function term. If this the case, it has to be eliminated first. This guarantees that, for the remainder of the function t is either a variable or a symbol. We then use the function "get-variable" to identify a suitable variable that can replace the function term. Technically, this function may return a symbol, but the treatment is identical, which is why we shall not distinguish these cases here. The identification of a suitable variable then works as follows.

function get-variable($C.r(v), O$)
  for every positive precondition p of O **do**
    if $p = C.r(v, v')$ **then**
      if is-fterm(v') **then**
        eliminate-fterms(v', O)
      return v'
  retrieve $\langle r, n, C' \rangle$ from C
  add new parameter v' of type C' to O
  add new precondition $C.r(v, v')$ to O
  return v'

This function first searches for an existing, positive precondition that identifies a value for the function. Since function terms may only be used for constraints that have at most one value, there can only be at most one such precondition. If such a precondition exists, its role filler (v', a variable or a symbol) may be used as the result. If no such precondition can be found, the function will create a new one and add it to the operator. To this end, a new parameter must be added to the action type, and to know the type of the variable we need to retrieve the role filler type from the role constraint. In practise, we also use the type name to generate a suitable variable name. Then a new precondition can be added that effectively binds the function to the role filler. And finally, the new variable may be returned.

Handling `nothing` The next construct that needs to be eliminated from the KEWI representation is any precondition that uses the role filler `nothing`. Note that this symbol does not occur in states and thus cannot be bound in traditional PDDL semantics. Simply adding this symbol to the state is not an option since other preconditions that require a specific value could then be unified with this state atom. For example, if we had an explicit atom that stated `holds(k1, nothing)` in our state, then the precondition `holds(?k, ?c)` of the load action type would be unifiable with this atom. An inequality precondition may solve this problem,

but only if the planner can correctly handle inequalities. The alternative approach we have implemented in KEWI is described in the following algorithm.

function eliminate-nothing(O)
  for every precondition $p = C.r(v, \bot)$ **do**
    replace p with $C.r.\ \bot\ (v)$
    if O has an effect $e = C.r(v, v')$
      add another effect $\neg C.r.\ \bot\ (v)$

The basic idea behind this approach is to use a new predicate to keep track of state-variables that have no values in a state. This is the purpose of the new predicate "$C.r.\ \bot$", indicating the role r of concept C has no filler for the given argument. This is a common approach in knowledge engineering for planning domains. For example, in the classic blocks world we find a "holds" relation for when a block is being held, and a predicate "hand-empty" for when no block is held.

The algorithm above uses this technique to replace all preconditions that have `nothing` as a role filler with a different precondition that expresses the non-existence of the role filler. To maintain this condition, it will also be necessary to modify the effects accordingly. This is done by adding the negation of this new predicate to corresponding existing effects.

Since this is pseudo code, the algorithm actually omits a few details, e.g. the declaration of the new predicate in the corresponding section of the PDDL domain, and the fact that the planning problem also needs to be modified to account for the new predicate. Both is fairly straight forward to implement.

State-Variable Updates Finally, the cases in which the value of a state-variable is simply changed needs to be handled. The approach we have adopted here is identical to the approach described in (Ghallab, Nau, and Traverso 2004). That is, when an effect assigns a new value to a state variable, e.g. $C.r(v, v_{new})$, we need to add a precondition to get the old value, e.g. $C.r(v, v_{old})$, and then we can use this value in a new negative effect to retract the old value: $\neg C.r(v, v_{old})$.

Example

In this section we shall look at a single operator taken from the DWR domain and compare its representation in PDDL as defined in (Ghallab, Nau, and Traverso 2004) with the equivalent operator in KEWI. Note that this comparison does not include the representation of the underlying ontology, which is rather trivial in the case of the PDDL version of the domain. However, there is one fundamental difference in the two ontologies, namely, that the PDDL version uses just one symbol `pallet` to denote all the pallets which are at the bottom of each pile. We consider this epistemologically inadequate. To avoid this discussion, we shall look at the move operator here, which does not interact with piles of containers directly, and therefore this ontological difference does not show up. In the PDDL version, the operator looks as follows:

```
(:action move
  :parameters
    (?r - robot ?from ?to - location)
  :precondition (and
    (adjacent ?from ?to)
    (at ?r ?from)
    (not (occupied ?to)))
  :effect (and
    (at ?r ?to)
    (not (occupied ?from))
    (occupied ?to)
    (not (at ?r ?from))))
```

The move operator takes three arguments: the robot to be moved and the two locations involved. There are three preconditions expressing that applicability of this operator depends on the two locations being adjacent, the robot initially being at the location from which the action takes place, and the destination location currently not being occupied. Note that the latter is a negative precondition which cannot be handled by all planners. The effects come in pairs and are somewhat redundant. The robot being at the destination of course implies that this location is now occupied. Similarly, the location that has just been vacated by the robot is now not occupied and the robot is not at that location.

The equivalent operator in KEWI exploits some of the features described above. However, it is important to note that the user interface does not provide a text editor that can be used to modify a PDDL-like representation. Instead, it consists of a web-form with fields for the various components that define an action type. For comparison, we have printed the internal KEWI representation in a Lisp-like syntax, which looks as follows:

```
(:action-type move
    (:arguments ( (?robot robot)
      (?from location) (?to location) ))
    (:precondition (and
      (:relation adjacent
        ( ?from ?to ))
      (:constr location.occupied-by
        ( ?from ?robot ))
      (:constr location.occupied-by
        ( ?to nothing ))))
    (:effect (and
      (:constr location.occupied-by
        ( ?to ?robot ))
      (:constr :not location.occupied-by
        ( ?from ?robot )))))
```

The KEWI version of this operator also requires three parameters. This is because none of the parameters is uniquely implied by any of the others. However, this is the only action in the DWR domain for which this is the case. All actions involving a crane have the location of the crane as another parameter, which would not be required in the KEWI version. The number of preconditions is not reduced either in this case. However, the KEWI version of the operator does not require a negative precondition since it exploits ontological knowledge about the occupancy of locations. Thus, this precondition can be reformulated using the `nothing`

symbol. The biggest difference between the two representations is in the effects, where KEWI only requires one effect for each of the pairs listed above. Again, this is due to the ontology that can be exploited, specifically to the fact that location can only be occupied by one robot at a time. Not having to represent such redundant effects reduces the risk of knowledge engineers forgetting to list such effects. Note that the second effect is negative, but could perhaps be expressed more elegantly by using the `nothing` symbol as in the preconditions. This is not permitted in the syntax at present.

Evaluation

This work is being carried out with an industrial partner, and we are using a real application of knowledge acquisition and engineering in the area of drilling and well control. The development of KEWI is in fact work in progress, and its evaluation is ongoing, and being done in several ways: (i) A drilling engineer is using KEWI, in parallel with the developers, to build up a knowledge base of knowledge about drilling artefacts, operations, procedures etc. (ii) We have created a hand-crafted PDDL domain and problem descriptions of a particular type of problem in drilling - that of executing emergency shut-in procedures. For the same problem area we have generated PDDL automatically from a tool inside KEWI, and we are in the process of comparing the two methods and the PDDL produced. An interface to a drill simulation system is being developed which will help in this aspect. (iii) We are working with another planning project in the same application, which aims to produce natural language explanations and argumentation supporting plans. In the future we believe to combine KEWI with this work, in order that (consistent with involving the user in model creation) the user will be able to better validate the planning operation.

Conclusions

In this paper we have introduced a knowledge engineering tool for building planning domain models, and given a formal account of parts of its structure and tools. Its characteristics are that (i) it has a user-friendly interface which is simple enough to support domain experts in encoding knowledge (ii) it is designed so that it can be used to acquire a range of knowledge, with links to operation via automated translators that create PDDL domain models (iii) it is designed to enable a groups of users to capture, store and maintain knowledge over a period of time, to enable the possibility of knowledge reuse.

Acknowledgements

The research was funded by the UK EPSRC Autonomous and Intelligent Systems Programme (grant no. EP/J011991/1).

References

Ai-Chang, M.; Bresina, J. L.; Charest, L.; Chase, A.; jung Hsu, J. C.; Jónsson, A. K.; Kanefsky, B.; Morris, P. H.; Rajan, K.; Yglesias, J.; Chafin, B. G.; Dias, W. C.; and Maldague, P. F. 2004. Mapgen: Mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems* 19(1):8–12.

Barreiro, J.; Boyce, M.; Do, M.; Frank, J.; Iatauro, M.; Kichkaylo, T.; Morris, P.; Ong, J.; Remolina, E.; and Smith, T. 2012. EUROPA: A platform for AI planning, scheduling, constraint programming, and optimization. In *4th International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*.

Currie, K., and Tate, A. 1991. O-Plan: The open planning architeture. *Artificial Intelligence* 52:49–86.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning*. Morgan Kaufmann.

Leshed, G.; Haber, E. M.; Matthews, T.; and Lau, T. A. 2008. Coscripter: automating & sharing how-to knowledge in the enterprise. In *CHI*, 1719–1728.

McCluskey, T. L., and Kitchin, D. E. 1998. A tool-supported approach to engineering HTN planning models. In *In Proceedings of 10th IEEE International Conference on Tools with Artificial Intelligence*.

Plch, T.; Chomut, M.; Brom, C.; and Barták, R. 2012. Inspect, edit and debug PDDL documents: Simply and efficiently with PDDL studio. *ICAPS12 System Demonstration* 4.

Schreiber, G.; Akkermans, H.; Anjewierden, A.; de Hoog, R.; Shadbolt, N.; de Velde, W. V.; and Wielinga, B. J. 1999. *Knowledge Engineering and Management: The CommonKADS Methodology*. Cambridge, Mass.: MIT Press, 2nd ed. edition.

Simpson, R.; Kitchin, D. E.; and McCluskey, T. 2007. Planning domain definition using gipo. *Knowledge Engineering Review* 22(2):117–134.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The anml language. *Proceedings of ICAPS-08*.

Vaquero, T. S.; Tonaco, R.; Costa, G.; Tonidandel, F.; Silva, J. R.; and Beck, J. C. 2012. itSIMPLE4.0: Enhancing the modeling experience of planning problems. In *System Demonstration – Proceedings of the 22nd International Conference on Automated Planning & Scheduling (ICAPS-12)*.

Vodrážka, J., and Chrpa, L. 2010. Visual design of planning domains. In *KEPS 2010: Workshop on Knowledge Engineering for Planning and Scheduling*.

Wickler, G.; Tate, A.; and Hansberger, J. 2013. Using shared procedural knowledge for virtual collaboration support in emergency response. *IEEE Intelligent Systems* 28(4):9–17.