Domain Knowledge in Planning: Representation and Use

Patrik Haslum
Knowledge Processing Lab
Linköping University
pahas@ida.liu.se

Ulrich Scholz
Intellectics Group
Darmstadt University of Technology
scholz@thispla.net

Abstract

Planning systems rely on knowledge about the problems they have to solve: The problem description and in many cases advice on how to find a solution. This paper is concerned with a third kind of knowledge which we term domain knowledge: Information about the problem that is produced by one component of the planner and used for advice by another. We first distinguish domain knowledge from the problem description and from advice, and argue for the advantages of the explict use of domain knowledge. Then we identify three classes of domain knowledge for which these advantages are most apparent and define a language, DKEL, to represent these classes. DKEL is designed as an extension to PDDL.

Knowledge in Planning

The knowledge input to a planning system may be divided in two distinct classes: problem specification and advice. The problem specification in turn typically consists of two parts: (1) a description of the means at the planners disposal, such as the possible actions that may be taken and resources that may be consumed, and (2) the goals to be achieved, including possibly a measure that should be optimized, constraints that should never be violated, and so on. Advice we term knowledge, of all kinds, intended to help the planner find a better solution, find it more quickly or even to find a solution at all.

There is often a certain difficulty in distinguishing the two, particularly since the same kind of knowledge, indeed the very same statement, may sometimes play one role and at other times another: *e.g.* constraints may be part of a problem specification, but there are also several planners that accept advice formulated as constraints. Nevertheless, two things always distinguish advice from the problem specification:

First, the problem specification defines what is a solution, advice does not. It may well be possible to find good solutions while ignoring, or even acting in conflict with, the given advice, and conversely, heeding poor advice may cause a planner to fail to find a solution even though one exists. It is, however, obviously never possible to find a solution in violation of the problem specification.

Second, the problem specification is, at least in theory, independent of the planning system used, or even of the fact that an automated planner is being used at all (apart from the fact that the specification must be expressed in a format understandable by the planner). What constitutes useful advice, by contrast, tends to be highly dependent on the type of planning system used.

Languages for Specification and Advice

Any automated planning system needs a means of accepting as input a problem specification, and in most cases this means is language. Consequently, many different planning problem specification languages, with a varying degree of similarity, have been used, but recently, PDDL (McDermott *et al.* 1998; Bacchus 2000; Fox & Long 2002b) has emerged as a kind of de facto standard. On a "specification vs. advice" scale, PDDL is strongly oriented towards specification, and even as a specification language, it has its shortcomings: there is for example no easy way to specify constraints, which, as mentioned above, may be an important part of a problem. To combat these shortcomings, several extensions of PDDL (or PDDL-like languages) have been proposed: PDDL2.1 (Fox & Long 2002b) adds the ability to express temporal and metric properties of actions as well as metric goals. PCDL (Baioletti, Marcugini, & Milani 1998) extends PDDL with a constraint vocabulary, which is then "compiled away" into standard PDDL. Many planners have added their own specific extensions, *e.g.* for constraints (Huang, Selman, & Kautz 1999) or invariants (Refanidis & Vlahavas 2001), and many use altogether different languages, *e.g.* to allow the expression of non-determinism (Bertoli *et al.* 2001) or of more elaborate action and resource models (Chien *et al.* 2000).

Languages for expressing plan constraints, whether they be specification or advice, tend to be quite closely related to the kind of planning algorithm used. Examples include Hierarchical Task Network (HTN) schemas, which have a long tradition as a means of expressing plan constraints (Tate 1977; Wilkins 1990; Nau *et al.* 1999; Wilkins & desJardins 2000), and more recently different temporal logics, as in *e.g.* TLPlan (Bacchus & Kabanza 2000) and TALplanner (Kvarnstrom & Doherty 2001).

Planners capable of accepting as input control knowledge of other kinds also use mostly specific languages. This is a natural consequence of the fact that the knowledge itself tends to be highly planner-specific.

Domain Knowledge

In between specification and advice, a third class of knowledge, commonly called *domain knowledge*, may be distinguished. Briefly, it consists of statements about a planning problem that are logically implied by the problem specification, but that are not part of the specification. We would like to amend this definition with the requirement that domain knowledge is "planner independent", *i.e.* not closely tied to the internal workings of any particular planning system, but such a requirement is difficult to formulate precisely.

There are several good reasons for making this distinction. First of all, domain knowledge is implied by the problem specification, so it can be derived from same, and in many cases derived automatically. In this way it is different from advice, which must be provided by a domain expert, or learned from experience over many similar problems. There is a large, and growing, body of work on automatic "domain analysis" of this kind.

Furthermore, good planner advice tends to depend on knowledge both about properties of the problem and the planner used to solve it. For a given planner, there is often a fairly direct mapping from certain classes of domain knowledge to useful advice for that planner. To take a simple example, in a regression planner an obvious use of state invariants is to cut branches of the search tree that violate an invariant. This is a sound principle, since a state that violates an invariant can never be reached and thus a goal set that violates the invariant is unreachable. The principle is founded on knowledge of how the planner works, but depends also on the existence of a certain kind of domain knowledge, namely state invariants.

On the other hand, domain knowledge in itself does not determine its use for advice. To continue the example, state invariants have many more uses: the MIPS planner uses them to find efficient state encodings (Edelkamp & Helmert 1999) and to find abstractions for generating heuristics (Edelkamp 2001), while in GRT (Refanidis & Vlahavas 2001) they are used to split the problem into parts and to improve the heuristic. In principle, the same planner can use the same domain knowledge in different, even mutually exclusive, ways.

For many classes of domain knowledge there exists algorithmic means of generating such knowledge, and indeed many planners do produce and make use of it: GRT, MIPS and STAN (Long & Fox 1999) use state invariants, FF (Hoffmann & Nebel 2001) uses goal orderings (Koehler & Hoffmann 2000), and IPP uses irrelevance information (Nebel, Dimopoulos, & Koehler 1997; Koehler 1999).

In these examples, the algorithms for generating domain knowledge can be, at least in principle, separated from the planning algorithm where it is used, but for practical reasons, the two are built together as one unit. We believe that making a practice out of this separation is good idea, as it enables "fast prototyping" of integrated planning systems, where existing implementations of different domain analysis techniques can easily be "chained" and coupled to existing planners. Although this is not necessary for building high performance planning systems, it would simplify the experimental evaluation of the impact of domain analysis on different planners, and thereby further the development of both automatic domain analysis and more flexible planners.

A Language for Domain Knowledge

In order to separate the generation and use of domain knowledge, we need some means of exchanging this knowledge between producer and consumer. What we propose is to "standardize" the expression of domain knowledge, using a language that builds on PDDL, to make this exchange as natural and easy as the passing of a problem specification to a planner. In short, what PDDL has done for planning problem specification, we wish to do for domain knowledge.

To this end, we have created the Domain Knowledge Exchange Language (DKEL). The language is an extension of PDDL and provides a means for items of domain knowledge to be stated as part of a PDDL domain or problem specification. The main goal of DKEL is to enable the kind of quick and easy prototyping of integrated planning systems outlined above. At the same time, it provides a limited taxonomy of different kinds of domain knowledge, with an attempt at a rigorous definition of the semantics for each kind.

DKEL is currently limited to a few classes of domain knowledge (described in the next section). We have selected these classes because they are reasonably well understood and obviously useful to planners of different kinds, but most of all because there exist domain analysis tools able to generate them.

Given that there already exists many formalisms for the specification of so-called "knowledge rich" planning problems, it is reasonable to ask why we propose yet another. The reply would be that DKEL fills a different niche: the kinds of knowledge expressible in DKEL are different from those expressible by constraint languages such as HTN schemas and temporal logics. In short, DKEL is a complement to, and not a replacement for, existing languages.

Implications of Explicit Domain Knowledge

DKEL augments the original domain or problem description with domain knowledge rather than altering or reducing it right away. Preserving the original structure of the domain and problem specification has several advantages: First of all, it is a prerequisite for the "chaining" of several analysis techniques described above. It also leaves the choice of what knowledge to apply, and how to apply it, up to the planner. As mentioned, turning domain knowledge into effective advice for a specific planner depends on knowledge of the workings of that planner, and including this in the exchange language would blur the separation between the generation and use of knowledge that it is meant to help achieve.

There are also problems with the use of domain knowledge. Not all knowledge is useful to all planners, and may even be detrimental if incorrectly used. Even if a particular item of knowledge is useful, the computational cost of inferring it may be higher than the benefit incurred by its use. Adding explicit domain knowledge to a problem specification increases the size of the specification, and although a planner may always choose to ignore useless items of knowledge, indiscriminate adding-on may blow specifications up to a size where the increased cost of simply reading and handling them outweighs any advantage. Note, however, that these problems are not intrinsic to explicit representations of domain knowledge, but only more prominent for them: In an integrated planning system, domain analysis algorithms can be customized to closely match needs, while explicit representations are intended for use with prefabricated, general tools.

Separating domain analysis from its use in planning means that the planner component loses control over how knowledge is generated, and must simply accept it as stated. Knowledge expressed in an interchange format like DKEL may have been added by the domain designer instead of being discovered by automatic analysis. Regardless of origin, it may be incorrect due to a flawed analysis, differences in the interpretation of knowledge statements, or even sheer malice. However, adopting an explicit representation for domain knowledge, such as DKEL, may also help in avoiding problems of the kinds mentioned before. It offers a human-readable intermediate format with a well-defined semantics, and encourages empirical evaluation of planning tools. Ultimately, the decision what domain knowledge generating components to couple to a specific planner still belongs with the designer of the integrated planning system.

Demarcating Domain Knowledge: Scope of DKEL

Distinguishing domain knowledge from other forms of knowledge, and thus finding the right scope for DKEL, is not easy. For example, it is not entirely clear where domain analysis ends and planning begins: Heuristic state evaluations done by a planner such as FF falls under our definition of domain knowledge as "statements logically implied by the problem specification", but we would not consider it such because what meaning, and relevance, would this information have to any other planner? Conversely, statements that are in fact not domain knowledge may appear syntactically indistinguishable from statements that are. For example, to a regression planner that uses state constraints to prune unreachable states from the search, advice to prune reachable but undesirable states could be given in exactly the same form. Only the fact that these constraints are not implied by the problem specification makes it advice rather than domain knowledge.

Another point is that domain knowledge is defined with respect to a problem instance, but what we really want to do is state knowledge about a planning domain, *i.e.* about all problem instances belonging to the domain. Because the concept of "domain" in PDDL is rather weak[1], we must in doing this exclude "unreasonable" problem instances. The semantics of DKEL statements, and language features such as `:context`, have been made with this in mind.

[1]The domain can for instance not specify the existence of any particular object, or sanity constraints on the initial state, nor restrictions on the goal. The first PDDL specification (McDermott *et al.* 1998) had some features along these lines, *e.g.* `:constants` and `situations` but they never gained widespread use.

Meta Knowledge

The semantics of DKEL statements are carefully and exactly defined, but they are in a sense not complete. For example, the meaning of the `:irrelevant` clause for actions is roughly "if there exists a plan, there also exists a plan that does not contain the irrelevant action", but this does not say whether the plan not containing the irrelevant action is of the same length (or cost according to the problem metric). Given two statements about action irrelevance, it is also not clear whether they can both be applied at the same time, or if doing so will make the problem unsolvable altogether. Another example is the `:replaceable` clause, which states that any occurrence of a particular action sequence can be replaced by a different action sequence, in any valid plan, but it does not specify if the replacement is valid in the presence of another action sequence in parallel,

These uncertainties could be resolved by adopting a stricter semantics for the various DKEL statements, but this would be likely to make the whole language too restricted to be useful. At the same time, most domain analyzers can be much more specific about properties of the domain knowledge they produce. For example, all `:irrelevant` action clauses produced by RedOp (Haslum & Jonsson 2000) can be safely used together, and some of them are also guaranteed not to increase the length (serial or parallel) of the plan.

We call this knowledge about properties of particular items of domain knowledge "meta knowledge" and ideally, we would like to be able to express it alongside domain knowledge in DKEL. However, what kinds and forms of meta knowledge are relevant is not clear to us, and therefore, at the moment, DKEL supports it only via "tags": domain knowledge items may be annotated with arbitrary symbols, intended to express such properties. A sketch of an ontology for meta knowledge is given in section "Current Form and Future Development" below.

Classes of Domain Knowledge

This section presents definitions of the semantics of three different classes of domain knowledge: state invariants, fact and action irrelevance, and replaceability of action sequences. These are the classes that can be expressed in DKEL. They represent by no means an exhaustive classification of domain knowledge, but together they cover a large part of the domain knowledge that is made explicit by existing automatic analysis techniques.

In defining the meaning of knowledge clauses, we consider for the most part plans to have the simple form of a sequence of atomic actions, although in some cases, *e.g.* state invariants, the meaning of a domain knowledge statement remains unchanged when slightly more complicated plan forms, such as partially ordered sets of actions, are used.

State Invariants

State invariants are probably the most commonly produced and used class of domain knowledge. They express properties of a planning domain that are invariant under action application, *e.g.* the uniqueness of a physical location of an object. State invariants in planning are commonly defined as a formula F on states such that if F is true in the initial state of a planning problem, F is true in all reachable states. The following is a typical example, taken from the blocksworld domain[2], which states that a block is either on the table or on exactly one other block:

```
(forall (?x) (and
 (or (clear ?x) (exists (?y)
     (implies (not (= ?x ?y)) (on ?x ?y))))
 (forall (?y ?z)
   (implies (and
     (not (= ?x ?y)) (not (= ?x ?z))
     (on ?x ?y) (on ?x ?z)) (= ?y ?z)))
 (not (exists (?y) (implies (not (= ?x ?y))
     (and (on-table ?x) (on ?x ?y)))))))
```

The formula is best explained in the terminology of TIM, see (Fox & Long 1998), p. 386f. The first of the three outer conjuncts corresponds to a state membership invariant, meaning that in every state and for every block `?x` at least one of `(on-table ?x)` or `(on ?x ?y)` is true, where `?y` is different block than `?x`. Likewise, the second and the third conjunct correspond to a identity and a uniqueness invariant, respectively. The second denotes that a block is on top of at most one other block and the third that a block is never simultaneously on top of another block and on the table. Then, for a problem with two blocks `A` and `B`, this formula specifies that exactly one of the facts `(on-table A)` and `(on A B)`, and analogously exactly one of the facts `(on-table B)` and `(on B A)`, is true (provided this was the case in the initial state).

We will generalize the above definition of state invariants slightly. First, we simply drop the reference to the initial state. As any state can be the initial state of a planning problem, an invariant according to the first definition is useful even if it becomes true in an intermediate state of a plan instead of in the initial state. Second, we consider invariants also on pairs of adjacent states. This allows us to express monotonicity properties on transitions among states. Although this extension may seem complicated, it fits naturally into the framework of DKEL.

Hence, our definition of a state invariant is (1) a formula F on states such that if F is true in a state s, F is true in all states reachable from s by application of a sequence of actions. In addition, a state invariant may be (2) a pair F_1, F_2 of a formula on states and a formula on pairs of states, respectively, such that if F_1 is true in a state s then F_2 is true for all pairs (s, s') where s' is reachable from s by the application of a single action or a set of non-conflicting actions in parallel.

With this definition we can formulate state invariants for a planning domain independently of any particular problem, even though their applicability clearly depends on the initial state of the problem. For example, intuition says that all the blocksworld state invariants given above are properties of the blocksworld domain, but it is easy to define problems whose initial state violates them. Our definition simply says that because such an initial state falsifies the antecedents of the state invariants, there is nothing said about the following state.

Also note that an invariant according to (1) remains an invariant, in the intuitive sense, also if the plan is parallel or partially ordered, if one makes the common assumption that the result of executing such a plan is the same as that of executing one of its linearizations. The definition does, however, not guarantee that the invariant formula holds during the execution of each action in the plan. In PDDL2.1, it is possible to specify effects at different time points in the execution of an action, and thus an action may falsify an invariant formula at the start but restore the truth of the formula at its end.

State invariants are explicitly and implicitly used by a variety of planners, among which are SATplan (Kautz & Selman 1992), STAN, GRT, and MIPS. A similar variety of tools calculate invariants from domain and problem descriptions, *e.g.* TIM, Discoplan (Gerevini & Schubert 1998), and a technique by Rintanen (2000).

Operator and Predicate Irrelevance

The difficulty of solving a planning problem increases, frequently exponentially, with the size of the problem specification. Unfortunately, for most planners it makes small difference how much of the specification is actually relevant for solving the problem goals. The larger and more complex problems get, the more likely is the presence of irrelevance (most of the real world is irrelevant for any of our tasks) and the greater is the cost of not realizing it. It is fair to say that the identification and efficient treatment of irrelevance is one of the key issues in building scalable planners.

As important as we consider the treatment of irrelevance to be, as difficult it is to define precisely. Nebel *et al.* (1997) identify three different kinds of irrelevance: (1) a fact or action is *completely irrelevant* if it is never part of any solution. This is a very weak criterion, since a plan can always contain redundant steps that contribute nothing to the achievement of the problem goals but make use of otherwise irrelevant facts or actions. Unreachable actions are of course completely irrelevant. (2) An initial fact or an action is *solution irrelevant* if its removal from the specification does not affect the existence of solution, and (3) an initial fact or an action is *solution-length irrelevant* if its removal does not affect the length of the shortest solution plan. This can obviously be generalized to any conceivable cost measure on plans.

We adopt solution irrelevance as the basis for our definition, since it seems the most intuitive and least complicated. Solution-cost preserving irrelevance is an important concept, but because of the unlimited number of measures, we relegate this property to meta knowledge. Thus we say that an action a is irrelevant if removing a from the set of actions available to the planner does not alter the existence of a solution. In other words, if there exists a plan, then there also exists a plan that does not contain the irrelevant action.

Concerning facts, the situation is more complicated, since there are several possible interpretations of what it means to "remove" a fact from the problem. Removing an "initial

[2]We use blocksworld as example domain throughout this paper. The blocksworld domain is simple, widely known, and allows the formulation of a wide variety of domain knowledge.

fact", *i.e.* one that is true in the initial state, can be simply defined to mean making its value in the initial state false (or unknown) instead. For facts that are not initial there is no such obvious interpretation, since removing a fact from the problem completely may have undesired side effects: If the removed fact appears as a precondition, an action may become applicable in a state where it was not applicable before, and thus the simplified problem may have a solution that is not a solution to the original problem.

Because of this, we choose only a simple definition of "initial fact irrelevance": A fact is *initial-irrelevant* if its truth value in the initial state does not affect solution existence.

Action irrelevance is usable by practically every planner, since its effect is only to reduce the size of the problem. This is particularly important for planners that work with an instantiated representation. Fact initial-irrelevance has been shown to be important for Graphplan and Graphplan-like planners (Nebel, Dimopoulos, & Koehler 1997). Domain analysis tools that produce irrelevance information include RIFO (Nebel, Dimopoulos, & Koehler 1997) and RedOp. RIFO implements several methods of detecting irrelevance, some of which are not guaranteed to be solution-preserving and therefore do not strictly fall within our definition. Still, since the knowledge produced by RIFO has been shown to be very useful in practice, we feel that the lack of a solution-preservation guarantee should be regarded as meta knowledge and indicated by a tag.

Replaceable Sequences of Operators

Planning problems tend to have numerous solutions and many of them are similar. They may differ perhaps only by a reordering of actions that do not interfere with each other, or by the substitution of a different object with identical properties, and recognizing this can improve the efficiency of search since only one of the equivalent sequences have to be considered (Fox & Long 1999; Taylor & Korf 1993). More generally, a sequence of actions may be "subsumed" by a different sequence, in the sense that wherever the first sequence occurs, the second can be substituted. We say that an action sequence T_1 is *replaceable* by an action sequence T_2 if in every executable action sequence containing T_1, replacing T_1 by T_2 also results in an executable sequence, which, in addition, achieves all the goals achieved by the original sequence.

An example of such a pair in the blocksworld domain are T_1 = `(move A B D)`∘`(move A D C)` and T_2 = `(move A B C)`: whenever a block is moved twice in a row, this sequence can be replaced by a single move directly to the destination of the second move. This is also an example where replaceability holds only in one direction, since replacing the second sequence by the first may result in an invalid plan, if `D` is covered by another block.

The replaceability relation is defined with respect to linear plans only: It leaves no guarantee that making the replacement in a plan where there exists actions parallel with the replaced sequence yields a valid plan. For example, if the sequence `(move E C F)`∘`(move G H D)` happens in parallel with T_1, the previous replacement yields a conflict: If `(move A B C)` is placed at the same time as `(move A B D)` then block `C` is still occupied, and if it is placed one step later, block `D` is not freed early enough. Note, however, that if replaceability between two sequences holds in the context of parallel totally ordered plans, it always holds also for linear plans. Thus, knowledge of replaceability as defined above may useful at least as a basis for computing replaceability for other kinds of plans.

Examples of automatically generated replaceability knowledge includes the result of RedOp and the RAS constraint of Scholz (1999). The latter also goes into replaceability for parallel plans. A common use of replaceability is "commutativity pruning", *i.e.* pruning from search all but one permutations of a sequence of commutative actions, used for example by GRT (Refanidis & Vlahavas 2001). An example of a different use is the "Planning by Rewriting" approach (Ambite & Knoblock 2001), although this uses a more elaborate model of replacement and hand-coded knowledge.

Other Classes of Domain Knowledge

Many classes of domain knowledge beside the three detailed above have appeared in the literature. They have all been implemented as part of planning systems, or in some cases as stand-alone tools, and thus are all candidates for future extensions of DKEL. Examples include

Landmarks: A landmark (Porteous, Sebastia, & Hoffman 2001) is a fact that must be achieved at some point in every solution to a planning problem. Different ordering relations on landmarks can be identified and used to prune from search candidate plans that achieve landmarks in violation of the order.

Goal orderings: Goal orderings (Koehler & Hoffmann 2000) allow a divide and conquer approach to planning. A goal ordering for a planning problem consists of two or more ordered subsets of its goals. Instead of planning for all goals at once, a planner can repeatedly search for a plan from one subset to the next, using the goal state of the previous plan as initial state. Then, the overall solution is the concatenation of the plans for the subgoals.

Symmetries: The detection of symmetry can considerably improve the performance of planning systems: If a candidate plan does not yield a solution, there is no use in considering a symmetric candidate. Fox and Long (1999; 2002a) describe how to find symmetries in planning problems.

Generic Types: Fox and Long (2000; 2001) define a generic type as a collection of types, characterized by specific kinds of behaviors, *e.g.* movable objects and lockable doors. Generic types are present in a variety of planning domains and are amenable to the application of specialized techniques. The identification of generic types allows to automatically compose a planner specialized for the planning problem at hand.

Another important class of knowledge in widespread use is general constraints on sequences of states and actions. It is common both as part of a problem specification (although not directly expressible in PDDL) and as a means of expressing advice. There are, however, good reasons why we have chosen not to include it in DKEL: There are already many languages for expressing constraints for these purposes, *e.g.* HTN schemas, temporal logic, and more. Such languages also tend to be highly expressive and quite complex. Even though constraints on action and state sequences can constitute domain knowledge, their main use is as either part of the specification, or as advice founded on the intuition of the domain designer. Also, there exists very few domain analysis tools that automatically discover knowledge of this kind.

The Domain Knowledge Exchange Language

This section describes how the three classes of domain knowledge detailed in the previous section are expressed in DKEL.

DKEL Design Principles

Our main goal in the design of DKEL has been to make a language that is useful in practice. In short, it should be simple, extendible, and as familiar as possible.

The most important design principle is simplicity, which does not only apply to the language definition but also to its intended use. In other words, we tried to keep things simple and ask the users of DKEL to do the same. On the other hand, the expressiveness of the language should be adapted to current (and, as far as possible, future) use, which motivates our restriction to three common classes of domain knowledge. In a trade-off with simplicity, we introduce a certain amount of "syntactic sugar", *i.e.* abbreviations for some common cases, for example the :set-constraint.

Finally, DKEL is designed as an extension of PDDL, so we expect the user to be familiar with this language. For this reason, we tried to keep DKEL as close to PDDL as possible and share some of the syntax with this language. For elements DKEL which are not described in this paper, please refer to the PDDL subset used in the AIPS 2000 Planning Competition (Bacchus 2000).

Stating Domain Knowledge in DKEL

DKEL clauses can be placed within either a domain, situation, or problem definition. Each location yields a different scope for the clause: If placed within a domain definition, a DKEL clause is valid for all problems of this domain. Analogously, a DKEL clause within a situation and a problem definition is valid only for problems that have the specified initial state and the specific problem, respectively. Note that the semantics of some DKEL clauses, *e.g.* state invariants, and the :context feature of the language (see below) allows domain descriptions to contain domain knowledge that is problem dependent to some extent.

DKEL clauses have the form of a list beginning with an identifier. Elements within a clause, like the elements of an action definition, consist of a keyword followed by some "content" in the form of a LISP expression, *i.e.* a single symbol or a list with balanced parentheses. The basic form of a DKEL clause is:

```
(<KNOWLEDGE_KIND> <ELEMENT>)

<KNOWLEDGE_KIND> ::=
 :replaceable | :irrelevant | :invariant

<ELEMENT> ::=
 [:tag <name>]*
 [:vars (<TYPED?-LIST-OF(VARIABLE)>)
   [:context <CONTEXT_FORMULA>] ]
 <CONTENT>+
```

Elements common to all clauses are :tag, :vars, :context, and <CONTENT>. The first allows a limited amount of meta knowledge, in the form of an arbitrary symbol, to be associated with the clause. Note that a clause may have more than one :tag element. Writing several instances of content within the same DKEL clause is equivalent to writing one clause with the same :tag, :vars, and :context for each of them.

Variables on the <ELEMENT> level act as universally quantified parameters to the content of the clause, allowing several instances of a domain knowledge item to be written in a single statement. The :context clause limits the possible instantiations of these variables. Thus, writing a DKEL clause with parameters is equivalent to writing one ground instance of the clause for each assignment of the variables that satisfies the context formula. For example, consider the :invariant clause in the next subsection: In a blocksworld problem with three blocks A, B, and C, it denotes three state invariants, one for each binding of ?x to a block.

The context formula is required to be "static", *i.e.* evaluable without reference to a particular state. This makes it possible (but not necessary) to convert all DKEL clauses to a set of ground instances in a preprocessing step. The restriction is reasonable, since for none of the classes of domain knowledge currently expressible in DKEL does validity depend on the state, but it may have to be lifted in the future if DKEL is extended to other kinds of domain knowledge.

To allow knowledge items in the domain definition to depend on properties of the problem instance, a context formula may contain two kinds of modal literals: (:init <literal(t)>) and (:goal <literal(t)>). They refer to the truth value of the literal in the initial and goal state of the problem, respectively.

Since even simple conjunctive goals in PDDL do no specify a complete state, there is a question of how to interpret negative :goal literals: Does (:goal (not <ATOM>)) mean "it is a goal that <ATOM> should be false", or does it mean "it is not a goal that <ATOM> should be true"? The most straightforward and general interpretation, and the one we choose for DKEL, is that (:goal <literal>) is true if and only if <literal> is entailed by the goal formula of the problem, even though this does make it more difficult to handle problems with complex goal formulas (see *e.g.* Kvarnström and Doherty (2001), Section 3.4, for a more detailed discussion). Consequently, the second possible interpretation suggested above is expressible as (not (:goal <ATOM>)).

State Invariants

An `:invariant` clause specifies a state invariant as first-order formula or as a set constraint. As defined in the previous section, this means that the given property is preserved by all operators. It does not necessarily mean it is true in every reachable state: Only if the `:invariant` clause stands within a `situation` or `problem` definition the property is required to be true in the initial state.

The syntax of an `:invariant` clause is as follows:

```
<CONTENT> ::=
   :formula <FORMULA>
 | :set-constraint (<CONSTRAINT_TYPE>
      <INTEGER> <LITERAL_SET>+)

<SET_CONSTRAINT> ::=
   exactly | at-most | at-least
 | decreasing | increasing

<LITERAL_SET> ::=
   <LITERAL(<TERM>)>
 | (:setof
     [:vars (<TYPED?-LIST-OF(VARIABLE)>)
       [:context <CONTEXT_FORMULA>] ]
    <LITERAL(<TERM>)>)
```

The motivation for introducing set constraints is to simplify the writing of common types of invariants. A set constraint specifies the literal set as a union of <LITERAL_SET>, each of which can either be a single literal or an instance of the `(setof VARS CONTEXT LITERAL)` construct. The latter denotes the set of literals entailed by the (closed) formula

```
(forall (VARS) (implies CONTEXT LITERAL)),
```

like the literals entailed by an action precondition. For example, the invariant given in the previous section may be expressed as follows using a set constraint:

```
(:invariant
 :vars (?x - block)
 :set-constraint (exactly 1
     (on-table ?x)
     (setof :vars (?y - block)
            :context (not (= ?x ?y))
       (on ?x ?y))))
```

The `setof` clause corresponds to the formula `(forall (?y) (implies (not (= ?x ?y)) (on ?x ?y)))`, where `?x` has already been bound on the <ELEMENT> level. In a blocksworld problem with blocks `A`, `B`, and `C`, if `?x` is bound to `A`, the formula denotes the set {`(on A B)`, `(on A C)`}.

The set constraint abbreviation is provided mainly because the corresponding first-order formulas quickly become very large: Imagine a blocksworld domain extended to have n tables, so that a block `?x` could be `(on-table1 ?x)`, `(on-table2 ?x)`, and so on. In this case, we need only to replace `(on-table ?x)` by the n new predicate schemata in the DKEL clause above, while formulating the same invariant in first-order logic requires a formula quadratic in size.

As it turns out, set constraints are well suited to express many of the invariants found by current analysis techniques. For example, the invariants found by TIM (identity, state membership, uniqueness, and fixed resource) and most of those found by Discoplan (implicative, single-valuedness, antisymmetry, OR, and XOR) all correspond to set constraints. Consider the following Discoplan XOR-constraint:

```
((XOR (ON ?X ?Y) (ON-TABLE ?X)) (BLOCK ?X))
```

Here, `?X` is universally quantified, `?Y` existentially quantified and `(BLOCK ?X)` is a supplementary condition that has to be true in the initial state. Hence, the constraint reads: "In every reachable state it holds that for all `?X` such that `(BLOCK ?X)` is true in the initial state, either there is a `?Y` such that `(ON ?X ?Y)` is true or `(ON-TABLE ?X)` is true". The one-to-one corresponding DKEL invariant is

```
(:invariant
 :vars (?x)
 :context (:init (block ?x))
 :set-constraint (exactly 1
      (on-table ?x)
      (setof :vars (?y) (on ?x ?y))))
```

Of course, set constraints can only describe a limited class of invariant properties, but for remaining invariants we can always resort to first-order formulas.

The semantics of set constraints are as follows: The constraints `exactly`, `at-most`, and `at-least` denote that exactly n, at most n, and at least n of the literals in the given set are true in a state, respectively. In TIM terminology, an `at-most` set constraint is the conjunction of the corresponding identity and uniqueness invariants, limited to the variable bindings that satisfy the context. Likewise, an `at-least` set constraint is the conjunction of the corresponding state membership and uniqueness invariants, again limited by the context. An `exactly` set constraint is the conjunction of the corresponding `at-most` and `at-least` set constraints.

The `decreasing` and `increasing` constraints are examples of the second type of invariants defined in the previous section, *i.e.* invariant properties on pairs of adjacent states. A `decreasing` (`increasing`) set constraint means that at most n (at least n) of the literals in the set are true in a state and that in any succeeding state, the number of true literals is the same or less (more). We give its semantics in first-order logic by quantifying TIM invariants over states. Then the first invariant formula $F_1(s)$ of `decreasing` is `(at-most i s)`, where the extra argument denotes the state that the invariant holds in. Formula $F_2(s, s')$ is a conjunction of $k+1$ implications `(implies((exactly i s) (at-most i s')))`, one for each i in the range $0 \leq i \leq k$. Here, s and s' denote adjacent states. The meaning of the `increasing` constraint may be expressed by a similar pair of formulas, with an upper limit given by the size of the fact set.

Operator and Predicate Irrelevance

The `:irrelevant` knowledge clause allows irrelevance information to be stated as part of the domain description instead of removing the irrelevant operator or predicate instances directly, thus preserving more of the original domain structure. The syntax is as follows:

```
<CONTENT> ::=
    :fact <ATOMIC-FORMULA(<TERM>)>
  | :action <OP_SCHEMA>
```

Any variables appearing in either fact or action schema should appear also in the `:vars` element of the clause.

If the clause contains an operator schema, the meaning is that any matching instance of that operator is solution-irrelevant, as defined in the previous section, *i.e.* if there exists a plan which contains such an action, there also exists a plan that does not. A predicate schema indicates that instances of this predicate are initial-irrelevant, as defined in the previous section, *i.e.* those instances can be removed from the initial state of the problem without affecting solution existence. The following is an example from the blocksworld domain:

```
(:irrelevant
 :vars (?x ?y ?z - block)
 :context (not (:goal (on ?x ?z)) )
 :action (move ?x ?y ?z))
```

It states that unless `(on ?x ?z)` is a goal, any instance of the `move` operator that places `?x` on `?z` is irrelevant.

Replaceable Sequences of Operators

A `:replaceable` clause specifies replaceability of operator sequences in the context of linear plans. The syntax of a `:replaceable` clause is as follows:

```
<CONTENT> ::=
 :replaced <OP_SEQUENCE_SCHEMA>
 :replacing <OP_SEQUENCE_SCHEMA>

<OP_SEQUENCE_SCHEMA> ::= (<OP_SCHEMA>*)

<OP_SCHEMA> ::= (<name> <TERM>*)
```

An example of a `:replaceable` clause from the blocksworld domain is the following:

```
(:replaceable
 :vars (?x ?y ?z - block)
 :replaced  ((move-from-table ?x ?y)
             (move-onto-table ?x ?y))
 :replacing ())
```

It states that it is always possible to replace the sequence of moving a block from the table onto a block and immediately back onto the table by the empty sequence. In other words, this subsequence can be removed from any solution plan.

Current Form and Future Development

DKEL, as presented in this paper, is a first step, not a final solution. It is the nature of a first step that there might be discussions about its direction. Specification languages for planning problems have evolved over many years, and PDDL is still undergoing development.

In the following, we identify some of the weaknesses DKEL currently exhibits, and discuss future developments to remedy those.

Coverage

DKEL does not offer a representation for every conceivable item of interesting domain knowledge. In fact, even the classification outlined in this paper does not cover all the kinds of domain knowledge that have been discussed in planning literature and used in planners up to now. The main reason why we have left it in such an unfinished state is that we believe the construction of a complete ontology of domain knowledge, and a matching representation, must be a project for the planning community, not only because of the scale of such a project but more importantly because an interlingua such as DKEL is intended to be is useless unless it is accepted by a large part of the community.

This said, we also think that DKEL, as presented here, is an adequate first step towards a more comprehensive representation. The three classes of knowledge it does cover have been selected as a starting point because they are fairly well understood and useful to a wide variety of planners, and because there exist techniques to automatically derive them from problem descriptions. In a sense, the language is a "snapshot" of the state of the art in domain analysis. As work in this area continues, we expect more kinds of domain knowledge fulfill these criteria, and we hope that they will also be incorporated into DKEL.

Finally, although DKEL is designed as an extension of PDDL, there is no reason to believe that similar extensions to other formalisms for specifying planning problems should not be of use: compared to constraint languages, such as *e.g.* HTN schemas, DKEL plays a different, and complementary, role.

Meta Knowledge

Neither have we provided a syntax or an ontology of the properties of items of domain knowledge which we have referred to as meta knowledge. Examples of such properties that may be important include:

Assumptions about domain, problem and plan. The validity of action sequence replaceability may depend on the assumption that the plan is linear, but instances of the replaceability relation may be valid also in the context of parallel or temporal plans. In a temporal planning domain, invariant and replaceability knowledge may also depend on exactly what action execution semantics are assumed.

Effects of applying domain transformations. Irrelevance and replaceability knowledge both describe (potential) changes to the planning domain and problem: as defined, these changes are guaranteed to preserve solution existence, but other properties of the solution, *e.g.* optimality with respect to number of actions, makespan, or the problem-defined metric, are not guaranteed to be preserved.

Compatibility and synergy. As already pointed out, action irrelevance statements may be mutually exclusive, in the sense that applying one such statement (by removing the action or actions from the domain) renders the other invalid. Less obviously, there may be synergy effects between domain knowledge items: For example, there may be state invariants not valid for the original domain and problem that become valid if a particular action replacement or irrelevance statement is consistently applied.

Origin and dependencies. With the proper meta knowledge attached, the origin of a particular item of domain knowledge is of no importance. However, as long as there is no detailed ontology of meta knowledge, it might be necessary to know what program produced the knowledge (and with what options), what other items of domain knowledge were used to derive it, and so on.

This, however, is merely a sketch, which may prove inadequate if DKEL is extended to cover more classes of domain knowledge. Although a need for a more structured classification of meta knowledge is sure to develop if DKEL becomes used in wider circles, such widespread use is also a prerequisite for designing an ontology that properly addresses that need.

Current Use of DKEL

Currently, there are two domain analysis tools that produce state invariants in DKEL form: version 2.0 of Discoplan[3] and TIM_dkel, a reimplementation of TIM.

RedOp[4] identifies actions that can be replaced by action sequences. This knowledge can be output in DKEL, either as `:irrelevant` or `:replaceable` statements.

One of the main goals of DKEL is to enable fast and easy prototyping of integrated planning systems built from existing preprocessing techniques and planners. Varrentrapp *et al.* (2002) demonstrate this with an on-line testbed for planning systems.[5]. Part of the testbed is a reimplementation of GRT that accepts DKEL invariants. Here it is also possible to download TIM_dkel.

DKEL, in its current form, has been subjected to relatively little in the way of evaluation. How does one evaluate a language, especially a language targeted at the role we have in mind for DKEL? While expressivity can be formally analyzed and compared, again, we believe the most important metric of the value of DKEL is acceptance.

Conclusions

Domain knowledge is an important resource for automated planners: It can be extracted automatically from the domain and problem specification by a variety of techniques, and in combination with knowledge of the workings of a planner it can be turned into effective advice for reducing search effort or improving the quality of plans found. The language DKEL has been conceived and designed as means for allowing easy integration of domain analyzers and planners in a flexible way. In a sense, this reduces the effort devoted to inventing efficient domain and problem specifications in exchange for finding a combination of tools and planner that efficiently solves the problem.

The explicit representation of domain knowledge also has other uses. For example, it opens up the possibility of reasoning about the planning process. An example of this is the planner HAP (Vrakas, Tsoumakas, & Vlahavas 2002), whose planning strategy is adjusted according to the existence and characteristic of domain properties. Statements of domain knowledge, *e.g.* a state invariant, are regarded as property of the corresponding domain, similar to details like the number of goal facts.

Two things we wish to stress. First, DKEL is aimed at describing a particular kind of knowledge about a planning domain: It is not a substitute for extensions to the expressivity of problem specification languages, or formalisms for “knowledge rich” domain description, it is a complement. Second, it is not final: Although useful in its current form, it will certainly need to be extended to meet future developments in planning and in domain analysis. Ultimately, the goal may be a unified and standardized language for planning problem specification, domain knowledge and planner advice, but it still lies far in the future.

[3] `http://prometeo.ing.unibs.it/discoplan`

[4] `http://www.ida.liu.se/~pahas/hsps/redop.html`

[5] `http://www.intellektik.informatik.tu-darmstadt.de/~planlib/Testbed`

References

Ambite, J. L., and Knoblock, C. A. 2001. Planning by rewriting. *Journal of Artificial Intelligence Research* 15:207–261.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

Bacchus, F. 2000. Subset of PDDL for the AIPS 2000 planning competition. `http://www.cs.toronto.edu/aips2000/pddl-subset.ps`.

Baioletti, M.; Marcugini, S.; and Milani, A. 1998. Encoding planning constraints into partial order planning domains. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 608–616.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proc. 17th International Conference on Artificial Intelligence (IJCAI'01)*, 473 – 486. San Francisco, CA: Morgan Kaufmann Publishers, Inc.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; Smith, B.; Fisher, F.; Barrett, T.; Stebbins, G.; and Tran, D. 2000. ASPEN – automating space mission operations using automated planning and scheduling. In *Proc. 6th International Symposium on Technical Interchange for Space Mission Operations*.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In Fox, M., and Biundo, S., eds., *Proc. 5th European Conference on Planning*, volume 1809 of *Lecture Notes in Artificial Intelligence*, 135–147. New York: Springer.

Edelkamp, S. 2001. Planning with pattern databases. In Cesta, A., and Borrajo, D., eds., *Proc. 6th European Con-*

ference on Planning, Lecture Notes in Artificial Intelligence. New York: Springer.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367–421.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning domains. In Dean, T., ed., *Proc. 15th International Joint Conference on Artificial Intelligence*, 956–961. Stockholm, Sweden: Morgan Kaufmann.

Fox, M., and Long, D. 2001. Hybrid STAN: Identifying and managing combinatorial optimisation sub-problems in planning. In Nebel, B., ed., *Proc. 16th International Joint Conference on Artificial Intelligence*, 445–452. Seattle, USA: Morgan Kaufmann.

Fox, M., and Long, D. 2002a. Extending the exploitation of symmetry analysis in planning. In Ghallab, M.; Hertzberg, J.; and Traverso, P., eds., *Proc. 6th Conference on Artificial Intelligence Planning & Scheduling*, 161–170.

Fox, M., and Long, D. 2002b. PDDL2.1: An extension to PDDL for expressing temporal planning domains. `http://www.dur.ac.uk/d.p.long/pddl2.ps.gz`.

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proc. 15th National Conference on Artificial Intelligence*, 905–912. Madison, USA: AAAI Press/MIT Press.

Haslum, P., and Jonsson, P. 2000. Planning with reduced operator sets. In Chien, S.; Kambhampati, S.; and Knoblock, C. A., eds., *Proc. 5th Conference on Artificial Intelligence Planning & Scheduling*, 150–158. Breckenridge, USA: AAAI Press.

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

Huang, Y.-C.; Selman, B.; and Kautz, H. 1999. Control knowledge in planning: Benefits and tradeoffs. In *Proc. 16th National Conference on Artificial Intelligence (AAAI'99)*, 511–517. Orlando, USA: AAAI Press/MIT Press.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In Neumann, B., ed., *Proc. 10th European Conference on Artificial Intelligence*, 359–363. Vieanna, Austria: John Wiley & Sons, Ltd.

Koehler, J., and Hoffmann, J. 2000. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research* 12:338–386.

Koehler, J. 1999. RIFO within IPP. Technical Report 126, Institute for Computer Science, University Freiburg.

Kvarnstrom, J., and Doherty, P. 2001. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence* 30(1):119 – 169.

Long, D., and Fox, M. 1999. Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research* 10:87–115.

Long, D., and Fox, M. 2000. Recognizing and exploiting generic types in planning domains. In Chien, S.; Kambhampati, S.; and Knoblock, C. A., eds., *Proc. 5th Conference on Artificial Intelligence Planning & Scheduling*, 196 – 205. Breckenridge, USA: AAAI Press.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C. A.; Ram, A.; Veloso, M.; Weld, D.; and Wikins, D. 1998. PDDL - the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In Dean, T., ed., *Proc. 16th International Joint Conference on Artificial Intelligence*, 968–973. Stockholm, Sweden: Morgan Kaufmann.

Nebel, B.; Dimopoulos, Y.; and Koehler, J. 1997. Ignoring irrelevant facts and operators in plan generation. In Steel, S., and Alami, R., eds., *Proc. 4th European Conference on Planning*, volume 1348 of *Lecture Notes in Computer Science*, 338–350. Toulouse, France: Springer.

Porteous, J.; Sebastia, L.; and Hoffman, J. 2001. On the extraction, ordering and usage of landmarks in planning. In *Proc. 6th European Conference on Planning*.

Refanidis, I., and Vlahavas, I. 2001. The GRT planning system: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research* 15:115–161.

Rintanen, J. 2000. An iterative algorithm for synthesizing invariants. In Kautz, H., and Porter, B., eds., *Proc. 17th National Conference on Artificial Intelligence*, 806–811. Austin, USA: AAAI Press/MIT Press.

Scholz, U. 1999. Action constraints for planning. In Biundo, S., and Fox, M., eds., *Proc. 5th European Conference on Planning*, volume 1809 of *Lecture Notes in Artificial Intelligence*, 148–160. New York: Springer.

Tate, A. 1977. Generating project networks. In *Proc. 5th International Joint Conference on Artificial Intelligence (IJCAI'77)*, 888 – 893. William Kaufmann.

Taylor, L., and Korf, R. 1993. Pruning duplicate nodes in depth-first search. In *Proc. 11th National Conference on Artificial Intelligence*, 756–761. AAAI Press.

Varrentrapp, K.; Scholz, U.; and Duchstein, P. 2002. Design of a testbed for planning systems. In McCluskey, L., ed., *AIPS'02 Workshop on Knowledge Engeneering Tools and Techniques for AI Planning*, 51–58.

Vrakas, D.; Tsoumakas, G.; and Vlahavas, I. 2002. Towards adaptive heuristic planning through machine learning. In Grant, T., and Witteveen, C., eds., *UK Planning and Scheduling SIG Workshop*, 12–21.

Wilkins, D. E., and desJardins, M. 2000. A call for knowledge-based planning. In *Proc. AIPS Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning*, 16 – 21.

Wilkins, D. E. 1990. Can AI planners solve practical problems? *Computational Intelligence* 6(4):232 – 246.