Table of Contents

Preface

The Planning Domain Description Language (PDDL) was first devised and proposed by a committee led by Drew McDermott in preparation for what turned out to be the first in a series of international planning competitions, held in 1998. Its role at that stage was simply to support a uniform foundation for the competitors in order to simplify the problem of domain encoding. Prior to its development, planners used a mixture of similar, but slightly different, domain description languages. The differences made sharing domains significantly harder and mitigated against a strong empirical evaluation within the subject. Although its role was originally primarily as a support for the competition, PDDL has been widely adopted in the research community because of the growing collection of resources available to use with it. The advantages of sharing effort on domain engineering, problem generators, domain engineering support tools, domain analysis tools, plan validation and visualisation tools and planners themselves make it clear that the research community has much to gain in a widespread use of a standard language.

Nevertheless, PDDL is not a panacea. Nor is it a static, stable standard. For the third competition, in 2002, Maria Fox and Derek Long led a committee that extended the core of PDDL (the fragment that had been widely accepted) in order to capture temporal planning domains and plan optimisation metrics. Some of the proposed extensions have already been tested and seen initial acceptance: others remain challenges that might be faced or might be superceded by other proposals for changes to PDDL. There are significant gaps in PDDL: for example, it does not contain language elements for expressing uncertainty, for representing certain kinds of constraints, for capturing rich domain knowledge, for modelling knowledge acquisition or for describing the capabilities of potential plan executives. Some of these features exist in existing research languages and others remain areas of speculation.

In order for PDDL to continue to serve the needs of the community, playing its role in unifying effort and encouraging sharing of resources which can contribute so much to the rapid developments in the research, it is important to make PDDL an open standard. It took the field 40 years to achieve a reasonable consensus on the expression of basic pre- and post-condition actions: it is unlikely that consensus will be achieved quickly on the expression of more complex planning domain features, many of which are very active research themes. This workshop is an opportunity for members of the research community to contribute to the debate that will progress PDDL towards a consensus. The debate occupies two levels. On the one hand is the very immediate question of what features of planning domains are most pressing for addition to PDDL and how can they best be integrated both syntactically and semantically with existing features. On the other hand is the question of how the process of the development of PDDL can be managed within the community to ensure that it does not either stagnate, because of lack of agreement about its future development, or fork into multiple incompatible directions, because of lack of coordination over its development.

This workshop will offer opportunities for discussion of aspects of both of these questions. There will be technical presentations of proposed extensions or modifications to the language, panels in which competing directions of development will be debated and, we, the workshop co-chairs, very much hope, audience participation in debate and discussion about the future of PDDL.

Planning research has moved forward quickly over the past decade. Few would disagree that PDDL has played an important part in that progress. Indeed, for many, the adoption of a standard planning domain description language can be seen as central to the research goal of development of a useful domain-independent planning system. The co-chairs hope that this workshop will contribute an important next step in the development of the standard and, ultimately, to increasing opportunities for planning research to find useful application.

Acknowledgements

We would like to thank the members of the organising committee who worked hard in reviewing papers on a tight schedule. We are also grateful to the ICAPS local arrangement chair, Piergiorgio Bertoli, and workshop coordinator, Marco Benedetti, for their help with the organisation and with the production of these proceedings.

Derek Long, Drew McDermott, and Sylvie Thiébaux

Organisers

Derek Long (co-chair)
University of Durham, UK
d.p.long@durham.ac.uk

Drew McDermott (co-chair)
Yale University, USA
drew.mcdermott@yale.edu

Sylvie Thiébaux (co-chair)
The Australian National University
sylvie.thiebaux@anu.edu.au

Piergiorgio Bertoli
IRST, Italy
bertoli@itc.it

Stefan Edelkamp
University of Freiburg, Germany
edelkamp@informatik.uni-freiburg.de

Jeremy Frank
NASA Ames, USA
frank@email.arc.nasa.gov

Adele Howe
Colorado State University, USA
howe@cs.colostate.edu

Michael Littman
Rutgers University, USA
mlittman@cs.rutgers.edu

Lee McCluskey
University of Huddersfield, UK
lee@zeus.hud.ac.uk

International Workshop on

PDDL

held during:

ICAPS 2003
Trento (Italy), June 9-13 2003

An Extension to PDDL for Hierarchical Planning

Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu

DIEE – Department of Electrical and Electronic Engineering
University of Cagliari
Piazza d'Armi, I-09123 Cagliari (Italy)
{armano, cherchi, vargiu}@diee.unica.it

Abstract

This paper describes an extension to PDDL, devised to support hierarchical planning. The proposed syntactic notation should be considered as an initial suggestion, headed at promoting a discussion about how the standard PDDL can be extended to represent abstraction hierarchies.

Introduction

Hierarchical planning exploits an ordered set of abstractions for controlling the search. This choice has proven to be an effective approach for dealing with the complexity of planning tasks. Under certain assumptions it can reduce the size of the search space from exponential to linear in the size of the solution (Knoblock 1991). The technique requires the original search space to be mapped into corresponding abstract spaces, in which irrelevant details are disregarded at different levels of granularity.

Two main abstraction mechanisms have been studied in the literature: action- and state-based. The former combines a group of actions to form macro-operators (Korf 1987), whereas the latter exploits representations of the world given at a lower level of detail.

The most significant forms of the latter rely on (i) relaxed and on (ii) reduced models. In relaxed models (Sacerdoti, 1974) a criticality value is associated to each predicate, so that operators' preconditions can progressively be relaxed, while climbing the abstraction hierarchy, by dropping those predicates whose criticality value is under the one that characterizes the current level. In reduced models (Knoblock 1994) each predicate is associated with a unique level of abstraction –according to the constraints imposed by the ordered monotonicity property; any such hierarchy can be obtained by progressively removing certain predicates from the domain (or problem) space.

From a general perspective, abstractions might occur on types, predicates, and operators. Relaxed models are a typical example of predicate-based abstraction, whereas macro-operators are an example of operator-based abstraction. In (Armano, Cherchi, and Vargiu 2003) some experiments on abstraction on all the three dimensions are presented.

Historically, several planning systems used abstraction hierarchies, e.g.: GPS (Newell and Simon 1972), ABSTRIPS (Sacerdoti 1974), ABTWEAK (Yang and Tenenberg 1990), PABLO (Christensen 1991), PRODIGY (Carbonell, Knoblock, and Minton 1990), but each of them introduced and adopted its own notation without following any standard. In other words, existing planning systems tailored for abstraction did not take into account the possibility of introducing a common notation.

To contrast the lack of a standard notation for supporting abstraction hierarchies, in this paper a suitable extension to PDDL 1.2 (McDermott et al. 1998) is proposed.

The remainder of this paper is organized as follows: After briefly framing abstraction hierarchies according to a theoretical perspective, the syntax of the proposed extension is given. Then, a sample of an abstraction hierarchy –described according to the proposed notation– is illustrated and commented, with particular emphasis on the mapping between abstraction levels. Finally, conclusions are drawn and future work is outlined.

The Proposed Extension to PDDL

According to (Giunchiglia and Walsh 1990), an abstraction is a mapping between representations of a problem. In symbols, an abstraction $f : \Sigma_0 \Rightarrow \Sigma_1$ consists of a pair of formal systems (Σ_0, Σ_1) with languages Λ_0 and Λ_1 respectively, and an effective total function $f_0 : \Lambda_0 \rightarrow \Lambda_1$.

Extending the definition, an abstraction hierarchy consists of a list of formal systems $(\Sigma_0, \Sigma_1, \ldots, \Sigma_{n-1})$ with languages $\Lambda_0, \Lambda_1, \ldots, \Lambda_{n-1}$ respectively, and a list of effective total functions $f_\kappa : \Lambda_k \rightarrow \Lambda_{k+1}$, (k=0, 1, …, *n*-2) devised to perform the mapping between adjacent levels of the hierarchy.

Assuming that standard PDDL is used to represent each Λ_k (k=0, 1, …, *n*-1), in this paper we focus on the problem of extending the standard for dealing with abstraction hierarchies, with particular emphasis on the mapping functions.

The syntactic notation of the proposed extension is given according to the Extended BNF (EBNF), whose basics are briefly recalled, to avoid ambiguities:

- each production rule has the form <syntactic element> ::= expansion;
- angle brackets delimit names of syntactic elements;
- square brackets surround optional material;
- an asterisk means "zero or more of";
- a plus means "one or more of".

Furthermore, let us point out that –here– ordinary parentheses are an essential part of the grammar we are defining and do not belong to the EBNF meta language.

To represent an abstraction hierarchy a new syntactic construct (*hierarchy*) has been defined, able to highlight the domains involved in the definition and the mapping between adjacent levels. Its syntax is:

```
<hierarchy> ::=
    (define (hierarchy <hierarchy name>)
     <domain-def>
     <mapping-def>*)

<domain-def> ::= (:domains <domain name>+)

<mapping-def> ::=
    (:mapping <mapping-pair>
     [:types <types-def>]
     [:predicates <predicates-def>]
     [:actions <actions-def>])

<mapping-pair> ::=
    (<source domain> <destination domain>)

<source domain> = <name>

<destination domain> = <name>

<types-def> ::= (<types-pair>+)

<types-pair> ::=
  (<destination type> <source type>)
<types-pair> ::= (nil <source type>)

<source type> = <name>

<destination type> = <name>

<predicates-def> ::= (<predicates-pair>+)

<predicates-pair> ::= (<predicate>  <PT>)
<predicates-pair> ::= (nil <PT>)

<predicate> ::=
  (<predicate name> <variable>*)

<variable> ::= ?<name>

<PT> ::= <typed-predicate>
<PT> ::= (and <PT>+)
<PT> ::= (or <PT>+)

<typed-predicate> ::=
  (<predicate name> <typed list>*)

<typed list> ::=
   <variable>+ - <type name>

<actions-def> ::= (<action-spec>+)

<action-spec> ::=
   <action-pair> | <action-def>

<action-pair> ::= (<action> <AT>)
<action-pair> ::= (nil <AT>)

<action> ::= (<action name> <variable>*)
<AT> ::= <action>
<AT> ::= (and <AT>+)
<AT> ::= (or <AT>+)

<action-def> ::=
  see the PDDL 1.2 standard definition
```

Let us briefly comment the main definitions that occur within the proposed extension to PDDL, focusing on the underlying semantics.

Hierarchy Definition

As specified by the syntax, the "*define hierarchy*" statement contains two subsections: `<domain-def>` and `<mapping-def>`.
The `:domains` field lists domains' names according to their abstraction level, from ground to the most abstract one.
The `<mapping-def>` definitions specify the mapping between adjacent levels. In general, *n* levels of abstraction require *n-1* `<mapping-def>` definitions. Therefore, a single-level hierarchy would result in omitting the `<mapping-def>` definition (i.e., in this case only the ground level exists).
It is worth noting that, although it would be desirable – for the sake of clarity– to give `:domains` and `:mapping` definitions (including `:types`, `:predicates`, and `:actions`) according to the ordering specified by the given grammar, nothing prevents from following a different ordering.

Mapping Definition

The `:mapping` field specifies, through the `<mapping-pair>` definition, the name of the source and destination domains, respectively. Given a source domain, the destination domain is unambiguously determined by consulting the `:domains` field. Nevertheless, for the sake of readability, the destination domain must be explicitly specified.

Types Definition. The `:types` field specifies how the type hierarchy is altered while translating between adjacent levels. Each `<types-pair>` is provided according to the following syntax:

```
(<destination type> <source type>)
```

It specifies that `<source type>` becomes `<destination type>` while performing "upward" translations. In particular, `<source type>` is disregarded when the first argument of the `<types-pair>` equals to `nil`.
By default, if a type is not mentioned in any pair, it is forwarded unaltered to the destination level.
If no `:types` field is provided, all constants and variables are forwarded to the destination level, labelling them with their `<source type>`.

Predicates Definition. The `:predicates` field declares how predicates are mapped between adjacent levels. Each `<predicates-pair>` expresses whether

a predicate [1] will be forwarded to the destination level. Generally speaking, three cases may arise:
- a predicate is forwarded unchanged: the pair can be omitted, being the default;
- a predicate is disregarded: the first argument becomes `nil`;
- a predicate is a logical combination of some predicates belonging to the source level: the second argument expresses the logical formula.

Note that the destination predicate accepts a list of untyped parameters, as –in this case– parameter types can be deducted from the `:types` mapping section. On the other hand, the source predicate needs to know the type of each parameter. This is required to avoid ambiguities, since there might be predicates with identical names, but different parameter types.
If the `:predicates` field is entirely omitted, then no predicate-based abstraction occurs. In other words, each predicate is forwarded without any change to the upper level.

Actions Definition. The `:actions` field describes how to build the set of operators for the destination domain. Four different mappings may occur:
- an action remains unchanged or some of its parameters are disregarded: the pair can be omitted by default;
- an action is removed: the first argument becomes `nil`;
- an action is a combination of actions belonging to the source domain ("`and`" meaning serialization, "`or`" meaning parallelization);
- a new operator is defined from scratch: the statement `<action-def>` is used (note that this definition is not expanded in the notation, since it follows the standard PDDL 1.2).

An Example of the Proposed Extension

As an example, let us consider the *depot* domain, taken from the AIPS 2002 planning competition (Long 2002). The domain was devised by joining two well-known planning domains: *logistics* and *blocks-world*. They have been combined to form a domain in which trucks can transport crates around, to be stacked onto pallets at their destinations. The stacking is achieved using hoists, so that the resulting stacking problem is very similar to a blocks-world problem with hands. Trucks behave like "tables", since the pallets on which crates are stacked are limited.
Let us suppose we want to create a two-level abstraction for the *depot* domain, composed by *depot-ground* and *depot-abstract*.
According to the above notation, we can start defining the hierarchy in the following way:

```
(define (hierarchy depot)
  (:domains depot-ground depot-abstract)
  ...
```

Since there are only two levels of abstraction, just one `:mapping` statement is needed. To express the mapping rules (on types, predicates, and operators) from the ground to the abstract level, the following statement must be introduced:

```
(:mapping
  (depot-ground depot-abstract)
  ...
```

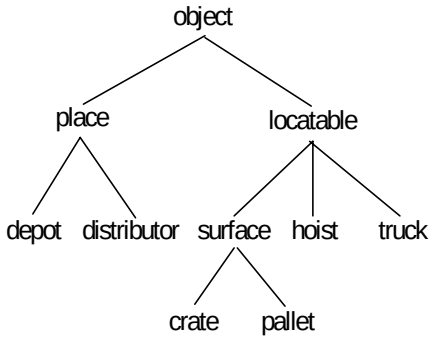


Fig. 1 - Type Hierarchy for the depot ground domain.

Let us start with abstracting types of the *depot* domain type hierarchy (as reported in Figure 1). We decided to disregard hoists and trucks, and not to distinguish between depots and distributors (i.e., considering both as generic places).
According to the proposed notation, the translation can be expressed in the following way:

```
:types
  ((place depot)
   (place distributor)
   (nil hoist)
   (nil truck))
```

The first two statements assert that both `depot` and `distributor` become `place` in the *depot-abstract* domain. The last two statements assert that both `hoist` and `truck` must be disregarded. Let us recall that, by default, the types not mentioned remain unchanged at the abstract level (e.g. `locatable`, `crate`, `place`, etc.).
The above notation entails the type hierarchy reported in Figure 2.

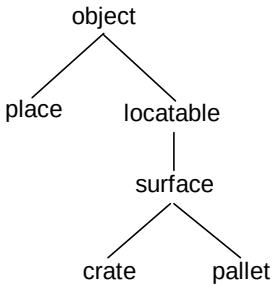


Fig. 2 - Type Hierarchy for depot abstract domain.

[1] Or a combination of predicates, obtained using logical *and*, *or*, *not* operators.

```
(in ?c - crate ?t - truck)
 (lifting ?h - hoist ?c - crate)
 (available ?h - hoist)
 (clear ?s - surface)
 (on ?c - crate ?s - surface)
 (at ?l - locatable ?p - place)
```

Fig. 3 – Predicates of the depot domain.

The choice of removing some types implies that some predicates might become meaningless at the abstract level. In particular, predicates accepting parameters of type `truck` or `hoist` cannot exist at the abstract level. Figure 3 lists the ground predicates of the *depot* domain. Since the `in` predicate accepts a `truck` as a parameter, it must be explicitly disregarded by the following statement:

```
(nil (in ?c – crate ?t – truck))
```

Similar considerations can be made for the `lifting` and `available` predicates.
The predicates `(clear ?s – surface)` and `(on ?c – crate ?s – surface)` remain unchanged and can be omitted in the `:mapping` field (being the default).

```
(define (hierarchy depot)
   (:domains depot-ground depot-abstract)
   (:mapping (depot-ground depot-abstract)
     :types
       ((place depot)
        (place distributor)
        (nil hoist)
        (nil truck))
     :predicates
       ((nil (lifting ?h – hoist ?c - crate))
        (nil (available ?h – hoist))
        (nil (in ?c – crate ?t – truck))
        (nil (at ?h – hoist ?p - place))
        (nil (at ?t – truck ?p - place)))
      :actions
       ((nil (drive ?t ?p1 ?p2))
        (nil (load ?h ?c ?t ?p))
        (nil (unload ?h ?c ?t ?p))
        (nil (lift ?h ?c ?s ?p))
        (nil (drop ?h ?c ?s ?p))
        ((lift-and-drop ?c ?s1 ?s2 ?p1 ?p2)
          (and (lift ?h ?c ?s1 ?p1)
               (drop ?h ?c ?s2 ?p2))))))
```

Fig. 4 – Hierarchy definition for the depot domain.

Note that `(at ?l – locatable ?p – place)` is overloaded, in the sense that it actually represents different predicates. Some examples of possible expansions are:

```
(at ?l – hoist ?p – distributor)
 (at ?l – truck ?p – depot)
 (at ?l – crate ?p – depot)
```

All expansions that accept any parameter whose type has been disregarded at the abstract level, must be explicitly removed. In this case, the following statements must be asserted:

```
(nil (at ?h – hoist ?p - place))
(nil (at ?t – truck ?p - place))
```

```
(define (domain elevator-ground)
   (:requirements :strips :typing)
   (:types passenger floor - object)

 (:predicates
   (origin ?person - passenger ?floor - floor)
   (destin ?person - passenger ?floor - floor)
   (above ?floor1 ?floor2 - floor)
   (boarded ?person - passenger)
   (served ?person - passenger)
   (lift-at ?floor - floor))

 (:action board
   :parameters (?f - floor ?p - passenger)
   :precondition
     (and (lift-at ?f) (origin ?p ?f))
   :effect (boarded ?p))

 (:action depart
   :parameters (?f - floor ?p - passenger)
   :precondition
     (and (lift-at ?f) (destin ?p ?f)
          (boarded ?p))
   :effect (and (not (boarded ?p))(served ?p)))

 (:action up
   :parameters (?f1?f2 - floor)
   :precondition
     (and (lift-at ?f1) (above ?f1 ?f2))
   :effect
     (and (lift-at ?f2) (not (lift-at ?f1))))

 (:action down
   :parameters (?f1?f2 - floor)
   :precondition
     (and (lift-at ?f1) (above ?f2 ?f1))
   :effect
     (and (lift-at ?f2) (not (lift-at ?f1)))))
```

Fig. 5 – The elevator domain.

Let us point out that more complex mapping rules are admissible. For example, two or more ground predicates could be combined to form a new abstract predicate. Let us consider the statement below:

```
((moveable ?c ?h ?s ?p)
 (and (lifting ?h – hoist ?c – crate)
      (at ?h – hoist ?p – place)
      (clear ?s – surface)
      (at ?s – surface ?p – place))
```

The new predicate `moveable` is introduced, which applies only when the specified group of ground predicates are true.
The mapping rules enforced on types and predicates may modify preconditions and effects of some ground operators. For example, consider the `drive` action:

```
(:action drive
  :parameters
   (?t - truck ?p1 ?p2 - place)
  :precondition
   (and (at ?t ?p1))
  :effect
```

```
(and (not (at ?t ?p1))(at ?t ?p2)))
```

Since the `(at ?t – truck ?p – place)` predicate has not been forwarded to the abstract level, the `drive` action could not require any such precondition or effect. Therefore, `drive` becomes meaningless at the abstract level, and must be removed throughout the following statement:

```
((nil (drive ?t ?p1 ?p2))
```

Similar considerations can be made for the `load` and `unload` actions:

```
(nil (load ?h ?c ?t ?p))
(nil (unload ?h ?c ?t ?p))
```

```
(define (hierarchy elevator)
  (:domains elevator-ground
            elevator-abstract)
  (:mapping
     (elevator-ground elevator-abstract)
     :predicates
       ((nil (lift-at ?f – floor))
        (nil (above ?f1 ?f2 - floor)))
      :actions
       ((nil (up ?f1 ?f2))
        (nil (down ?f1 ?f2))
        (nil (board ?f ?p))
        (nil (depart ?f ?p))
        ((load ?f ?p) (board ?f ?p))
        ((unload ?f ?p) (depart ?f ?p)))))
```

Fig. 6 – Hierarchy definition for the elevator domain.

At this point, one may want to join the remaining actions `lift` and `drop` to form a new abstract operator (say `lift-and-drop`). According to the proposed extension, the new operator is defined as:

```
((lift-and-drop ?c ?s1 ?s2 ?p1 ?p2)
 (and (lift ?h ?c ?s1 ?p1)
      (drop ?h ?c ?s2 ?p2)))
```

Moreover, the `lift` and `drop` actions can be ignored:

```
(nil (lift ?h ?c ?s ?p))
(nil (drop ?h ?c ?s ?p))
```

Alternatively, the new abstract operator `lift-and-drop` could be introduced from scratch as follows:

```
(:action lift-and-drop
  :parameters
   (?c - crate ?s1 ?s2 – surface
    ?p1 ?p2 - place)
  :precondition
    (and (at ?c ?p1) (on ?c ?s1)
         (clear ?c) (at ?s2 ?p2)
         (clear ?s2))
  :effect
    (and (not (at ?c ?p1))
         (at ?c ?p2)(clear ?s1)
         (not (clear ?s2))
         (on ?c ?s2)
         (not (on ?c ?s1))))
```

For the sake of completeness, the entire hierarchy definition for the *depot* domain is summarized in Figure 4.

In the above example, we started by abstracting the type hierarchy. It is worth pointing out that this choice is not mandatory; in fact abstraction could also be started by specifying the mapping of predicates or actions.

To better illustrate an alternative approach, let us consider another example applied to the *elevator* domain (Koehler and Schuster 2000), whose ground definition is reported in Figure 5.

The type hierarchy of elevator is very simple and contains only two types: `passenger` and `floor`. Thus, let us abstract the domain from predicates.

In particular, one may decide to disregard `(above ?f1 ?f2 – floor)` and `(lift-at ?f – floor)`, so that the lift is always available and moveable from a floor to another. This choice has an influence on actions: `up` and `down` become meaningless, whereas preconditions and effects of `board` and `depart` undergo some modifications on their abstract

```
(define (domain blocks-ground)
  (:requirements :strips :typing)
  (:types block - object)
  (:predicates
    (on ?x - block ?y - block)
    (ontable ?x - block)
    (clear ?x - block)
    (handempty)
    (holding ?x - block))

  (:action pick-up
     :parameters (?x - block)
     :precondition
      (and (clear ?x)(ontable ?x)
           (handempty))
     :effect
      (and (not (ontable ?x))
           (not (clear ?x))
           (not (handempty))(holding ?x)))

  (:action put-down
     :parameters (?x - block)
     :precondition (holding ?x)
     :effect
      (and (not (holding ?x))(clear ?x)
           (handempty)(ontable ?x)))

  (:action stack
     :parameters (?x - block ?y - block)
     :precondition
      (and (holding ?x) (clear ?y))
     :effect
      (and (not (holding ?x))
           (not (clear ?y))(clear ?x)
           (handempty)(on ?x ?y)))

  (:action unstack
     :parameters (?x - block ?y - block)
     :precondition
      (and (on ?x ?y)(clear ?x)(handempty))
     :effect
      (and (holding ?x)(clear ?y)
           (not (clear ?x))(not (handempty))
           (not (on ?x ?y)))))
```

Fig. 7 – The blocks-world domain.

```
(define (hierarchy blocks)
  (:domains blocks-ground blocks-abstract)
  (:mapping
    (blocks-ground blocks-abstract)
    :predicates
      ((nil (handempty))
       (nil (holding ?b - block)))
     :actions
      ((nil (pick-up ?b))
       (nil (put-down ?b))
       (nil (stack ?b1 ?b2))
       (nil (unstack ?b1 ?b2))
       ((pick-up&stack ?b1 ?b2)
          (and (pick-up ?b1)(stack ?b1 ?b2)))
       ((unstack&put-down ?b1 ?b2)
          (and (unstack ?b1 ?b2)
               (put-down ?b1))))))
```

Fig. 8 – Hierarchy definition of the blocks-world domain.

counterparts (say `load` and `unload`, respectively). Figure 6 shows the described hierarchy definition for the elevator domain.

As an example of abstraction starting from actions, let us consider the *blocks-world* domain, reported in Figure 7. In this case the type hierarchy cannot be abstracted, as it contains only the type `block`. In this domain two macro-operators can be identified: `pick-up&stack` and `unstack&put-down`. The decision of adopting these operators entails a deterministic choice on which predicates have to be forwarded / disregarded while performing upward translations. More explicitly `(handempty)` and `(holding ?b – block)` must be disregarded, meaning that the hand can be considered always available. Figure 8 shows the corresponding hierarchical definition of the blocks-world domain, according to the proposed notation.

Conclusions and Future Work

In this paper a novel extension to the standard PDDL 1.2 has been proposed, devised to support hierarchical planning. The extension introduces the *hierarchy* construct, which encapsulates an ordered set of domains, together with a set of mappings between adjacent levels of abstraction. Since mappings are given in term of types, predicates, and operators, three subfields in the <mapping-def> have been introduced, to represent the abstraction over such dimensions. The extension described in this paper should be considered as an initial proposal, headed at promoting a discussion about how the standard PDDL can be enriched with additional constructs able to represent abstraction hierarchies.

As for the future work, the possibility of extending the notation to encompass PDDL 2.1 (Fox and Long 2002) is being investigated.

References

Armano, G., Cherchi, G., and Vargiu, E. A Parametric Hierarchical Planner for Experimenting Abstraction Techniques. *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, Acapulco, Mexico, August 2003, to appear.

Carbonell, J.C., Knoblock, C.A., and Minton, S. PRODIGY: An integrated architecture for planning and learning. In D. Paul Benjamin (ed.) *Change of Representation and Inductive Bias.* Kluwer Academic Publisher, 125--146, 1990.

Christensen, J. *Automatic Abstraction in Planning*. PhD thesis, Department of Computer Science, Standford University, 1991.

Fox, M., and Long, D. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. Technical Report, Department of Computer Science, University of Durham, UK, 2001.

Giunchiglia, F., and Walsh, T. A theory of Abstraction, Technical Report 9001-14, IRST, Trento, Italy, 1990.

Koehler, J., and Schuster, K. Elevator Control as a Planning Problem. *Proceedings of the 5th International Conference on AI Planning and Scheduling*, 331--338, AAAI Press, Menlo Park, 2000.

Korf, R.E., Planning as Search: A Quantitative Approach. *Artificial Intelligence*, 33(1):65--88, 1987.

Knoblock, C.A. Search Reduction in Hierarchical Problem Solving. Proceedings of the 9th National Conference on Artificial Intelligence, 686--691, Anaheim, CA, 1991.

Knoblock, C.A. Automatically Generating Abstractions for Planning. *Artificial Intelligence*, 68(2):243--302, 1994.

Long, D. *Results of the AIPS 2002 planning competition*, 2002, Url: http://www.dur.ac.uk/d.p.long/competition.html.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C.A., Ram, A., Veloso, M., Weld, D., and Wilkins, D. PDDL – The Planning Domain Definition Language, Technical Report CVC TR-98-003 / DCS TR-1165, Yale Center for Communicational Vision and Control, October 1998.

Newell A., and Simon H.A. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, NJ, 1972.

Sacerdoti, E.D. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115--135, 1974.

Yang Q., and Tenenberg, J. Abtweak: Abstracting a Nonlinear, Least Commitment Planner. *Proceedings of the 8th National Conference on Artificial Intelligence*, 204--209, Boston, MA, 1990.

Formalizing Resources for Planning

Tania Bedrax-Weiss, Conor McGann and **Sailesh Ramakrishnan**
QSS Group, Inc.
NASA Ames Research Center
Moffett Field, CA 94035
{tania,cmcgann,sailesh}@email.arc.nasa.gov

Abstract

In this paper we present a classification scheme which circumscribes a large class of resources found in the real world. Building on the work of others we also define key properties of resources that allow formal expression of the proposed classification. Furthermore, operations that change the state of a resource are formalized. Together, properties and operations go a long way in formalizing the representation and reasoning aspects of resources for planning.

Introduction

Historically, allocation of tasks to resources has been considered part of the scheduling problem, and largely omitted from the planning literature. In recent times, the importance of such resource allocation decisions in planning has been recognized (Smith, Frank, & Jónsson 2000; Long *et al.* 2000). Though progress has been made to meet this challenge (Laborie 2001), the state of the art has not yet advanced to a point where we have a comprehensive treatment of resources as an inherent part of a planning framework. More specifically, efforts to characterize the types of resources of interest have been incomplete and largely constrained by the availability of efficient algorithms to reason with them. Furthermore, approaches to natively incorporate resources into domain descriptions have been largely absent.

We believe that the absence of explicit types of resources

- obfuscates the semantics of the model,
- impedes detection of domain modeling errors,
- complicates the mapping to efficient implementations that could be tailored to particular resource types, and
- hinders domain analysis.

For example, resources have not yet been incorporated explicitly in the PDDL 2.1 specification (Fox & Long 2003) although they can be represented through the use of functional expressions. This is illustrated in Figure 1 which describes an action *fly* consuming a resource *fuel*. In the action definition, the fuel consumption is expressed as an effect decreasing the level of *fuel*. Fuel is not identified explicitly as a resource. Furthermore, the inherent properties of *fuel* and the way in which it is allowed to change are not represented.

In this paper we present a classification scheme that will circumscribe a large class of resource types found in the real world. We use and build upon the large body of work both in the planning and scheduling communities. We first develop, through exploration of real world examples, a description of resources from a planning perspective. We define a set of properties that characterize a resource. We then present a classification scheme based on these properties. We go on to explain how this classification will identify transactions on resources. We present examples throughout to illustrate the terms introduced. Where possible, we follow the PDDL 2.1 syntax in the hope that it will be familiar to the reader. We then review related work in PDDL and other languages, in terms of their methods and limitations to address the needs outlined. We conclude with a brief synopsis and discussion of future work.

```
(:durative−action fly
   :parameters (?p − plane ?t − traveller ?a ?b − location)
   :duration (= ?duration (flight−time ?a ?b))
   :condtion (and (at start (at ?p ?a))
                  (at start (at ?t ?a))
  (over all (inflight ?p))
  (over all (aboard ?t ?p))
  (at start  (>= (fuel−level ?p) (* (flight−time ?a
   ?b) (consumption−rate ?p))))))
   :effect (and (at start (not (at ?p ?a)))
                (at start (not (at ?t ?a)))
                (at end (at ?p ?b))
                (at end (at ?t ?b))
 (at start (inflight ?p))
                (at end (not (inflight ?p)))
                (at end (not (aboard ?t ?p)))
                (at end (decrease (fuel−level ?p) (* (flight−time ?a
    ?b) (consumption−rate ?p))))))
```

Figure 1: Example of an activity on a resource in PDDL 2.1

Classification

Classification by Example

The following examples are designed to illustrate the various features of resources that are of interest to the modeler.

Consider the cargo bay of the space shuttle. Many items of different sizes are placed in the bay and consume volume.

The space is used when an item is placed in it. It is made available again when the item is removed. We can consider the space as an example of a reusable resource. In contrast consider the fuel in a fuel tank. Once consumed it is destroyed permanently. Fuel, in this case, is an example of a consumable resource. If the fuel tank can be refueled then it is an example of a replenishable resource. Process byproducts which are never used are examples of producible resources.

The battery on a planetary rover is an example of a continuous resource. Within the capacity of the battery any amount of energy can be drawn at a time. In contrast disk space on a hard drive is consumed in discrete chunks (bytes). This is an example of a discrete resource.

A printer is an example of a single-capacity resource since it prints only one job at a time. On the other hand, a passenger aircraft contains numerous seats representing a multi-capacity resource.

A fuel container typically has a fixed volume, and therefore a fixed capacity. Alternatively, a battery whose capacity degrades over time is an example of a variable capacity resource.

Seats on an airplane are also examples of a deterministic resource because the state of the resource is known precisely. The energy of a battery is an example of a stochastic resource because of the inherent uncertainty in the amount of the resource available.

The fuel tank in a car is an example of an exclusive resource because refueling is not allowed while the engine is running. Data bandwidth is an example of a shared resource because multiple activities can use the bandwidth simultaneously.

A cargo bay has specific restrictions for both weight and volume. Loading a cargo bay consumes both weight and volume at different rates. Weight and volume are two distinct dimensions of the same resource so this is an example of a multi-dimensional resource (Smith & Becker 1997).

Keeping these examples in mind, we proceed to define properties that precisely categorize resources.

Resource Properties

In this section we present a set of properties that can be used to describe a quantitative resource i.e. a resource with capacity and availability described in terms of numeric quantities. We assume all measures of quantity are represented by a generic unit rather than actual units of measure. We also assume that all conversion operations are defined and occur outside of the resource definitions and operations. Furthermore, we assume that all quantities are greater than zero.

We adopt the following notation for domain definitions. Let $\mathbb{R}$ be to the domain of real numbers; let $\mathbb{N}$ be the domain of natural numbers;[1] let $\mathbb{Z}$ be the domain of whole numbers; let $\mathbb{T}$ be the domain of time. Notice that we explicitly defer commitment to whether time is represented by natural numbers or real numbers, leaving the choice to implementation. Furthermore, let $\Re$ be the universe of resources and let $\mathbb{X}$ be the universe of transactions.[2] Let $[t_1, t_2] = \{x \in \mathbb{T} \mid t_1 \leq x \leq t_2\}$ be an interval of time. Let t equal to $[t, t]$ by definition.

The following properties are defined for a resource.

1. *Level*: The amount of available resource. The level is defined as $\mathcal{L}$:$\Re \times \mathbb{T} \rightarrow [\mathcal{L}_{min}, \mathcal{L}_{max}]$ where $\mathcal{L}_{min}, \mathcal{L}_{max} \in \mathbb{R}$ for continuous resources and $\mathcal{L}_{min}, \mathcal{L}_{max} \in \mathbb{Z}$ for discrete resources. We assume that plans may have temporal flexibility so the level is given by an interval.
2. *Level Limit*: The instantaneous physical limit of available resource. The level limit is given by an upper and a lower bound and is defined as $\mathcal{LL}$:$\Re \times \mathbb{T} \rightarrow \langle \mathcal{LL}_{min}, \mathcal{LL}_{max} \rangle$ where $\mathcal{LL}_{min}, \mathcal{LL}_{max} \in \mathbb{R}$ for continuous resources and $\mathcal{LL}_{min}, \mathcal{LL}_{max} \in \mathbb{Z}$ for discrete resources.
3. *Rate*: The change in level per unit of time. Formally, $\mathcal{R}$:$\Re \times \mathbb{T} \rightarrow [\mathcal{R}_{min}, \mathcal{R}_{max}]$ where $\mathcal{R}_{min}, \mathcal{R}_{max} \in \mathbb{R}$ for continuous resources and $\mathcal{R}_{min}, \mathcal{R}_{max} \in \mathbb{Z}$ for discrete resources. Because plans may have temporal flexibility, the level is given by an interval.
4. *Rate Limit*: The limit imposed on the rate. The rate limit is given by a lower and an upper bound and is defined as $\mathcal{RL}$:$\Re \times \mathbb{T} \rightarrow \langle \mathcal{RL}_{min}, \mathcal{RL}_{max} \rangle$ where $\mathcal{RL}_{min}, \mathcal{RL}_{max} \in \mathbb{R}$ for continuous resources and $\mathcal{RL}_{min}, \mathcal{RL}_{max} \in \mathbb{Z}$ for discrete resources.
5. *Transactions*: The set of all transactions that may or must occur by a given instant of time, defined by $\mathcal{T}$:$\Re \times \mathbb{T} \rightarrow \mathbb{X}$
6. *Completed Transactions*: The set of all transactions that must occur by a given instant of time, defined by $\mathcal{CT}$:$\Re \times \mathbb{T} \rightarrow \mathbb{X}$
7. *Pending Transactions*: The set of all transactions that may overlap a given instant. $\mathcal{PT}$:$\Re \times \mathbb{T} \rightarrow \mathbb{X}$
8. *Transaction Count*: The number of total transactions at an instant of time, defined by $\mathcal{TC}$:$\Re \times \mathbb{T} \rightarrow [\mathcal{TC}_{min}, \mathcal{TC}_{max}]$ where $\mathcal{T}_{min}, \mathcal{T}_{max} \in \mathbb{N}$. Because plans may have temporal flexibility, the transaction count is given by an interval.
9. *Transaction Limit*: The limit imposed on the number of concurrent transactions. The transaction limit is defined as $\mathcal{TL}$:$\Re \times \mathbb{T} \rightarrow \langle \mathcal{TL}_{min}, \mathcal{TL}_{max} \rangle$ where $\mathcal{T}_{min}, \mathcal{T}_{max} \in \mathbb{N}$. Because a resource may have upper and lower limits on the transaction count, the transaction limit is given by an interval.
10. *Horizon*: The time interval over which the resource can process transactions and answer queries. Formally, $\mathcal{H} = [\mathcal{H}_s, \mathcal{H}_e]$, where $\mathcal{H}_s, \mathcal{H}_e \in \mathbb{T}$.

Variations of properties 1 through 9 can be defined to capture consumption and production on the resource, e.g. production rate, production level, consumption transaction count, consumption level limit, consumer transactions. It should be noted that the levels, rates, counts and transaction sets are derived values based on the occurrence of transactions on the resource. In contrast, limits are imposed directly to capture constraints on the state. It should also be

[1] We assume that $\mathbb{N}$ includes the number zero.

[2] Transactions will be described in more detail in the next section.

noted that limits are not used in the definitions of the properties they intend to constrain. This is deliberate, since no commitment is given regarding the enforcement policies of constraints in these definitions.

In addition to the formal definitions provided for each property, certain relationships can be observed among them for any given instant of time.

1. The maximum transaction count is equal to the sum of the maximum producing transaction count and the maximum consuming transaction count. $\mathcal{TC}_{max} = \mathcal{TC}_{prod,max} + \mathcal{TC}_{cons,max}$
2. Similarly, the minimum transaction count is equal to the sum of the minimum producing transaction count and the minimum consuming transaction count. $\mathcal{TC}_{min} = \mathcal{TC}_{prod,min} + \mathcal{TC}_{cons,min}$
3. The maximum rate of change of the resource is the difference between the maximum production rate and the minimum consumption rate. $\mathcal{R}_{max} = \mathcal{R}_{prod,max} - \mathcal{R}_{cons,min}$
4. The minimum rate of change of the resource is the difference between the minimum production rate and the maximum consumption rate. $\mathcal{R}_{min} = \mathcal{R}_{prod,min} - \mathcal{R}_{cons,max}$
5. The set of completed transactions is a subset of the set of all transactions. $\mathcal{CT} \subseteq \mathcal{T}$
6. The set of pending transactions is a subset of the set of all transactions. $\mathcal{PT} \subseteq \mathcal{T}$
7. No transaction can belong to both the pending transactions and the completed transactions. $\mathcal{PT} \cap \mathcal{CT} = \emptyset$
8. A transaction is either a pending or a completed transaction. $\mathcal{CT} \cup \mathcal{PT} = \mathcal{T}$
9. The maximum resource level at an instant of time is a function of the quantities of the completed production transactions, the completed consumption transactions, and the quantities of some subset of the pending transactions. The subset is chosen so as to maximize the the overall resource level. $\mathcal{L}_{max} = \sum_i qp_i(t) - \sum_i qc_i(t) + \sum_i qx_i(t)$,
   where
   $$\begin{aligned} qp_i(t) &: \exists\, \texttt{produce}(r, qp_i(t)) \in \mathcal{T}_{prod} \cap \mathcal{CT}, \\ qc_i(t) &: \exists\, \texttt{consume}(r, qc_i(t)) \in \mathcal{T}_{cons} \cap \mathcal{CT}, \\ qx_i(t) &: \exists\, \texttt{produce}(r, qx_i(t)) \in S \text{ or} \\ &\quad\ \exists\, \texttt{consume}(r, qx_i(t)) \in S \end{aligned}$$
   where $S \subseteq \mathcal{PT}$ is chosen so as to maximize the sum.
10. The maximum resource level at an instant of time is a function of the quantities of the completed production transactions, the completed consumption transactions, and the quantities of some subset of the pending transactions. The subset is chosen so as to minimize the the overall resource level. $\mathcal{L}_{min} = \sum_i qp_i(t) - \sum_i qc_i(t) - \sum_i qx_i(t)$,
    where
    $$\begin{aligned} qp_i(t) &: \exists\, \texttt{produce}(r, qp_i(t)) \in \mathcal{T}_{prod} \cap \mathcal{CT}, \\ qc_i(t) &: \exists\, \texttt{consume}(r, qc_i(t)) \in \mathcal{T}_{cons} \cap \mathcal{CT}, \\ qx_i(t) &: \exists\, \texttt{produce}(r, qx_i(t)) \in S \text{ or} \\ &\quad\ \exists\, \texttt{consume}(r, qx_i(t)) \in S \end{aligned}$$
    where $S \subseteq \mathcal{PT}$ is chosen so as to maximize the sum of the $qx_i(t)$.

```
(:resource fuel−in−a−tank
    :level−limit [0 2000]                ;all tanks in this domain
                                         ;can hold atmost 2000 units
    :production−rate−limit [0 100]       ;for all tanks in the domain
                                         ;atmost 100 units can be
                                         ;pumped in per unit time
    :consumption−rate−limit [0 100]
    :num−transactions−limit [0 4]        ;atmost 4 transactions can be
                                         ;done on this resource.
)

(fuel−in−a−tank fueltank1
    :level−limit (                       ;represents a limit profile
          (over [0 6] [0 1000])          ;where upto 6 am and after
          (over [6 18] [0 2000])         ;6 pm the tank can hold 1000
          (over [18 24] [0 1000])        ;and between 6am−6pm hold
          )              ;.              ;2000

    :production−rate−limit [0 42.5]
    :consumption−rate−limit [0 75.5]
    :num−transactions−limit [0 3]
    :level 876.34                        ;in the init state, this
                                         ;particular tank has 876 units
                                         ;in it.
    :num−producers−limit [0 2]           ;this tank has two inlet
                                         ;valves
    :horizon [9 17]                      ;fuel can be drawn only during
                                         ;normal working hours.
    )
```

Figure 2: Example in pseudo-PDDL describing some properties of a resource

$\mathcal{L}_{max}$ and $\mathcal{L}_{min}$ represent what is informally called the "resource envelope". Different algorithms (Laborie 2001; Muscettola 2002) compute the envelope with varying degrees of accuracy. The degree of accuracy depends on how carefully the set S is chosen.

There are other distinctions that can be expressed in terms of the properties described above. For example, persistence and expiration dates can be described in terms of the horizon. Similarly, scheduled unavailability can be described in terms of limits on the number of transactions or the rates of change. Other researchers (Powell, Shapiro, & Simao 2001) have mentioned concepts such as active and passive and have characterized numbers of simultaneous producers and consumers. These concepts can be expressed in terms of the properties defined above.

Figure 2 presents an example illustrating the use of these properties in specifying a resource. Restrictions included in the definition of the resource *fuel-in-tank* restrict the properties of all instances of that resource. Restrictions in the definition of the instance *fuel-tank1* impose further restrictions for that particular instance alone. Additionally, the level-limit of *fuel-tank1* is expressed as a profile over time. In this case, the profile is piecewise constant i.e. between the hours of 6 am and 6 pm the maximum level-limit and the minimum level-limit are constants (2000 and 0 respectively).

Resource Categories

The set of resource categories informally introduced earlier provide a means to qualitatively describe the nature of a resource. The set of properties introduced earlier provide a means to quantitatively describe the nature of a resource. In this section we formalize the former in terms of the latter.

A resource can be categorized based on how it may be produced or consumed.

1. *Consumable*: A consumable resource is a resource that is decreased by some activities but is not produced by any activities in the system. For example, a resource such as ammunition may be depleted by firing a weapon. In a mode where the plan of attack prohibits resupply then additional ammunition may not be obtained. This means that $\mathcal{L}$ is monotonically non-increasing.
2. *Producible*: A producible resource is a resource that is created by some activities but is not consumed by any activities in the system. A waste-product of an industrial process may be an example of this. This means that $\mathcal{L}$ is monotonically non-decreasing.
3. *Replenishable*: A replenishable resource can be both produced and consumed as part of the same system. Any ordering of production and consumption transactions are allowed on the resource, e.g battery power which may be produced if it is in the charger, as well as consumed, if the device it powers is turned on.
4. *Reusable*: A reusable resource is a replenishable resource that is produced and consumed with the additional constraint that producing and consuming transactions must happen in tandem. That is, for any interval of time, two consecutive consumption or production transactions are not allowed.

We can further describe resources as discrete or continuous based on the quantities produced or consumed.

1. *Discrete*: The resource is consumed, produced, or used in discrete quantities. For example, disk space is allocated in chunks of bytes. A printer consumes single sheets of paper from the paper bin. If a resource is discrete, the levels, limits and rates are all discrete.
2. *Continuous*: The resource is consumed, produced, or used in continuous quantities, e.g energy stored in a battery. If a resource is continuous, the levels, limits, and rates are all continuous.

Depending on the amount (divisibility or unit) we can distinguish between single and multiple capacity resources.

1. *Single Capacity*: The resource can be thought of as one unit which must be consumed as a whole. This characteristic implies a restriction on the level limit such that

$$\mathcal{LL} = \begin{cases} 0 & \text{if it is being consumed} \\ 1 & \text{otherwise} \end{cases}$$

2. *Multiple Capacity*: The resource represents multiple units which can be used or consumed by different operations. This characteristic implies a restriction on the level limit such that $\mathcal{LL}_{max} > 1$ and $\mathcal{LL}_{min} \neq \mathcal{LL}_{max}$.

The variation of capacity over time allows us to distinguish between fixed and variable capacity resources.

1. *Fixed capacity*: The level limit of a resource is fixed over time, e.g. a gas tank has a fixed capacity to store gasoline. This characteristic imposes a constraint on the level limit such that the level limit is constant with respect to time.
2. *Variable capacity*: The level limit of a resource is a function of time, e.g. a battery whose capacity degrades over time. This characteristic agrees with our definition of level limit.

The level of certainty with which one can determine the capacity allows us to classify resources as deterministic or stochastic.

1. *Deterministic capacity*: The capacity is precisely determined. This characteristic agrees with our definition of level limit.
2. *Stochastic capacity*: The capacity is determined probabilistically. This characteristic is only mentioned for completeness since we provide no formal treatment of this characteristic in this paper.

If simultaneous transactions are allowed to operate on resources then the resource is shared as opposed to exclusive.

1. *Shared*: Using the resource at less than full capacity allows others to use the resource simultaneously, e.g. a battery on a rover typically provides energy simultaneously to a number of devices. This characteristic imposes a constraint on the transaction limit such that $\mathcal{TL}_{max} > 1$ for all time instants.
2. *Exclusive*: Only one activity is allowed to access the resource at a time, regardless of the amount of resource used, e.g. a restaurant table that seats 10 but only 6 people are seated at the table. Even though it seats 10, the remaining 4 seats are made unavailable. This characteristic imposes a constraint on transactions such that $\mathcal{TL} = \langle 0, 1 \rangle$ for all time instants.

There are two additional characteristics noted in (Smith & Becker 1997) that are also important: single-dimensional vs. multi-dimensional, and pooled vs. not pooled. A resource is multi-dimensional if it has more than one aspect to its quantity and each aspect must be updated together but at different rates. A cargo bay with constraints on weight and volume is an example of a multi-dimensional resource. Each *load* activity would simultaneously consume both weight and volume in different quantities. A resource may be aggregated into a resource pool. This allows the domain model to abstract out details of individual resources while preserving the individual nature of each resource for final allocation. We have not had the time to further explore these concepts. For the purpose of this paper we consider all resources to be single-dimensional and omit treatment of resource pools.

Figure 3 gives an example of three resources which have been defined in terms of the categories described above. Qualitative categories impose restrictions on quantitative properties. Note that as before, properties may be additionally restricted.

```
;A camera that can be used to take a picture at a time
(:resource camera
   :categories (exclusive
      fixed
      single−capacity
      discrete
      deterministic
      reusable))


;Earth communication window with a fixed bandwith
(:resource Earth−Communication
   :categories (shared
      fixed
      multi−capacity
      continuous
      deterministic
      reusable)
   :horizon ((540 550) (1020 1030) )

;Solid state data storage disk on which data can be written and erased
;after downloading.
(:resource data−storage−disk
   :categories (exclusive
      fixed
      multi−capacity
      discrete
      deterministic
      replenishable))
```

Figure 3: Examples in pseudo-PDDL describing resources based on their categories

Operations on Resources

Resource categories provide a means to qualitatively describe the nature of a resource. Resource properties, on the other hand, provide a means to quantitatively describe both its nature and its current state. Given this model, the operations to effect state change (transactions) can be formally defined. Throughout, we take the view that all operations on resources should be context free, i.e. we make no assumptions about the activity context that is causing the transaction.

Resource Transactions

Resource transactions are caused by actions in a plan to change the state of a resource. Transactions on resources have fixed semantics that are fully defined based on each transaction type. We assume that transactions can always be applied and that if no other transactions occur in the period (instant or interval) of time over which they're applied, the effect is always observed.[3]

The term *quantity* is used in transaction specifications to indicate the amount of the given resource to be transacted. In the general case, quantity may vary over time. In this case, quantity is a function of time returning a number, an element of $\mathbb{R}$ for continuous resources or an element of $\mathbb{N}$ for discrete resources. The precise semantics of the *quantity* function, however, is implementation specific and we define it externally. Computing with these functions has to be well-defined, so we require that all *quantity* functions be commutative and convolvable. It is also likely we could enforce additional restrictions given the particulars of a resource description and more explicit insight into the semantics of a *quantity* function. At this time, no scheme has been developed to formalize the semantics of these functions to allow such checks for correctness by analysis of the model.

We define the following transactions.

1. *Consume*: Consumption can occur either at an instant of time or over an interval of time.
   - `at` t `consume`(r, q) If a consume transaction and no other transactions are operating on a resource at the specified time, the transaction has the effect of reducing the level by the specified quantity at that instant of time. This transaction also increases the transaction count and consumer transaction count by one.
   - `over` $[t_s, t_e]$ `consume`$(r, q(t))$ If a consume transaction and no other transactions are operating on a resource over the specified time interval, the transaction has the effect of reducing the level by the quantity evaluated at each instant of time in the interval. This transaction also increases the transaction count and consumer transaction count by one for all $t_s \leq t \leq t_e$.

   A *consume* transaction is only allowed on resources which are *replenishable* or *consumable*.
2. *Produce*:
   - `at` t `produce`(r, q) If a produce transaction and no other transactions are operating on a resource at the specified time, the transaction has the effect of reducing the level by the specified quantity at that instant of time. This transaction also increases the transaction count and producer transaction count by one.
   - `over` $[t_s, t_e]$ `produce`$(r, q(t))$ If a produce transaction and no other transactions are operating on a resource over the specified time interval, the transaction has the effect of reducing the level by the quantity evaluated at each instant of time in the interval. This transaction also increases the transaction count and producer transaction count by one for all $t_s \leq t \leq t_e$.

   A *produce* transaction is only allowed on resources which are *replenishable* or *producible*.
3. *Use*: A use operation is defined in terms of consume and produce transactions such that a consume transaction occurs at the start of the use operation and a produce transaction occurs at its end. The quantity consumed is equal to the quantity produced. This operation increases the total number of transactions by two. `over` $[t_s, t_e]$ `use`(r, q) which evaluates to
   `at` t_s `consume`(r, q) and
   `at` t_e `produce`(r, q).
   A *use* transaction is only allowed on *reusable* resources and is allowed to use only a fixed quantity.

[3] If there is any concurrency of transactions it is not always possible to say that if a production of 5 units occurs, the level := level + 5.

Queries and Constraints on resources

Once a resource is defined in terms of its properties, planners can access state of the resource during planning by querying these properties at any instant of time. It is possible to gain access to how a resource changes over time through complex queries that perform arithmetic calculations of the properties over time. The information provided by these queries is very useful in guiding a planner in its search to find a resource compliant plan. Specifically, queries about excess level of resource and the number of transactions provide a planner with the means to look ahead in the planning process and backtrack if necessary. For example, the planner can attempt to balance the consumption of the resource by prefering time intervals where there is the excess capacity is large.

The limit properties of a resource already constrain the operations that can be performed on a resource. For example, the level-limits constrain the total amount of production and consumption, while the transaction count limits constrain the number of activities producing or consuming a resource. More significantly, using the ability to build complex queries over properties, additional constraints can be incorporated into the action definitions (in conditions and effects). Since the properties are defined over time, it is also possible to define constraints that hold only over specific time periods. Complex and dynamic domains can be more naturally expressed through these constraints.

Related Work

Our work is focused on the development of a scheme for classifying and reasoning with resources in planning and we confine ourselves to a discussion of work related to this effort. For a broader examination of the issues of incorporating resources into planning systems see (Long *et al.* 2000). For a discussion of algorithmic issues more pertinent to implementation see (Laborie 2001; Muscettola 2002; Srivastava 2000; Zhou & Smith 2002).

Our work shares some common ground with other efforts to formalize representations and reasoning with resources (Smith & Becker 1997; Yang & Geunes 2001). We differ primarily in our extension to cover epistemological issues, and also in the details of the classfication.

IxTeT (Laborie & Ghallab 1995), ILOG (Laborie 2001) and O-Plan2 (Tate, Drabble, & Kirby 1994) provide examples of planning frameworks that have incorporated complex resource handling capabilities. The scope of their representation is, not surprisingly, constrained by the functionality available in their systems. They each have similar expressive power to handle resources that can be variations of the cross-product of {single-capacity, multi-capacity} and {reusable, consumable, replenishable}. They do not separate the concepts of shareability and capacity, nor do they support the notion of requiring a resource without diminishing its level. No support is provided for resource pools, though approximations are proposed which provide aggregation by creating a new resource of aggregated capacity, e.g. a pool of 10 people becomes a resource of capacity 10. Though they do provide explicit type support for resources, the type structures defined do not account for the variety of characteristics identified in this paper.

More recently, a number of planners have been developed which exploit resource constraints and resource state to control search (Do & Kambhampati 2002; Koehler 1998; Haslum & Geffner 2001; Kvarnstrom & Magnusson 2002). Our work draws from these efforts in developing commonly used terms, transactions, queries and constraints on resources.

Deficiencies in PDDL 2.1 (Fox & Long 2003) with respect to dealing with resources have been documented (Kvarnstrom & Magnusson 2002; Frank, Golden, & Jonsson 2003). For example, treatment of resources implicitly through numeric variables excludes access to aspects of resources that could be useful as operator preconditions, constraints or control rules. Furthermore, the necessity to reference resources as preconditions and effects in this form imposes an artificial serialization of activities. We further believe that the absence of explicit type support for resources makes models more difficult to comprehend due to the natural semantics of resources being implicit. It also makes it more error-prone since one cannot check compatibility of operations on the resource, e.g. producing a consumable resource. Furthermore, it makes it more difficult to map to efficient implementations that could be tailored to particular resource types.

Figure 4 is a representation of the example in figure 1 using our richer representation. A key feature is the representation of the continuous consumption of fuel over an interval of time in the effect of the action *fly*. The semantic of the consume statement is to evaluate the consumption rate (an external function that is time dependent) at various instants of time (determined by the planner implementation) to determine the quantity to be consumed. This quantity is then used in the consume operation of the resource to suitably change its level.

Conclusion

In this paper we have presented a classification scheme which circumscribes a large class of resources found in the real world. Building on the work of others we have also defined key properties of resources that allow formal expression of the proposed classification. Furthermore, operations which alter, query and constrain resource state have been presented. Together this work forms a classification and semantic description that goes a long way in formalizing the representation and reasoning aspects of resources for planning.

Due to the limits of time, a number of outstanding issues have been deferred. For example, no formal treatment is given to resource pools i.e. the ability to aggregate distinct heterogeneous resources into a pool which can be treated as a single resource and still allow allocation of transactions to individual members. Nor have the semantics for multi-dimensional resources been formalized, e.g. simultaneous treatment of the weight and volume aspects of a cargo bay. It may be worthwhile to consider explicitly modeling these composite semantics and provide operations that reflect this.

The properties of resources clearly support expression of temporally scoped limits. This seems adequate to represent

```
(:resource fueltank−in−a−airplane
  :properties (exclusive fixed multi−capacity continuous deterministic
  consumable)
)

(:durative−action fly
   :parameters (?p − plane ?t − traveller ?a ?b − location)
   :duration (= ?duration (flight−time ?a ?b))
   :condtion (and (at start (at ?p ?a))
                  (at start (at ?t ?a))
  (at start  (>= (level (fuel−tank ?p) (* (flight−time ?a
   ?b) (max−consumption−rate ?p))))))
   :effect (and (at start (not (at ?p ?a)))
                (at start (not (at ?t ?a)))
                (at end (at ?p ?b))
                (at end (at ?t ?b))
                (over [start end] (inflight ?p))
                (at end (not (inflight ?p)))
                (at end (not (aboard ?t ?p)))
                (over [start end] (consume  (fuel−tank ?p)
   (consumption−rate ?p)))))

(:objects fueltank1 plane1 city1 city2)
(:init(plane plane1)
(airport city1)
(airport city2)
(fueltank−in−a−airplane fueltank1
        :level−limit [0 2134.5]
        :level  872.7
        :consumption−rate−limit [0 10.6])
     (= (fuel−tank plane1) fueltank1))

(:extern  (#t (consumption−rate ?p)))
```

Figure 4: Examples in pseudo-PDDL describing resources and their use in an action

important patterns such as shift rotations or recurring communication windows. However, no methods have been defined to concisely impose cyclic or recurring patterns of constraints on resource properties. This will be investigated in the context of integration of resources into a formal modeling language.

A key issue for reasoning with resources is balancing the costs of checking resource constraints with the benefits that may be obtained to inform search. In scheduling, the set of activities is fixed. In planning, new activities are introduced throughout the process. Consequently, the challenge of deriving useful information from resource propagation is much greater in planning than in scheduling. This issue is highlighted by reported degradations of planner performance (Srivastava & Kambhampati 1999) as more resources are added to the problem, which is counter-intuitive. Approaches to address this have studied deferment of resource reasoning until all activities have been selected (Srivastava 2000), which effectively recasts resource reasoning as a scheduling problem addressed after the planning phase. An important extension of the work described in this paper is to explore techniques which exploit the rich semantics of the proposed representation. More sophisticated queries may be utilized to inform search. More powerful constraints may be specified to make stronger inferences despite the fact that all activities have not been selected. More refined, explicit characterizations of resources and transactions may allow specialization of resource handling algorithms based on model descriptions.

Throughout this paper it has been assumed that all resources are deterministic. It is an open question how the formalism described in this paper could or should be extended to reflect uncertainty. Furthermore, it is assumed that the semantics of external functions describing transaction quantities are defined outside of the model. Although some progress has been made in dealing with this issue, i.e. by requiring such functions to be convolvable and commutative as a prerequisite for admissibility, it is preferable to describe and enforce the semantics of such functions explicitly and unambiguously. It seems plausible that one could require registration of external functions with formal signatures. One could further introduce specified type mechanisms to characterize different classes of functions. The means to achieve this more generally is an open question.

With a view to practical application, and in support of further experimentation, we plan to incorporate the formalism proposed, together with any extensions we may develop, into formal modeling languages for domain, problem and heuristic descriptions. We also plan to integrate the proposed formalism to a planning framework such as that described in (Frank & Jónsson 2003). This effort is likely to raise many interesting issues and trade-offs in terms of modeling fidelity and computational complexity. Furthermore, domain analysis techniques shall be investigated to aid model specification and allow domain-independent specialization of implementations. This work would require mapping data structure and algorithm choices to particular classes of resources.

Acknowledgements

The authors would like to acknowledge the input of Jeremy Frank, Ari Jónsson and Paul Morris, who collaborated in early discussions preceding this work. We also wish to acknowledge the support of the NASA Intelligent Systems Program.

References

Do, M. B., and Kambhampati, S. 2002. Sapa: A scalable multi-objective heuristic metric temporal planner. *Journal of Artificial Intelligence Research*. To appear.

Fox, M., and Long, D. 2003. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*. To appear.

Frank, J., and Jónsson, A. 2003. Constraint-based attribute interval planning. *Constraints Journal, Special Issue on Planning*. To appear.

Frank, J.; Golden, K.; and Jonsson, A. 2003. The loyal opposition comments on plan domain description langauges. Sumitted to ICAPS Workshop on PDDL.

Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *IJCAI-01 Workshop on Planning with Resources*.

Koehler, J. 1998. Planning under resource constraints. In *European Conference on Artificial Intelligence*, 489–493.

Kvarnstrom, J., and Magnusson, M. 2002. TAL planner in the third international planning competition: Extensions and control rules. *Journal of Artificial Intelligence Research*. To appear.

Laborie, P., and Ghallab, M. 1995. Planning with sharable resource constraints. In *14th International Joint Conference on Artificial Intelligence*, 1643–1649.

Laborie, P. 2001. Algorithms for propagating resource constraints in ai: Existing approaches and new results. In *IJCAI-01 Workshop on Planning with Resources*.

Long, D.; Fox, M.; Sebastia, L.; and Coddington, A. 2000. An examination of resources in planning. Technical report, Department of Computer Science, University of Durham.

Muscettola, N. 2002. Computing the resource envelope for stepwise constant resource allocations. In *Principles and Practice of Constraint Programming*, 139–152.

Powell, W. B.; Shapiro, J. A.; and Simao, H. P. 2001. A representational paradigm for dynamic resource transformation problems. Technical Report CL-01-05, Department of Operations Research and Financial Engineering, Princeton University, Princeton, NJ 08544.

Smith, S. F., and Becker, M. 1997. An ontology for constructing scheduling systems. In *1997 AAAI Symposium on Ontological Engineering*. AAAI Press.

Smith, D.; Frank, J.; and Jónsson, A. 2000. Bridging the gap between planning and scheduling. In *Knowledge Engineering Review, 15(1):61–94, 2000*. Cambridge University Press.

Srivastava, B., and Kambhampati, S. 1999. Scaling up planning by teasing out resource scheduling. In *ECP*.

Srivastava, B. 2000. Realplan: Decoupling causal and resource reasoning in planning. In *AAAI/IAAI*, 812–818.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-plan2: An architecture for command and control. In Fox, M., and Zweben, M., eds., *Intelligent Scheduling*. San Mateo, California: Morgan Kaufmann.

Yang, B., and Geunes, J. 2001. Resource-constrained project scheduling: Past work and new directions. Technical Report 2001-6, Department of Industrial and Systems Engineering, University of Florida.

Zhou, Q., and Smith, S. F. 2002. An efficient consumable resource representation for scheduling. In *3rd International NASA Workshop on Planning and Scheduling for Space Applications*.

Extending PDDL to nondeterminism, limited sensing and iterative conditional plans

Piergiorgio Bertoli[1], **Alessandro Cimatti**[1], **Ugo Dal Lago**[2], **Marco Pistore**[1,3]
[1] IRST - Istituto Trentino di Cultura, 38050 Povo, Trento, Italy
[2] Università di Bologna, 40127 Bologna, Italy
[3] Università di Trento, 38050 Povo, Trento, Italy
{bertoli,cimatti}@irst.itc.it
dallago@cs.unibo.it
pistore@dit.unitn.it

Abstract

The last decade has witnessed a dramatic progress in the variety and performance of techniques and tools for classical planning. The existence of a de-facto standard modeling language for classical planning, PDDL, has played a relevant role in this process. PDDL has fostered information sharing and data exchange in the planning community, and has made international classical planning competitions possible.

At the same time, in the last few years, non-classical planning has gained considerable attention, due to its capability to capture relevant features of real-life domains which the classical framework fails to express. However, no significant effort has been made to achieve a standard mean for expressing non-classical problems, making it difficult for the planning community to compare non-classical approaches and systems.

This paper provides a contribution in this direction. We extend PDDL in order to express three very relevant features outside classical planning: uncertainty in the initial state, nondeterministic actions, and partial observability. NPDDL's extensions are designed to retain backward compatibility with PDDL, and with an emphasis on compactness of the representation. Moreover, we define a powerful, user-friendly plan language to go together with NPDDL. The language allows expressing program-like plans with branching and iterations structures, as it is necessary to solve planning problems in the presence of initial state uncertainty, nondeterminism and partial observability. We are testing NPDDL's ability to cope with a variety of problems, as they are handled by a state-of-the-art planner, MBP.

Introduction

Planning is an extremely active field of research. Because of its potential in terms of real-life applications, a wide variety of approaches have been developed, and several powerful automated planning systems have been designed to cope with complex problems. The existence of PDDL, a de-facto standard language for planning has been crucial for fostering the reuse of models, establishing a common repository of problems, and comparing and integrating systems, as it is evident from the results of the international competitions (McDermott 2000; Bacchus 2000; Fox & Long 2002).

PDDL is however limited to "classical" planning problems, and it is unable is to capture many relevant features that are important for modeling real world domains. In particular:

- the initial situation may be only partially specified;
- it is often unrealistic to assume that actions have a fully predictable outcome;
- the status of the domain may be only partially observable by the plan executor, and sensing might convey unreliable results. In general, in most cases it is unrealistic to assume the omniscience of the plan executor;
- problems of interest may often go beyond planning to reach a condition; in general, it is highly desirable to express properties about the whole execution of a plan, to state e.g. safety or maintenance requirements;
- sequences of actions are not sufficient to express solutions for problems and domains with the aforementioned features. More complex structures, e.g. loops and conditions, are required.

The growth of scientific interest for expressive planning, taking into account nondeterminism, partial observability and complex temporal goals, is evident. A number of publications and events (e.g.(C. Boutilier & Koenig 2002; Cimatti *et al.* 2001a)) have taken place, and increasingly many powerful planning systems are designed to deal with (combinations of) the features above, using a variety of approaches (Bonet & Geffner 2000a; Weld, Anderson, & Smith 1998; Smith & Weld 1998; Bertoli *et al.* 2001; Castellini, Giunchiglia, & Tacchella 2001; Kabanza 1999; Doherty & Kvarnström 2001; Rintanen 1999). However, no significant effort has been made to provide a standard mean for expressing nondeterministic, partially observable planning domains.

This paper presents the NPDDL language, a first contribution in the direction of a general PDDL-like language for planning with incomplete information. We first describe an extension to the current standard PDDL2.1 (Fox & Long 2001; Ghallab *et al.* 1998) to allow for description of nondeterministic, partially observable planning domains. Furthermore, we add a rich, user-friendly plan language that captures the iterative, branching plan structures needed to plan for domains involving the aforementioned features. The language we describe is the input to MBP (Bertoli *et al.* 2001), a state-of-the art planner integrating plan synthesis, valida-

tion and simulation within the planning via symbolic model checking framework. The input language to MBP also provides a means to express a rich class of temporal requirements; this shows the potential for further extension of the standard.

The paper is structured as follows. We first present a conceptual reference model for planning with incomplete information. We then introduce the NPDDL syntax, and show how it allows for the description of the features of interest. We present some results, compare NPDDL with the related work, and discuss some open issues. A BNF characterization of NPDDL is available at `http://sra.itc.it/tools/mbp/npddl.ps`.

The Framework

As a reference example, we consider a simple domain featuring uncertainty in the initial condition, nondeterminism and partial observability, and model a planning problem for it. The domain consists of a line of rooms that can be traversed by a robot. A printer is situated at one end of the line, and it may print a paper everytime its exit tray is empty. Each printed paper has a banner, where the destination room is reported. The banner can be read by the robot. The robot can pick up a paper at the printer, and can leave it at an office. The robot can only check the printer tray's status when at the printer's place. To identify its position, the robot is equipped with a sensor that detects whether the printer is in the same room. For this domain, we consider the problem of having the robot correctly deliver all the papers queued at the printer - the length and content of the queue being unknown, and the robot being initially positioned anywhere. A solution plan must consider the available sensing, and requires an iterative conditional structure whose execution is possibly infinite.

We rely on a simple, general framework to provide a semantic foundation to our work; this is depicted in fig.1. In our view, a domain is a (possibly nondeterministic) finite state machine, whose state evolves according to the actions received as input, and to the previous state. The domain conveys information to the plan by means of observations. We think of a plan as a deterministic finite state machine, which determines the actions to be performed according to the observations from the domain, and to its state. The execution of a plan in the domain can be thought of as an iteration where (i) an observation is given as input to the plan, (ii) the status of the plan evolves and the next action is determined, (iii) the action affects the domain status and the possible observations. A formal characterization of the framework is provided in (Bertoli *et al.* 2002b). In the following it is sufficient to limit the discussion to the underlying intuitions.

With this approach, it is possible to encompass incompletely specified initial conditions, and nondeterministic action effects. Incompletely specified initial conditions are represented by specifying a set of possible initial states of the domain. In our example, the initial states cover every possible position of the robot, and every possible content of the printer queue.

Action effects are characterized by associating actions with transitions from state to state. Nondeterministic action effects are obtained by associating an action with several transitions from the same initial state to different result states. The domain is therefore characterized as a relation, rather than as a partial function as in the deterministic case. In our example, the effect of picking a paper may result in several states, where the printer queue may be either empty, or may have on top papers with different banners.

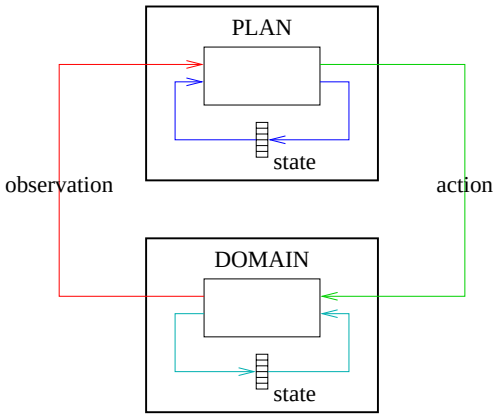


Figure 1: Planning Framework

Our modeling of observations maps states of the domain, which may not be directly observable, into observation variables, the value of which can be directly inspected at run time by the executor. Our approach, that is somewhat similar to (Goldman & Boddy 1996; Bonet & Geffner 2000b; Nourbakhsh & Genesereth 1996), also allows us to capture noisy/unreliable and partial sensing, where information is available only under specific conditions. This is achieved by relating values to observation variables not as a function of the domain state, but as relations. In this way, when an observation variable conveys no information, it can assume any value in its range, nondeterministically. As a special case, an observation O may be undefined over a domain state, by making every value of O possible on S. Within this framework, different forms of sensing can be modeled. With "automatic sensing" (Tovey & Koenig 2000), information can always be acquired, as usual in embedded controllers, where a signal from the environment is sampled and acquired at a fixed rate, latched and internally available. Observations resulting from the execution of "sensing actions" (Cassandra, Kaelbling, & Littman 1994; Weld, Anderson, & Smith 1998; Pryor & Collins 1996; Bonet & Geffner 2000a) can be modeled by representing in the domain state the last executed action, and exploiting the possibility for describing undefinedness. In our example, the domain features three observations: one, in case the robot is at the printer's place, indicates whether the tray is full or not, and provides no information otherwise; another signals whether the robot is at the printer's place or not; the last signals to the robot the current banner of the carried paper. These are easily modeled exploiting the possibility of unreliable/noisy sensing.

The idea of plans as finite state machines makes it possible to express complex courses of actions. In particular, we take into account the possibility of branching (which is able to exploit the information acquired at run-time in order to tackle the nondeterminism of the domain) iteration (which enables to express possibly cyclic, trial-and-error courses of actions). Such more expressive plans are required to solve problems under partial observability and with nondeterminism.

The NPDDL Language

NPDDL is a language that extends PDDL to encompass the intuitions outlined in previous section. NPDDL provides a set of independent extensions to describe incompletely specified initial conditions, nondeterministic action effects, and partial observability.

NPDDL starts from the standard PDDL 2.1. In particular, it starts from the level 2 of PDDL 2.1, thus inheriting a compact and clearly defined set of constructs to handle numeric and conditional effects, and first-order quantification, as well as STRIPS effects on predicates. The aspects related to the higher levels (level 3 to level 5) of PDDL2.1, such as durative actions, are not taken into account with this version of NPDDL.

In the following, we assume that we are referring to a specific ground planning domain, where $\mathcal{F}$ is the finite set of fluents. The instantiation of operators and predicates to the objects in the domain in NPDDL follows the very same schema as in PDDL. Our discussion is based on possibly non-boolean fluents; each $f_i \in \mathcal{F}$ is associated with a finite range of values. An assignment, denoted in the following by $a, a_1, a_2, \ldots$, is an expression that maps a fluent f_i on a value v_j of the associated range. (In the following, without loss of generality, we do not explicitly treat predicates, that can be seen as fluents with values over the binary range `boolean`.) NPDDL allows for n-ary constructs whenever PDDL does. However, we restrict our discussion to their binary version, leaving the trivial generalization to the reader.

Incompletely Specified Initial Conditions

PDDL allows for the specification of completely characterized initial condition, i.e. of a single initial state. This is described with the `:init` statement, containing a set of conjunctive assignments $i \doteq \{i_1, \ldots, i_n\}$. Each conjunctive assignment is a (possibly nested) conjunction of fluent assignments. The initial state specification i is in fact an implicit conjunction of fluent assignments; the top-level conjunction is left implicit. Fluents in $\mathcal{F}$ that are not explicitly assigned are treated according to the Closed-World Assumption (CWA) and given a value (by convention, we assume the first value in the range is given). More formally, let $\textsc{Assigned}(i)$ be the set of fluents assigned by i. Let $F \subseteq \mathcal{F}$ be a set of fluents, and $\textsc{Cwa}(F)$ the CWA-implied assignment to the set of fluents F, then the initial state is identified by

$$\textsc{Initial}(i) \doteq i \wedge \textsc{Cwa}(\mathcal{F} \setminus \textsc{Assigned}(i))$$

The set of assigned fluents is computed as follows:

1. $\textsc{Assigned}(\text{“}\texttt{(assign } f\ v\texttt{)}\text{”}) = \{f\}$
2. $\textsc{Assigned}(\text{“}\texttt{(and } i_1\ i_2\texttt{)}\text{”}) = \textsc{Assigned}(i_1) \cup \textsc{Assigned}(i_2)$

In NPDDL an incompletely specified initial condition is characterized by describing the set of possible initial states. To allow for multiple initial states, NPDDL introduces a `oneof` construct ($\texttt{oneof } i_1 \ldots i_n$), meaning that exactly one of the specifications described by i_j is active. Thus a statement of the form ($\texttt{oneof } i_1 \ldots i_n$) is associated to a set of partial assignments. The corresponding set of initial states is simply obtained by applying CWA to each of those partial assignments. The `oneof` construct can be combined arbitrarily with PDDL constructs, in order to allow for an independent description of uncertainty over distinct sets of domain fluents. As such, a generic PDDL initial state specification i is associated to a set of partial assignments $\textsc{AssSet}(i)$. This is defined as follows, considering the semantics of `oneof`, and that of `and` (which distributes upon the set of possible assignments):

- $\textsc{AssSet}(\text{“}\texttt{(assign } f\ v\texttt{)}\text{”}) = \{\texttt{(assign } f\ v\texttt{)}\}$
- $\textsc{AssSet}(\text{“}\texttt{(and } i_1\ i_2\texttt{)}\text{”}) = \bigcup\{\texttt{(and } b_1\ b_2\texttt{)}\}$ where $b_j \in \textsc{AssSet}(i_j)$, for $j \in \{1, 2\}$
- $\textsc{AssSet}(\text{“}\texttt{(oneof } i_1\ \texttt{...}\ i_n\texttt{)}\text{”}) = \bigcup_{1 \leq j \leq n} \textsc{AssSet}(i_j)$

The initial states are then built by complementing each partial assignment with the CWA, as follows:

$$\textsc{Initials}(i) \doteq \{\textsc{Initial}(i_j) | i_j \in \textsc{AssSet}(i)\}$$

Notice that, if the initial condition is completely specified, NPDDL maps back to PDDL: $\textsc{AssSet}(i) = \{i\}$ and $\textsc{Initials}(i) = \{\textsc{Initial}(i)\}$.

NPDDL also contains the `unknown` construct, to express the set of all possible assignments to a generic fluent f. This construct enables us to avoid explicitly listing every possible value of f. “(`unknown` f)” is equivalent to specifying a `oneof` statement on any possible value of the type of f:

```
(unknown f) ≐
  (oneof (assign f v1)...(assign f vn))
```

where $\{v_1, \ldots, v_n\}$ is the finite range of f.

Example Initially, the status of the queue is unspecified, and the robot may be in any room. This can be easily expressed in NPDDL by a conjunction of `unknown` statements.

```
(:init
   (unknown (robot_room))
   (unknown (paper_at_printer))
   (not (papers_around))
   (not (arm_busy)))
```

In the appendix, we provide a full NPDDL modeling for the reference domain and problem.

Nondeterministic Action Effects

Action effects in PDDL

In PDDL, actions are deterministic: the execution of an action in a domain state S result in a single outcome S'. The

way S' is determined depends on the interaction of PDDL's effect features: conditional effects, quantifiers and inertia handling. In order to describe actions in NPDDL, we first reduce the general structure of PDDL to a simple normal form. First, universal quantifications can be eliminated by replacement with n-ary conjunctions. Then, we observe that nested conditional effects can be eliminated by rewriting. In Appendix, we show that it is enough to consider top-level conditional effects whose branches are mutually exclusive. When executing an action A featuring such a set of top-level conditional effects, exactly one of the conditions holds, triggering the associated condition-free effect E. E's outcome consists in the (possibly partial) assignment described by E, complemented by assigning "inertially" those fluents not assigned by E:

$$\textsc{Outcome}(E) \doteq E \wedge \textsc{Inertia}(\mathcal{F} \setminus \textsc{Assigned}(E))$$

where $\textsc{Inertia}(E)$ assigns each fluent in E its current value. Notice the similarity with the way the initial state is computed.

Action Effects in NPDDL

NPDDL allows for nondeterministic actions, whose execution on a domain state S may have several possible outcomes. NPDDL uses the `oneof` construct in action effects for this purpose; intuitively, in an action effect, (`oneof` $e_1 \ \ldots \ e_n$) indicates that exactly one of the e_i effects will take place, and, as such, it is associated to a set of partial assignments (those resulting from e_i). Since NPDDL allows for a general combination of `oneof` statements with PDDL's constructs, a generic condition-free NPDDL effect e is associated to a set of partial assignments. This is computed by $\textsc{AssSet}(E)$.

The set of possible outcomes of a nondeterministic effect E simply results from the set of (possibly partial) assignments $\textsc{AssSet}(E)$:

$$\textsc{Outcomes}(E) \doteq \{\textsc{Outcome}(e_i) | e_i \in \textsc{AssSet}(E)\}$$

Notice that, if the effect is fully deterministic, $\textsc{AssSet}(E) = \{E\}$, implying $\textsc{Outcomes}(E) = \{\textsc{Outcome}(E)\}$, i.e. NPDDL maps back to PDDL.

Example Uncertain action effects are in that, when picking a paper, it may be the last or not, and on the information reported by the banner. This is modeled in the `pick_paper` operator

```
(:action pick_paper
 :precondition (and (paper_at_printer)
                    (not (arm_busy))
                    (= (robot_room) 0))
 :effect  (and
            (arm_busy)
            (unknown (paper_banner))
            (unknown (paper_at_printer))))
```

It is possible to use the `unknown` construct in action effects in order to express the assignment of a fluent to any value. As for the initial condition, this avoids the explicit listing of assignment for each possible fluent value. The `unknown` construct in action effects is handled similarly to the case of initial states: "(`unknown` f)" is equivalent to

(`oneof` (`assign` f v_1)...(`assign` f v_n))

where $\{v_1, \ldots, v_n\}$ is the finite range of f.

Compactness of the representation NPDDL is designed to compactly model domains where actions could possibly have high branching rates, and problems with a possibly large number of initial states. To this end, it is very important that `oneof` constructs can be arbitrarily nested and combined with other operators, in order to compactly specify problems with high degrees of uncertainty. A solution where `oneof` constructs are allowed only at top level would result in very clumsy specifications, since it would force considering every combination of the effects of nondeterminism/initial uncertainty over each fluent. This is also clear in the reference example, where independent facets of initial uncertainty (robot position and print queue) would otherwise result in a lengthy state enumeration.

The characterization provided in this section is by no means intended to suggest a practical way to deal with NPDDL, e.g. to build a parser and a domain constructor. One of the challenges in planning with nondeterministic domains is to be able to internally represent domains without having to enumerate their initial states and the possible action outcomes. Symbolic techniques (Rintanen 2002; Bertoli *et al.* 2001; Castellini, Giunchiglia, & Tacchella 2001) appear to have significant leverage in this respect.

Partial Observability in NPDDL

In order to allow for partial observability, NPDDL introduces a notion of "observation". Similar to (Bonet & Geffner 2000b; Nourbakhsh & Genesereth 1996), and in accord to the framework, in order to model unreliable sensing, observations are defined as arbitrary relations from the domain state to finitely valued observation variables, to be intended as the "sensors" available to the plan executor.

In the concrete syntax of NPDDL, this is achieved by a parametric `:observation` construct. An observation variable V is characterized by a boolean formula over V and the domain fluents. The intuition is that the formula defines the relation between the value of the observation variable, and the values of the (possibly unobservable) domain fluents. The formula is arbitrary, with the only (semantic) constraint that every domain state must correspond to at least one value of V. For the sake of simplicity, a domain fluent f can be declared `:observable`. This amounts to introducing a new observation, the value of which faithfully reports the values of f.

Example The sensing described in the example can be modeled as follows. Notice the way NPDDL allows describing a `paper_in_printer` sensor which does not provide information (is undefined) unless the robot is appropriately placed at the printer.

```
(:observable (paper_banner) - room_number)

(:observation (robot_at_printer) - :boolean
```

```
   (iff (robot_at_printer) (= (robot_room) 0)))

(:observation (paper_in_printer) - :boolean
   (and
     (imply (paper_in_printer)
          (or (> (robot_room) 0)
              (paper_at_printer)))
     (imply (not (paper_in_printer))
          (or (> (robot_room) 0)
              (not (paper_at_printer))))))
```

Problems in NPDDL

When dealing with nondeterminism and partial observability, a plan may admit a set of different executions, possibly resulting in different final states. This makes it necessary to specify whether every execution is required to be successful or not, and whether the possibility of having infinite executions is accepted. These natural requirements are captured by specifying that plans have to be "weak", "strong", "strong cyclic" (`:weakgoal`,`:stronggoal`, `:strongcyclicgoal` resp). Intuitively, a plan is "weak" if it admits at least one successful finite execution; it is "strong" if every admissible execution is finite and successful; it is "strong cyclic" if every (possibly infinite) execution always admits a possibility of succeeding finitely. See (Cimatti *et al.* 2001b) for formal definitions.

Moreover, several problems, e.g. classical planning problems or conformant planning problems, define implicit assumptions about the observability of the domain. To support users in naturally specifying these, NPDDL introduces an optional `:observability` keyword. This allows users to specify that a plan must be synthesized under e.g. full observability assumptions, regardless of the observations specified in the domain. Default observability is assumed to be `:full`, to retain backward compatibility with PDDL.

Example A possible expression of desired goal consists in the following:

```
(:observability :partial)
(:strongcyclicgoal
   (and
        (not (arm_busy))
        (not (papers_around))
        (not (paper_at_printer))))
```

Plans in NPDDL

In classical planning, a plan is simply a set of partially ordered actions. Nondeterminism entails the necessity for iterative structures; partial observability requires the introduction of branching in the plan language. NPDDL supports a high-level plan language. A plan may have local, internal variables, different from the ones of the domain, containing information to encode, for instance, the progress of the plan. We call such variables "plan variables"; plan variables are in a finite number, and feature finite ranges. The basic steps of the language, differently from what happens in a simple programming language, must take into account the issue of execution, with the delivery of actions to the domain actuators. The basic construct is `evolve`, with syntax

`(evolve <assignment>`$^+$ `<action-call>)`

that specifies a set of assignments to plan variables, followed by an action. Unless assigned, plan variables retain their previous value. The `action` construct, with syntax

`(action <action-call>)`

is a variation on `evolve` that does not alter the plan state. The `done` construct indicates that the plan has to be intended as terminated; no specification is given upon which actions are produced by a plan after `(done)` is executed.

The plan language also provides a number of imperative-style constructs:

- *Sequencing Commands*: `(sequence <command>`$^+$`)` corresponds to sequentially executing all the commands in the specified order.
- *Branching Commands*: `(if <condition> <plan_then> <plan_else>)`, with its usual semantic.
- *Iterative Commands*: `(while <condition> <plan>)` and `(repeat <plan>)`. The `while` semantics is standard; `repeat` causes an endless looping execution.
- *Labeling and Jumping Commands.* Labeling may refer to a command or to a given point inside a command; this is reflected by two alternative syntaxes, namely `(label <name> <command>)` and `(label <name>)`. The `(goto <name>)` construct has the usual semantics.

Example Consider the following course of actions, solving the example problem. "Loop forever, acting as follows: first the robot re-positions at the printer's room, then, if some paper is there, it picks it up, and delivers it to the proper room; if no paper is there, the plan terminates." This is represented by the following plan. Once picked a paper, to deliver it to the proper room, the robot traverses x rooms, where x is the paper banner content. Unless the printing queue is initially empty, the reference problem may require an infinitely executing plan, e.g. one like the following.

```
(define (plan deliver_paper)
(:domain paper_delivery)
(:problem continuous_delivery)
(:planvars known_room - room_number)
(:init (= (known_room) 1))
(:body
 (repeat
   (sequence
     (while (not (robot_at_printer))
       (action (move_left)))
     (if (not (paper_in_printer))
        (done)
        (sequence
          (evolve
            (assign (known_room) 0)
            (action (pick_paper)))
          (while (< (known_room) (paper_banner))
            (evolve
               (assign (known_room)
                       (+ (known_room) 1))
               (action (go_right))))
          (action (leave_paper)))))))))
```

A plan can be converted into a Finite-State Machine. The conversion procedure, that results in a definition of the tran-

sition relation of the FSM, is the following:

- Sequencing, branching and iterative constructs are removed by introducing label/jump pairs, and splitting the plan into a set of labeled plans whose bodies only includes `goto` and basic commands.
- `goto` occurrences are eliminated by inlining the bodies of the labeled plan they refer to.
- The set of labeled plans is translated into an equivalent finite state machine, whose state space is augmented with a variable ranging over the set of all used labels; this additional variable serves to model a "plan program counter" to appropriately sequentialize the executions of labeled plans.

Related Work and Discussion

Several works present PDDL-like languages to deal with some aspects related to nondeterminism (Bonet & Geffner 2000a; Kabanza 1999; Smith & Weld 1998; Weld, Anderson, & Smith 1998). The language of GPT (Bonet & Geffner 2000a) adopts a PDDL-like syntax, extended with a quantitative model with probabilities and rewards. Such issues were purposefully avoided in NPDDL, where a qualitative model is assumed. If we compare the language of GPT and NPDDL disregarding quantitative issues, however, NPDDL exhibits an improved flexibility in expressing nondeterminism. In particular, the GPT language does not allow for a general description of nondeterminism in the form of nested/conjoined nondeterministic assignments. The result raises an issue of compactness in the domain/problem descriptions, as an explicit enumeration of all of the possible initial states (or action outcomes) is in order. Our clean treatment of nondeterminism, on the other hand, opens up the possibility of extending NPDDL with quantitative aspects. In fact, the characterization in terms of sets of assignments presented in this paper seems to be amenable to a straightforward extension, that results in a quantitative model without suffering from the problems related to the explicit enumeration of initial states and transitions. The extension is the object of an ongoing investigation. As far as observability is concerned, the GPT language is based on "knowledge-gathering actions", making it difficult to express automatic sensing, where observations can be automatically gathered at each "cycle". Although NPDDL focuses on automatic sensing, action-dependent sensing can be easily encoded by suitably defining observed values as undetermined unless a given action is formerly executed.

The PDDL-like languages used in CGP (Smith & Weld 1998) and SGP (Weld, Anderson, & Smith 1998) allow the description of incompletely specified initial conditions, but are limited to deterministic actions. Sensing in SGP is restricted to knowledge-acquisition actions, and the underlying model is very limited.

SimPlan (Kabanza 1999) is able to deal with initial uncertainty and nondeterminism, but does not admit partial observability. The ADL/STRIPS interface of SimPlan allows for a very limited forms of nondeterminism, resulting from the combinations of controllable actions with environment actions. A more accurate handling of nondeterminism is only possible using SimPlan's custom interface language to specify a domain transition relation, but the specification of complex domains turns out to be very cumbersome. NADL (Jensen & Veloso 2000) is a language that allows for expressing qualitative models of nondeterministic domains in a multiagent setting. Exogenous actions are supported, as well as a limited form of concurrency for actions that involves a notion of resource constraint.

Both SimPlan and NADL try to address the fact that often a nondeterministic domain is naturally described as the composition of a (possibly deterministic) plant with some form of nondeterminism due, for instance, to uncontrollable agents in the domain. For instance, the reference example could be extended with the notion of doors between rooms, that are being opened/closed e.g. possibly by non controllable agents (this example is remarkably similar to the kid doors in (Kabanza, Barbeau, & St-Denis 1997)). The problem with this kind of dynamics is in the fact that it is not naturally described in the style of PDDL, where operators describe the possible tasks of the agent the activity of which is being planned for. Every action should have this environment dynamics in its effects, e.g. doors being nondeterministically open or closed. In fact, even if a `do-nothing` action is performed, the effect of doors possibly being open or closed by other agents should be taken into account. Similar examples arises when a system has a certain dynamics (e.g. a timer being set in a certain situation and expiring after N time units). In a PDDL-style characterization, each action should have the effect of decrementing the timer value, and possibly making the "expired" predicate become true. A design choice underlying NPDDL was to retain the operator-based description of actions. Although we believe that the current expressive power is enough to characterize many interesting domains, whether the modeling is *natural* this is an open issue. A principled analysis is in order for the extension of NPDDL to deal with this important issue.

Somewhat less related are high level action languages such as AR (Giunchiglia, Kartha, & Lifschitz 1997) and C (Giunchiglia & Lifschitz 1998), that deal with the problem of providing expressive languages for domain description in presence of nondeterminism. AR deals with ramification constraints, can represent different forms of nondeterminism, and its semantics is defined in terms of a minimization procedure to solve the frame and the ramification problem. C is an action language based on causal explanation, allowing for nondeterminism and concurrency. In both cases, the underlying semantics and the representation style are very far from PDDL's, and observability issues are not taken into account.

Finally, (Petrick & Bacchus 2002) discusses a different approach to nondeterminism, based on encoding actions at the knowledge level, and exploiting "knowledge databases" to trigger knowledge-level derivations. Knowledge-level reasoning is not explicitly addressed by NPDDL, although for several domains/problems suitable knowledge-level abstractions are possible within standard PDDL.

Results

NPDDL is the input language of the MBP planner. MBP integrates plan synthesis, plan validation and plan simulation; MBP handles plans in the format described in this paper. MBP applies some restrictions to the language; in particular, observations are currently to boolean variables and have a fixed structure; the `number` type is not allowed, and union types are not implemented. Several NPDDL examples have been designed, e.g. the "maze" benchmark for partial observability (Bertoli, Cimatti, & Roveri 2001), and domains taken from the Power Supply Restoration (Bertoli *et al.* 2002a). Moreover, MBP supports two extended goal languages for expressing maintenance goals, safety goals, liveness goals. Most of these classes identify constraints over the plan execution, e.g. a certain property must hold throughout the whole execution, or it must never hold throughout the execution, and so on. As such, they can be captured by a temporal logic that deals with nondeterminism, such as CTL (Emerson 1990). More recently, the necessity of expressing intentionality in the goals to achieve high-quality plans has been highlighted ((Pistore, Bettin, & Traverso 2001)). MBP allows for CTL and for EaGLe ((Dal Lago, Pistore, & Traverso 2002)), an extended temporal language to express intentionality into goals.

Example We consider a variation of the original goal, adding the requirement that, everytime a robot has a paper and is in the right room to deliver it, it should leave it with no delay. This can be modeled as a CTL goal:

```
(:ctlgoal
     (and
      (ag
       (imply (= (robot_paper_x) (robot_x))
              (not (next (arm_busy)))))
      (aw
         (and
            (not (papers_around))
            (ef (not (paper_at_printer))))
         (not (paper_at_printer)))))
```

Conclusions

In this paper we have presented NPDDL, an extension to PDDL for planning in nondeterministic domains. NPDDL retains all the features of PDDL, and allows for a compact characterization of incompletely specified initial conditions and nondeterministic action effects. NPDDL provides a clear solution to the issues related to the interaction between closed world assumption and initial condition, and nondeterministic action effects and law of inertia. The model for observations underlying NPDDL allows to characterize and combine action-dependent and automatic sensing. In addition, NPDDL allows to model complex plan structures and temporally extended goals. NPDDL is implemented in MBP, has been used to model complex real-world problems, and will hopefully prove to be an adequate starting point for a standard extension to PDDL. In this sense, we are studying ways to improve its flexibility, e.g. by suitably extending the PDDL `:requirements` to allow certain features (e.g. support for temporal goals) without forcibly requiring them.

Acknowledgements

The starting point of this work is a number of discussions that were in August 2001 at IJCAI, including Enrico Giunchiglia, David Smith, Blai Bonet, Keith Golden, Jussi Rintanen. We thank Marco Roveri and Paolo Traverso for many fruitful discussions on the topic.

References

Bacchus, F. 2000. AIPS-2000 planning systems competition. On-line report at http://www.cs.toronto.edu/aips2000/.

Bertoli, P.; Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2001. MBP: a Model Based Planner. In *Proc. of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*.

Bertoli, P.; Cimatti, A.; J.Slaney; and S.Thiebaux. 2002a. Solving power supply restoration problems with planning via model checking. In *Proceedings of ECAI'02*.

Bertoli, P.; Cimatti, A.; M.Pistore; and Traverso, P. 2002b. Plan validation for extended goals under partial observability (preliminary report). In *Proc. of AIPS-02 Workshop on Planning via Model Checking*.

Bertoli, P.; Cimatti, A.; and Roveri, M. 2001. Conditional Planning under Partial Observability as Heuristic-Symbolic Search in Belief Space. In *Proceedings of IJ-CAI'01*.

Bonet, B., and Geffner, H. 2000a. Planning with Incomplete Information as Heuristic Search in Belief Space. In Chien, S.; Kambhampati, S.; and Knoblock, C., eds., 5^{th} *International Conference on Artificial Intelligence Planning and Scheduling*, 52–61. AAAI-Press.

Bonet, B., and Geffner, H. 2000b. Planning with Incomplete Information as Heuristic Search in Belief Space. In *Proc. AIPS 2000*, 52–61.

C. Boutilier, T. D., and Koenig, S., eds. 2002. *"Artificial Intelligence Journal, Special Issue on Planning with Uncertainty and Incomplete Information"*. Elsevier.

Cassandra, A.; Kaelbling, L.; and Littman, M. 1994. Acting optimally in partially observable stochastic domains. In *Proc. of AAAI-94*. AAAI-Press.

Castellini, C.; Giunchiglia, E.; and Tacchella, A. 2001. Improvements to sat-based conformant planning. In *Proc. of 6th European Conference on Planning (ECP-01)*.

Cimatti, A.; Giunchiglia, E.; Geffner, H.; and Rintanen, J., eds. 2001a. *Proc. of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2001b. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. Technical report, IRST, Trento, Italy.

Dal Lago, U.; Pistore, M.; and Traverso, P. 2002. Planning with a Language for Extended Goals. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI02)*.

Doherty, P., and Kvarnström, J. 2001. TALplanner: A temporal logic-based planner. *The AI Magazine* 22(1):95–102.

Emerson, E. A. 1990. Temporal and modal logic. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier. chapter 16, 995–1072.

Fox, M., and Long, D. 2001. PDDL2.1: an extension to pddl for modelling time and metric resources.

Fox, M., and Long, D. 2002. The third international planning competition: Temporal and metric planning. In *Proc. of Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS02)*.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl—the planning domain definition language.

Giunchiglia, E., and Lifschitz, V. 1998. An action language based on causal explanation: Preliminary report. In *AAAI/IAAI*, 623–630.

Giunchiglia, E.; Kartha, G. N.; and Lifschitz, V. 1997. Representing action: Indeterminacy and ramifications. *Artificial Intelligence* 95(2):409–438.

Goldman, R., and Boddy, M. 1996. Expressive Planning and Explicit Knowledge. In *Proc. of AIPS-96*.

Jensen, R., and Veloso, M. 2000. OBDD-based Universal Planning for Synchronized Agents in Non-Deterministic Domains. *Journal of Artificial Intelligence Research* 13:189–226.

Kabanza, F.; Barbeau, M.; and St-Denis, R. 1997. Planning control rules for reactive agents. *Artificial Intelligence* 95(1):67–113.

Kabanza, F. 1999. Simplan - theoretical background.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

Nourbakhsh, I., and Genesereth, M. 1996. Assumptive planning and execution: a simple, working robot architecture. *Autonomous Robots Journal* 3(1):49–67.

Petrick, R., and Bacchus, F. 2002. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of AIPS'02, Toulouse, France*, 212–221.

Pistore, M.; Bettin, R.; and Traverso, P. 2001. Symbolic techniques for planning with extended goals in non-deterministic domains.

Pryor, L., and Collins, G. 1996. Planning for Contingency: a Decision Based Approach. *J. of Artificial Intelligence Research* 4:81–120.

Rintanen, J. 1999. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intellegence Research* 10:323–352.

Rintanen, J. 2002. Backward plan construction for planning with partial observability. In M. Ghallab, J. H., and Traverso, P., eds., *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS02)*.

Smith, D. E., and Weld, D. S. 1998. Conformant graphplan. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, 889–896. Menlo Park: AAAI Press.

Tovey, C., and Koenig, S. 2000. Gridworlds as testbeds for planning with incomplete information. In *Proceedings of the National Conference on Artificial Intelligence*.

Weld, D. S.; Anderson, C. R.; and Smith, D. E. 1998. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, 897–904. Menlo Park: AAAI Press.

PDDL Conditional effects rewriting

In the following we outline a procedure for rewriting a generic effect (with nested conditional effects) into an effect featuring only non-nested conditional effects. The procedure is based on a set of rewrite rules, and introduces, in the intermediate steps, a SWITCH construct that generalizes the when dealing with a set of mutually exclusive CASEs. The result of the procedure is a PDDL term, where the intermediate SWITCH and CASE have been eliminated. Rule 1 is used as the first rewriting step to translate whens into binary SWITCHes. Rules 2, 3 are used to bring switches to top-level, eliminating nesting within other switches or into PDDL constructs. Finally, rule 4 is used to transform the top-level switch into a set of whens, making use of the fact that the switch features mutually exclusive conditions covering all CASEs.

Rule 1: when **elimination**

```
(when <condition> <effect>)
```

becomes

```
(SWITCH
   (CASE <condition> <effect>)
   (CASE (not <condition>) true))
```

Rule 2

```
(and
    (SWITCH
       (CASE <condition_1^1> <effect_1^1>)
       ...
       (CASE <condition_{n_1}^1> <effect_{n_1}^1>))
    ...
    (SWITCH
       (CASE <condition_1^m> <effect_1^m>)
       ...
       (CASE <condition_{n_m}^m> <effect_{n_m}^m>)))
```

becomes

```
(SWITCH
   (CASE
      (and <condition_1^1> ...  <condition_1^m>)
      (and <effect_1^1> ...  <effect_1^m>))
   ...
   (CASE
      (and <condition_{n_1}^1> ...  <condition_{n_m}^m>)
      (and <effect_{n_1}^1> ...  <effect_{n_m}^m>)))
```

Rule 3

```
(SWITCH
   (CASE <condition^1> <effect^1>)
   ...
   (CASE <condition^{i-1}> <effect^{i-1}>)
   (CASE <condition^i>
      (SWITCH
         (CASE <innercondition^1> <innereffect^1>)
         ...
         (CASE <innercondition^m> <innereffect^m>)))
   (CASE <condition^{i+1}> <effect^{i+1}>)
   ...
   (CASE <condition^n> <effect^n>))
```

becomes

```
(SWITCH
   (CASE <condition^1> <effect^1>)
   ...
   (CASE <condition^{i-1}> <effect^{i-1}>)
   (CASE (and <condition^i> <innercondition^1>)
         <innereffect^1>)
   ...
   (CASE (and <condition^i> <innercondition^m>)
         <innereffect^m>)
   (CASE <condition^{i+1}> <effect^{i+1}>)
   ...
   (CASE <condition^n> <effect^n>))
```

Rule 4: when **introduction**

```
(SWITCH
   (CASE <condition^1> <effect^1>)
   ...
   (CASE <condition^m> <effect^m>))
```

becomes

```
(and
   (when <condition^1> <effect^1>)
   ...
   (when <condition^m> <effect^m>))
```

NPDDL Conditional Effects Rewriting

The following rule is used, in conjunction to rules 2 and 3, to bring switches to top-level, in the case of nondeterminism.

```
(oneof
      (SWITCH
         (CASE <condition_1^1> <effect_1^1>)
         ...
         (CASE <condition_{n_1}^1> <effect_{n_1}^1>))
      ...
      (SWITCH
         (CASE <condition_1^m> <effect_1^m>)
         ...
         (CASE <condition_{n_m}^m> <effect_{n_m}^m>)))
```

becomes

```
(SWITCH
   (CASE
      (and <condition_1^1> ...  <condition_1^m>)
      (oneof <effect_1^1> ...  <effect_1^m>))
   ...
   (CASE
      (and <condition_{n_1}^1> ...  <condition_{n_m}^m>)
      (oneof <effect_{n_1}^1> ...  <effect_{n_m}^m>)))
```

The Complete NPDDL Model

```
(define (domain paper_delivery)
  (:types room_number)
  (:predicates
    (arm_busy)
    (papers_around)
    (paper_at_printer))
  (:functions
       (robot_room)   - room_number
       (paper_banner) - room_number)

  (:action move_left
   :precondition (not (= (robot_room) 0))
   :effect       (assign (robot_room) (- (robot_room) 1)))

  (:action move_right
   :precondition (not (= (robot_room) (sup room_number)))
   :effect       (assign (robot_room) (+ (robot_room) 1)))

  (:action pick_paper
   :precondition (and (paper_at_printer)
                      (not (arm_busy))
                      (= (robot_room) 0))
   :effect  (and
              (arm_busy)
              (unknown (paper_banner))
              (unknown (paper_at_printer))))

  (:action leave_paper
   :precondition (arm_busy)
   :effect  (and
              (not (arm_busy))
              (when (not (= (robot_room) (paper_banner)))
                    (papers_around)))))

(:observable (paper_banner) - room_number)

(:observation (robot_at_printer) - :boolean
   (iff (robot_at_printer) (= (robot_room) 0)))

(:observation (paper_in_printer) - :boolean
   (and
     (imply (paper_in_printer)
          (or (> (robot_room) 0)
              (paper_at_printer)))
     (imply (not (paper_in_printer))
          (or (> (robot_room) 0)
              (not (paper_at_printer))))))


(define (problem continuous_delivery)
        (:domain paper_delivery)
        (:typedef room_number - (range 0 50))
        (:init
           (unknown (robot_room))
           (unknown (paper_at_printer))
           (not (papers_around))
           (not (arm_busy)))
        (:observability :partial)
        (:strongcyclicgoal
           (and
                (not (arm_busy))
                (not (papers_around))
                (not (paper_at_printer)))))
```

Extending PDDL for Hierarchical Planning and Topological Abstraction

Adi Botea and **Martin Müller** and **Jonathan Schaeffer**
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{adib,mmueller,jonathan}@cs.ualberta.ca

Abstract

Despite major progress in AI planning over the last few years, many interesting domains remain challenging for current planners. Topological abstraction can reduce planning complexity in several domains, decomposing a problem into a two-level hierarchy. This paper presents LAP, a planning model based on topological abstraction. In formalizing LAP as a generic planning framework, the support of a planning language more expressive than PDDL can be very important. We discuss how an extended version of PDDL can be part of our planning framework, by providing support for hierarchical planning and topological abstraction. We demonstrate our ideas in Sokoban and path-finding, two domains where topological abstraction is useful.

Introduction

AI planning has recently achieved significant progress in both theoretical and practical aspects. The last few years have seen major advances in the performance of planning systems, in part stimulated by the planning competitions held as part of the AIPS series of conferences (McDermott 2000; Bacchus 2001; Fox & Long 2002). However, many hard domains still remain a great challenge for the current capabilities of planning systems.

Abstraction is a natural approach to simplify planning in complex problems. For instance, humans often create abstract plans that they try to follow during their search. In this paper we present topological abstraction, a technique for reducing planning complexity in hard domains. Based on this abstraction model, we also discuss extending the PDDL language so that it supports hierarchical planning and abstraction. In exploring how PDDL can be extended, the meaning of "abstraction" can be more general than our topological approach. Topological abstraction is only one of a general class of hierarchical relationships that PDDL may not be able to express well.

Our abstraction approach reformulates the state representation, grouping related low-level features in local clusters. The clustering aims to catch local relationships inside clusters and keep cluster interactions as low as possible. In effect, the initial problem is decomposed into a two-level hierarchy of sub-problems, each being much simpler than the initial one. At the local level, each cluster has associated a local problem that solves the local constraints. There is also a global planning problem where clusters are treated as black-boxes and local state features are hidden away. The reason why we call this topological abstraction is that an important class of applications of which the approach is suitable have a spatial structure such as a grid. In such a domain, we group atomic grid squares into abstract clusters such as rooms in a building.

Motivation

Many interesting domains are hard to deal with when no abstraction is present. Examples of such domains are Sokoban and path-finding. In these domains, a hierarchical problem decomposition based on topological clustering can lead to significantly better performance. Our preliminary work using these domains as a testbed has already shown an impressive potential of the topological abstraction.

Sokoban is a puzzle with many similarities to a robotics application. In this domain, a man in a maze has to push stones from their initial positions to designated destinations called *goal squares* (see Section 3 for a detailed description of the rules). Both the AI planning and the single-agent search communities agree that this is a hard domain. The game is difficult for a computer for several reasons including deadlocks (positions from which no goal state can be reached), the large branching factor (can be over 100 – if we consider as moves all the stone pushes in the man reachable area), and long optimal solutions (can be over 600 moves). Another problem is that all known lower-bound heuristic estimators for the solution length are either of low quality, or expensive to compute.

Humans, who solve Sokoban puzzles much easier than state-of-the-art AI applications, abstract the maze into rooms and tunnels and use this high-level representation to create abstract plans. Following the humans' example, an AI application can cluster atomic squares into more abstract features such as rooms connected by tunnels, reducing the complexity of the hard initial problem. In effect, a large number of atomic squares is replaced by a few abstract, more meaningful features such as rooms and tunnels.

In the domain of path-finding, an agent on a map has to find a (shortest) path from its current position to a destination position. The map topology can have many forms, such as a battlefield, the interior of a building, etc. The problem is important in commercial computer games, robot planning,

military applications, etc. The efficiency of the path-finding algorithms is often crucial, as they have to produce solutions in real-time and use limited resources. The classical solving strategy represents the maze as a grid of atomic cells and uses a search algorithm such as A* on that graph. An action is to move to an adjacent cell that is not part of an obstacle. The representation of states in the search space greatly influences the efficiency of the search. A fine granularity of the map leads to a large search space, requiring serious time (and possibly space) resources. A much more efficient problem representation is to abstract the map into connected clusters such as rooms, large obstacle-free areas, bridges, etc. As in Sokoban, the abstract map representation is a small graph of connected clusters, with a much reduced search space.

To the best of our knowledge, in some commercial games the search space is abstracted by human experts, who define the abstract clusters by hand. Our contribution is an automated abstraction method, especially useful when human expertise is expensive or not available. For instance, a bomb can destroy a bridge, changing the landscape dynamically and invalidating the previous abstract representation. Also, the user of a game may be allowed to define new map configurations, which have to be abstracted from scratch.

In standard planning domains such as Logistics, topological abstraction of the real world is part of the domain definition. In Logistics, several packages have to be transported from their initial location to various destinations. A Logistics problem has a map of cities connected by airline routes. Transportation inside cities can be done by truck (there is one truck in each city). Cities are abstracted, being treated as black boxes. Inside a city, a truck can go from any point to any destination at no cost. However, in the real world, transportation within a city is a subproblem that can involve considerable costs. In this context, removing human expertise and automatically obtaining abstracted models of the real world is an important research problem.

The Planning Language Support

Topological abstraction is appropriate for several application domains. Our goal is to build a general planning framework where topological abstraction is automatically performed for different planning domains. In such a framework, the robustness of the planning language used to describe the domain and problem instances is very important. Many parts of our abstraction framework could more easily be expressed when using a more general planning language. The language support for hierarchical planning in general should deal with representing the abstraction levels, and modeling relationships and communication across the levels. The language support for abstraction should cover several issues, such as problem reformulation, automatic abstraction, adaptive abstraction, and a hybrid problem representation. In this paper, *hybrid representation* refers to using both low-level features (for the part of the problem representation space not abstracted yet) and abstract features (for the already abstracted part of the space) to represent a problem state. In our framework, problem reformulation means to replace a low-level domain and problem representation by an equivalent abstract representation, which is easier to solve. We want to represent the abstracted problem explicitly, as an independent planning problem written in a language such as PDDL. This allows solving the abstract problem with no interaction with the initial low-level formulation. Another advantage of representing abstraction as part of the PDDL formulation is that, at one moment, we can use a hybrid state representation, using both low-level and abstract features for state description. When planning is done repeatedly in a fixed environment, an adaptive abstraction, which is performed as the system learns more about the environment, is also valuable. For instance, in a path-finding problem the map is initially represented at the low-level. An adaptive abstraction algorithm builds the clusters gradually, as the planning agent discovers more and more parts of the map. Before the abstraction is completed, the planning is done using a state representation composed of both atomic squares (for the unexplored parts of the map) and abstract clusters (for the explored parts of the map).

Adaptive abstraction can naturally be related to planning with uncertainty. We can consider that the part of the problem not abstracted yet is in a sense unknown to the planner. Using abstraction this way also required the domain description (or, more generally, the planning and plan execution framework) to handle uncertainty. PDDL currently doesn't do this, and topological abstraction can't handle this without a treatment of uncertainty. Even if this is an interesting topic, in this paper we don't focus on how an extended PDDL can be used to better handle uncertainty. We keep our discussion limited to hierarchical planning and abstraction issues.

The rest of the paper is structured as follows: In the next section we review the related work. In the third section we highlight our abstraction framework and briefly describe how we applied it to Sokoban and path-finding. We point out some features that can easier be addressed using a more robust planning language. In the fourth section we discuss extending PDDL to support hierarchical planning and topological abstraction. The last section presents our conclusion.

Related Work

Abstraction is a frequently used technique to reduce problem complexity in AI planning. Automatically abstracting planning domains has been explored by Knoblock (Knoblock 1994). His approach builds a hierarchy of abstractions by dropping literals from the problem definition at the previous abstraction level. Bacchus and Yang define a theoretical probabilistic framework to analyze the search complexity in hierarchical models (Bacchus & Yang 1994). They also use some concepts of that model to improve Knoblock's abstraction algorithm. In this work, the abstraction consists of problem relaxation. In our approach, abstraction means to reformulate a problem into an equivalent hierarchical representation. The abstract problem is solved independently from the initial problem formulation.

Long *et al.* use *generic types* and *active preconditions* to reformulate and abstract planning problems (Long, Fox, & Hamdi 2002). As a result of the reformulation, subproblems of the initial problem are identified and solved by

using specialized solvers. Our approach has similarities with this work. Both formalisms try to cope with domain-specific features at the local level, keeping the global problem as generic as possible. The difference is that we reformulate problems as a result of topological abstraction, whereas in the cited work reformulation is obtained by identifying various generic types of behavior and objects such as *mobile objects*.

Using topological abstraction to speed-up planning in a reinforcement learning framework has been proposed in (Precup, Sutton, & Singh 1997). In this work, the authors define macro actions as *offset-casual* policies. In such a policy, the probability of an atomic action depends not only on the current state, but also on the previous states and atomic actions of the policy. Learning macro actions in a grid robot planning domain induces a topological abstraction of the problem space.

Previous experiments showed that planning in a low-level Sokoban formulation was too hard for state-of-the-art generic planners (McDermott 1997; Junghanns & Schaeffer 1999). Culberson performed a theoretical analysis of Sokoban, showing that this domain is PSPACE-complete (Culberson 1997). The state-of-the-art Sokoban solvers are Junghanns' *Rolling Stone* (Junghanns 1999; Junghanns & Schaeffer 2001) and *deep green*, developed inside the Japanese Sokoban community (Junghanns 1999). These applications can find solutions for two thirds of the standard 90-problem test suite [1].

Local Abstraction in Planning

In this section we present an overview of our abstraction model, called LAP (Local Abstraction in Planning). We also show how the model can be applied to domains such as Sokoban and path-finding. We use the model and these domains as a basis to motivate the need for a PDDL extension supporting hierarchical planning and abstraction. Thus, implementation details and analysis of experimental results are not our focus here. We rather consider issues such as hierarchical planning, automatic clustering, adaptive clustering, and hybrid state representation.

The Model Overview

LAP is a planning model based on a topological abstraction of the state representation. A clustering of the problem representation space is used to group related low-level features. The goal of the clustering process is to group together related atomic pieces and keep cluster interactions low. The abstraction allows us to decompose the initial problem into a hierarchy of sub-problems in a divide-and-conquer manner. For each cluster we define a local problem, which solves the local constraints of that cluster. The global problem uses an abstract problem description, where global states are characterized by states of abstract features. Each feature is a cluster that represents several atomic elements of the space.

At the global level, our abstraction approach leads to a much more compact state representation. For instance, a *room* in a robot planning domain is an abstract feature encoding many low-level objects such as atomic-size *squares*. Since one cluster is a complex feature representing several atomic features, cluster states can have many possible values. It is therefore natural to represent the global abstract states as tuples of cluster values. Using this representation, our abstraction model can be defined as a special case of the Simplified Action Structures (SAS) model (Bäckström & Klein 1991; Bäckström & Nebel 1995). In a SAS model, the global state space is a cross product of sub-spaces describing states of the problem features. Actions have associated three types of feature sets: *precondition* sets, *effect* sets, and *prevail* sets. The precondition set identifies the features used to check the action preconditions, the effect set contains the features whose states are changed by the considered action, and the prevail set contains the features whose values are preserved after the action has been applied. Below we point out the properties that differentiate our model from other existing SAS structures. First, in the LAP formalism, an abstract action changes either one state component or two components, leaving the rest of the tuple unchanged. In other words, the planning agent is only allowed to do local processing inside a cluster or perform an action affecting two clusters. Second, a transition between two clusters is possible only if the two clusters are neighbors. Third, when checking the preconditions of an operator, the only preconditions that matter are the values of the components that are changed by the action effect. For instance, in the Sokoban domain, when transferring a stone between two adjacent rooms A and B, the local stone configuration of other rooms is not relevant.

At the global level, we use abstract planning actions called macro-actions. Checking the preconditions of a macro-action uses cluster states rather than states of atomic features. The effects of a macro-action also change cluster states. The model does not guarantee the solution optimality. If for each action of an optimal abstract solution we compute an optimal sequence of atomic moves, the resulting low-level solution is not guaranteed to be optimal.

LAP in Sokoban

Sokoban is a single player game created in Japan in the early 1980s. Figure 1 shows an example of a Sokoban problem. The puzzle consists of a maze which has two types of squares: inaccessible *wall squares* and accessible *interior squares*. Several *stones* are initially placed on some of the interior squares. There is also a *man* that can walk around by moving from his current position to any adjacent *free* interior position. A free position is an interior square that is not occupied by either a stone or the man. If there is a stone next to the man and the position behind the stone is free, then the man can *push* the stone to that free square. The man moves forward to the initial position of the stone. The goal of the puzzle is to push all the stones to some specific marked interior positions called *goal squares*.

The game is difficult for a computer for several reasons including deadlocks, the large branching factor, and long optimal solutions. Also, all known lower-bound heuristic estimators are either of low quality, or expensive to compute.

[1] The test suite is available at `http://xsokoban.lcs.mit.edu/xsokoban.html`.

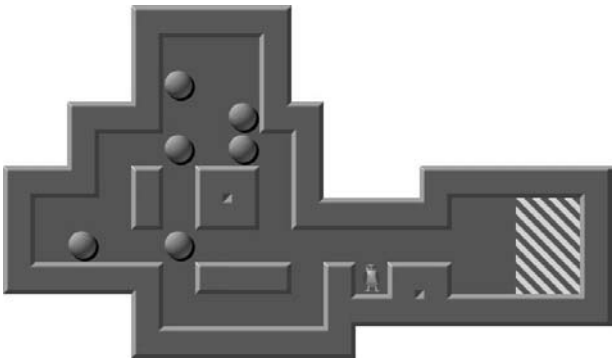


Figure 1: Problem #1 in the standard 90 problem Sokoban test suite. The six goal squares are the marked ones at the right end of the maze.

A simple planning representation of Sokoban can be obtained by translating all the low-level properties of the game into a planning language such as PDDL. For instance, a one-square push becomes one planning action. We call this *plain Sokoban*. This simple representation leads to very poor results, as it does not allow for an efficient handling of the long-range properties of the game. The domain formulation can be significantly improved by applying the LAP model to plain Sokoban. We call the new abstracted formulation *abstract Sokoban*. In abstract Sokoban, the clustering produces a decomposition of the maze into *rooms* and *tunnels*. Rooms and tunnels become clusters in the abstract representation. If there are k clusters extracted from the maze, the abstract description of a state is a tuple of integer values

$$s = (s_1, s_2, ..., s_k)$$

where s_i represents the internal state of cluster i. The internal state of a cluster is a complete description of the stone configuration and, in the case of rooms, the area reachable by the man inside that room. The abstract planning actions are: (a) to re-arrange stones inside a room, so that the man can walk between two designated entrances and (b) to transfer a stone between two rooms, or between a room and a tunnel. In case (a) we have *unary* operators, as they only change the status of one cluster. In case (b) we have *binary* operators, as they change the status of two adjacent clusters.

Figure 2 illustrates how our abstraction model works in Sokoban. At the global level, a search problem is transformed into a graph (R_i, T_j), where the nodes R_i represent rooms and the edges T_j represent tunnels. In effect, the global problem has a much smaller search space. Besides the global planning problem, we also obtain several local search problems, one for each room. Local problems check the action preconditions for the global planning problem. For instance, if the abstract action is to transfer a stone from room A to room B, we have to check that the local stone configurations allow this macro-action. Tunnels are so simple that the associated local problems are trivial.

State of the art in solving Sokoban is about 60 out of 90 problems solved by *Rolling Stone* and *deep green*. Our system, called *Power Plan*, has solved 25 problems so far. We consider our preliminary results very encouraging, as they show a great reduction of the problem complexity. For comparison, we could not solve any problem from the standard test-suite by using a non-abstracted representation of Sokoban. Using a partial abstraction (i.e., only tunnels were abstracted), we solved only one problem. We believe that we can further improve our performance in Sokoban in the future. The limitations are on the lack of domain-specific knowledge of Power Plan, not on the abstraction approach.

LAP in Path-Finding

In path-finding, an agent on a map has to find a (shortest) path from his current position to a destination position. The map topology can have many forms, such as a battlefield, the interior of a building, etc. The problem is important in computer games, robot planning, military applications, etc. The classical solving strategy is based on single agent search. In this approach, the map is represented as a grid of atomic cells, and a search algorithm such as A* is used to search for the solution. An action is to move to an adjacent cell, if the destination cell is not an obstacle (Yap 2002).

Our abstraction model groups atomic cells into abstract clusters, reducing the size of the search space dramatically. Actions become moving between two entrances of a cluster (crossing a cluster), rather than moving from one cell to the next. In our experiments, we split the maze into equal-size boxes which become abstract clusters. For example, a 100×100 map can be decomposed into 100 10×10 boxes (clusters). For each edge common to two adjacent clusters, we identify entrances for communication between clusters. An entrance is an obstacle-free part of the edge bounded by two obstacles. For each entrance, we define one *transition point* at the middle of that entrance. No points other than the transition points can be used for moving from one cluster to another.

We can identify a global problem and several local problems, one for each cluster. Processing performed inside a cluster is part of the local problem associated to that cluster. For each pair of transition points on the border of the same cluster, we compute an optimal path between them that is contained in that cluster. Since in path-finding different problem instances use a fixed map but different $(start, target)$ node pairs, this pre-processing phase is performed once and re-used for *many* problem instances. For $n \in \{start, target\}$, we also compute optimal paths from n to the transition points located on the border of the cluster that contains n.

At the global level, we define the *abstract search graph*, whose nodes are $start$, $target$, and the transition points. Optimal paths between the nodes become weighted edges. The abstract graph can easily be updated for different problem instances, as we only have to update information about $start$ and/or $target$. Since the map if fixed, the rest of the abstract graph is fixed too. Searching in the abstract graph is the global problem. In Figure 3 we illustrate this abstraction process on a 20×20 map.

We compared our method with A* performed at the atomic level. Our first results show a great reduction in the search effort. Searching in the abstract graph expands one

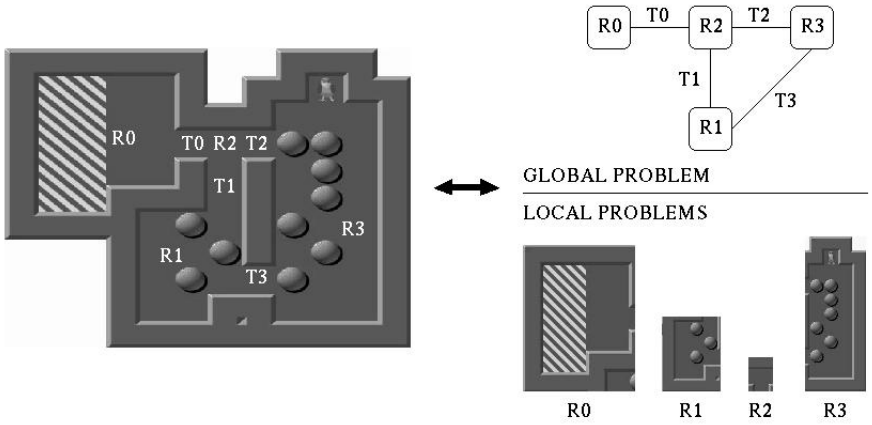


Figure 2: A Sokoban problem (#6 of the test suite) is decomposed into several abstract sub-problems. There is one global problem as well as one local problem for each room. Rooms and tunnels are denoted by R and T, respectively.

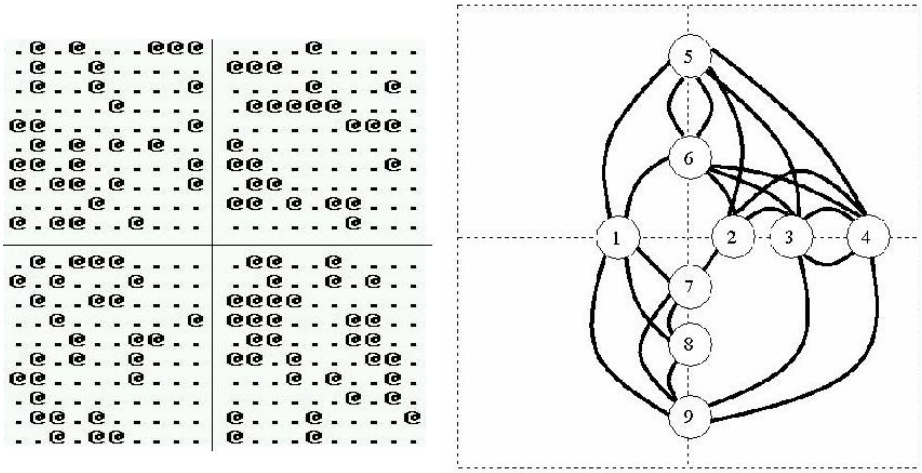


Figure 3: A 20×20 map is decomposed into 4 abstract clusters. The obstacles on the map are represented by '@'. A search space having 280 non-obstacle squares is replaced by a small abstract graph having 9 nodes (not including $start$ and $target$). The abstract search space is shown on the right side of the picture.

order of magnitude less nodes than searching at the atomic level. The length of the solution computed with our method is very close to the optimal value. When parameters such as the maze size or the obstacle rate are varied, the average increase of the solution length (as compared to the optimal value) is consistently under 1%. These results are especially valuable as, in path-finding, the speed of the search algorithm is often crucial, while the solution optimality condition can be relaxed. For instance, in a commercial computer game, most of the CPU cycles are allocated to other game modules such as the graphics engine. In addition, solutions don't have to be optimal. Sub-optimal solutions that look realistic will do, too.

We plan to extend our work in path-finding in many directions, including natural clustering, adaptive abstraction and hybrid map representation. As we show in the next section, when exploring these directions in a generic planning framework, the support of the planning language can be very important. Even if our technique for maze clustering turned out to be quite efficient, we still want to explore how to discover more natural clusters such as rooms inside a building. Natural clusters can lead to simpler local problems. For instance, if the cluster does not contain any obstacle square, then optimal paths between entrances are computed instantly, reducing the pre-processing costs. In a path-finding problem, parts of the map can be unknown for the planning agent. For instance, this is the case when the map is so large that it does not fit into the computer's memory. In this case, the feature clusters can be built dynamically, as the planning agent discovers more and more parts of the map. When the map is partially abstracted, the search can be performed in a space representation containing both abstract local clusters (for the

already explored parts of the map) and atomic-size squares (for the unknown parts of the map).

Extending PDDL

In this section we discuss in more detail how a more powerful version of PDDL would be beneficial for using abstraction and hierarchies in AI planning. To better express our vision, we use Sokoban and path-finding as case-study domains.

Since topological abstraction is useful in several domains, we plan to extend our work in the direction of developing a generic planning framework based on this type of abstraction. A significant part of this generic framework can be provided by a more general planning language. As we have pointed out before, the language support that we need mainly refers to hierarchical planning and topological abstraction. The support for hierarchical planning should allow us to express an abstracted problem as a sum of subproblems having different abstraction levels. The support for abstraction includes modeling the relation between low-level and abstract features, automatic abstraction, adaptive abstraction, and using hybrid state representations. The language should also support users to guide the abstraction process by encoding hints that should be considered by the abstraction algorithm.

The rest of the section is structured in two subsections. The first subsection identifies challenges about extending PDDL. The second subsection shows concrete steps toward solving some of the issues pointed out in the first subsection.

Challenges

In this subsection we identify some challenges that we have faced that could be more easily addressed by using a more general planning language. In our experiments with abstract Sokoban (Botea, Müller, & Schaeffer 2002; Botea 2002), we used TLPlan (Bacchus & Kabanza 2000), a generic planner which allows users to plug in domain specific knowledge as C code libraries. Next we show the parts of our framework that had to be implemented as custom code.

Since the planning language did not support hierarchical planning, we represented only the global component as a planning problem (i.e., formulated in PDDL and solved by the generic planner). The solver for the local problems was implemented as custom code in C. This approach definitely has the advantage of efficiency. On the other hand, it clearly points out the need for a unified hierarchical planning framework. In such a framework, users should have the opportunity to describe domains using linked levels of abstraction. As opposed to hierarchical task networks, our abstraction approach defines boundaries between problem components. Users should be supported in expressing topological features of domains such as connectivity of spatial structures within the domain.

All the abstraction levels of a problem (i.e., in our approach, both the global problem and the local problems associated to clusters) should be represented as part of a single PDDL problem formulation. This formulation also encodes relationships between problem components, as well as other useful information that users may consider for explicit representation. The issue of PDDL expressiveness should be kept separated from the strategy adopted for solving planning problems. However, we want to point out that a unified framework may also mean a common solving strategy. In the unified model, we could perform both the high level planning and the low-level computation. The same solving engine can be used to solve both levels of the problem.

The translation of a planning problem from the low-level representation to the hierarchical representation is also part of the planning framework. This means that the abstraction process is also integrated in the model. PDDL could be extended to formalize the codification of rich control knowledge, including hints about how best (in the encoder's view) to decompose a problem into components.

Integrating the abstraction process within the model leads to the interesting and more general debate whether PDDL should be a user-level language or a machine-level language.

In the first scenario (i.e., user-level), the language is used only as an interface through which the planner communicates to the outside world. The planner input (i.e., domain and problem definition) is formulated in PDDL. The planner's output (i.e. a sequence of actions representing the problem solution) can also be considered as a PDDL sequence. All the internal problem representations used by the planner are not part of the generic planning framework. In this scenario, there is a gap in the framework between the low-level representation and the abstracted representation of a problem. Since internal problem representations are hidden and planner specific, we cannot have both a low-level and an abstract PDDL formulation for the same problem. We either start the solving process with an abstracted PDDL formulation, or perform the abstraction internally, being unable to access the internal abstracted problem representation. In our previous experiments in Sokoban we performed the abstraction *a priori*, as a separate pre-processing step. The input of the planner was a PDDL formulation of the global component of the abstracted problem.

The more interesting scenario is to consider PDDL as a machine-level planning language. This allows us to integrate the abstraction process into the framework. Planners can use the language to express internal problem representations at various stages of abstraction. Several possible problem representations, having different abstraction degrees, can be formulated in PDDL and used internally by the planner. This sets a better framework for planners to automatically discover useful abstractions and represent them in PDDL. This also induces greater task modularity and standardization. Abstracted problem representations can be produced and used by other solvers. Last but not the least, when written in PDDL, internal problem representations are easier to understand for humans.

Language Extension Ideas

In this subsection we propose some solutions to the challenges presented above. First, we show how an abstracted problem can be formulated in an unitary framework. Second, we provide concrete ideas about supporting the abstraction process. We include language features that allow users

to guide the abstraction.

When expressed in the extended PDDL, the abstract problem formulation should actually be a set of inter-dependent sub-problems defined in the same PDDL file. There is a global problem and several local problems, one for each cluster. The file should also contain inter-relationships occurring across the abstraction levels. To identify the sub-problems, we can introduce two new keywords to the language: `global` and `local`. These keywords are used for the problem formulation, not the domain formulation. The first keyword is part of the global problem header. This problem is described using abstract state features and abstract actions. When we use a hybrid state representation, low-level features can be part of the global state description too. However, to keep our presentation simple, we ignore this possibility for now. The keyword `local` introduces the description of a local problem. For instance, the statement:

```
local:  room1
```

can be the header of the local problem associated to the abstract object called `room1` (which was defined inside the global problem). The header should be followed by the PDDL description of the problem. This problem description is at the low level.

Since the global states are described using new abstract features, we need new types of global actions, that change cluster states. The global actions should also be included in the global problem definition. The initial low-level actions become part of the local problems. We point out again that how to define abstract actions should be kept separated from the problem of extending the language. What we consider more relevant is the mechanism for computing action preconditions and effects in the global problem. This mechanism actually establishes the relationship between the local level and the global level of a problem.

The local problems do not interact directly. The only problem interaction allowed in our model is between the global problem and a local problem. At the global level, the clusters are treated as black boxes. When solving the global problem, the planner may need information about the clusters. This information is necessary to check action preconditions (i.e., whether the current state of a cluster allows performing a certain action) and compute action effects (i.e., the resulting internal state of a cluster when a certain action is performed). The cluster information is provided by the local problem associated with the corresponding cluster.

The planner starts solving the global problem. When information about a cluster is necessary, the planner stops solving the global problem and computes the needed piece of knowledge by performing a search in the corresponding local problem. After the information needed at the global level becomes available, the solving of the global problem resumes. There are several ways to optimize this problem solving approach at the local level. First, when local problems are small enough, they can be solved *a priori* (i.e., compute and store all the information about the corresponding cluster that may be needed for the global problem). Second, the results of on-demand local computations can be cached and re-used when needed again. Third, several equivalent cluster states can be merged to compose one abstract state of a cluster.

In Sokoban, we performed the problem abstraction as an application-specific method, with no interference with the generic planning framework. However, we want to develop generic abstraction methods, integrated in our planning model. Since it is often hard to find "good" abstractions by using generic methods only, we consider that language features allowing users to guide the abstraction process are useful.

At one extreme, the user's hints could actually complete the abstraction process. For instance, for each atomic square $square_i$ we can declare:

(hint (belongs_to square$_i$ room$_j$)).

This series of statements shows precisely how to build the abstract clusters.

On the other hand, the abstraction process can be automated. The rest of our discussion focuses on this case. The user can assist the abstraction process by encoding hints about how best to decompose a problem into components. A very simple example is the following:

```
(hint (= (max_cluster_size 10))),
```

stating that a cluster should contain at most 10 atomic squares.

Another possible language extension supporting automatic abstraction is the following. Let us assume that the language accepts the declaration:

```
(abstracts room square)
```

as part of the domain formulation. `square` is a predicate instantiated in the low-level problem description, `room` defines an abstract feature, and `abstracts` is a key word of the language. The semantic of this statement is that the domain can be topologically abstracted by building rooms out of (closely related) squares. When a problem is initially loaded to the planning system, no room object is instantiated. Since the system knows that squares can be grouped together to form rooms, a clustering method can be used to discover rooms for the given problem. The clustering algorithm could be either domain-specific or generic. The algorithm can use hints that the user formulates as part of the domain or problem definition. When formulated in the domain definition, the hints apply to all problem instances of that domain. When formulated in the problem definition, the hints apply to the considered problem.

The computed clusters replace the corresponding low-level features in the global problem representation. These low-level features become part of the local problems corresponding to those clusters.

The discussion on hierarchical planning and abstraction applies to both our test cases, Sokoban and path-finding. In addition, path-finding is a good test-bed for adaptive abstraction and hybrid state representation. As in the Sokoban example, the declaration:

```
(abstracts cluster square)
```

can be part of the domain definition in path-finding. Initially, no cluster object is instantiated, since the planning agent

did not explore the map at all. On this map, the planning agent is requested to perform many searches, for different start and destination points. For instance, in a commercial game, there can be many characters that have to travel across the map. Moreover, one character can do many trips during one game. As the planning agent performs more and more searches, it also learns more about the map, being able to abstract the already explored parts. In effect, the state representation changes gradually, as more and more low-level squares are replaced by abstract clusters. After building one more abstract cluster, the global problem changes, replacing several low-level squares by an abstract feature. Also, one more local problem, corresponding to navigation within the newly created cluster, is added. Before the abstraction is completed, we need to be able to represent a state as a mixture of both low-level squares and abstract clusters. This means that the global problem accepts both abstract clusters and low-level squares for state representation.

Conclusion

Topological abstraction is a powerful technique for reducing problem complexity in AI planning and single-agent search. The method is based on a clustering of the initial problem representation space. The clustering catches local relationships inside clusters and keeps cluster interactions as limited as possible. In effect, the initial problem is decomposed into a two-level hierarchy of sub-problems, each being much simpler than the initial one. At the local level, each cluster has associated a local problem that solves the local constraints. There is also a global planning problem which uses clusters as features in the global state description.

Since this model is useful in several application domains, it is worth to build a generic planning framework using topological clustering. In such a framework, the expressiveness of the planning language can have a great importance. In this paper we discussed an extension of the PDDL language supporting hierarchical planning and topological abstraction. We pointed out challenges that could be better solved with a more general planning language. We also presented ideas about how to solve these challenges. We demonstrated our ideas using Sokoban and path-finding, two domains where a hierarchical approach based on topological abstraction can be beneficial.

Acknowledgment

This research was supported by NSERC and iCORE. Also, the authors would like to thank the reviewers of this paper. Their feedback truly helped us improve the quality of this paper.

References

Bacchus, F., and Kabanza, F. 2000. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence* 16:123–191.

Bacchus, F., and Yang, Q. 1994. Downward Refinement and the Efficiency of Hierarchical Problem Solving. *Artificial Intelligence* 71(1):43–100.

Bacchus, F. 2001. AIPS'00 Planning Competition. *AI Magazine* 22(3):47–56.

Bäckström, C., and Klein, I. 1991. Planning in Polinomial Time: The SAS-PUBS Class. *Computational Intelligence* 7(3):181–197.

Bäckström, C., and Nebel, B. 1995. Complexity Results for SAS + Planning. *Computational Intelligence* 11(4):625–655.

Botea, A.; Müller, M.; and Schaeffer, J. 2002. Using Abstraction for Planning in Sokoban. To appear in *Proceedings of the 3rd International Conference on Computers and Games (CG'2002), Edmonton, Canada*.

Botea, A. 2002. Using Abstraction for Heuristic Search and Planning. In Koenig, S., and Holte, R., eds., *Proceedings of the 5th International Symposium on Abstraction, Reformulation, and Approximation*, volume 2371 of *Lecture Notes in Artificial Intelligence*, 326–327.

Culberson, J. 1997. SOKOBAN is PSPACE-complete. Technical report, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada. `ftp://ftp.cs.ualberta.ca/pub/TechReports/1997/TR97-02`.

Fox, M., and Long, D. 2002. The Third International Planning Competition: Temporal and Metric Planning. In *Preprints of The Sixth International Conference on AI Planning and Scheduling, Toulouse, France*, 115–118.

Junghanns, A., and Schaeffer, J. 1999. Domain-Dependent Single-Agent Search Enhancements. In *Proceedings IJCAI-99, Stockholm, Sweden*, 570–575.

Junghanns, A., and Schaeffer, J. 2001. Sokoban: Enhancing Single-Agent Search Using Domain Knowledge. *Artificial Intelligence* 129(1–2):219–251.

Junghanns, A. 1999. *Pushing the Limits: New Developments in Single-Agent Search*. Ph.D. Dissertation, University of Alberta.

Knoblock, C. A. 1994. Automatically Generating Abstractions for Planning. *Artificial Intelligence* 68(2):243–302.

Long, D.; Fox, M.; and Hamdi, M. 2002. Reformulation in Planning. In Koenig, S., and Holte, R., eds., *Proceedings of the 5th International Symposium on Abstraction, Reformulation, and Approximation*, volume 2371 of *Lecture Notes in Artificial Intelligence*, 18–32.

McDermott, D. 1997. Using Regression-Match Graphs to Control Search in Planning. `http://www.cs.yale.edu/HTML/YALE/CS/HyPlans/mcdermott.html`.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 21(2):35–55.

Precup, D.; Sutton, R.; and Singh, S. 1997. Planning with Closed-loop Macro Actions. In Working notes of the 1997 AAAI Fall Symposium on Model-directed Autonomous Systems, 1997.

Yap, P. 2002. Grid-based path-finding. In Cohen, R., and Spencer, B., eds., *Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence*, 44–55.

A Multiagent Planning Language

Michael Brenner
Institute for Computer Science
University of Freiburg, Germany
Georges-Koehler-Allee, Geb. 052
79110 Freiburg
brenner@informatik.uni-freiburg.de

Abstract

This paper discusses specific features of planning in multiagent domains and presents concepts for a multiagent extension of PDDL, the Multiagent Planning Language MAPL ("maple"). MAPL uses non-boolean state variables and thus allows to describe an agent's ignorance of facts as well as a simplified mutex concept. The time model of MAPL is based on Simple Temporal Networks and allows both quantitavive and qualitative use of time in plans, thereby subsuming the plan semantics of both partial order plans and PDDL 2.1.

Introduction

In this paper we describe some properties specific to planning in multiagent systems and, resulting from these properties, propose a multiagent extension of PDDL, the Multiagent Planning Language MAPL (pronocunced "maple"). By Multiagent Planning (MAP) we denote any kind of planning in multiagent environments, meaning on the one hand that the planning process can be distributed among several *planning* agents, but also that individual plans can (and possibly must) take into account concurrent actions by several *executing* agents. We do not impose any relation among planning and executing agents: one planner can plan for a group of concurrent executers (this corresponds roughly to planning with PDDL 2.1 but necessitates extensions allowing more execution flexibility), several planners can devise one shared plan (linear or not) or, in the general case, m planners plan for n executing agents. In the specific, yet common case of n agents, each having both planning and executing capabilities we speak of *autonomous agents*. Note that we do neither assume cooperativity nor competition among agents.

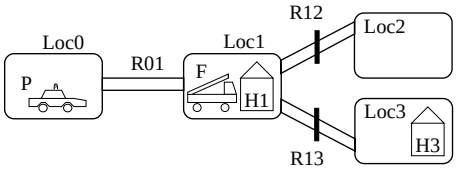


Figure 1: A multiagent planning problem

As a motivating example, fig. 1 shows a simple MAP problem as it appears in the RoboCupRescue simulation (Kitano *et al.* 1999). There are two autonomous agents: police force P and fire brigade F. They have different capabilities: P clears blocked roads, F extinguishes burning houses, both can move on unblocked roads. Each action has a duration which may vary because of specific execution parameters (e.g. location distance, motion speed) and/or intrinsic unpredictability. For this example, we assume a duration of 30 to 180 minutes for *clear*, 1 to 4 hours for *extinguish*, and 2 to 4 minutes for *move*. The speed and thus the duration of *move* is controlled by each agent while the duration intervals for *clear* and *extinguish* can only be estimated. The agents' knowledge and goals are differing, too: P wants the roads to be clear, but is unaware of the state of all roads except $R01$. F wants all burning houses extinguished, knows that $H1$ and $H3$ are burning, but also that it cannot reach $H3$ because road $R13$ is blocked.

Even in this trivial example we can make some general observations about planning in MAS that will motivate the concepts introduced in the rest of the paper.

(1) *Concurrent acting* is central to MAS (P can move to $Loc1$ and start clearing $R13$ while F is extinguishing $H1$). (2) *Metric time* is needed to realistically describe action durations and their relations. (3) Synchronizing on actions of unknown (at least to some agent) duration demands *qualitative* use of time (e.g. "after P has cleared $R13$"). A specific usage of qualitative time in MAP is (4) synchronization on communicative acts, for example "after P has informed me that $R13$ is now clear".

While many recent planning formalisms allow some degree of concurrency, most fail in providing either (2) or (3). PDDL 2.1, for example, supports metric time but enforces planners to assign exact time stamps and durations to all events (Fox and Long 2002). In contrast, the concurrency model of (2001) augments partial order plans with concurrency, thus allowing flexible, synchronized execution, but makes no difference between plans that take seconds and ones that take years. None of the planning models known to us allows to synchronize on communicative acts.

To summarize, PDDL is, in its current form, inadequate for representing MAP problems and their solutions, namely because of the following missing features:

1. **beliefs:** if more than one agent is manipulating the world (unlike assumed by classical planners) facts about it cannot only be true or false, but also simply *unknown* to an agent (e.g. P not knowing whether road $R12$ is clear or

not). Instead of using some kind of possible world semantics, we propose to give up the propositional representation of facts in PDDL and move on to ternary or even n-ary state variables.

2. **model of time:** Not only *quantitative* ("duration is 30 minutes"), but also *qualitative* ("F moves to Loc3 *after* P has cleared R13") models of time are needed to represent and coordinate multiple-agent behavior. To that end, we propose to exchange PDDL's time-point semantics with a semantics of temporal relations among actions that can be both quantitative or qualitative.
3. **degrees of control:** An agent may exploit another agents' actions in her own plan ("F moves to $L3$ after P has cleared $R13$"), but cannot (by removing them from her plan) prevent them from happening. Even her own actions might be only partially controllable by the agent, e.g. duration of *move* (controllable) and *extinguish* (uncontrollable). PDDL must allow to describe controllable and uncontrollable events so that agents can exploit their differing properties during planning.
4. **plan synchronization:** PDDL 2.1's plan semantics forces agents to attribute exact time points to all actions, thus making synchronization of (partial) plans very hard (when trivial merging is impossible). More importantly, for reasons of flexibility and security it is often best to share as little information as possible. To achieve such *minimum synchronization* we suggest not only to change the temporal model but to allow *speech acts* as synchronizing (meta-)actions in a plan. E.g., all F needs to know about P's plan is that at some point P will have cleared $R13$. F can thus enter a speech act TOLD(P,F,R13=CLEAR) into her own plan that has R13=CLEAR as effect and allows F to plan on with that knowledge.

Our extension of PDDL will provide these features. Fig. 2 shows part of a MAPL description for the Rescue domain. Fig. 3 shows a MAPL plan of agent F for the problem given in Fig. 1

```
(:state-variables
  (pos ?a - agent) - location
  (connection ?p1 ?p2 - place) - road
  (clear ?r - road) - boolean)
(:durative-action Move
  :parameters (?a - agent ?dst - place)
  :duration (:= ?duration (interval 2 4))
  :condition
    (at start (clear (connection (pos ?a) ?dst)))
  :effect (and
    (at start (:= (pos ?a) (connection (pos ?a) ?dst)))
    (at end (:= (pos ?a) ?dst))))
```

Figure 2: Excerpt from a MAPL domain description

The remainder of the paper presents MAPL solutions to these problems: first, we show how to describe beliefs conveniently as non-binary state variables. Then we present the temporal model of MAPL and its representation of events and actions. The concepts of control over and mutual exclusivity among events are introduced in the following sections,

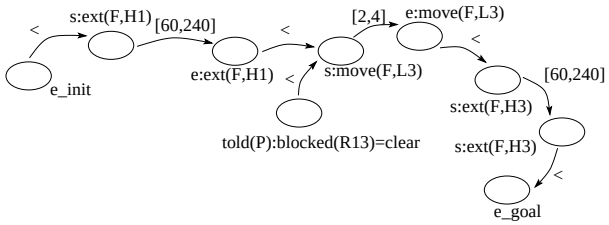


Figure 3: F's plan including a reference speech act by P

preparing the ground for the definition of MAPL's plan semantics. Finally, we show how speech acts can provide synchronization between plans of several agents.

Beliefs and other state variables

One main feature distinguishing MAPL from PDDL is the use of non-propositional state variables: in MAP we must dismiss the Closed-World Assumption (CWA) that everything not known to be true is false – the truth value might also be simply *unknown* to an agent. There are several possibilities to represent such belief states, for example sets of possible states (possible worlds) could represent all possible combinations of states for unknown facts. Another possibility is to represent each of the three possible states of a fact (true, false, unknown) by a unique proposition and to assure that exactly of one these propositions hold in any given state. This is similar to the representation of negation proposed in (Gazen and Knoblock 1997): explicit negation of a fact is compiled away in a planning domain by introducing a special proposition representing the negated fact and assuring in the planning domain that only one of the two facts can hold in a state.

However, we do not see any genuine merit in a propositional representation of states; the simplest way to represent beliefs it to allow state variables to have more than just the two values *true* and *false*. We will therefore not only allow ternary state variables (with values *true*, *false* and *unknown*), but *n-ary state variables*, meaning that a state variable v must be assigned exactly one of its n possible values in any given state. Among others, Geffner(2000) uses the same concept and gives an extended formal description and justification.

For example, in our Rescue domain the state variable `(pos F)` could have any of the values `Loc0`, `Loc1`, `Loc2`, `Loc3` or the new "default" value `unknown` that is a possible value for each state variable. Our new CWA will then be that every state variable the value of which is not specified in a state (or cannot be deduced otherwise) is believed to be `unknown`.

Note that a compilation approach similar to the one of (Gazen and Knoblock 1997) is still possible: every n-ary state variable can be compiled down to a set of propositions that must be ensured to be mutually exclusive. This ensurance is implicit in the definition of n-ary state variables and thus gives domain designers a natural way to describe important invariants of a domain, for example that an object

can only be at one location at a time or may have only shape or color.

Definition 1 *A* planning domain *is a tuple* $D = (T, O, V, type)$ *where* T *is a set of types,* O *a finite set of objects,* V *the set of state variables.* $type : O \cup V \rightarrow T$ *assigns a type to each object and state variable.* $dom : V \rightarrow \mathcal{P}(O)$ *with* $dom(v) := \{o \in O | type(o) = type(v)\} \cup \{unknown\}$ *gives the possible values for state variable* v*. A* state variable assignment *is a pair* $(v,o) \in V \times dom(v)$*, also written* $(v = o)$.

Temporal model

Quantitative models of time are necessary to describe exact temporal relations between actions of differing duration. Level 3 of PDDL 2.1 provides a simple, yet expressive means to model durative actions. However, the time-point semantics for plans proposed in (Fox and Long 2002) is overly restrictive. In forcing planners to assign exact time points to every action in a plan it takes away the execution flexibility offered by plan semantics based on action order. Sequential, Graphplan-like ordered, or partially ordered plan semantics can easily deal with action durations that are unknown (in general or to a specific agent) because they offer qualitative notions of time like "after" or "before". MAPL is an approach to take the best of both worlds and combine quantitative and qualitative models of time. The key idea is to give up the time-point semantics for plans and go back to ordering constraints among events, but to make these constraints more flexible than those of total or even partial-order planning. Precisely, the temporal component of a MAPL plan corresponds to a Simple Temporal Network (Dechter *et al.* 1991) the constraints of which are intervals describing the temporal relation among events (instantaneous state changes). Note that in lieu of the term *action* we use the more neutral *event* here to reflect that state changes are not necessarily actively brought about by an agent but can also be observations of "natural" changes in the environment.

Definition 2 *An* event[1] e *is defined by two sets of state variable assignments: its* preconditions $pre(e)$ *and its* effects $\mathit{eff}(e)$*. For assignments* $(v = o)$ *in the preconditions* [*effects*] *of an event we will also write* $(v == o)$ $[(v := o)]$.

Relating events by ordering (i.e. temporal) constraints is central to partial-order planning (but is also implicit in classical time-step based planning). To allow for a quantitative model of time, we will extend each constraint with an interval expressing the possible variation in two events' temporal distance.

Definition 3 *A* temporal constraint $c = (e_1, e_2, I)$ *associates events* e_1, e_2 *with an interval* I *over the real numbers, describing the values allowed for the temporal distance between the occurrence times* t_{e_1} *and* t_{e_2} *of the events:* (e_1, e_2, I) *is satisfied iff* $t_{e_2} - t_{e_1} \in I$*.* I *can be open, closed or semi-open.*

[1] In this paper we assume *ground* events and actions. Instantiation of actions schemas (Fig. 4) includes instantiation of the state variable schemas (like `pos(?a)`) as well. When a state variable is used functionally, i.e. it represents its value in a given state (like `(pos ?a)` in `(connection (pos(?a) ?p))`, instantiation implies creation of ground actions for every possible value $o \in dom(v)$. There, v is replaced by o and $(v == o)$ is added to the preconditions.

```
(:durative-action Move_F_Loc2[Loc1_R12]
  :parameters (?a - agent ?dst - place)
  :duration (:= ?duration (interval 2 4))
  :condition (and
       (at start (== (pos F) Loc1))
       (at start (== (connection Loc1 Loc1) R12))
       (at start (clear R12))
  :effect (and (at start (:= (pos ?a) R12))
               (at end (:= (pos ?a) Loc2))))
```

Figure 4: Instantiated Move action

Using intervals, we can express that the duration of an action is undetermined that an agent is ignorant of it. The main advantage of the interval constraints, however, is that we can express *quantitative* relations in a quantitative manner: "e_x occurs *after* e_y" is expressed by the constraint $(e_x, e_y, \mathbb{R}^+)$; "e_x occurs *at the same time as* e_y" by $(e_x, e_y, [0, 0])$. To give qualitative descriptions of concrete, quantitative constraints we will use the abbreviation $(e_1 \prec e_2) \in C$ for the expression $\forall I.\ (e_1, e_2, I) \in C \rightarrow I \subseteq \mathbb{R}^+$, i.e. e_1 occurs sometime before e_2. $(e_1 \preceq e_2) \in C$ is defined similarly for sub-intervals of $\mathbb{R}_0^+$.

With such constraints we do not need definite time points any more: all that is important to describe a plan is the *relations* among the actions and events. As usual in partial-order planning, the initial state can be represented by a special event e_0 such that constraints with e_0 can be seen as absolute times. However, in MAP, there may be a different initial event for every agent. To be able to synchronize on absolute times if necessary, we can (but need not) assume a common clock. It is modeled as a special event e_{tr}, the temporal reference point, also called the Big Bang event because it lies before all other events and is thus the point where time starts. All agents know e_{tr} and thus can describe *absolute times* as constraints with e_{tr}.

Definition 4 *A* durative action *is a tuple* $a = (e_s, e_e, I, e_{inv})$ *where* e_s, e_e *are events (called the start and end event),* $I \subseteq \mathbb{R}^+$ *is an interval representing the temporal constraint* (e_s, e_e, I) *of the form* $e_s \preceq e_e$*, and* e_{inv} *is an event with* $\mathit{eff}(e) = \emptyset$*, called the* invariant event*. An* instantaneous action *is a durative action* $a = (e, e, [0, 0], e_{inv})$ *where* $pre(e_{inv}) = \mathit{eff}(e_{inv}) = \emptyset$*. For a set of actions* Act*,* E_{Act} *denotes the set of start and events of actions in* Act*.*

It is clear that when only using instantaneous actions and constraints of the form $(e_x, e_y, \mathbb{R}^+)$ between them, we come back to partial-order plans. On the other hand, when using durative actions with constraints of the form $(e_x, e_y, [d, d])$, i.e. exact durations and delays, we will create PDDL 2.1 plans. Thus, MAPL subsumes both partial-order and PDDL plans.

Before describing the semantics of MAPL plans we will introduce two more concepts describing events: the first,

control, allowing planners to distinguish between endogenous and exogenous events, the second, mutual exclusiveness (or, relatedly, read-write locks) describing events that must not occur concurrently.

Control

There are two kinds of durative actions: those in which duration is controlled by the executing agent (e.g. reading a book) and those in which the environments determines the duration (e.g. boiling water). In the former case, the agent (or its corresponding planner) can *choose* the delay from start to end event, in the latter case the end event may happen *at any time* during the interval given by the constraint. For any set of actions Act_a of an agent a we assume there is a *control function* $c_a : E_{Act} \rightarrow \{a, env\}$ describing whether the agent or the environment controls the occurrence time of an event. As agents can normally decide at least the start time of an action we assume that $c_a(e) = a$ for start events e_s.

When multiple planners communicate and share parts of their plans, a planner has to store for each event in a plan the executing agent controlling the event. As each planner will plan for at least one agent, the control concept is a natural way to model which events the planner can influence and how. Durations of actions where both start and end events are controlled by the planner (i.e. executing agents associated with the planner) can be manipulated in the limits of the constraining interval. Actions in which only the start event is controlled by the planner can at least be added or removed from the plan at will. Actions and events not under control of the planner cannot simply be removed from the plan; that would be self-deception because removal would not prevent their occurence. Their occurence must be taken into account during planning and plans should be valid for every possible duration in the limits of the constraining interval. (Similar, but more sophisticated concepts are developed in (Vidal and Fargier 1999; Tsamardinos *et al.* 2002).)

Mutex events and variable locks

Concurrency is a key notion in MAS. In Multiagent Planning it appears at two levels: as concurrent actions in a *plan* (or distributed over several plans by different agents) and as concurrent *planning*. Both levels are closely related: concurrency conflicts at the plan level must be detected and resolved during planning. For the plan level we define:

Definition 5 *Two events are* mutually exclusive (mutex) *if one affects a state variable assignment that the other relies on or affects, too.* $mutex(e_1, e_2) :\Leftrightarrow$
$(\exists (v{:=}o) \in \mathit{eff}(e1)\ \exists (v,o') \in pre(e_2) \cup \mathit{eff}(e_2)) \vee$
$(\exists (v{:=}o) \in \mathit{eff}(e2)\ \exists (v,o') \in pre(e_1) \cup \mathit{eff}(e_1))$

This definition corresponds to mutex concepts in single-agent Planning, e.g. in PDDL 2.1 or Graphplan(Blum and Furst 1997). From a Distributed Systems point of view, however, the mutex definition describes a *read-write lock* on the state variable v that will prevent concurrent access to the same resource v because this may lead to indeterminate values of v. Interestingly, the correspondence between mutual exclusive events and locks on state variable is more visible in a formalism like MAPL that, by the use of non-boolean state variables, seems to be a step closer to "imperative" distributed programming than the more declarative style of STRIPS and PDDL in which the state variable concept is hidden behind the Closed World Assumption and ADD/DEL effects instead of state variable updates.

In the next section, we will use the mutex definition to describe non-interference in concurrent plans. In another paper we introduce the related concept of state variable *responsibility* among agents to solve lock/mutex conflicts during distributed *planning* (Brenner 2003).

Plans

Definition 6 *A* multiagent plan *is a tuple* $P = (A, E, C, c)$ *where* A *is a set of* agents, E *a set of events,* C *a set of temporal constraints over* E*, and* $c : E \rightarrow A$ *is the* control function *assigning to each event an agent controlling its execution.*

We can now start to describe when a plan is valid, i.e. executable. We will split this definition into two aspects: temporal validity, meaning that there are no inconsistencies among temporal constraints in the plan, and logical consistency, meaning that no actions do logically interfere or are disabled when they shall be executed in the plan.

To simplify the next definitions we assume the set C of temporal constraints to be always *complete*, i.e. $\forall e_1, e_2 \in E \exists I.\ (e_1, e_2, I) \in C$. This is no restriction because we can assume C to contain the trivial constraints $(e, e, [0, 0])$ for all events $e \in E$ and $(e_1, e_2, (-\infty, \infty))$ for unrelated events $e_1 \neq e_2$.

Definition 7 *A set of temporal constraints* C *is* consistent *if* $\neg\exists e_1, e_2, \ldots, e_n.(e_1 \prec e_2) \in C \wedge (e_2 \prec e_3) \in C \wedge \cdots \wedge (e_n \prec e_1) \in C$*. A multiagent plan* $P = (A, E, C, c)$ *is* temporally consistent *if* C *is consistent.*

This is a reformulation of the consistency condition for Simple Temporal Networks (STNs) (Dechter *et al.* 1991) as (E, C) is in fact an STN[2]. Using the Floyd-Warshall algorithm (Cormen *et al.* 1992), consistency of an STN can be checked in $O(n^3)$. In planning, new events and constraints are repeatedly added to a plan while consistenty must be kept. To check this, we have developed an incremental variant of the algorithm (omitted from this paper) that checks for consistency violations caused by a constraint newly entered into the plan. This algorithm is in $O(n^2)$ (for every addition of a constraint).

Definition 8 *A multiagent plan* $P = (A, E, C, c)$ *is* logically valid *if the following conditions hold:*

1. *No mutex events* $e', e'' \in E$ *can occur simultaneously:*
   $\forall e', e'' \in E.mutex(e', e'') \rightarrow (e' \prec e'') \in C \vee (e'' \prec e') \in C$

For any assignment $(v == o)$ *in the precondition of any event* $e \in E$ *there is a* safe achieving event $e' \in E$*:*

[2] We are aware that STN consistency is not adequate for plans with uncontrollable action durations. We are working to integrate the concept of *dynamic controllability* into our framework (Vidal and Fargier 1999).

2. $(e' \prec e) \in C \wedge (v := o) \in \mathit{eff}(e)$ *(achieving event)*
3. $\forall e'' \in E\ \forall (v := o') \in \mathit{eff}(e'').\ o' \neq o \ \rightarrow\ (e'' \prec e') \in C\ \vee\ (e \prec e'') \in C$ *(safety)*

Conditions 2 and 3 define plans as valid if there are no open conditions and no unsafe links, an approach well-known from partial order planning(Nguyen and Kambhampati 2001; Weld 1994). Condition 1 (similarly used in GraphPlan(Blum and Furst 1997)) describes threats caused by conflicting effects that do not necessarily cause unsafe links. This happens especially when events violate invariants of durative actions.

Definition 9 *A* planning problem *for an agent a is a tuple $Prob_a = (Act, c_a, e_0, e_\infty)$ where Act is a set of actions, c_a is the control function for Act, and e_0, e_∞ are special events describing the initial and goal conditions.*

We will now define when a plan solves a problem. We do not need to and cannot use happening sequences like PDDL 2.1 because of MAPL's plans being partially ordered. Instead we will reduce the question to a check for temporal and logical validity of a new plan that is obtained as a combiniation of the problem with the solution plan.

Definition 10 *A multiagent plan $P = (A, E, C, c)$ is* valid *if it is both temporally consistent and logically valid. A plan P is a* solution *to a problem $Prob_a = (Act, c_a, e_0, e_\infty)$ of agent a if the following conditions are satisfied*

1. *c is consistent with c_a: $c_a(e) = x\ \rightarrow\ c(e) = x$ and*
   $\forall (e_s, e_e, I, e_{inv}) \in Act.$
   $[c(e_e) = a\ \rightarrow\ \forall (e_s, e_e, I') \in C.\ I' \subseteq I]\ \wedge$
   $[c(e_e) = env\ \rightarrow\ \forall (e_s, e_e, I') \in C.\ I' = I]$
2. $\forall (e_s, e_e, I, e_{inv}) \in Act.$
   $e_s \in E\ \rightarrow\ (e_e \in E \wedge e_{inv} \in E)\ \wedge$
   $(e_s \prec e_{inv}) \in C \wedge (e_{inv} \prec e_e) \in C$
3. *for $C' = C \cup \bigcup_{e \in E} \{(e_0, e, \mathbb{R}^+), (e, e_\infty, \mathbb{R}^+)\}$*
   $P' = (A, E \cup \{e_0, e_\infty\}, C', c)$ is valid.

In words these conditions can be described as follows:

(1) the plans uses actions controlled by the agent in the way they are specified in the problem: the agent controlling an event is the same in the problem and in the plan; only actions in which the planner can control start and end event can be tightened during planning (complete control).

(2) durative actions and their invariants are used as expected: for each action appearing in a plan, its start, end, and invariant event must all appear in the plan as well as constraints describing their appearance in the natural order: $e_s \prec e_{inv} \prec e_e$. Note that no "pseudo" time points must be associated with invariants but that it suffices to have constraints forcing them to hold anytime between the start and end events.

(3) executing the plan in the initial state reaches the goals. Though looking simple, this last condition is the most important: when initial and goal events are added to the plan with constraints describing that the initial event (goal event) happens before (after) all others in the plan, then temporal and logical validity of the resulting plan signifies that the plan solves the problem.

Note that the solution plan is not required to contain only actions from Act: a plan can solve an agent's problem even if it contains not a single action of that agent!

Speech acts as synchronizing events between plans

An agent using a fact in his plan need not know how, why or by whom it has been achieved. In temporally uncertain domains the agent must even plan not knowing *when* exactly the fact will become true. To enable planning under these different kinds of ignorance, we will allow agents to use different kinds of possibly virtual *reference events* in their plans. As the same event may appear in plans of different agents this provides an implicit coordination among those plans while still allowing the knowledge about causal or temporal links of the event with others to vary largely from agent to agent.

A basic reference event that we will only briefly mention here is e_{tr}, the temporal reference point lying before all other events. All agents know e_{tr} and thus can describe *absolute times* as constraints with e_{tr}.

For MAP it is most important that agents can coordinate and exchange knowledge about the domain and their plans. This can be done with *communicative events* (i.e. speech acts). For now, we propose only the simple communicative act of the form $\text{TELL}_{v,o}$ with $\mathit{pre}(\text{TELL}_{v,o}) = \{(v, o)\}$ and $\mathit{eff}(\text{TELL}_{v,o}) = \emptyset$ and its counterpart $\text{TOLD}_{v,o}$ with $\mathit{pre}(\text{TOLD}_{v,o}) = \emptyset$ and $\mathit{eff}(\text{TOLD}_{v,o}) = \{(v, o)\}$.

By entering new information into the current plan with TOLD agents can use it like any effects of other events: as preconditions of new actions and as temporal reference in constraints. It is the latter use that is especially helpful: the TOLD event provides automatic synchronization with another agents plan. E.g. fig. 3 shows how the fire brigade synchronizes on the police clearing a road without knowing when or how this is done. Only the minimum of information necessary for coordinated action is communicated. This is important both for privacy reasons and to keep individual knowledge bases conveniently small.

Having communication explicitly anchored in the plan has several advantages. First and foremost, "being told something" is one of the simplest means for modeling "observations" of world changes not brought about by an agent himself. This way, we do not need complex semantics for information gathering or conditional plan execution.

For the speaking agent, the communicative act represents a commitment to inform the other of a specific fact during execution. It is not enough, for example, that a police agent promises to clear a road *during planning*, but that also the fire agent somehow has to be informed *during execution* that this promise has been realized. Anchoring the speech act in the plan thus is a "physical" representation of the link between the commitment made during planning and its fulfillment during execution.

During distributed planning this means, on the other hand, that plans synchronized by speech acts also commit the agents to coordinate changes to their plans. If, for example, the police agent decides at some point during planning

that he must revise his decision to clear R13, the TELL event will remind him to inform the fire brigade of this change. The speech act can thus represent a distributed backtracking point, a concept similarly used in Distributed CSP solving (Yokoo and Hirayama 2000).

The basics for a distributed planning algorithm using speech acts both to exchange missing information and to synchronize are presented in (Brenner 2003).

Conclusion and future work

We have presented basic concepts for the Multiagent Planning Language MAPL, an extension of PDDL that supports planning for and by Multiagent Systems. MAPL's temporal model can be used to describe exact, quantitative temporal relations as well as flexible, quantitative ones. It might therefore be useful not only for multiagent scenarios but for every domain where execution flexibility is important after planning has been completed. MAPL's use of non-boolean state variables makes it easier for domain designers to describe basic invariants like "an object can have only one location at a time". It also sheds some light on the relation between mutually exclusive actions in Planning and similar concept in Distributed Computing like read-write locks on variables.

We have defined temporal and logical validity of MAPL plans as well as what it means to solve a specified planning problem. As, in contrast to PDDL 2.1, MAPL plans are partially ordered we cannot and do not need to define happening sequences or induced simple plans for MAPL plans. This also avoids associating invariants with "pseudo" time points.

In another paper (Brenner 2003), we present the first single agent and distributed planning algorithms for MAPL domains. These algorithms are as preliminary as the definition of MAPL's syntax and semantics. Exciting future work is possible now: we are currently working on a parser and a small domain suite to test both the expressivity of the language and the powers and limits of our algorithms.

MAP has been a topic of interest in AI for quite some time. However, not much work has been published, neither in the field of Multiagent Systems (MAS) nor in Planning; furthermore, what has been published is mostly stand-alone work that has not led to a steady development in MAP research. In our view, this is due to an unfavorable separation of the (single-agent) planning phase and the (multi-agent) coordination and execution phase, resulting in AI Planning researchers concentrating mostly on the former and MAS researchers almost exclusively dealing with the latter. This separation is only possible with strong assumptions that narrow the generality of the proposed approaches, for example the assumption in MAS research that the actual planning of each agent can either be handled by classical single-agent planning methods or is eased by a given hierarchical task decomposition. The AI planning community, on the other hand, has only recently fully acknowledged the need for sophisticated models of concurrent plan execution (earlier exceptions include most notably work by M. Ghallab(Ghallab and Laruelle 1994)). MAPL is an attempt to show possible extensions of PDDL 2.1's representation in a way that allows flexible execution *after* and easy coordination *during* the planning process. We hope that our representation will allow to conveniently describe largely differing MAP domains for which researchers can propose and cross-evaluate very different algorithmic approaches, thus promoting the field of Multiagent Planning.

References

Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2), 1997.

Craig Boutilier and Ronen Brafman. Partial order planning with concurrent interacting actions. *Journal of Artificial Intelligence Research*, 2001.

Michael Brenner. Multiagent planning with partially ordered temporal plans. In *Proc. IJCAI*, 2003.

T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49, 1991.

Maria Fox and Derek Long. *PDDL 2.1: an Extension to PDDL for Expressing Temporal Planning Domains*, 2002.

B. Gazen and C. Knoblock. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In *Proc. ECP '97*, 1997.

H. Geffner. Functional STRIPS: a more flexible language for planning and problem solving. In Jack Minker, editor, *Logic-Based Artificial Intelligence*. Kluwer, 2000.

M. Ghallab and H. Laruelle. Representation and control in IxTeT, a temporal planner. In *Proc. of AIPS '94*, 1994.

H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjoh, and S. Shimada. RoboCupRescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *Proc. 1999 IEEE Intl. Conf. on Systems, Man and Cybernetics*, 1999.

XuanLong Nguyen and Subbarao Kambhampati. Reviving partial order planning. In *Proc. IJCAI '01*, 2001.

Ioannis Tsamardinos, Thierry Vidal, and Martha Pollack. CTP: A new constraint-based formalism for conditional, temporal planning. *Constraints Journal*, 2002.

Thierry Vidal and Hélène Fargier. Handling contingency in temporal constraint networks. *Journal of Experimental and Theoretical Artificial Intelligence*, 11, 1999.

Daniel Weld. An introduction to least commitment planning. *AI Magazine*, 15(4), 1994.

Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: a review. *Autonomous Agents and Multi-Agent Systems*, 3(2), 2000.

The Loyal Opposition Comments on Plan Domain Description Languages

Jeremy Frank and Keith Golden and Ari Jonsson
Computational Sciences Division
NASA Ames Research Center, MS 269-2
frank@email.arc.nasa.gov
Moffett Field, CA 94035

Abstract

In this paper we take a critical look at PDDL 2.1 as designers and users of plan domain description languages. We describe planning domains that have features which are hard to model using PDDL 2.1. We then offer some suggestions on domain description language design, and describe how these suggestions make modeling our chosen domains easier.

The Loyal Opposition

PDDL has served as the underpinnings of the Planning Competition, and has had an enormous impact on the planning community as a whole. PDDL and STRIPS have served as a *lingua franca* for deterministic planning domains, making it possible for researchers to compare techniques on the same problems and enabling meaningful comparisons of those techniques. The result has been considerable progress in planning algorithms.

Recently, however, the problems of interest to the planning community have changed. Goal achievement by itself is no longer enough, and the form of the goals has changed as well. Temporal planning requires meeting goals that include temporal constraints and resource constraints. While it is possible to model planning domains with temporally extended state and resources using purely propositional modeling languages, it is expensive in modeler effort and results in huge domain descriptions. By contrast, software domains such as data processing, web services integration and and information integration generally require incomplete information, incompletely known dynamic universes and sensing actions, but do not, generally, require rich models of time.

In this paper, we take a critical look at the design of PDDL 2.1. Our perspective is that of researchers in planning and scheduling who have considerable experience in designing and using planning domain description languages. We will describe planning domains that pose problems for PDDL 2.1. We will then describe what we believe are the core set of features for modeling planning domains. We will show how these core features simplify the modeling of the domains of interest.

PDDL and its Discontents

In this section, we describe some domain models and their PDDL 2.1 representations, and then discuss reasons why the PDDL 2.1 model is problematic.

Temporal Constraints and Instantaneous Events

PDDL 2.1 requires that propositions hold for a non-zero amount of time before they are used as preconditions for actions. However, PDDL's underlying representation of states is not based on intervals, and so this makes it difficult to specify the interaction between multiple events that modify the same proposition. It will also be difficult to see what domain axioms resulted in the separation of two events that modify the same proposition in the final plan.

Part of the reason for this is that sometimes actions require additional constraints to decide whether the action sequence is legal, and PDDL 2.1's semantics forbid such plans when there is a possibility of plan execution failure. The following example appears in (Fox & Long 2003) . The model includes action A with preconditions $P \vee Q$ and effect R, and action B with preconditions P and effect $\neg P$ and S. These two actions are considered mutually exclusive because a plan in which A and B execute concurrently may fail. We might have A and B execute at the same time in a state where $\neg Q$ holds, for example. PDDL assumes that the executions of A and B are actually ordered, and that the order is arbitrary. Thus, if the state happens to be one where $\neg Q$ holds, it may be that B happens first, making A fail both its preconditions.

Modeling this scenario by asserting the intervals of time over which P and Q hold, and forcing actions to declare how long the precondition must hold, clarifies the situation. For example, if A requires P to hold for 1 unit of time prior to A's execution, then in order for the plan to be legal, B can't happen at a time that makes P true for less than a unit of time. This makes it possible to both post and check the constraints that must hold for both A and B to happen concurrently. If P has held for long enough, then A and B can happen at the same instant, even in a state where $\neg Q$ holds.

A related problem is that it is hard to express certain temporal constraints in PDDL 2.1. For example, suppose that an action Take-Picture requires a proposition stable to hold for a period lasting from at least 5 seconds before to at least 5 seconds after the Take-Picture ends. Furthermore, the camera expends energy only during the time of the exposure, which lasts 24 seconds. During the intervening 5 seconds before and after the exposure, the camera could be involved in other activities such as changing the filters, but can't slew (as this results in vibration).

PDDL 2.1 does not allow direct expression of the constraints on stability. As we said previously, there is no way

```
(:durative-action take-picture
:parameters (?s - satellite ?c - camera ?o - observation)
:duration (= ?duration 24)
:condition (and (at start (pointing-at (?s ?o)))
   (at start (on ?c))
   (at start (stable ?s))
   (over all (on ?c))
   (over all (stable ?s))
:effect (and (captured-but-not-stabilized ?o)
   (at start (decrease (energy ?s 5)))
   (at end (increase (energy ?s 5)))))

(:durative-action stabilize-before
:parameters (?s - satellite)
:duration (= ?duration 5)
:condition (over all (stable ?s))
:effect (and(at end (stable ?s))))

(:durative-action stabilize-after
:parameters (?s - satellite ?o - object)
:duration (= ?duration 5)
:condition (and(at-start (captured-but-not-stabilized ?o))
   (at start (stable ?s))
   (over all (stable ?s)))
:effect (and (at end (not(captured-but-not-stabilized ?o)))
   (at end (captured ?o))))
```

Figure 1: Temporal Constraints in PDDL 2.1

in PDDL 2.1 to write a condition that requires a proposition to hold for an interval of time. Thus, we can't even require that Stable hold for 5 seconds before Take-Picture can start. We can write an action Stabilize-Before that lasts for 5 seconds, with an effect Stable, after which Take-Picture can be performed. Take-Picture also asserts Stable throughout the 24 second interval, and properly asserts that the instrument must be on, and affects the power properly. Now, however, we must ensure that stability is held for another 5 seconds. We can do so by asserting a condition captured-but-not-stabilized for the observation. The only way to assert captured, then, is to execute the stabilize-after action. The only remaining problem is that these actions should be executed back-to-back to enforce our original constraint, and PDDL 2.1 has no way of ensuring that this happens.

Notice that PDDL 2.1 does not allow direct assertions that states hold for a period of time, but rather uses action duration to indirectly affect the amount of time that propositions hold during a plan. More generally, PDDL preserves the notion of preconditions and effects from STRIPS, which simply doesn't make sense when considering plans with overlapping concurrent activities. The stability condition described above is more than a persistent precondition, in that it outlasts the end of the action. PDDL has no mechanisms to specify such requirements.

Exogenous Events

Consider a domain containing a spacecraft that collects data which can be transmitted back to Earth only at specific times. PDDL 2.1 forces this to be modeled using constraints on the times that communication actions can begin and end. This must be accomplished by using a conditional effect that enumerates, in the condition, the possible start times for the communication windows. For instance, suppose that communication windows are open from times 1 to 5, and from times 7 to 10. The model in Figure 2 describes how this works. Notice also that we have a function compute-open-window-duration that determines how long the action takes based on the start time.

While this enforces the correct behavior in the sense that communication windows are open at the correct times, it doesn't actually have the desired semantics; if a use-window action occurs at a time when no window is available, the action occurs and uses time, but the communication window is not open for use, when in fact we would like the action to fail. Furthermore, this approach requires rewriting the model for each new situation. In particular, the conditions and the function compute-open-window-duration now must be revised for each new set of communication windows desired.

The problem is that PDDL does not allow the user to specify specify *exogenous events* that are known to be in the plan. For a treatment of exogenous events in temporal planning we refer the reader to (Smith & Jonsson 2002). Exogenous events require no explanation, and may establish states that can be used by other actions. Often, a null action is assumed to have established all the propositions in the initial state; the ability to include many such exogenous actions at different times is therefore a natural extension to planning domain initial states. Including exogenous events will make domain models more general; for example, the constraints in the satellite domain governing communication activities can be removed, and a general condition establishing the existence of a communication window can be used instead. The initial

```
(:durative-action use-window
:parameters (?s - ground-station)
:duration (= ?duration (compute-open-window-duration(?s ?start))
:condition (at start (closed ?s)))
:effect (and (when (or (and (> (?start 1)) (< (?start 5)))
   (and (> (?start 7)) (< (?start 10))))
   (over-all (in-use ?s))
   (at end (not(in-use ?s))))))
```

Figure 2: Exogenous Events in PDDL 2.1

state will contain a listing of all communication windows.

Continuously Varying Quantities

Suppose we have an aircraft that can be refueled in flight. We would like to model the amount of fuel in the fuel tank while the aircraft is both consuming fuel and being refueled. As pointed out in (Fox & Long 2003), PDDL 2.1 offers two options for modeling this domain; discrete durative actions and continuous durative actions. Discrete durative actions force all modification of the fuel amount to occur at the end of the action. Under some circumstances this assumption eliminates plans that are legal. Suppose the plane has 5 units of fuel left when the refueling action begins, and consumes 1 unit of fuel per unit of time. Suppose further that the refueling operation adds 20 units of fuel but takes 10 units of time. Modeling with discrete updates, the plane would be out of fuel before the refueling action finishes.

Continuous durative actions allow the fuel state to be continuously updated, and so the situation described above can be avoided, since the net fuel change is to add one unit of fuel per unit of time during refueling. This increased power comes at a high price. The function governing the amount of fuel must be evaluated at arbitrary points during the action, and this imposes strict requirements on concurrent actions updating the same numeric quantity in the plan.

The requirement for continuous updating of variables results in complex and overly restrictive semantics. It is unnecessary to allow such unrestricted access to continuously varying quantities. First, (Fox & Long 2003) indicate that plan validation only checks the values of continuously varying quantities at finitely many points, which implies that high order nonlinear functions cannot be validated. Thus, the power of the approach is not actually used. It is not clear whether such high order variations are simply forbidden, or whether model correctness is sacrificed in such cases. Second, modelers usually have a good idea of the conditions under which values of variables are needed, and can build the models to correctly account for these situations. As pointed out in (Fox & Long 2003), an alternative model is possible in which the correct value of the fuel was only guaranteed to be visible at the start and end of the action. The action of refueling would change the rate of fuel consumption, necessitating a new action, Flying-and-Refueling. After the completion of the refueling, if further flying was needed, then Flying would resume.

As discussed in (Fox & Long 2003), a model such as this one is weaker than the model using continuous durative action, in the sense that more states are needed, more parameters may be needed to propagate values through Flying-while-Refueling, and there is no way to access the value of fuel in the middle of the Flying or Flying-while-refueling actions. But the need for doing so has been eliminated, because the Refueling action now changes the state to Flying-and-Refueling, and all that is needed is the value of the fuel level at the end and beginning of the actions to ensure the fuel value is propagated correctly. The arguments in (Fox & Long 2003) indicate that all possible concurrent actions affecting fuel may need to be considered. While this is true, the approach taken in PDDL 2.1 actually infers these concurrent actions during the plan validation phase, and assumes that checking the endpoints of the actions is sufficient to validate the constraints. In situations where this is not sufficient, the modeler *must* take the burden on, to the extent they see fit when modeling the application. Thus, the PDDL 2.1 approach makes the commitment to model fidelity for the modeler, and is inappropriate for more complex cases.

Resources

PDDL 2.1 uses numeric expressions to model resources. We have seen examples of resources modeled this way in Figure 3. However, this approach makes it difficult to do some very useful reasoning about resources. Techniques like edge finding (Baptiste & Pape 1996) and resource envelopes (Muscettola 2002; Laborie 2003) require an explicit notion of activities using resources in order to work. In addition, modeling resources solely through numeric expressions tends to hide information from humans reading models, as well as forcing modelers to hide obvious resources in the model by using the numerical expressions. While techniques like TIM (Long & Fox 2000) can be used to infer the presence of resource behavior in planning domains, we feel there are significant advantages to explicitly declaring these parts of domain models. Note that complex numeric expressions may still be necessary to determine the actual amount of resource consumption or production; for example, a model of solar panel power production will require complex numerical constraints to determine the actual impact on the resource. However, an explicit declaration of resources and explicit declaration of resource use by actions can be beneficial.

Infinite and Dynamic Domains

In order to ensure that the number of actions and propositions is finite, PDDL permits only a finite number of objects, which must be explicitly enumerated, and does not allow arguments to actions or predicates to include numeric expressions (numbers being the only non-finite domains permitted

```
(:durative-action fly
:parameters (?x - airplane ?y - waypoint ?z - waypoint)
:duration (= ?duration (travel-time ?y ?z))
:condition (and (at start (at (?x ?y)))
   (over all (inflight ?x))
   (over all (>= (fuel-level ?x) 0)))
:effect (and (at start (not (at ?x ?y)))
   (at end (at (?x ?z)))
   (at start (inflight ?x))
   (at end (not (inflight ?x)))
   (decrease (fuel-level ?x) (* #t (fuel-consumption-rate ?x)))))

(:durative-action midair-refuel
:parameters (?x - airplane)
:condition (inflight ?x)
:effect (increase (fuel-level ?x) (* #t (refuel-rate ?x))))
```

Figure 3: Flying and Refueling in PDDL 2.1

in PDDL). It also forbids functions that return objects, which could be used to introduce infinitely many new objects. The justification for these restriction is that many planners rely on being able to enumerate the actions and propositions in a planning problem.[1]

Since PDDL is designed for the planning competitions, tailoring it to the limitations of the planners competing is reasonable. However, from the perspective of modeling certain real-world domains, having such a requirement encoded in the language definition is problematic. For example, (Fox & Long 2003) point out that it is impossible to write a PDDL action to fly at a certain altitude. Indeed, an action to drive at a given speed or a given distance would also be ruled out for the same reason.

This requirement also makes PDDL unsuitable for modeling software domains. Software domains include information integration, web services, data processing, and other domains where the agent interacts in a software environment. These domains are typically characterized by a large, incompletely known and often dynamic universe. Since PDDL 2.1 was not designed to handle sensing, we will defer the discussion of incomplete information. Instead, we focus on dynamic universes. In PDDL 2.1, actions that create new objects must be modeled by enumerating all objects that might appear during planning ahead of time, either explicitly or implicitly. For example, in the Settlers domain, newly created machines are modeled using an integer counter. Such an approach is inadequate for describing software domains. For example, consider the following command, which creates a new archive of the files in directory `~/papers`:

```
zip papers.zip ~/papers
```

This action fails to conform to PDDL restrictions in two ways; first, it creates a new object, the archive `~/papers.zip`. Second, one of its arguments is a string, which is not a finite type. Actually, both arguments are strings, but one of them, `~/papers`, designates an existing object, a directory, and the number of directories is finite. Following the general advice in (Fox & Long 2003), we could use the directory as an argument, rather than referring directly to its pathname, so that argument would be finite. However, the other argument designates a file yet to be created, so there is no existing object for which it is an attribute.

One might insist that such open-ended choices in action selection create an unreasonable burden for the planner, but nothing could be further from the truth. Either the choice will be constrained by the problem specification, in which case there may be no choice at all, or it will not, in which case the choice doesn't matter; any random string will suffice. From a constraint reasoning perspective, it is a trivial problem.

String and numeric arguments are ubiquitous in software domains. In addition to file creation, many image processing commands take numeric arguments, such as thresholds, values for scaling, rotating, brightening, compression factors, etc., and positions for cropping or overlapping images. None of these values are attributes of existing objects, but they are controls that the planner should be able to set, because whether, and how well, the plan achieves the goal depends on the values of those numeric arguments. For example, a user may want to scale an image to just fit on her screen, maintaining the same orientation and aspect ratio. The appropriate scale value depends on the horizontal and vertical extents of the image and of the screen. Another user may want to combine two images which are at different resolutions. Doing so will require scaling one of the images so that its resolution matches the other; the appropriate scale value depends on the resolutions of the two images, and possibly additional constraints, such as memory and the resolution of the final image.

One could imagine handling the object creation by listing, in advance, all of the objects that could be created. For example, we could have a few hundred "blank" file objects, and instantiate them as needed when new objects are created. However, this is not just inelegant and inefficient; it is also inadequate. Consider the reverse of the archive creation action above: archive extraction:

[1] A philosophical argument is also offered: that there are only finitely many objects in the world; we agree that this is technically true, but for all practical purposes it is false, and the number of possible actions in many worlds of interest is essentially infinite.

`unzip papers.zip`

This action will create new copies of every file in the archive `papers.zip`, preserving the original directory structure from `~/papers`, but rooted in the current directory. Since a single action can create an unbounded number of new objects, listing all of the new objects up front is clearly infeasible.

(Fox & Long 2003) raise the concern in connection with infinite domains that extensional interpretations of quantified preconditions are no longer possible. This is not a problem in the examples we have discussed because, although domains are dynamic, in any given state, any object domain is finite and can be directly determined from the execution trace that led to it. In fact, there is no way to introduce infinite numbers of new objects unless actions can have an infinite number of effects, which impossible to describe unless we already allow quantification over infinite domains. In other words, there is no way to get quantification over infinite domains unless we already have it.

A related issue is that an extensional interpretation of universally quantified *goals* may not be possible, because the universe at the time the goal is achieved is not known at planning time, even if the agent has complete information and all actions are deterministic. Indeed, there could be many goal states with quite different universes, depending on the execution traces followed to reach the goal. However, this can be solved quite simply by interpreting the Herbrand universe with respect to the initial state, as is done in (Golden & Weld 1996).

Domain Description Languages: An Ontological Approach

The essence of modeling is abstraction, and the essence of abstraction is simplification – omitting details so that the model is simpler than the thing being modeled. Choosing the right abstraction for a problem makes the problem much easier to solve. Domain description languages should enable modelers to abstract away details of planning domains that they feel are irrelevant to the task at hand.

There is an implicit agenda in the expansion of PDDL to gradually encompass more and more features that are needed for various planning domains. One possible conclusion of this is a grand-unified domain description language (GUDDL). As we have pointed out in this paper, some of the problems with PDDL 2.1 stem from an attempt to shoehorn time and resources into a STRIPS-based language. We anticipate similar problems as other features are encompassed. Domain modelers will tend to reject a language with unneeded features if the presence of those features proves to be a burden, either in increased computational complexity or increased modeling difficulty.

STRIPS and earlier versions of PDDL impose one set of abstractions: instantaneous action, the STRIPS assumption concerning persistence of states, and so on. Planning frameworks like CAIP (Frank & Jónsson 2003) impose different abstractions, such as the failure to distinguish state and action. PDDL 2.1 offers yet another set of abstractions, the ability to assert only local temporal constraints and tying temporally extended states to actions with duration, and the freedom from modeling the interaction of some concurrent actions that modify the same quantities. Incomplete information, offers additional options for abstraction. Some representations opt for a list of all possible states, others for a probability distribution over all possible states, in which the underlying representation is propositional. Still others reject the propositional abstraction, to allow sensors that return (possibly continuous) values, but give up explicit case analysis afforded by enumerating states.

It is unlikely that a modeler will model a behavior in two different ways in the same model. Having a language that supports different abstractions of the same underlying concept also makes the language clumsy to use and makes model validation more difficult. While some of the abstractions are hierarchical, forcing a planning domain to support the most concrete leads to both inefficiency and frustration on the part of domain modelers who don't use the power of the language. Furthermore, it is unlikely that different abstractions will be needed in the same model [2] A domain modeler is unlikely to want to put both continuously changing quantities and discretely changing quantities into the same model, for example. That same modeler, however, may want unary resources in a model which also has continuously changing quantities. In order to unify the languages, the resulting language will have to be less abstract, and the language will become more unwieldy.

A Common Core

As an alternative to a single language for all planning domains, we propose a common core for use in many planning domain description languages. The core must contain the essential elements of all planning domains, and provide a common set of concepts that can be used to develop many planning languages. These languages can have different syntax, and different underlying implementations that play to the strengths of the particular additional components, but depend on the same set of underlying ideas.

We believe that all planning domains require the following components:

1. A notion of *state*. States must be allowed to contain *numerical arguments*, but are fundamentally discrete statements about the world.
2. A notion of *objects* or *attributes*, which take on states.
3. A notion of the conditions governing *state transitions*. States may either end on their own or be terminated by an event or action. The rules governing these state transitions must be encoded in the domain description. Sometimes these transitions may be uncertain, and sometimes they may be conditional, but they must be described.
4. A notion of the *requirements* states impose on plans. States must be explained; either an event establishes them or they are exogenous. Furthermore, states generally impose conditions on the plan. Again, sometimes the requirements may be conditional.

Modelers must describe the set of objects that exist in the world, and enumerate the states they can

[2] A potential exception to this is unified agent models, in which models of different levels of abstraction must be coordinated. However, it is unlikely that the same planner will work with the different levels of abstraction.

take on. For example, a satellite object may take on the states `Take-Picture, Idle, Communicate` or `Slew`. This is a generalization of the idea that propositions are either true or not true at any given instant. By declaring the set of states an object can take on, the modeler also declares a number of mutual exclusion constraints. That is, objects can be in only one state, and the set of possible states is enumerated in the model. Notice how the semantics of negation are affected. In the example of the satellite domain, if a satellite is not `Idle` then by closure it must be in one of the other states.

A state takes the form `P(`$x_1 ... x_n$`)` where `P` is a predicate and x_i are parameter variables. We will refer to the additional variables o_P, s_P and e_P, the object, start and end variables of the state. Extending the set of variables this way makes it easy to post precedence constraints among states, and also to make decisions about what object of a class of objects takes on a required state. The constraint $P_s < P_e$ is implicitly understood to hold for all states. We view these additional variables as being implicit parameters to all states.

Domain Axioms provide the means to explain states, assert the conditions that states impose on plans, and describe the rules of state transitions. This is accomplished using *constraints* on the variables of states. Domain axioms take the form: `P(`$x_1 ... x_n$`)` $\wedge G(X) \rightarrow$ `Q(`$y_1 ... y_j$`)` $\wedge H(Z)$ where $X \subset \{x_1 ... x_i\}$ and $Z \subset \{x_1 ... x_i, y_1 ... y_j\}$, G is a set of conditions and H is a set of constraints. If a state `P` is in the plan, then some other states must be in the plan, and some constraints must hold among the variables representing those states. Notice that the states that must be in the plan are not necessarily new states; they can be states established some other way. Thus, planners must decide whether to reuse existing states or establish new state instances. The conditions G allow us to specify that some of the variables in P must take on certain values for the axiom to apply. The constraints in H allow us to impose limitations on the possible ground states `Q` that can be in the plan along with `P`.

The conditions can be used to dictate the transitions between states. Constraints can be posted among the parameters to limit the legal sets of predicates as well as imposing ordering constraints among the states in the plan. As an example, suppose that in the satellite domain a `Take-Picture` can be followed by another `Take-Picture` or an `Idle` state. The rules

`Take-Picture(`p, s`)`$\wedge$ `eq(`s`,take-picture)` $\rightarrow$
`Take-Picture(`q, t`),eq(`e_{tp1}, s_{tp2}`)`
`Take-Picture(`p, s`)`$\wedge$ `eq(`s`,idle)` $\rightarrow$
`Idle(),eq(`e_{tp}, s_i`)`

ensure these conditions hold.

Constraints are also a natural way to model both disjunctive preconditions and conditional effects. For example, the rules

`Take-Picture(`p, s`)`$\wedge$ `eq(`p`,idle)` $\rightarrow$
`Idle(), eq(`s_{tp}, e_i`)`
`Take-Picture(`p, s`)`$\wedge$ `eq(`p`,warmup)` $\rightarrow$
`Warmup(), eq(`s_{tp}, e_w`)`

indicate that two possible preconditions can hold for a `Take-Picture` action, either an `Idle` action or a `Warmup` action. This easily enables back chaining from exogenous events.

Constraints also replace numerical expressions in PDDL 2.1. Let us consider the in-flight refueling model of Figure 3. This would be modeled in the following way: the action

`Fly(`s, i, c, y`)` $\wedge$ `eq(`s`,refuel)` $\rightarrow$
`Fly-and-refuel(`t, j, d, b`),`
`fuel-cons(`s_f, e_f, i, c`),`
`eq(`j, c`), eq(`y, b`)`

computes the fuel consumption for the `Fly` action, which determines how much fuel is available when the refueling begins. We post equality constraints to ensure that the destination of the original `Fly` operation persists in the new action. The `Fly-and-Refuel` action looks similar:

`Fly-and-Refuel(`s, i, c, y`)` $\wedge$ `eq(`s`,fly)` $\rightarrow$
`Fly(`t, j, d, b`), fuel-prod(`s_f, e_f, i, c`),eq(`j, c`),eq(`y, b`)`

In this case, we post the constraint that fuel is produced instead of consumed, but otherwise the state axioms look very similar.

Exogenous events are simply assertions that actions take place in a plan. One way of thinking about exogenous events is that they are simply a set of simple domain axioms that always hold. Since they are volatile in the same way that goals and initial states are, they properly belong in the initial state file, It is very convenient, however, that we can express them using the same underlying concepts that we use to express the domain axioms. Returning to the communication windows example, we can express the assertion that a communication window is in a plan as follows:

`TRUE` $\rightarrow$ `Comm-Window()`

and constraints that a communication window is followed immediately by a closed communication window are written

`TRUE` $\rightarrow$
`Comm-Window(), No-Comm-Window(),`
`eq(`e_c, s_n`)`

Pros and Cons of a Constraint-Based Representation

A number of aspects of PDDL 2.1 make it difficult to build planners that work by means other than progression. For example, suppose that a goal is to fly an airplane to a city, but nothing in the goal specifies the remaining fuel. A progression planner can simulate the `Fly` action with the current fuel and check for action success. However, a regression planner must figure out how to invert the functions in the domain axioms to determine the minimum amount of fuel needed to perform the action in the city of origin. Constraints make this easier, since they are simply relations on the legal values of the variables. In a sense, however, this moves the problem to the underlying support system to enforce the relation correctly. However, this is not required. Domains can be written that involve only successor state axioms, or only involve explanatory axioms. Thus, if a modeler knows that only progression planning is needed, only the successor axioms need to be put in the model.

As with PDDL, generic states can be introduced without fixing the entity that takes on that state. Since this is just another variable, it can be constrained just like any other variable. However, as we said earlier, mutual exclusion is enforced on objects as a part of the semantics of objects. For PDDL, this must be done by other means, either using

hand-coded domain axioms or propositions or numeric expressions to simulate unary resources.

PDDL uses the STRIPS axiom to ensure that propositions that are not negated persist in time. This is a little harder to do using our framework. Since properties of objects are manifested as parameters in the states, we need to ensure they are propagated from state to state using equivalence constraints, as we saw in the `Fly-and-Refuel` example.

PDDL actions can change the value of many propositions at once. Synchronizing state changes using the concepts we describe is also simple. We can write a rule that forces many objects to change their states all "simultaneously".

An advantage of our approach over PDDL is that we can write rules that require unconditional state changes. However, in some cases, we are actually forced to do so; an example is "idle" states where we would like to persist some state information.

We have eliminated explicit actions from our representation. Part of the reason for this is that, when states have duration, there is a blurring of the distinction between temporally extended states and actions with duration. In many domains, some properties that appear "static" are really "active"; a spacecraft pointing at Earth is performing many functions in order to do so, for example. Finally, some states may only hold for a short time, as opposed to continuing indefinitely. Since actions can be mimicked using parameters of states, and since most propositional planners assume actions to be instantaneous, we feel this imposes no great burden. As we discuss later, actions can be introduced at the syntactic level if desired. However, the underlying semantics is concerned only with state transitions.

Extensions

How can the core be extended? We describe three principal extensions: states with temporal extent, uncertainty, and dynamic domains. All of these extensions are very natural and the fundamental concepts we have described above make it easy to create languages that support these features.

State Duration and Metric Temporal Constraints

States can be extended to have duration, and constraints then govern duration and the temporal relationship between states. As an example consider the `Take-Picture` state. Suppose its duration is 24 seconds. Then we have the following rule: `Take-Picture(`p, s`)`$\rightarrow$
`addeq(`$s_{tp}, 24, e_{tp}$`)`
States are now more properly called *intervals*. Note that this is a very natural extension given the representation described previously; we merely add more constraints on the start and end variables of states.

More generally, we can post any constraints in Allen's algebra. For example, consider the satellite domain in which the `Take-Picture` state required the satellite to be `Stable` for 5 seconds before and after the action. Consider the domain description in Figure 1. Compare it to the following:

`Take-Picture(`p, s`)`$\rightarrow$
`Stable(`t`), eq(`s, t`),`
`addeq(`$s_S, -5, s_T$`),addeq(`$e_S, 5, e_T$`)`

We can concisely express the constraint that a `Take-Picture` state requires a `Stable` state that "contains" it, and express the exact constraints that must hold between the temporal variables of the states. Furthermore, we can also ensure not only that some state occurs in the plan, we can ensure that it happens at a particular time.

If states have duration, we can no longer employ the STRIPS axiom, States do not necessarily persist indefinitely; we must write the successor axioms and frame axioms for all states. However, this does not impose a serious burden on the modeler in most cases. A domain axiom can indicate that a particular state can last indefinitely, but their successors must be enumerated in case the state is terminated. For example, consider the `Idle` state in the satellite example. In the event that a state terminates, we must describe what states can follow it. Termination is accomplished by assigning or constraining the duration of the state. Defining successors can be done a number of ways, but an easy way is to use a parameter of the state to define the possible successors, then use conditions as we have described in previous examples. The rules would look like this:

Uncertainty

Uncertainty can be added in several different flavors to accommodate the needs of the domain. For example, in a contingency planning context, one might only wish to provide the set of possible outcomes. Those wishing a description more like MDPs can provide probability distributions over action transitions. If we revisit the satellite domain, we see that the rules need to be augmented in these cases. Suppose that trying to take a picture may fail because the shutter does not open. We can do so by introducing a special set of world-choice variables for each state, which are "set" by the world. For example, suppose the Take-Picture action either results in a Camera-Ready state or a Camera-Broken state, conditional on an outcome, !o, which the planner has no control over:

`Take-Picture(`p, s`)`
$\wedge$ `eq(`s`,take-pic),eq(`!o`,ready)`$\rightarrow$
`Camera-Ready(),eq(`e_{tp}, s_r`)`
`Take-Picture(`p, s`)`
$\wedge$ `eq(`s`,take-pic),eq(`!o`,broken)`$\rightarrow$
`Camera-Broken(),eq(`e_{tp}, s_b`)`

A richer representation of uncertainty allows us to specify a probability distribution over possible outcomes. We can augment the above example by associating probabilities with the different values of the outcome variable !o. Notice we need only do this for successor state transitions, not explanatory axioms.

A more complex task is to handle continuous probability distributions over the outcomes of actions. Uncertainty can be represented in terms of unknown values of variables. For example, uncertainty over the start time of an event can be expressed as an interval representation for the start-time variable. Again, we must take care to distinguish between uncertainty, where the world chooses, and temporal flexibility, where the agent chooses. More sophisticated representations can add probability distributions over values in the interval. For example, if the `Take-Picture` action results in an uncertain amount of onboard storage use, we can

imagine extending the set of constraints to involve continuous probability distributions as constraints over quantities. Sensing actions can constrain the parameters of the distributions, for example. However, the fundamental notion of constraints over variables in the states still holds.

More importantly for software domains, we can represent uncertainty over the value of an object attribute, such as the pathname or size of a file. Here we see a significant advantage over propositional representations, because these attributes have infinite domains; representing the possibilities as a list of worlds is impossible. Instead, we leave the domain of the variable open, to indicate that it could have any value, or partially open, to indicate that it is restricted to a particular subset of values.

In addition to uncertainty over the attributes, we can represent uncertainty over the objects themselves using the same representation. It is not necessary to list all objects that could exist in the world; it is sufficient to represent the actions that can discover new objects and dynamically introduce new variables as needed to describe new objects as they are discovered. We can represent sensors that return arbitrary numbers of new objects by making the world-choice variables $!o$ universally quantified (Golden & Weld 1996; Golden & Frank 2002).

Dynamic Domains

Dynamic domains arise both in the context of sensing, when a new object in the world is detected, and object creation, when an action in the plan leads to the creation of new objects whose states must be reasoned about. We can handle sensing and object creation using a similar approach. A newly created value is similar in most respects to a newly sensed value, the differences being that, in the case of object creation, the world changes and the planner has some control over the outcome. We can represent a new object, such as the output of a data-producing action, as a variable whose value is a skolem function of the corresponding action. As in the case of sensing, if the number of objects that will be created is unknown (because it depends on an unknown number of inputs, for example), we can represent the effect using universal quantification, where one variable is used to represent a set of objects.

A simple extension to the form of domain axioms enables this. Recall that domain axioms can lead to the creation of a new state for an object, if an existing state isn't appropriate. Thus, there is already precedent for constraints that hold to justify the existence of a new state. We can extend the form of domain axioms to enable the creation of new objects as well. For example, consider the zip file creation action. Let us suppose that the states a zip file can be in are `Idle, Compressing, Uncompressing, Moving`, and the properties of interest of zip files are its size, whether or not it is compressed (represented by a boolean) and its location. We can write this as follows:

`Zip(`l, p`)` $\rightarrow$
  `new Zip-File, Idle(`m, s, c`),eq(`c`,false)`
  `zipsizeof(`s, p`),eq(`m, l`),eq(`s_i, e_z`)`

The keyword `new` indicates that the zip file we are asserting properties of is created and is like the approach used in (Golden 2002). The semantics of this can ensure that no state before the time of creation can be asserted. However, we can also impose the usual constraints on the `Idle` state of the file, along with asserting the files initial size, location, and compressed state.

Syntax

The fundamental construct we have used in the descriptions above is that the presence of a state in a plan implies some other states must exist, and that there are some constraints. We can wrap these ideas in a number of convenient syntactic constructs. We will describe a variety of these in this section.

We will begin with simple domains where states do not have duration and metric temporal constraints are not used. We can use syntax that posts ordering constraints on the states directly:
`P` $<$ `Q` and translate this into the constraints on variables. If temporal constraints and states with duration are used, we can use the Allen's algebra names or other convenient labels to express temporal constraints. In the case of the constraint that the satellite must be stable while taking pictures, this constraint is written

`Take-Picture(`p, s`)` $\rightarrow$
  `contained-by[5][5] Stable(`t`)`

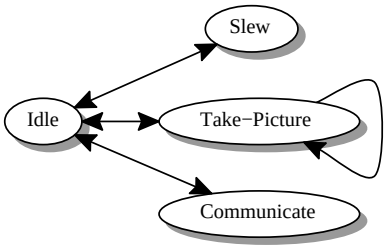
Equivalence constraints can be posted by simply using the same variable names in the parameter lists of the states.

For those who want to build models with a distinction between state and action, this can be accomplished. Actions would depend on objects being in particular states, and would ensure that some objects have new states. A simple transformation would augment each state with action parameters and the axioms can be rewritten to ensure that the proper constraints are posted among the variables of different states.

As we said previously, since properties of objects are represented by parameters of states, a mechanism is needed to propagate values to states where they are involved in constraints. However, syntax can conceal these details from the modeler. For example, objects can be created with a fixed set of variables, and the states can use these variable names in constraints. The underlying reasoning system can then decide whether new variable instances are needed and post the appropriate constraints. In the satellite domain, `Idle` states normally would propagate the amount of data in the onboard storage unit to the next `Take-Picture` or `Communicate` state. However, the action need not name the variable representing the data amount, and the underlying system would simply use the variable representing the last computed quantity in the constraint involving the next state. Notice that this syntax is similar to the PDDL 2.1 syntax, but with a different interpretation.

The astute reader will notice that, since mutual exclusion is enforced on object states, that state machines or timed automata are a good representation for many planning domains. The transformation between these representations and our fundamental language of states and constraints is also very straightforward. Rules relating the states of different objects are represented by synchronizations across different state machines.

As we said previously, domain axioms can be thought of as implications that always hold. However, another way of



Idle → Slew
Idle → Take−Picture
Idle → Communicate
...

Figure 4: A simple state machine representation of the satellite domain. The rules implied by the state machine appear below the figure.

thinking about them is as partial plans. As such, we can assert that actions take place and have constraints among their variables, without deciding when they take place, or even whether they are ordered or not. Syntax describing partial plans can take a wide variety of forms.

Finally, resource declarations can be added to a language to augment numerical constraints, enable technologies like edge finding and envelope calculations, and to add descriptive clarity to model definitions. With an explicit resource declaration, we can replace axioms designed to enforce mutual exclusion with a unary resource shared by many actions, as well as numerical expressions meant to simulate resources.

For example, consider the case of a unary resource, stability, used by 5 different actions: three Take-Picture actions (one for each of three instruments on a satellite), communication, and slewing. All the actions but the slewing action require stability, and slewing makes the satellite unstable. We might write this model as follows:

```
Resource unary stability
Take-Picture(p, s)→
  uses Stability (s_t − 5, e_t + 5)
Communicate()→
  uses Stability (s_t, e_t)
Slew()→
  uses Stability (s_t, e_t)
```

Each state now declares how it uses the resource. The usage time can be constrained using mathematical functions of the start and end times of the activity in any way the domain modeler sees fit.

Now consider a mutli-capacity reusable resource such as power. Resources now must declare their resource impact as well as the time span during which the resource is affected. As an example:

```
Resource multi reusable power 5
Take-Picture(p, s)→
  uses power (s_t − 5, e_t + 5, 5)
```

Finally, we consider renewable resources, where each activity can consume the resource or produce the resource. In this case, we must allow for the possibility that an activity could change the resource in different ways at different timepoints, in general.

```
Resource multi renewable power 5
Take-Picture(p, s)→
  uses power (s_t − 5, 5)
```

This looks quite similar to the declaration of a function in PDDL, but there is an important difference: it is easier to understand that the resource is a complex, flexibly scoped constraint that can be reasoned about as a single entity. This simplifies modeling as well as revealing the reasons for mutual exclusion of actions. The computational burden is wholly shifted to the implementation, where it can be efficiently handled in any way the implementer sees fit. All of these declarations can be converted into simple arithmetic constraints, or they could be used as the input to edge finding, envelope calculations, or other sophisticated techniques.

A final syntax issue is that of functional representations versus object based representations. PDDL 2.1 uses a functional representation, and allows objects to be passed as arguments to functions. Other planning domain languages use a notion of objects with attributes, where attributes can be accessed using syntax that resembles that in object oriented programming languages. Neither of these approaches is fundamentally incompatible with a constraint-based representation such as the one we have proposed. The two approaches offer different representational transparency in the model and in the way in which planners access the information, but can represent the same things.

A Challenge for the Community

In this paper, we do not advocate a single planning domain description language. Even though the fundamental concepts we have described appear quite general and powerful, it would be easy to create a single, very clumsy language supporting many features using these concepts. However, we believe that using these concepts as a starting point will make it easier for language designers to extend the basic language in a wide variety of ways and create good languages for accomplishing many modeling tasks.

Several existing plan domain description languages make use of some of the ideas presented here. Numerous languages have more flexible temporal representations (Jónsson *et al.* 2000; Smith & Jonsson 2002), use constraints rather than functions (Frank & Jónsson 2003), and use dynamic domains (Golden & Frank 2002). All of these languages have their pros and cons. Language designers should be sensitive to the strengths and weaknesses of these languages for the various purposes they are used for, and consider how the language is likely to be used. The challenge for the planning community is not to search for one language that fits all needs, but to search for the core elements of languages that are most suitable for modeling different planning domains.

References

Baptiste, P., and Pape, C. L. 1996. Edge-finding constraint propagation algorithms for disjunctive and cumula-

tive scheduling. In *Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group*.

Fox, M., and Long, D. 2003. Pddl 2.1: An extension of pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*.

Frank, J., and Jónsson, A. 2003. Constraint based attribute and interval planning. *Journal of Constraints*.

Golden, K., and Frank, J. 2002. Universal quantification in a constraint-based planner. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling*.

Golden, K., and Weld, D. 1996. Representing sensing actions: The middle ground revisited. In *Proc. 5th Int. Conf. Principles of Knowledge Representation and Reasoning*, 174–185.

Golden, K. 2002. Dpadl: An action language for data processing domains. In *Proceedings of the 3rd NASA Intl. Planning and Scheduling workshop*, 28–33.

Jónsson, A. K.; Morris, P. H.; Muscettola, N.; Rajan, K.; and Smith, B. 2000. Planning in interplanetary space: Theory and practice. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*.

Laborie, P. 2003. Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. *Artificial Intelligence* 143(2).

Long, D., and Fox, M. 2000. Recognizing and exploiting generic types in planning domains. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling*.

Muscettola, N. 2002. Computing the envelope for stepwise-constant resource allocations. In *Proceedings of the Eighth International Conference on the Principles and Practices of Constraint Programming*.

Smith, D., and Jonsson, A. 2002. The logic of reachability. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling*.

Model-Based Planning for Object-Rearrangement Problems*

Max Garagnani and **Yucheng Ding**
Department of Computing
The Open University
Walton Hall, Milton Keynes - MK7 6AA
{M.Garagnani,Y.Ding}@open.ac.uk

Abstract

We describe a model-based planning representation, aimed at capturing more efficiently the basic *topological* and *structural* properties of a domain. We specify the syntax of a domain-modelling language based on the proposed representation. We report the experimental results obtained with a prototype system (called PMP, Pattern-Matching Planner) able to represent and solve planning problems expressed in this language. The performances of PMP on a set of five domains are compared with those of a second planner, adopting the *same* search algorithm but using a classical STRIPS propositional language. Preliminary results show a superior performance of PMP on *all* of the chosen domains.

Introduction

During the past few years, the planning community has put a significant effort into developing systems able to exploit *domain-specific* knowledge to carry out a more 'informed' search (e.g., knowledge about the generic type and structure of a domain (Fox & Long 2001; 2002), control knowledge and structure of desirable solutions (Bacchus & Kabanza 2000)(Nau *et al.* 1999), heuristics (Hoffmann & Nebel 2001)(Haslum & Geffner 2000), problem constraints and domain invariants (Kautz & Selman 1999)(Gerevini & Schubert 1998)). In spite of the leap in the scale and complexity of the problems solved that this effort has produced, current applications are still limited to narrow, well-defined domains, and do not exhibit the flexibility and adaptability that characterise human planners (Wilkins 1997)(Wilkins & desJardins 2001). To a large extent, the cause of this limitation is the fact that, in addition to *domain-specific* knowledge, planning in the real world requires using *common-sense* knowledge and reasoning (including reasoning by analogy, abstraction, learning, and dealing with uncertainty and incomplete knowledge), a type of inference which has proven particularly hard to automate in all areas of AI.

This work is guided by the hypothesis that one of the main factors preventing modern planning systems from carrying out fast and effective common-sense reasoning (and, hence, from scaling well to realistic domains) lies in their adoption of *inefficient problem representations*. In particular, most current planners rely on '*propositional*' domain descriptions languages (e.g., STRIPS, ADL, PDDL and descendants). Such formalisms are not always appropriate for modelling real-world problems, particularly when these require a substantial amount of common-sense reasoning about *spatial* and *topological* relations between objects. Indeed, even the most recent versions of PDDL (Fox & Long 2003) require the basic physical properties and *constraints* of the world (e.g., the fact that an object cannot be simultaneously in two different places) to be declared and/or dealt with *explicitly*. The adequate encoding and exploitation of such constraints turns out to be crucial for achieving good performances in large and realistically complex problems (e.g., see (Kautz & Selman 1998)).

An alternative to adopting propositional (or 'sentential') formalisms consists of using *model-based* (or 'analogical') domain descriptions. In a model-based representation, the world state is encoded as a data structure which is *isomorphic* to (i.e., a model of) the *semantics* of the problem domain. For example, in their seminal work, Halpern and Vardi proposed the adoption of a Kripke structure to model the 'possible-worlds' knowledge of a group of interacting agents (Halpern & Vardi 1991). Because of their isomorphism with the world state, a key feature of model-based representations is their ability to *implicitly* embody constraints that other representations must make explicit, and, hence, to improve the efficiency of the reasoning process (Myers & Konolige 1992). On the other hand, model-based formalisms tend to be less expressive and more limited in scope than propositional languages.

In this paper, we propose a model-based planning representation, able to capture implicitly, more efficiently and naturally the basic, common-sense *structural* and *topological* constraints (expressing spatial and 'containment' relationships, respectively) of a domain. Although model-based, the representation is sufficiently expressive to allow the encoding of a significant set of domains, in which the planning performances are notably improved.

The rest of the paper is organised as follows: first of all, we delimit the class of domains included within the scope of this investigation and describe the general features of the new representation. Secondly, we specify the syntax of a description language, which allows encoding domains using a simplified version of the general representation proposed.

*This work was partially supported by the UK Engineering and Physical Sciences Research Council, grant no. GR/R53432/01

Thirdly, we discuss preliminary results obtained with a prototype planner on a set of five domains, and conclude by pointing out advantages, limitations and possible extensions of the proposed approach.

A Model-Based Representation

The planning representation that we describe here has been developed to allow the efficient and natural encoding of object-rearrangement (or, simply, *move*) domains. These can be defined as problems that require planning the changes of *position* (location) of a finite set of objects on the basis of their spatial and topological relations, subject to a set of constraints. The Tower of Hanoi (ToH) represents a prototypical example of this class, in which the positions of a set of objects (disks) have to be changed according to a set of rules (constraining the movement of the disks). Other examples of this class are the Briefcase domain, Gripper, Blocksworld (BW), Grid, Logistics and Eight-puzzle. Notice, however, that although not explicitly of a 'move' nature, some domains are isomorphic to (and can be treated as) object-relocation problems. For example, if *activities* are represented as objects, and locations denote *time points* or *intervals*, then the problem of scheduling a number of tasks over a given time period can be seen as that of re-assigning to each 'object' (activity) an appropriate 'location' (start/end time point), subject to various constraints. More in general, any *state* change of an object can be modelled as a change of *position*, given an appropriate reformulation of the domain.

The basic entities of our representation are '*nodes*', '*places*' and '*edges*'. 'Nodes' represent instances of the types of (mobile) objects present in the domain (e.g., physical objects, agents, resources, etc.). 'Places' denote different locations of the domain, and can be thought of as *qualitatively* distinct areas of space containing sets of objects. A place can contain nodes, other '*sub*-places', or both. 'Edges' are pairs of places and nodes, and express spatial and topological relationships between them. An edge may 'connect' two places, two nodes, or a place and a node. Nodes, places and edges can be associated to unique labels.

The sub-places of a place are places themselves, and can be used to define the *internal structure* of a place. A place may be defined so that it is subject to specific restrictions, limiting, for example, the *type* and *number* of nodes that it can contain. The sub-places of a place may also be connected by edges. A place containing no connected sub-places will be called '*unstructured*'.[1] Nodes, places and edges are defined using three separate type hierarchies, in which the properties of a type are inherited by all of its instances and sub-types.

Figure 1.(a) shows an example of node and place hierarchies for the well-known Briefcase domain. The types "OBJECT" and "PLACE" lie at the roots of the two hierarchies. The place hierarchy specifies that a "Location" place will be allowed to contain any number of nodes of type "OBJECT" (i.e., instances of "Portable" or "Mobile"). A place of type "Briefcase" can only contain "Portable" nodes.

[1] In general, one can see nodes as places required to be always empty, or places as nodes which contain other nodes.

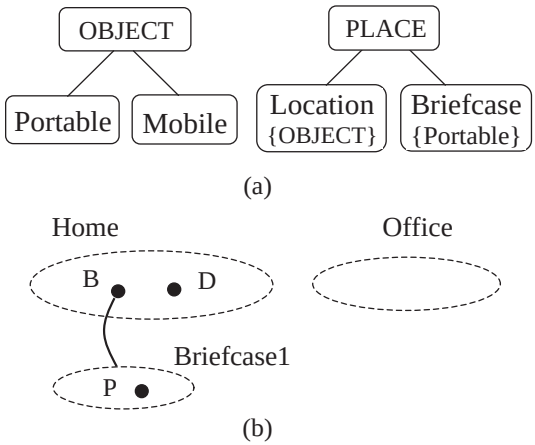


Figure 1: Briefcase world: (a) type hierarchies, and (b) initial state for the "Get-paid" problem

Figure 1.(b) contains a graphical representation of a possible encoding of a state in terms of places, nodes and edges. Nodes are represented as (labelled) filled circles, places are denoted by dashed ellipses, and edges are depicted as bold arcs (in this example, the edge hierarchy can be assumed to contain only the root class "EDGE"). The state represented in Figure 1.(b) corresponds to the set of propositions {`at(Home,B), at(Home,D), in(P), at(Home,P)`}, the initial state for the "Get-paid" problem. Nodes 'P' and 'D' are "Portable" objects (the types of the various instances are not shown in the figure), whereas 'B' (the briefcase) is an instance of "Mobile". Notice also the presence of different types of places: 'Home' and 'Office' are instances of "Location", while 'Briefcase1' is a place of type "Briefcase". The state contains a single edge, connecting node 'B' with place 'Briefcase1' and encoding the *association* between the briefcase node and its contents.

As a second example, Figure 2 models the Blocksworld domain. Part (a) of the figure describes node and place hierarchies, according to which a "Cell" place is allowed to contain up to one ('[1]') node of type "OBJECT", while a "Stack" place can contain any number of "Cells" as sub-places. Part (b) shows a graphical representation of a three-block problem. This example demonstrates the use of *structured* places and edges as place-to-place connectors. In particular, each of the three "Stacks" S_1-S_3 contains four connected "Cells", some of which contain a node. Although in this example they are not connected, these three places could, in turn, be linked by (possibly labelled) edges. The nodes labelled 'A', 'B' and 'C' are instances of the type "Block", while the three nodes labelled 'T' are of type "Table". Notice that such 'T' nodes refer to distinct objects of the current state which do not need to be *discriminated* at this level of the representation, and that have been assigned the same label. We will refer to this kind of objects as *generic instances* of a type, in that they cannot be distinguished from each other, but can still be discerned from other entities (even of the same type) having a unique name. In this example, none of the internal sub-places or edges have been associated to a unique label.

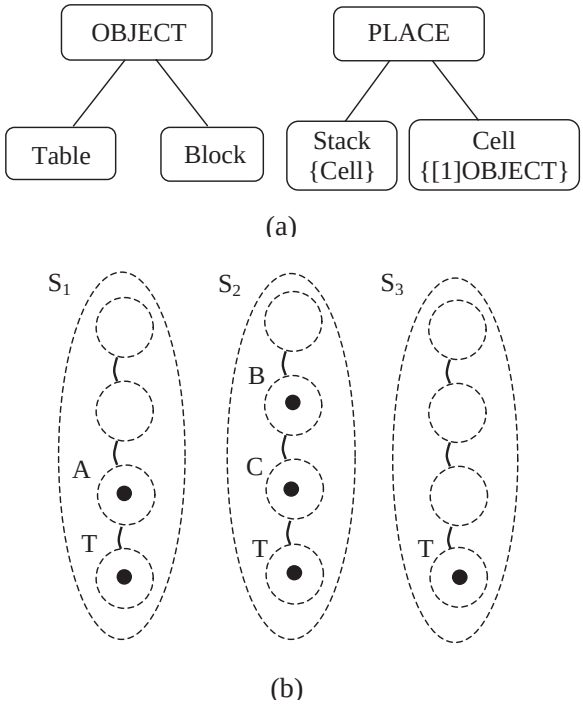


Figure 2: Blocksworld: (a) type hierarchies; (b) graphical representation of the state {`On(A,Table), On(B,C), On(C,Table), Clear(A), Clear(B)`}

Representing Change

Having described the basic elements of the state representation, let us move on to the definition of the formalism for specifying the set of possible *action schemata*, necessary for identifying the legal transformations of the state. In this representation, the world state is a collection of places, nodes and edges. In general, any of these elements will be allowed to be moved from their current position, or even to be *added* to or *removed* from the state. However, nodes are often the only *mobile* objects of the domain, while places and connecting edges cannot be affected during plan execution and can be regarded as forming an underlying *stationary* structure. For example, in Blocksworld, the internal cells of the three stacks can be regarded as 'fixed' places, while the blocks and the 'table' nodes can be moved from one place to another (subject to appropriate constraints).

In this analysis, we assumed that the possible transformations of the state are limited to the *movement* of nodes and places, while the edges connecting such entities remains unchanged.[2] When a node (or a place) moves, all edges connecting it to other entities remain 'attached' to it, and all contents of a place move with it. In addition, the movement of places and nodes is restricted by the general constraints of the domain (e.g., number and types of nodes allowed in a place) specified by the type hierarchies.

Consider, for example, the action schema '*Put-in*' for the Briefcase domain, illustrated graphically in Figure 3.(a). The left-hand side (*preconditions*) of the schema specifies the situation holding in the two relevant ('loosely' connected) places before the execution of the action. (Notice that the absence of nodes in one of the places should not be interpreted here as requiring such place to be empty – this is clarified below). The right-hand side (*effects*) depicts the same set of places and nodes after the action has produced a new node arrangement.[3]

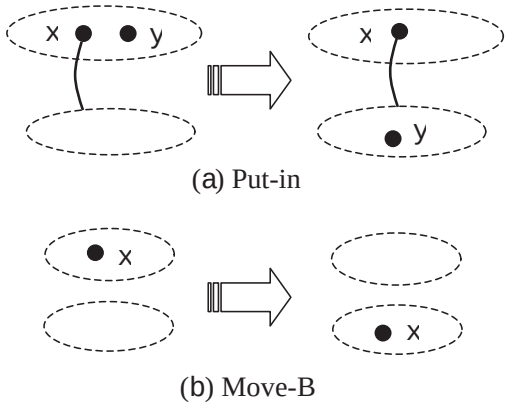


Figure 3: Briefcase domain: action schemata

In general, the preconditions of an action may contain a *conjunction* of 'groups' of connected places and nodes. An action schema can be applied in a state S when *each and every one of the elements* – places, nodes and edges – *present in its preconditions* 'matches' *(can be bound to) a distinct*[4] *element of* S, so that each precondition group is bound to a (distinct) group in S having the same *topological structure* ('pattern' of places and edges) and node arrangement. The definition of '*match*' is as follows: two types match *iff* one is sub-type of the other; two instances match *iff* they are the same instance (i.e., they have the same name); an instance x and a type T match *iff* x is an instance of T, or x is an instance of any of T's sub-types. A precondition group consisting of a place containing n nodes matches any place (of the appropriate type) containing *at least* n nodes (of the appropriate type). For two edges to match, they must connect matching entities. In summary, for an action to be applicable, each of its precondition groups must 'overlap' with a distinct part of the current state. In the previous example, the preconditions of the *Put-in* operator contain only one group, which is easily mapped to corresponding elements of the state shown in Figure 1.(b).

Figure 3.(b) contains the graphical representation of the more interesting action schema '*Move-B*', which allows the movement of the briefcase from a location to another. The preconditions of this action contain *two* groups, which must be bound to distinct places of the state. For the action schema to be applied correctly, node '`x`' must be bound to

[2] This implies that the initial number of entities – nodes, places and edges – remains *constant* throughout plan execution.

[3] The *Take-out* action schema can be obtained simply by reading the *Put-in* operator 'backwards'.

[4] Notice that *generic* instances of a type still represent distinct elements of the state.

an instance of "Mobile", and the two places to (distinct) instances of "Location". If these constraints on the type of the nodes and places were not enforced, these elements could be bound to incorrect instances and lead to illegal moves (such as moving a "Portable" node like 'D' directly across locations, or a "Mobile" (briefcase) node inside a "Briefcase" place). Notice that the *Put-in* operator (and its mirror-image *Take-out*) will also be subject to appropriate type requirements, although in this case the topological structure of the preconditions and effects is sufficient to guarantee that the application of the schema (in both directions) to a semantically correct state will always produce a correct state, even when multiple briefcases are present.

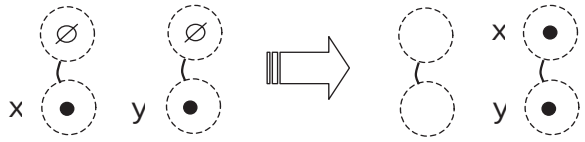


Figure 4: Blocksworld: '*Put-On*' action schema

Finally, consider the BW *Put-on* action schema, represented graphically in Figure 4. Let us assume the types of the nodes 'x' and 'y' to be, respectively, "Block" and "OBJECT", and all places to be of type "Cell". Notice the use of empty-set symbols in the preconditions, requiring the two top cells to be empty. With reference to the state depicted in Figure 2.(b), the first group of the preconditions would match the cell containing 'A' (and the empty cell immediately above), or, alternatively, the cell containing 'B' (and the one just above). Similarly, the second group would match the same cells, and, in addition, the two cells at the bottom of 'S_3' ('y' is of type "OBJECT"). Hence, this schema could be applied in the current state to move 'A' onto 'B' or vice versa, or either of the two blocks on the table. Notice that for this action schema to be used in a 'backward' search, it would be necessary to require an extra empty cell on top of the second group, in order to guarantee that node 'x' (being removed from the top of 'y') is 'clear'.

In general, the nodes, places and edges present in the initial state of a problem are not necessarily *preserved* throughout the plan execution: new objects can be dynamically 'produced', existing objects can change their *state* or be 'consumed', and relations between places might change as a result of some of the actions. For example, in a 'house construction' domain in which places represent the locations that the workers can currently reach, the *addition* of a new floor or of a new scaffolding will change the set of places and the connections between them. Although such 'non-conservative' state transformations have not been considered here, the model could be easily extended to include a set of *primitive* actions which allow the addition and removal of nodes, places or edges (subject to appropriate preconditions), enabling the representation of this type of dynamics.

A Prototype Language

In order to assess the effectiveness of the representation, a prototype planner has been implemented. The domain-description language developed for the planner consists of a restricted, 'diagrammatic' version of the formalism described in the previous section. In particular, in the implementation, the *internal* structure of places is confined to be a one- or two-dimensional *grid* of sub-places. Hence, places can either be unstructured, or have their internal structure equivalent to that of a matrix (or vector), in which cells are considered as a collection of *adjacent* locations and are allowed to contain up to *one* object (node).[5] Since nodes have been represented simply as labels, a place is a (structured or unstructured) collection of strings.

A further simplification of the language implemented consists of not allowing the use of connecting *edges*, which encode topological and spatial relationships. This restriction is compensated in part by the possibility of places to have a predefined, fixed internal structure (which can be augmented with a set of dedicated spatial relationship – this is discussed below in more details), and in part by the possible use of *identical names* across hierarchies, which allows, for example, a node and a place to be assigned the same label (i.e., to be somewhat connected). As mentioned before, dynamic changes of the underlying topological structure of the domain have not been allowed.

In spite of the limited expressiveness, this language still contains most of the fundamental characteristics of the general model, and allows a preliminary assessment of the validity of its basic assumptions. The language developed is sufficiently flexible to be able to model naturally and efficiently a small subset of the benchmark domains (including BW, Briefcase, Miconic, Gripper and Eight-puzzle), which we used to carry out a set of experiments. The results of such experiments are reported in the final part of this section. In what follows, we illustrate briefly the syntax of the language and its semantics, exemplifying the description with declarations taken from the BW domain.

Type Hierarchies

The type hierarchies are declared using a syntax similar to that of PDDL. To describe the syntax, we adopt the extended BNF (EBNF) formalism, used in the original PDDL definition (McDermott *et al.* 1998):

<types-def> ::= (:ObjectTypes <typed-list (name)>)
<types-def> ::= (:PlaceTypes <place-type>$^+$)
<typed-list (x)> ::= x^* | x^+- <type> <typed-list (x)>
<place-type> ::= <name> {<type>[::<dimension>]}
<type> ::= <name> | (either <type>$^+$)
<dimension> ::= 1 | 2

The pipe character ('|') indicates disjunction (e.g., <dimension> can be either '1' or '2'). The category <name> can be any string of characters, not necessarily beginning with a letter. The 'typed list' is a parameterised production that generates a (possibly empty) list of object-type names and 'IS-A' relations, using the minus sign ("$-$") as in PDDL to indicate a 'parent' type. '`object`' and '`place`' are both predefined types, and '`object`' is used as default

[5] Notice that, for simplicity, the notion of 'adjacency' in two-dimensions is restricted to *vertical* and *horizontal* pairs of cells.

terminator of any typed list. The optional parameter in the place type declaration (surrounded by square brackets) allows the specification of the internal structure of a type of place (array of one or two dimensions)[6]. In absence of such parameter, the place type is assumed by default to be unstructured. As in PDDL, the 'either' construct allows an object type to be defined as the union of several types. Below is an example of type declaration for the BW domain:

```
(define (domain blocksworld)
   ...
   (:ObjectTypes block table)
   (:PlaceTypes stack {object::1})
   ...
)
```

Types '`block`' and '`table`' are declared (by default) as sub-types of '`object`'. Places of type '`stack`' will be *one*-dimensional arrays ("::1") of '`object`'s. Notice that this syntax does not allow the specification of an upper limit on the number of elements contained in an unstructured place, or on the number of 'cells' composing a structured place. That is, a '`stack`' place (vector) could, at this point, contain any number of cells, each one containing up to one '`object`'. The upper limit on the number of elements (or cells) contained in a place will be determined at 'run time', by the specific problem instance.

Action Schemata

The following EBNF productions specify the syntax for the declaration of an action schema:

```
<action-def>   ::= (:action  <name>
                       :parameters (<typed-list (variable)>)
                       <body-def>)
<body-def>     ::= :pre (<place>*)
                   :post (<place>*)
<place>        ::= <name> { [<relation>] <object>+ }
<object>       ::= <variable> | <emptySpace>
<variable>     ::= <name>
<relation>     ::= ⋆ | / | ↔ | ↕
<emptySpace>   ::= _
```

The action declaration consists of a unique name, a list of parameters, and lists of 'pre'- and 'post'-conditions (effects). The parameters are names of variables, coupled with the respective types. The pre- and post-conditions consist of lists of place types, each containing as argument a list of parameters and, possibly, 'empty spaces'. The empty-space symbol ('`_`') is used in the preconditions to require the presence of empty cells in structured places, or to require the availability of 'space' for an object in non-structured ones.

For an action to be applicable, all the elements present in the preconditions must be bound to appropriate elements of the current state, as explained earlier on. Notice that *all* the places listed in the preconditions must be in a *1-1* mapping with those listed in the postconditions, following the *order* in which they are listed. In addition, the number of objects present in each place must remain constant[7]. Below is an example of '*Put-on*' action schema in Blocksworld:

```
(:action Put-on
    :parameters (x y - block)
    :pre  (stack {x _ } stack {y _ })
    :post (stack {_ _ } stack {y x })
)
```

This declaration can be easily mapped to the graphical representation of Figure 4. By default, the elements listed inside a *structured* place are interpreted as being required to be *adjacent* (i.e., to occupy adjacent cells). Thus, blocks '`x`' and '`y`' are guaranteed to be 'clear' by the presence of an empty cell adjacent to them.[8] The two '`stack`' places listed in the preconditions are mapped to the two places listed in the effects, in the order specified. The content of each cell of a *structured* place listed in the preconditions (identified by a parameter or an empty space) is *replaced* with the object occupying the *same position* in the postconditions, following the *order* in which the objects are listed. For example, the cell containing the object that gets bound to parameter '`x`' will end up containing an empty-space ('`_`'), and the (currently empty) cell adjacent to '`y`' will contain '`x`'. Elements specified in a non-structured place cannot be required to belong in any specific spatial relationship.

In order to require more complex spatial relationship to hold between elements contained in a structured place, the language has been endowed with a set of (optional) <relation> symbols, representing a limited version of the more general feature of labelled *edge*. These symbols can be used inside structured places to require that a certain spatial relationship, different from that of adjacency, hold between the specified elements. For example, '$\star$' indicates that the elements that follow can be (with respect to each other) "anywhere within the place"; '$\leftrightarrow$' means "anywhere on the same row", '$\updownarrow$' means "anywhere in the same column", and '/' requires that the two following elements be located in adjacent cells of the same column (the last two relationships are only needed for two-dimensional arrays). The chosen set of relationships is by no means complete, and can be easily extended with other, more complex, ones. An example of usage of these relationships is demonstrated by the Miconic (elevator) domain description, reported in Appendix A.

Initial State and Goal

We use an actual initial state and goal declarations for a specific BW problem (the 'Sussman anomaly') to illustrate informally the syntax adopted for describing initial state and goal. The correct formalisation can be easily inferred from this example and the EBNF rules used earlier for the types and actions declarations:

[6] Notice that unlike square brackets, which are meta-symbols, *curly* brackets are terminal symbols of the language being defined.

[7] An 'empty space' is treated as a special type of object.

[8] This assumes that all the '`block`' objects will be arranged in the '`stack`'s so as to have always at most *one* empty cell adjacent to them, which represents the space on 'top' of them – e.g., see Figure 2.(b).

```
(define (problem Sussman)
   (:domain blocksworld)
   ...
   (:Objects A B C - block T - table)
   (:Places s1 s2 s3 - stack)
   (:init
          s1 [T A C _ ]
          s2 [T B _ _ ]
          s3 [T _ _ _ ])
   (:goal
          stack {C B A})
)
```

Notice the similarity between the declaration of the initial state and the graphical representation shown in Figure 2.(b). In the declaration of the initial state, the contents of a *structured* place are delimited by square brackets, indicating the actual 'start' and 'end' of the array. Hence, the declaration implicitly specifies the *maximum* number of required cells, and the *exact position* of all the objects and empty spaces ('_') within them. For unstructured places, the contents will be delimited by *curly* brackets (as for normal sets), and the declaration will specify the contents and the maximum number of objects that a place is able to contain.

A *goal* is syntactically equivalent to a precondition list. It consists of a 'conjunction' of places required to contain specific sets of objects (nodes), possibly subject to specific spatial relationships. The conditions for achieving a goal are analogous to those required for the application of an action schema: a goal g is *achieved* in a state S when all groups specified in g can be bound to distinct groups of S, such that, for each place of g, all the elements contained in it match distinct elements in the corresponding place of S (which satisfy the same spatial relationships, if appropriate).

In goals – like in action preconditions – the objects listed inside a *structured* place are required to occupy consecutive cells of the place, in the same order specified. However, the 'pattern' of objects specified can be placed *anywhere* within the place. More formally, if L is a type of structured place, and $x_1, x_2, \ldots, x_n$ are objects, the goal $L\{x_1 x_2 \ldots x_n\}$ requires an instance of a place of type L to contain the listed objects in *any* n consecutive cells, such that (x_1, x_2), (x_2, x_3), (x_3, x_4), $\ldots$, (x_{n-1}, x_n) are all pairs of adjacent elements. Thus, for example, for a state S to achieve the goal of the '`Sussman`' problem presented earlier, it must be the case that S contains a place of type '`stack`' in which ('C','B'), and ('B','A') are pairs of adjacent elements. Two examples of stacks of four cells that satisfy such requirements are '[T C B A]' and '[C B A _]'.

Experimental results

Ideally, the evaluation of the efficiency gain (or loss) resulting from the adoption of a new domain representation – let us call it 'α' – with respect to another (propositional) representation, 'β', could involve the following four steps: (1) measuring the performance of a state-of-the art planning system adopting β a on a select set of problems; (2) 'switching' the domain representation to α; (3) measuring the performance of the system on the same set of problems; (4) comparing the results. However, this method of assessment presents several drawbacks. The first one concerns the fairness of the evaluation in itself. After more than three decades of research based almost exclusively on propositional domain-modelling languages, planning algorithms have become more and more sophisticated and geared towards purely sentential descriptions. In fact, one could say that their efficiency results mainly from their ability to exploit some of the inherent properties of such representations. Taking a state-of-the-art planning system, designed specifically for propositional descriptions, and simply 'switching' its representation into a completely different one (assuming that this is possible) would not produce a system 'comparable' with the original one. Indeed, the new formalism might be incapable of replicating some of the particular techniques upon which the propositional planner might rely for achieving good performances. At the same time, new, different reasoning modes may become available in the new representation, which could lead to gains in performance. However, modifying the algorithm to take advantage specifically of such features would lead to the implementation of an entirely different system, not comparable with the original one.

The second methodological issue concerns the actual value and feasibility of the assessment. Even assuming that a specific system can be identified such that all of its sophisticated search mechanisms can be correctly 'translated' in the new representation, what would be the significance of an evaluation carried out on such a specific case? The results would not apply to all planning algorithms, not even to those adopting the same propositional language. In addition, replicating correctly all of the various features of the planner in the new representation would require a considerable effort, and would still leave a margin of uncertainty on the soundness of the outcome.

The above considerations suggest that a fair evaluation should be based on a very simple, 'primitive' planning algorithm, in which the few mechanisms at the basis of the search can be encoded in a straightforward manner in both representations. One possible candidate that immediately springs to mind is the original STRIPS planner. However, although old and inefficient, STRIPS is quite a complex system; obtaining, understanding and modifying the original software, written more than thirty years ago, did not seem a very practical approach.

Therefore, in order to assess the value of the new representation, we decided to build two simple prototype planners, adopting exactly the *same search engine* (a traditional, uninformed, breadth-first forward state-space search), but differing in the way they represent domains and problems. The first, propositional planner (which we called 'PP') adopts a classical (typed) STRIPS representation. The second planner (which we called 'PMP', Pattern-Matching Planner) adopts the 'diagrammatic', simplified version of the general model-based representation described in the previous section. The two planners were developed using the same programming environment and language (Java), and were run on the same machine. We tested the planners on the same problems for the five chosen domains (BW, Gripper, Miconic, Eight-puzzle and Briefcase). In carrying out these

Table 1: Time taken (in sec.) by PP and PMP for solving Blocksworld problems with four, five and six blocks.

	BW-n-problem instance (n = no. of blocks)			
Planner	BW-4-0	BW-4-1	BW-5-0	BW-6-0
PP	0.52	1.50	7.43	144.42
PMP	0.08	0.44	0.34	1.60
PMP(b)	0.19	0.54	2.13	36.23

experiments we hoped to demonstrate that: (1) PMP actually produced correct solutions; (2) its performances were at least 'comparable' with those of PP. The very first results that we obtained were encouraging: in the four BW problem instances used, PMP was not only producing correct solutions, but also performing much better than PP, with a speed-up factor varying from more than three to as much as *ninety* times faster (see the first two rows of Table 1).

We reasoned that such difference in performance had to do with the fact that the specific representation of BW in PMP is implicitly *constrained*. For example, in PP the space available on the table is not limited; hence, at any point of the search, any block on top of another can be put on the table. By contrast, in PMP the problem representation is limited to *three* stacks.[9] Hence, many of the moves which are possible in the standard propositional representation are no longer available in PMP, which can avoid exploring them. This significantly prunes the number of possible paths in the search space. In order to evaluate the performance of PMP without this 'inherent' advantage, we added to the description of the problems a number of redundant empty stacks, so that the total number of stacks equalled the total number of blocks. This 'evened up' the number of possible paths in the two representations, as the space available on the table in PMP was now sufficient to allow the same number of moves as in the propositional representation. We repeated the same experiments, adding five extra problems to the set of tests (all taken from the benchmark problems used in the AIPS'00 International Planning Competition). The new results for PMP on the original four experiments are reported in the last row of Table 1 (labelled 'PMP*(b)*'), while the results for the complete set of instances are plotted in Figure 5.

As expected, the presence of the redundant stacks caused a loss in performance, evident from the comparisons of the PMP and PMP*(b)* data. However, PMP still took significantly less than PP to find the same (shortest) plan in *all* of the problems. Figure 5 shows that the increasing difficulty of the problems produces (exponentially) bigger differences in performance in favour of PMP. One possible explanation for this speed-up may be that the problem representation in PMP is still more constrained than the propositional one. In other words, many of the search paths that are considered in PP may be redundant or non-feasible, and are completely ignored in PMP. For example, the ground action-schemata in PP may contain duplicate or unadmissable instances (e.g., move a block on top of itself) that are *implicitly* avoided by PMP's object-based representation.

[9] The problems had been chosen so that three stacks were sufficient for finding the same optimal solution found by PP.

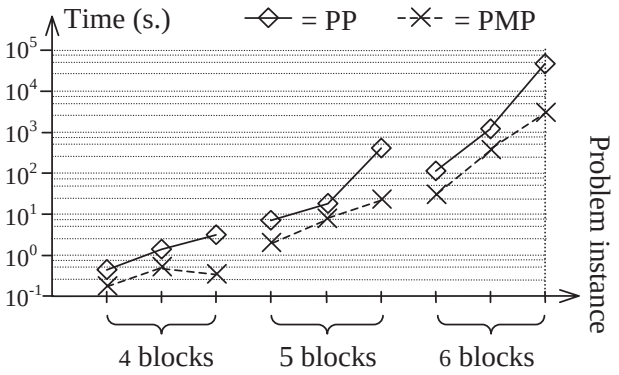


Figure 5: Blocksworld: results for PP and PMP planners.

In order to further investigate the effectiveness of the representation, we carried out experiments on four other domains – namely, Gripper, Miconic, Briefcase and Eight-puzzle. The results are plotted in Figures 6, 7, 8 and 9, respectively. In Figures 6 and 8 the problems have been ordered by increasing PP-time; notice that the *sum* of the two numbers identifying each problem increases monotonically.

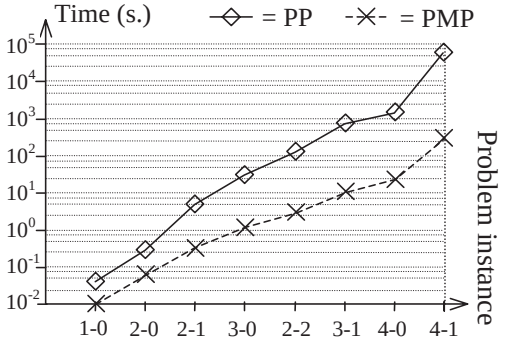


Figure 6: Gripper domain. Problem instance 'm-n' consists of two rooms, containing respectively m and n balls.

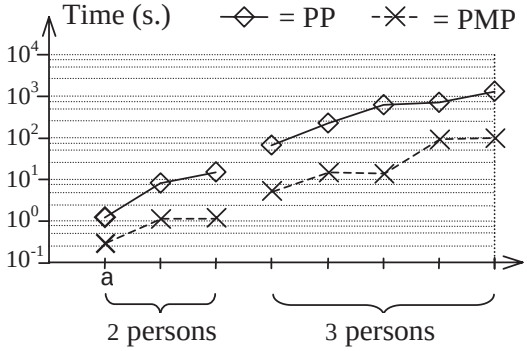


Figure 7: Results for the Miconic domain. All problems contain four floors (except for problem *(a)*, which has three).

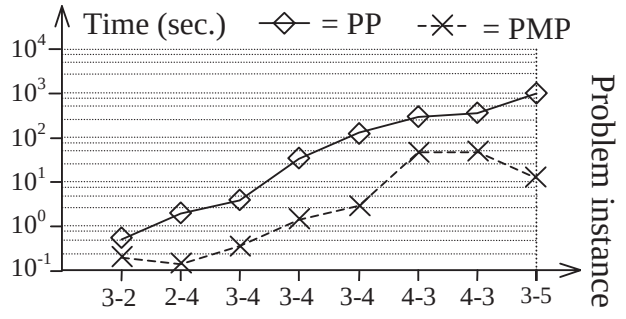


Figure 8: Briefcase domain. Problem instance 'm-n' contains one briefcase, m locations and n portable objects.

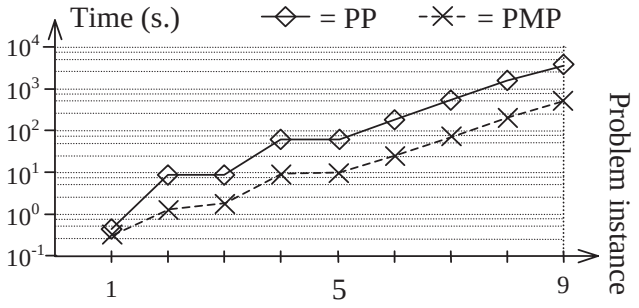


Figure 9: Eight-puzzle domain.

Being confined to a small set of domains not *randomly* chosen from the entire population, these results cannot be assumed to form a representative sample from which general conclusions can be drawn. However, albeit preliminary, they demonstrate the effectiveness of the proposed approach, and give a clear indication of the performance improvements that the new representation can yield (with respect to a standard propositional representation) in the context of *move* problems.

Related work

One of the few recent works adopting a model-based view to planning is that of Giunchiglia and Traverso (Giunchiglia & Traverso 1999), who treat planning as a model-checking problem. Their approach is more abstract and general than the one proposed here: the representation is not based on a model structurally isomorphic to the world, but on a graph (FSM) representing the possible states in which the world can be and the possible transitions between them. Although more general, this representation fails to *implicitly* capture the spatial and topological constraints of the domain.

In the work of Cesta and Oddi (Cesta & Oddi 1996), planning is seen as the problem of deciding the behavior of a dynamic domain described as a set of state variables subject to constraints. This view can be seen as complementary to the approach taken in this work, as it is based on the idea of representing a domain as a set of possible state *changes*, instead of possible object *moves*. However, once again, the representation is not capable to *implicitly* embody the topological and structural constraints of the problem.

The object-centred domain-description language (OCL) developed by Liu and McCluskey (Liu & McCluskey 2000) allows modelling a domain as a set of *objects* subject to various constraints. The idea of an object-based representation is, in many respects, analogous to that of a model-based approach. However, OCL does not allow the specification of the spatial features of the domain; hence, some of the constraints which are implicit in the proposed model still have to be explicitly declared in OCL (e.g., the fact that if an agent holds an object, the location of the object must coincide with that of the agent).

The work of Long and Fox (Long & Fox 2000) on the abstract structure of domains and generic types is also relevant in this context. Long and Fox have developed domain analysis techniques which allow the automatic detection and exploitation of generic types of objects (such as mobiles and portables) in a domain, given its propositional description. In essence, the *move* problems considered here can be seen as generalizing and combining two of the abstract classes identified by Long and Fox (namely, those of 'transportation' and 'construction' domains).

An earlier example of work integrating model-based and propositional representations is the hybrid problem-solving system of Myers and Konolige (Myers & Konolige 1992). Myers and Konolige proposed a formal framework for combining analogical and deductive reasoning. The system they implemented could reason about 'diagrammatic' structures isomorphic to the current world state. However, the system did not allow the model to undergo any *change* during the reasoning process, and, hence, could not be used to solve *planning* problems. In addition, the efficiency of the representation was not evaluated against that of a propositional language through a comparative study, as it has been done here. The latter approach and other related ones fall within the area of 'diagrammatic reasoning' (Glasgow, Narayanan, & Chandrasekaran 1995), and are particularly relevant to the simplified, diagrammatic version of the representation which was used in the experiments.

Discussion

In this paper we have introduced and assessed a new planning representation, able to capture more efficiently the basic *topological* (containment relationships between places and objects, or objects and objects) and *structural* (spatial relationships between places, or internal structure of a place) properties of a domain. In addition, we have described the syntax of a specific domain-description language, representing a simplified version of the general model proposed. The performance of a prototype system (PMP) on a subset of five domains has been compared with that of an equivalent propositional planner (PP), adopting an *identical* planning algorithm but using a STRIPS domain-description language. The results show that PMP is superior to PP on *all* of the chosen instances of problems. We believe that the speed-up of the PMP planner is the result of the ability of the new representation to (1) capture naturally and *implicitly* some of the basic, common-sense physical constraints of the domain,

and to (2) *organize* in appropriate structures the relevant entities of the domain, allowing more efficient reasoning about the object-rearrangement aspects of the problem.

The adoption of a simpler, model-based, spatially-structured representation that allows effective commonsense reasoning also enables new reasoning modes, which can support easier and more efficient *learning*, *heuristic-extraction* and *abstraction* techniques. For example, it is easy to see that the graphical description of the action schemata shown in Figures 3 and 4 could be *inferred* automatically using machine-learning and image-processing techniques. In addition, the use of edges as node-to-place connections can easily support abstraction: a group of objects contained in the same place (or attached to the same object) can be represented as a *single* entity, allowing the system to abstract away from the details of the parts and enabling abstraction to take place at different levels.

The use of a model which replicates the spatial and topological aspects of the real world also provides the planner with an implicit guidance on the ordering of subgoals. Consider, for example, the 'Sussman anomaly' problem in BW, presented earlier. A propositional description of the problem does not provide *a priori* information on the order in which the two subgoals {`(On A B)`,`(On B C)`} must be achieved. By contrast, in the proposed representation, the goal-pattern G = `stack`{`C B A`} allows regressing a new goal G' in which 'A' has been picked up from the top of 'B' (by reversing step *Put-on*(A,B)), but does *not* permit the regression of a situation in which 'B' has been removed from the top of 'C' (via step *Put-on*(B,C)). In fact, consider the action schema *Put-on*(x,y): its effects include leaving block x 'clear'. While goal G makes no requirements on the content of the cell to the immediate right (read 'top') of 'A' (which may thus be assumed empty), the cell to the right of 'B' is currently occupied by 'A'. This prevents x from being bound to 'B'. Hence, a backchaining planner would be (correctly) forced to begin with addressing subgoal `(On A B)` – i.e., to execute *Put-on*(A,B) as final step.

Finally, a domain-modelling language (such as the one described earlier) based on the proposed representation has a straightforward graphical interpretation, and, hence, should result more intuitive and easier to use for the non-experts; this, together with better performances, is expected to facilitate the take-up of AI planning technology and its wider application to the real world.

The proposed representation, however, is also limited in several ways. Perhaps its most obvious weakness is the fact that it does not offer any means for describing the non spatial or topological aspects of a domain (such as time or metric quantities), which, on the other hand, could be modelled using a propositional language. However, the approach described here should be seen as representing the extreme of a *continuum* of possible domain-modelling languages, in which propositional and analogical aspects of a domain can be present in different degrees. One of the future directions of this work consists of extending the representation to allow the use of propositional expressions. A more expressive version, for example, could allow nodes to consist of complex objects, having an internal (static or dynamic) structure and properties. In such an extended model, the action preconditions could require the *attributes* of the specified objects (such as size, color, shape, etc.) to satisfy specific (qualitative or quantitative) constraints.

In conclusion, the results of this investigation demonstrate the potential benefits of the proposed representation, and motivate further work on model-based planning formalisms and on their use in conjunction with current propositional paradigms. This paper lays the foundations for future developments in this direction.

References

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

Cesta, A., and Oddi, A. 1996. DDL.1: A formal description of a constraint representation language for physical domains. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*. IOS Press (Amsterdam). 341–352. (Proceedings of the 3rd European Workshop on Planning (EWSP95), Assisi, Italy, September 1995).

Fox, M., and Long, D. P. 2001. STAN4: A Hybrid Planning Strategy Based on Sub-problem Abstraction. *AI Magazine* 22(4).

Fox, M., and Long, D. 2002. Extending the exploitation of symmetry analysis in planning. In *Proceedings of the 5th International Conference on AI Planning and Scheduling*. Toulouse, France: AAAI press.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research – Special issue on the 3rd International Planning Competition*. (forthcoming).

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

Giunchiglia, F., and Traverso, P. 1999. Planning as model checking. In Biundo, S., and Fox, M., eds., *Proc. of the 5th European Conference on Planning (ECP-99)*, 1–20.

Glasgow, J.; Narayanan, N.; and Chandrasekaran, B., eds. 1995. *Diagrammatic Reasoning*. Cambridge, MA: MIT Press.

Halpern, J. Y., and Vardi, M. Y. 1991. Model Checking vs. Theorem Proving: A Manifesto. In Allen, J. A.; Fikes, R.; and Sandewall, E., eds., *Principles of Knoweldge representation and Reasoning: Proceedings of the 2nd International Conference (KR91)*, 325–332.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proceedings of the 5th International Conference of AI Planning Systems (AIPS 2000)*, 140–149. Breckenridge, Colorado: 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.

Kautz, H., and Selman, B. 1998. The role of domain-specific knowledge in the planning as satisfiability frame-

work. In *Proceedings of the 1998 International Conference on AI Planning Systems (AIPS-98)*, 181–189. Pittsburgh, PA: AAAI Press, Menlo Park, CA.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and Graph-based planning. In *Proceedings of the 16th International Joint Conference on AI (IJCAI-99)*. Stockholm, Sweden: Morgan Kaufmann.

Liu, D., and McCluskey, T. 2000. The OCL Language Manual, Version 1.2. Technical report, Department of Computing and Mathematical Sciences, University of Huddersfield (UK).

Long, D., and Fox, M. 2000. Automatic synthesis and use of generic types in planning. In *Proceedings of the 5th International Conference on AI Planning and Scheduling Systems (AIPS'00)*, 196–205. Breckenridge, CO: AAAI Press.

McDermott, D.; Knoblock, C.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – the planning domain definition language. Version 1.7. Technical report, Department of Computer Science, Yale University (CT). (available at www.cs.yale.edu/homes/dvm/).

Myers, K., and Konolige, K. 1992. Reasoning with analogical representations. In Nebel, B.; Rich, C.; and Swartout, W., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR92)*, 189–200. Morgan Kaufmann Publishers Inc., San Mateo, CA.

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on AI (IJCAI-99)*, 968–973. Stockholm, Sweden: Morgan Kaufmann.

Wilkins, D., and desJardins, M. 2001. A call for knowledge-based planning. *AI Magazine* 22(1):99–115.

Wilkins, D. 1997. That's something I could not allow to happen: HAL and Planning. In Stork, D., ed., *HAL's Legacy: 2001's computers as dream and reality*. Cambridge, MA: MIT Press. chapter 14, 305–331.

Appendix A

A.1 – Miconic domain description

```
(define (domain miconic)
  (:ObjectTypes person lift)
  (:PlaceTypes building {object::2}
               lift {person})

  (:action board
     :parameters (P - person L - lift)
     :pre (building{↔ L P} lift{ _ })
     :post (building{ L _ } lift{ P })
  )

  (:action depart
     :parameters (P - person L - lift)
     :pre (building{↔ L _ } lift{ P })
     :post (building{ L P } lift{ _ })
  )

  (:action move-up
     :parameters (L - lift)
     :pre (building{/ L _ })
     :post (building{/ _ L})
  )

  (:action move-down
     :parameters (L - lift)
     :pre (building{/ _ L })
     :post (building{/ L _ })
  )
)
```

A.2 – Miconic problem instance

```
(define (problem mic-01)
   (:domain miconic)

   (:Objects A B C D E - person
             lf - lift)
   (:Places bd - building lf - lift)
   (:init bd [ [_  A B _ ]
               [lf D _ _ ]
               [_  _  C _ ] ]
          lf { _ _ }
   )
   (:goal bd [ [lf A _ _ ]
               [_  D _ C ]
               [_  B _ _ ] ]
   )
)
```

A Domain Description Language for Data Processing

Keith Golden
NASA Ames Research Center
MS 269-2
Moffett Field, CA 94035
kgolden@email.arc.nasa.gov

Abstract

We discuss an application of planning to data processing, a planning problem which poses unique challenges for domain description languages. We discuss these challenges and why the current PDDL standard does not meet them. We discuss DPADL (Data Processing Action Description Language), a language for describing planning domains that involve data processing. DPADL is a declarative, object-oriented language that supports constraints and embedded Java code, object creation and copying, explicit inputs and outputs for actions, and metadata descriptions of existing and desired data. DPADL is supported by the IMAGEbot system, which we are using to provide automation for an ecological forecasting application. We compare DPADL to PDDL and discuss changes that could be made to PDDL to make it more suitable for representing planning domains that involve data processing actions.

1 Introduction

Earth-observing satellites return terabytes of data per day, providing global daily coverage across multiple spectral bands at a variety of resolutions. These observations can be used in countless ways: to monitor changes in Earth's climate, assess the health of forests and farms, and track critical short-term events, such as severe storms. However, doing all this in a timely manner is a significant challenge, which will require greater levels of automation. To go from raw "level 0" satellite data to high-level observations or predictions such as "decreased vegetation growth" or "high fire risk" requires many data-processing steps, from filtering out noise to running simulations. There are often many data sources to choose from, and many ways to process the data to produce the desired data product. These choices involve tradeoffs along many dimensions, including data quality, temporal and spatial resolution and coverage, timeliness, CPU usage, storage and bandwidth.

We use planning technology to automate this data processing. We represent data-processing operations as planner actions, descriptions of desired data products as planner goals, and use a planner to generate data-flow programs that output the requested data. We are working with Earth scientists to provide planner-based automation to an ecological forecasting system called the Terrestrial Observation and Prediction System, or TOPS (Nemani *et al.* 2002) (http://www.forestry.umt.edu/ntsg/Projects/TOPS/). We have developed a planner-based softbot (**soft**ware ro**bot**), called IMAGEbot, to generate and execute data-flow programs (plans) in response to data requests. The data-processing operations supported by IMAGEbot include image processing, text processing, managing file archives and running scientific models. Some aspects of the planner are described in (Golden & Frank 2002).

In the course of developing IMAGEbot, we considered available domain description languages, especially PDDL, for representing data processing actions, but found them unsuitable. We discuss the features of these domains that are problematic for PDDL and the changes to the PDDL that would be needed to handle them. To deal with these issues, we developed a new language called DPADL, for Data Processing Action Description Language. We considered basing our language on PDDL, which is attractive in that it has become a standard for much of the planning community, but decided instead to base the syntax on a different widely-used language: Java. This decision was driven by practical considerations, such as our desire for the language to be usable by software developers, the appropriateness of an object-oriented language to describe the complex data structures that arise in data-processing domains, and the fact that Java is the language that both TOPS and IMAGEbot are written in, to name a few.

In the remainder of the paper, we discuss language features relevant to representing data-processing domains, how those features are implemented in PDADL, and what issues there would be in including those features in PDDL:

- First-class objects (Section 2): Data files often have complex data structures. The language should provide the vocabulary for describing these structures.
- Constraints (Section 4): Determining the appropriate parameters for an action can be more difficult than determining which action schemas belong in the plan. Parameter values can depend on other actions or objects in the plan. The language should provide the ability to specify such constraints where they are needed.
- Integration with a run-time environment (Section 4): Sensing and acting in a complex software environment requires "hooks" into that environment, both to obtain information and to initiate operations.
- Metadata goals (Section 5) and inputs (Section 7): The inputs and outputs of data-processing plans are data, so

the language should be expressive enough to describe requested and available data . Since a data file contains information about past states of the world, metadata should be able to describe how the content of the data depends on the past state of the world.

- Object creation and copying (Section 6.3): Many programs create new objects, such as files, sometimes by copying or modifying other objects. The language must provide a way of describing such operations.
- Data-flow plans (Section 8): Since the purpose of plans is to process data, they should take the form of data-flow programs, in which outputs of one action are fed into inputs of another.

At the end of each section, we present a BNF grammar covering the language elements described in that section. For example, the top-level production rule for a domain description is:

DOMAIN ::=	(TYPE | FUNCTION | ACTION | GOAL | STATE | <INLINE_CODE>)+ <EOF>

where symbols in SMALL CAPS are non-terminals, symbols in <ANGLE_BRACKETS> are terminals, and keywords are underlined.

2 First-class objects

Data files (and other entities in a software environment) typically have a complex, hierarchical structure, which can be described in terms of object composition. Representing these data structures explicitly as first-class citizens not only makes domains simpler to encode and understand, but provides valuable information to the planner. Thus, we decided that DPADL should be an object-oriented language. Although the identification of objects and object attributes is important, an object-oriented syntax, such as our Java-based syntax, is less so; the same information could be expressed in a PDDL-style relational syntax, just not as concisely.

DPADL allows the definition of new types corresponding either to structures (objects) or primitive types, such as integers or strings. The keyword for introducing a new type declaration is **`type`**. Here and elsewhere in the paper, DPADL text is rendered in `typewriter` font, and keywords are **`bold`**. We use ellipses (...) to indicate that text has been omitted for the sake of brevity. For example,

```
static type Filename extends String
```

introduces a new type, `Filename`, which is a subtype of `String`, a predefined type. The predefined types are `int`, `unsigned`, `float`, `String`, `Object` and `boolean`. The keyword **`static`** means that no instance of `Filename`, once created, can ever be changed.[1] A type that is not static is **`fluent`**.

Subtypes of `Object` may be used to represent Java objects. For example,

```
static type Tile extends Object
            mapsto tops.modis.Tile
```

means that the type `Tile` corresponds to the Java class `tops.modis.Tile`. As we discuss in Sections 4 and 6.4, the agent can manipulate Java objects in the course of constraint reasoning or action execution by executing in-lined Java code.

Alternatively, when there is a small number of instances of a type, we can define it by listing all possible instances. This is similar to enumerated types in C/C++, but without the restriction to integral values.[2] Listing values in this way is useful for constraint reasoning, since the domain of a variable corresponding to such a type can be initialized with the set of possible values.

```
static type ImageFormat =
{"JPG", "GIF", "TIFF", "PNG", "XCF", ...};
```

As in C/C++, enumerated values can have symbolic names attached to them.

```
static type ProjectionType =
{LAZEA=11, GOODE_HOMOL=24, ROBINSON=21, ...};
```

Like classes in C++ and Java, types can have attributes. For example, file attributes include pathname and owner:

```
type File extends Object {
  static key Path pathname;
  User owner;
  ...
}
```

The keyword **`key`** is used to indicate that `pathname` is a unique identifier for a file, so two files that have the same pathname must in fact be the same file. This is not correct if we access files on multiple machines, in which case we should use the host machine as an additional key.

In addition to the subtype relation, designated using **`extends`**, we can specify that one type **`implements`** another, meaning it inherits all the attributes of the other type but is not an instance of that type.[3] This is useful in cases where two objects share the same structure but cannot be used interchangeably. For example, a file archive, such as a tar file, contains records that reflect all the properties and contents of individual files, but are not themselves files. We say that `TarFile.Record` **`implements`** `File`. This is especially useful when used in conjunction with **`copyof`** (Section 6.3), since a record in a tar file can be a copy of a file, or vice versa.

When referring to an attribute of an object, we use a Java-like syntax. For example, `f.filename` refers to the `filename` attribute of the object represented by the variable `f`. Attributes can take arguments. For example, `pic.pixelValue(x,y)` refers to the value of the pixel at the x,y coordinates of the image `pic`. Although the syntax resembles that of Java method calls, `pixelValue(x,y)` is simply a parameterized attribute, and can be used in exactly the same contexts. For example,

[1] This is a departure from the Java meaning of `static`.

[2] In PDDL, all types other than numbers are effectively enumerated types, since all objects of each type must be explicitly declared. Since the Closed-World Assumption is not at all reasonable for data processing domains, DPADL does not impose this requirement.

[3] This is a departure from the Java meaning of `implements`.

```
pic2.pixelValue(x,y) = pic1.pixelValue(y, x+5);
```

describes an effect that transposes an image to the left by 5 pixels.

The object-oriented notation is convenient, but not essential. Any object description can be translated into an equivalent relational form by translating each attribute description into a relation in which the first argument is a reference to the object, the second argument is the value of the attribute, and the remaining arguments are the arguments of the attribute. From the example above, we would define a PDDL relation (pixelValueR ?image ?pvalue ?x, ?y). Additionally, the action descriptions or domain axioms would need to be modified to enforce the fact that

- Two objects are equal if and only if their key attributes are equal and
- An attribute can have only one value, so (pixelValue IMAGE-56, BLACK, 10, 10) is mutually exclusive with (pixelValue IMAGE-56, ?v, 10, 10), for all ?v $\neq$ BLACK.

Alternatively, we could provide additional syntax to convey the same information while maintaining a relational representation, as was done in the SADL language (Golden & Weld 1996).

Explicitly identifying objects is not just useful to the domain developer, but also to the planner. For example, the planner can reduce search by exploiting the fact that attributes of static objects don't change once the object is created. Additionally, Section 6.3 discusses the role attributes play when objects are copied.

TYPE	::=	(static | fluent)? type ((<IDENTIFIER> **=** **{** MEMBERS **}**) | (TYPESPEC)) (TYPEBODY | **;**)
MEMBERS	::=	((<IDENTIFIER> **=**)? LITERAL) (**,** MEMBERS)?
TYPESPEC	::=	PRIMTYPE | (<IDENTIFIER> extends TYPENAME (implements TYPENAME)* (mapsto <CLASSNAME>)?
TYPEBODY	::=	**{** (MEMDEF | CTRSPEC | TYPE)* **}**
MEMDEF	::=	(static | fluent)? key? TYPENAME <IDENTIFIER> (PARAMS)? (MEMBODY | **;**)
MEMBODY	::=	**{** (CTRSPEC)* **}**
PRIMTYPE	::=	int | unsigned | float | String | Object | boolean
TYPENAME	::=	<IDENTIFIER> | PRIMTYPE
QUALTYPE	::=	TYPENAME (**.** <IDENTIFIER>)*

3 Functions and relations

The object-attribute notation is just a special case of a functional notation, which DPADL also supports. Functions, like types, may be static or fluent. The value of a fluent function changes over time, whereas the value of a static function does not. For example,

```
fluent float temp(float lon, float lat);
```

declares a function that takes two real values, representing longitude and latitude, and returns a real value representing the temperature at that location. Functions, like attributes, may have zero arguments, in which case the parentheses are omitted. For example,

```
fluent Date currentDate;
```

specifies that `currentDate` is a fluent function taking no arguments.

Functions over objects have been mentioned as a possible future extension to PDDL (Fox & Long 2003). While that would make it easier to describe data-processing domains in PDDL, we should note that functions in DPADL are merely a notational convenience; they allow us to avoid explicitly stating the mutual exclusions to specify that, for example, a file can have only one size, but semantically they are no different from relations in which one of the arguments is restricted to a single value. In particular, they do not play the same role that functions play in first-order predicate logic. DPADL does not support domain axioms, which could be used to generate an arbitrary number of object references through repeated function composition.

To indicate that a function is undefined for particular arguments, we use the keyword **`null`** to represent invalid values. The type of **`null`** is a subtype of all types, but **`null`** will not match any value except itself.

FUNCTION	::=	(static | fluent) TYPENAME (<IDENTIFIER> | <OPERATOR>) (**(** PARAMS? **)**)? (**;** | **{** (CTRSPEC)* **}**)
PARAMS	::=	(PARAMDEF (**,** PARAMS)?) | :rest PARAMDEF
PARAMDEF	::=	QUALTYPE <IDENTIFIER>

4 Constraints

In data-processing domains, we need to be able to express thresholds, intervals over space or time, mathematical functions, and more complex calculations. In DPADL, these are all represented using constraints. PDDL supports numeric functions, which are used to specify how quantities change over time. PDADL constraints can serve the same role, but are more flexible; they can perform arbitrary calculations or sense information from the environment. However, they are also more limited than PDDL functions, in that they cannot represent and reason about quantities that change continuously over time, such as fuel (Fox & Long 2003).

Formally speaking, a constraint is simply a relation that holds over a set of variables, so we could view any functions, object attributes or types as constraints. However, thus far, we have only shown how to *declare* functions, attributes and types, not (with the exception of enumerated types) how to *define* them. To reason about constraints, we need definitions, not just declarations. For example, consider the following declaration.

```
float foo(float x);
```

Given the value of `x`, we know there must be some value `y = foo(x)`, but we have provided no way to determine what that value is. Viewing `foo` as a constraint is valid but pointless.

We provide two alternative ways of specifying the definition of a constraint; it may be selected from a library of predefined constraint definitions or defined in terms of arbitrary Java code embedded in the type and function declarations. The constraint reasoning system supports constraints over all primitive types as well as Java objects. It can also handle constraints involving universal quantification, as discussed in (Golden & Frank 2002).

Constraint definitions can only be given for statics. Any function defined as a constraint must be determined only by that constraint; no action may affect it. This restriction provides a clear division of labor between causal reasoning and constraint reasoning.

4.1 Type constraints

Formally, a type is a unary relation that is true for all instances of the type and false for all non-instances. But in the type declarations of Section 2, we did not define what those relations were. It is fine to say `Filename` **`extends`** `String`, but given a String, how do we know if it is a valid filename?

One possibility might be to define `Filename` as an enumerated type; that is, we list all valid filenames. The obvious problem with this is that there are, for all practical purposes, infinitely many of them. A better option is to specify a regular expression that concisely specifies all valid filenames:

```
static type Filename extends String {
   constraint Matches(true, this, "~[/]+");
}
```

means that filenames must contain at least one character, and they cannot contain the character '/'. In Unix, this is, in fact, the only practical limitation on filenames. `Matches` is a constraint from the constraint library requiring a string to a match a regular expression. All string constraints are actually defined in terms of operations on regular expressions, so Matches is, in a sense, the simplest. The keyword **`this`** designates an instance of the type being defined, in this case a filename.

Constraints can also be defined in terms of inlined Java code, as discussed in the next section.

4.2 Attribute constraints

We can define attributes as constraints as well. One reason for doing this is to support *procedural attachment*: specifying program code that provides the definition of the attribute. For example, if we have a DPADL object that corresponds to a Java object, we must specify what methods to call on the Java object to determine the values of the attributes as declared in DPADL:

```
static type Tile extends Object
            mapsto tops.modis.Tile {
  key String uniqueId {
    constraint {
      value(this) = $this.getUID()$;
      this(value) = $Tile.findTile(value)$;
    }
...
```

The attribute `uniqueId` is declared as a **`key`** of a (mosaic) `Tile`, meaning there is a one-to-one mapping between tiles and their unique identifiers. Given a tile, we should be able to obtain its unique identifier, and given a unique identifier, we should be able to obtain the corresponding tile. The embedded Java code provides instructions for performing these mappings. The `uniqueId` attribute of a `Tile` can be determined by calling the `getUID` method on the `Tile`, and a `Tile` object corresponding to a given `uniqueId` can be determined by calling the method `findTile`, with the `uniqueId` as an argument. The text preceding the "=" is a "signature" specifying the return value and parameters of the following Java code. The keyword **`value`** refers to the value of the attribute being defined, in this case `uniqueId`. The keyword **`this`** refers to an object of the type being defined, in this case `Tile`. Thus, **`value`**`(`**`this`**`)` means that given an object of type `Tile`, we can obtain the value of the `uniqueId` attribute by executing the following Java code (delimited by `$`). Conversely, **`this`**`(`**`value`**`)` means that given a `uniqueId`, we can find the corresponding `Tile`.

The above constraint will only be enforced if there is a singleton domain for some tile or ID variable. It is also possible to define constraints that work for non-singleton domains, by indicating that an argument or return value represents an interval (delimited by `[]`) or a finite set (delimited by `{}`). For example, one attribute of a `Tile` is that it `covers` a given longitude, latitude. Given a particular longitude and latitude, the constraint solver can invoke a method to find a single tile that covers it, but it can do even better. Given a rectangular region, represented by intervals of longitude and latitude, it can invoke a method to find a set of tiles covering that region.

```
boolean covers(float lon, float lat) {
  constraint {

    ...
    // returns the set of tiles covering
    // a given lon/lat range.
    {this}([lon], [lat], d=day, y=year,
           p=product, value)
      = {$ if(value)
             return tm.getTiles(lon.max,
                                lat.min,
                                lon.min,
                                lat.max,
                                d, y, p);
           else return null; $};
  }
}
```

In this example, the signature is more complicated. `{`**`this`**`}(...)` means that the return value of the Java code is a set (specified by `{...}`) of Tiles (specified by **`this`**). The first two arguments, lon and lat, are surrounded by `[...]`, indicating that the variable domains should be intervals. The next three arguments, d, y, and p are defined as being equal to the Tile attributes day, year and product, not shown in this example. Finally, **`value`** is the boolean value of the `covers` relation, true if and only if the tile covers the specified lon/lat.

The Java code is also more complex. Unlike the previous example, it has a conditional and an explicit return call. If **value** is true, then it returns the result of the method getTiles. Since lon and lat are intervals, we refer to their maximum and minimum values to specify the bounding box of interest. If **value** is false, it returns **null**, meaning the set of tiles could not be determined, since there is no method for returning the tiles outside of a bounding box.

4.3 Function constraints

Functions, like attributes, can have constraints associated with them, the only difference being that the constraints cannot reference the keyword **this**, because there is no object to reference. Infix mathematical operators are also functions, and they can be defined for any type, using a syntax similar to that used for C++ operator overloading. For example to specify that the "+" operator can be used to concatenate strings, as in Java, we can write

```
static String operator+ (String s1,
                         String s2) {
  constraint Concat(value, s1, s2);
}
```

where Concat is a constraint from the constraint library, specifying that the first argument is the concatenation of the remaining arguments.

4.4 Restrictions

Minimum requirements on the inlined code used to define constraints are:

1. The code may not do anything other than calculate the domain of a variable and return it. That is, it may not have any side-effects.
2. If the code is called multiple times with the same arguments, it will always return the same calculated domain. This requirement precludes using constraints to represent values that change during the course of planning.
3. If the domains corresponding to one or more of the arguments is reduced, then the calculated domain will be a subset of the original domain.

If these requirements are not met, then the results are undefined. With them, we can view each constraint as some unknown relation and the procedures as sensors that provide limited information about the extension of the relation.

```
CTRSPEC  ::= constraint ( <IDENTIFIER>
             ( ARGS ) _;_ ) | { ( JAVACTR )+ } )
JAVACTR  ::= ( CTRARG CTRARGS _=_
             <INLINE_CODE> _;_ )
             | ( [ CTRARG ] CTRARGS _=_
             [ <INLINE_CODE> _,_
             <INLINE_CODE> ] ; )
             | ( { CTRARG } CTRARGS _=_
             { <INLINE_CODE> } _;_ )
CTRARG   ::= <IDENTIFIER> | value | this
CTRARG2  ::= (CTRARG | [ CTRARG ] | { CTRARG })
             ( _=_ ADDITIVE )?
CTRARGS  ::= ( CTRARG2 [ _,_ CTRARGS ] )
```

5 Goals

Goals are used primarily to describe data products that the system should produce. Data product descriptions should specify at least the following:

- Data semantics: the information represented by the data. That is, what facts about the world can be inferred from the data contents.
- Data syntax: how the information is coded in the data. For example, what pixel values in an image are used to represent the information.
- Time: what time the information pertains to. For example, we need to be able to distinguish between rainfall last week and rainfall last year.

Time is an optional argument of all fluents. The mapping between semantics and syntax is specified using the keyword **when**. For example, to request a file that contains gridded temperature values over a particular region, using the LAZEA projection and a particular mapping (tempEncoding) from temperatures to pixel values, we could write:

```
forall int x, int y, float lon, float lat,
       float t;
when (tempEncoding(temperature(lon, lat)) == t
     && proj(LAZEA, x, y, lon, lat)
     && 0 <= x < MAXX && 0 <= y < MAXY) {
  file.pixelValue(x, y) == t;
}
```

We will call the expression inside the parentheses following the keyword **when** the left-hand side (LHS) of the goal, and we will call the expression in the braces the right-hand side (RHS).

A key aspect of DPADL is that all data descriptions are purely causal; we describe how the data content of the file causally depends on the (earlier) state of the world. An advantage of this representation is that standard temporal projection techniques can be used to to determine how a succession of data-processing operations affect the data content of the final output.

A **when** condition describes an implication, but an implication between conditions that hold at two different times. The LHS implicitly refers to the time the goal is posted (unless an earlier time is specified), and the RHS refers to the final state (whenever the goal achieved). Because the agent cannot change the past, the only way to achieve the goal is to make sure the RHS is satisfied, subject to the conditions given by the LHS.

More formally, a **when** goal can be be described as follows. Let s_0 be the initial state and let Σ be the set of states indistinguishable from s_0 consistent with the agent's knowledge in the initial state (i.e., the set of all possible worlds). Let π be a plan consisting of a sequence of actions, and let $do(\pi, s)$ be the state reached from s by executing π. The goal **when**$(\Phi(\vec{x}))$ $\{\Psi(\vec{x})\}$ is achieved by π if $\forall\vec{x}\forall s \in \Sigma((s \models \Phi(\vec{x})) \Rightarrow (do(\pi, s) \models \Psi(\vec{x})))$.

The only formal difference between conditions in the LHS and conditions in the RHS is the time that they refer to, but this provides a sufficient foundation for describing data

goals, since the important characteristic of data is that it stores information about the past. Thus, we use temporal goals to describe how the past information of the world determines the future content of the data:

- The semantics of the desired data (e.g., temperature) is specified in terms of fluents in the LHS of the goal, because it concerns properties of the world that hold when the goal is specified (or earlier), properties that are not affected by the agent in pure data-processing domains.
- The data syntax (e.g., pixelValue) is specified in terms of static predicates in the RHS, because it concerns properties of data that may not exist at the time the goal is given, properties that must be affected by the agent to produce the requested data. Optionally, predicates describing syntax could also appear on the LHS, to represent goals of converting file formats, etc. For example, we might specify a goal of making all the red pixels in an image blue: **when**`(input.color == RED) {output.color == BLUE;}`
- Constraints (e.g., `0 <= x < MAXX`) are specified in the LHS of the goal because, being static, they must hold in the initial state and cannot be affected by the agent. Since variables involved in constraints can appear in the RHS, this is not a practical limitation.

To use these conventions in PDDL, we would need to extend PDDL to specify goals that refer to an earlier state of the world in addition to the final state. PDDL 2.1 can refer to time, but only the start and end times of actions. It would also be necessary to relax the CWA, if these domains are to be remotely interesting.

```
GOAL      ::= goal <IDENTIFIER> ( PARAMS? ) {
              ( (output | forall | exists) PARAMS
              ; )* OREXP }
OREXP     ::= CONDEXP+ ( || CONDEXP+ )*
CONDEXP   ::= ( when ( ANDEXP ) { CONDEXP* }
              ( else { CONDEXP* } )? ) | EQUAL ;
ANDEXP    ::= EQUAL ( && ( EQUAL ) )*
EQUAL     ::= RELATION ( ( == | != ) RELATION
              )*
RELATION  ::= ADDITIVE ( ( < | > | <= | >=
              ) ADDITIVE )*
ADDITIVE  ::= MULTIPL ( ( + | - ) MULTIPL )*
MULTIPL   ::= UNARY ( ( * | / | % ) UNARY )*
UNARY     ::= ( + | - | ! )? PRIMEXP
PRIMEXP   ::= ( ANDEXP ) | ( FUNEXP | this )
              ( . FUNEXP )* | LITERAL
FUNEXP    ::= <IDENTIFIER> ( ( ARGS ) )?
LITERAL   ::= <INTEGER_LITERAL>
              | <FLOATING_POINT_LITERAL>
              | <CHARACTER_LITERAL>
              | <STRING_LITERAL> | null
              | true | false
ARGS      ::= ADDITIVE ( , ARGS )?
```

6 Actions

Actions can include data sources (which provide data based on the state of the world) and filters (which provide data based on their inputs), so preconditions and effects describe inputs and outputs as well as the state of the world. Additionally, actions must be executable, so the procedure for executing an action (i.e., Java code) is part of the action description.

```
ACTION    ::= action <IDENTIFIER> ( PARAMS ) {
              (( input | forall) PARAMS ; )
              | (output OUTPUTS ; )
              | PRECOND | EFFECT | EXEC )* }
OUTPUTS   ::= PARAMDEF (copyof <IDENTIFIER>)?
              ( , OUTPUTS)?
```

6.1 Inputs, outputs and parameters

As in PDDL (McDermott 2000), actions are parameterized, and parameters are typed. In addition to ordinary parameters, two kinds of variables are recognized as unique and are treated somewhat differently; namely, inputs and outputs.

Outputs represent objects (e.g., files) generated as a result of executing the action. An output does not exist before the corresponding action is executed, and is always distinct from all other objects.

Inputs represent objects that are required by the action but are not required to exist after the action has been executed. Inputs may come from outputs of other actions or they may be preexisting objects. In the former case, all preconditions describing attributes of a given static input must be supported by the same action, since only one action can have produced the output, and once it is created, no action can change it.

Ordinary parameters are essentially like the parameters passed to method or function calls in C or Java; they refer to primitive values or objects that may exist before the action is executed and may persist afterward.

In addition to parameters, inputs and outputs, actions can refer to universally quantified variables and introduce variables corresponding to new objects with the **new** keyword, discussed in Section 6.3.

To extend PDDL to handle input and output parameters, it would be necessary to allow for object creation (which requires a dynamic universe), and to allow the values of certain variables to be unbound at planning time, provided it can be proven that they will be bound at execution time.

6.2 Preconditions

Preconditions describe the conditions that must be true of the world and of the inputs in order for the action to be executable. Thus, action preconditions need to reference the input variables and the prior world state, but cannot reference the output variables, which describe objects that don't exist in the prior state.

Low-level actions, such as filters, can be described purely in terms of the syntactic properties of the input files. For example, an image-processing operation doesn't care whether the pixels of the input image represent temperatures in Montana or a bowl of fruit. All that matters are the values of the pixels. Thus, the preconditions for these actions should refer only to properties of the data that hold in the prior state. Similarly, simple sensors (data sources) depend only on the immediate state of the world, so their preconditions should

only refer to conditions of the world that hold in the prior state.

However, some high-level actions, such as ecological models, expect their inputs to represent certain information about past states of the world, such as temperature or precipitation, so it is appropriate for the preconditions of these actions to specify the information content of their inputs, not just the structure. The descriptions the information content of these inputs will be in terms of states other than the prior state. For example, an ecological model might require a file containing temperature data from last Tuesday. In other words, preconditions, like goals, can include metadata descriptions, which are described in exactly the same way, using the keyword **when**.

The LHS of a **when** precondition, like the LHS of a goal, refers to past states. The RHS, however, rather than referring to the final state, refers to the start of execution of the corresponding action. Conventions for describing data inputs in preconditions are the same as the conventions for describing goals: The LHS specifies the semantics of the data file and the RHS specifies the syntax. Any constraints must appear in the LHS.

Preconditions are introduced with the keyword **precond**, and introduce a condition, which may be disjunctive.

PRECOND ::= precond OREXP

6.3 Effects

Effects, introduced with the keyword **effect**, are used to describe the outputs generated by an action. Outputs depend on the state of the world (in the case of sensory actions) or the inputs (in the case of filters), so effects need to be able to reference both the prior state and next state and both the input and output variables.

EFFECT ::= effect (WHENEXP)+

Conditional effects Like goals and preconditions, conditional effects are introduced using the keyword **when**, but here the LHS refers to the prior state (and input variables), not the initial state. The RHS describes the next state and output variables, so the combination of the two describes how the output depends on the input (or on the state of the world). This is no different than conditional effects in PDDL.

As with goals, there are conventions for describing data effects.

- data sources are described using conditional effects, in which conditions on the LHS are either constraints or fluents describing the state of the world and conditions on the RHS are statics describing the syntax of the output data.
- Filters are described using conditional effects, in which conditions on the LHS are either constraints or statics describing the syntax of the input data, and conditions on the RHS are statics describing the syntax of the output data.

In order to restrict the language to only describe data-processing domains, we do not allow fluents to appear on the RHS of any effect. This means that actions cannot change the world except by creating objects (e.g., files) that satisfy certain conditions based on the current (or past) state of the world. This restriction could easily be lifted, allowing us to describe arbitrary planning domains, but imposing it allows the use of specialized planning algorithms that take advantage of unique properties of pure data-processing domains. An alternative would be to run a simple preprocessor that checks whether a domain is a data-processing domain and runs a specialized planner if it is.

Every atomic RHS expression involves setting the (possibly boolean) value of a function or attribute or creating a new object. A static attribute can only be set if it is an attribute of a newly created object. Since we restrict predicates on the RHS of effects to static predicates, that means all that actions can do is produce data; they cannot change the world or alter preexisting data.

For example, to describe a threshold action, which sets output pixels to either BLACK or WHITE, depending on whether the corresponding input pixels are below or above a given threshold `thresh`, we can write:

```
action threshold (unsigned thresh) {
  input Image in;
  output Image out copyof in;
  forall unsigned x, unsigned y;
  effect when ((x < in.xSize)
               && (y < in.ySize) {
    when (in.valueAt(x, y) <= thresh) {
        out.valueAt(x, y) = BLACK;
    } else {
        out.valueAt(x, y) = WHITE;
    }
  }
}
```

The keyword **else** has the same meaning as in C or Java. The keyword **copyof** is explained below.

WHENEXP	::=	(when **(** ANDEXP **)** **{** (WHENEXP)* **}** (else **{** (WHENEXP)* **}**)?) | CONSEQNT
CONSEQNT	::=	ASSIGNMNT | NEWDECL
ASSIGNMNT	::=	CFUN (**.** CFUN)* (**=** (EQUAL | NEWEXP))? **;**
CFUN	::=	<IDENTIFIER> (**(** CARGS **)**)?
CARGS	::=	(ADDITIVE | NEWEXP) (**,** CARGS)?

Object creation and copying Output variables implicitly describe newly created objects, but it is sometimes necessary to explicitly refer to object creation in action effects. For example, an output may be a complex object, such as a file archive or a list, with an unbounded number of complex sub-elements. Since each of those sub-elements is (possibly) newly created, we need some way of describing their creation. We do so using the keyword **new**.

Additionally, newly created objects may be copies of other objects, possibly with minor changes. Listing all the ways the new objects are the same as the preexisting objects can be cumbersome and error-prone, so we would like to simply indicate that one is a copy of the other, and then

specify only the ways in which they differ. We do so using the **copyof** keyword.

Suppose we have an action whose input, `in`, is a collection of JPEG files and whose output, `out`, is a new collection, in which the files from the input are compressed with quality of 0.75.

```
forall Image orig;
when(in.contains(orig)) {
  out.contains
     (new Image copyof orig {
       quality = min(orig.quality, 0.75); });
}
```

When an object is copied, all attributes of the original object are inherited by the copy, unless explicitly overridden. For example, the new Image is identical to the original in every way, except in quality, which is set to 0.75. Note that this is one way in which attributes of objects are different from other relations on objects. `in.contains(orig)` is an attribute of `in`, but not an attribute of `orig`, so after `orig` is copied, `in.contains(copy)` is not true but, for example, `copy.format == JPEG` is true.

The copy and the original need not be the same type, as long as they inherit from or implement a common parent type. All attributes common to both types are copied.

Formally, we can describe new objects as Skolem functions of the actions, inputs and quantified variables that they depend on. The semantics of **copyof** can be specified in a manner similar to Reiter's solution to the frame problem (Reiter 1991). Let a be an action with an effect "**new** T n **copyof** i," and let p represent an attribute common to the type of i and type T. Let $n = sk(a,i,v)$ be a Skolem function of action a, input i and variables v appearing in a. We will write $p(n)$ to designate the value of the attribute p of object n. Let Π^a be the precondition of action a, let $do(a,s)$ be the state reached by executing action a in state s. Without loss of generality, assume that for each possible value v of attribute p, there is a single conditional effect of the form "**when** $(\gamma^v_{p(n)}(a))$ `{n.p = v;}`." If a has no direct effects concerning $p(n)$, then $\gamma^v_{p(n)}(a,s)$ is false for all v. If a unconditionally sets $p(n) = x$, then $\gamma^x_{p(n)}(a,s)$ = true. Because the effects of a are assumed to be consistent, $\gamma^v_{p(n)}(a,s)$ can be true for at most one value of v.

The successor state axiom for $p(n)$ is:

$$\Pi^a(s) \Rightarrow p(n, do(a,s)) = \begin{cases} v & \text{if } \gamma^v_{p(n)}(a,s) \\ p(i) & \text{otherwise} \end{cases}$$

That is, assuming a is executable ($\Pi^a(s)$), the value of $p(n)$ after a is executed is v if a has an effect that sets it to v. Otherwise, it is the value of $p(i)$. A similar axiom must be given for each attribute common between i and n.

The advantage of **copyof** is purely syntactic, since it could be replaced by a large number of conditional effects, one for each attribute of the object being copied. However, since the number of attributes can be quite large, the reduction in the size and apparent complexity of action descriptions can be substantial. This is exactly analogous to the advantage of the STRIPS assumption as a solution to the Frame Problem, in that we avoid specifying conditions that stay the same. The only difference is that the properties that "persist" are actually copied from one object to another. Using conditional effects would also make it harder for a planner to distinguish effects that result in progress toward some goal from those that simply propagate a condition from one file to another. Although this could be determined using domain analysis, that would be making life harder for both the planner and domain modeler with no apparent advantage for either.

NEWDECL	::=	new QUALTYPE <IDENTIFIER> (copyof <IDENTIFIER>)? (({ (ATTRIBUTES)* }) | ;)
NEWEXP	::=	new QUALTYPE (copyof <IDENTIFIER>)? (({ ATTRIBUTES * }) | ;)
ATTRIBUTES	::=	FUNEXP = (EQUAL ; | NEWEXP)

6.4 Execution

The action descriptions include instructions for actually executing the action. These instructions are written in Java, which enables us to write actions that correspond to any operation that can be performed by the Java runtime environment, including invoking methods on objects or making system calls. All parameters and inputs corresponding to Java objects or primitives may be referenced in the Java code, and outputs must be initialized.

We pose the requirement that the results of execution are accurately reflected by the stated effects of the action. There is, of course, no way to verify this requirement, but that's the case for execution in any planning domain.

EXEC	::=	exec <INLINE_CODE> ;

7 States

A typical component of planning problems is a specification of the "initial state," from which the goal must be achieved. In PDADL, a significant amount of state information is communicated through the execution of inlined code during constraint reasoning, which can be used to "query the world" to determine the current state. However, static state information is also useful, especially metadata descriptions for stable data sources. The language provides the ability to define multiple named states through the **state** keyword. States may be thought of as dumbed-down actions that have no preconditions and can only be "executed" in the initial state. As with goals, metadata descriptions are specified using the **when** keyword. As with goals, the LHS can refer to the current state or earlier, but the RHS refers to the current state, not the final state. The conventions for describing the semantics and syntax of data are the same as they are for goal descriptions.

The RHS of metadata state conditions can only contain static predicates describing the data and the LHS can only contain fluents and constraints. Recall that the LHS of goals could contain static predicates, which allowed us to express goals that relate the contents of one data file to the contents of another. Metadata formulas in the initial state can only relate data contents to the current or past state of the world. A

consequence of this restriction is that the predicates appearing on the LHS are completely disjoint from the predicates appearing on the RHS.

In addition to metadata, state conditions can also include unconditional fluent literals describing simple facts such as the names and locations of files.

```
STATE ::= state <IDENTIFIER> {
          ( forall PARAMS _; )
          | WHENEXP )+ }
```

8 Plans

A DPADL plan is a triple $< \mathcal{N}, \mathcal{A}, \mathcal{C} >$, where $\mathcal{N}$ and $\mathcal{A}$ are a set of nodes and arcs in the form of a directed acyclic graph (DAG). The nodes represent actions. The goal is represented as a node with only incoming arcs, and the initial state is represented as a node with only outgoing arcs. An arc $A \in \mathcal{A}$ is a tuple $< p, o_p, c, i_c >$, in which $p \in \mathcal{N}$ is the producer or source node, o_p is an output variable of p, $c \in \mathcal{N}$ is the consumer or target node and i_c is an input variable of c. We refer to the arcs in $\mathcal{A}$ as "I/O links," because they link the output of the producer to the input of the consumer. $\mathcal{C}$ is a constraint network, which is a triple $< V, D, C >$, where V is a set of variables appearing in actions $n \in \mathcal{N}$, D is a set of domains of those variables, representing their possible values, and C is a set of constraints, each of which defines a relation on some subset of the variables in V.

A plan is valid if

- All of the variables in V corresponding to action parameters are grounded (i.e., have singleton domains in D), $\mathcal{C}$ is solved. See (Golden & Frank 2002) for a discussion of how the constraint network is solved.
- All of the constraints corresponding to goals or preconditions are in C.
- Each input i_n of each action $n \in \mathcal{N}$ has a corresponding arc $< p, o_p, n, i_n >$, such that the constraint $i_n = o_p$ is in C and every precondition (excluding constraints) associated with i_n is supported by p. A disjunctive precondition is supported if one of the disjuncts is supported, a conjunctive precondition is supported if all of the conjuncts are supported, and a precondition of the form **"when** (Φ) $\{\psi\}$," where ψ is a literal and Φ is conjunctive, is supported by p if
  - It is a constraint in C **or**
  - $\Phi \models \psi$ **or**
  - There is a corresponding effect **"when** (Φ') $\{\psi'\}$" in p, such that $\psi' \models \psi$,[4] subject to the constraints in C, and either $\Phi \models \Phi'$ or the subgoal "**when**(Φ) $\{\Phi'\}$" is supported **or**
  - There is no effect **"when** (Φ') $\{\psi'\}$" in p, such that $\psi' \models \psi$ or $\psi' \models \neg\psi$, but there is an effect **"when** (Φ'') $\{o_p$ **copyof** $i_p\}$," where i_p is an input of p, and the subgoal **"when** (Φ) $\{\psi\{i_n/i_p\} \wedge \Phi''\}$" is supported.
- Each precondition not associated with any input is true in the initial state.

[4] Entailment can be determined using unification.

Since we are restricting our consideration to pure data-processing domains, we can ignore "sibling" subgoal clobbering. Actions only create new objects; they don't change the world or existing objects, so there is no opportunity for parallel branches to interact with each other. Note also that there is nothing preventing us from having multiple I/O links coming in to a single input, providing redundant ways of producing that input. At execution time, the agent will need to choose which to use, but deferring this choice to execution time can provide flexibility and robustness, since some data source may be unavailable, late, or of poor quality.

9 Conclusions and Related Work

We have described DPADL, an action language for data processing domains, which is used in the IMAGEbot system. The parser for the language, and a planner that supports the language, are fully implemented, and the whole system is fully integrated with the TOPS ecological forecasting system, which is under ongoing development; IMAGEbot can sense, plan and act in the TOPS domain.

We have compared DPADL to PDDL and discussed some of the reasons PDDL is not suitable for data processing domains. Of these, the most important are that it relies on the CWA, provides no support for inputs, outputs and object creation, and is very limited in the kinds of constraints that can be expressed. These problems could be addressed by less radical changes to the PDDL language. Some features, such as the use of **when** expressions in goals and the initial state and quantification over potentially infinite sets, are necessary for describing data processing on a causal level. We have found this low-level causal representation quite conducive to planning, since standard planning techniques can be used to correctly reason about the result of chaining multiple data-processing actions together. With a more abstract representation, paradoxically, more effort would be required of the domain designer .

DDL, the language used in the Europa planner (Jönsson *et al.* 2000), the descendent of the Remote Agent planner that flew on-board Deep Space One, supports constraints and rich temporal action models. In fact, the constraint reasoning system we use was taken from Europa. DDL supports a limited ability to create new objects, but not as a consequence of action execution. DDL domain descriptions are quite different from those of either PDDL or DPADL. Rather than describing actions in terms of preconditions and effects, DDL uses explanatory frame axioms. That is, for every condition that could be achieved, the domain designer must specify how to achieve it, listing all actions that could support it and other conditions that must be satisfied. DDL is also timeline-based and makes no distinction between states and actions. While these may be good design decisions for spacecraft domains, they are not appropriate for data processing domains.

DAML-S (Ankolenkar *et al.* 2002) and WSDL (Christensen *et al.* 2002) are languages for describing web services, both based on XML. DAML-S is the more expressive, allowing the specification of types using a description logic and allowing one to specify preconditions and post-conditions, which might be used by a planning agent. However, we don't believe that description logics are expressive

enough to describe the data-processing operations that we need to support.

The Earth Science Markup Language (ESML; http://esml.itsc.uah.edu) is another language based on XML, under development at the University of Alabama in Huntsville to provide metadata descriptions for Earth Science data. Unlike DAML-S and WSDL, ESML is well suited to describing the complex data structures that appear in scientific data. Unlike DPADL, it is only intended to describe data files, not data processing operations, but it does provide explicit support for describing the syntax and semantics of data files and allows the specification of constraints in the form of equations. Although it is less expressive and more specialized than DPADL, it is a promising metadata standard for Earth Science. In the near future, we hope to support conversion between ESML and DPADL metadata specifications.

Near the far end of the expressiveness spectrum, the situation calculus (McCarthy & Hayes 1969) provides plenty of expressive power, but at a price: planning requires first-order theorem proving. We opted instead to make our language as simple as possible, but no more so. DPADL does not support domain axioms, nondeterministic effects or uncertainty expressed in terms of possible worlds, and much of the apparent complexity of the language is handled by a compiler, which reduces complex expressions into primitives that a simple planner can cope with. Despite the superficial similarity to program synthesis (Stickel *et al.* 1994), DPADL action descriptions are not expressive enough to describe arbitrary program elements, and the plans themselves do not contain loops or conditionals.

Of the many planning domain description languages that have been devised, the closest to DPADL is ADLIM (Golden 2000), on which it is based. Advances over ADLIM include tight integration with the run-time environment (Java) and constraint system and a Java-like object-oriented syntax that makes it natural to describe objects and their properties. As discussed in Sections 2 and 6.3, this encodes valuable information used by the planner.

Collage (Lansky & Philpot 1993) and MVP (Chien *et al.* 1997) were planners that automated image manipulation tasks. However, they didn't focus as much on accurate causal models of data processing, so their representation requirements were simpler.

Acknowledgments

I am indebted to Wanlin Pang, Jeremy Frank, Ellen Spertus and Petr Votava for helpful comments and discussions. This work was funded by the NASA Computing, Information and Communication Technologies (CICT) Intelligent Systems program.

References

Ankolenkar, A.; Burnstein, M.; Hobbs, J. R.; Lassila, O.; Martin, D. L.; McDermott, D.; McIlraith, S. A.; Narayana, S.; Paolucci, M.; Payne, T. R.; and Sycara, K. 2002. DAML-S: Web service description for the semantic web. In *Proceedings of the 1st Int'l Semantic Web Conference (ISWC)*.

Chien, S.; Fisher, F.; Lo, E.; Mortensen, H.; and Greeley, R. 1997. Using artificial intelligence planning to automate science data analysis for large image database. In *Proc. 1997 Conference on Knowledge Discovery and Data Mining*.

Christensen, E.; Curbera, F.; Meredith, G.; and Weerawarana, S. 2002. Web Services Description Language (WSDL) 1.1. Technical report, World Wide Web Consortium. Available at http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

Fox, M., and Long, D. 2003. Pddl 2.1: An extension of pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*.

Golden, K., and Frank, J. 2002. Universal quantification in a constraint-based planner. In *Proc. 6th Intl. Conf. AI Planning Systems*.

Golden, K., and Weld, D. 1996. Representing sensing actions: The middle ground revisited. In *Proc. 5th Int. Conf. Principles of Knowledge Representation and Reasoning*, 174–185.

Golden, K. 2000. Acting on information: a plan language for manipulating data. In *Proceedings of the 2nd NASA Intl. Planning and Scheduling workshop*, 28–33. Published as NASA Conference Proceedings NASA/CP-2000-209590.

Jönsson, A.; Morris, P.; Muscettola, N.; and Rajan, K. 2000. Planning in interplanetary space: Theory and practice. In *Proc. 5th Intl. Conf. AI Planning Systems*.

Lansky, A. L., and Philpot, A. G. 1993. AI-based planning for data analysis tasks. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence for Applications (CAIA-93)*.

McCarthy, J., and Hayes, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*. Edinburgh University Press. 463–502.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 2(2):35–55.

Nemani, R.; Votava, P.; Roads, J.; White, M.; Thornton, P.; and Coughlan, J. 2002. Terrestrial observation and prediction system: Integration of satellite and surface weather observations with ecosystem models. In *Proceedings of the 2002 International Geoscience and Remote Sensing Symposium (IGARSS)*.

Reiter, R. 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V., ed., *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*. Academic Press. 359–380.

Stickel, M.; Waldinger, R.; Lowry, M.; Pressburger, T.; and Underwood, I. 1994. Deductive composition of astronomical software from subroutine libraries. In *Proceedings of the 12th Conference on Automated Deduction*.

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 `situation`s 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.

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.* 17*th 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-*

[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`

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.

Position paper on the infrastructure supporting the development of PDDL

Thomas McCallum *
Centre for Intelligent Systems and their Applications
School of Informatics, The University of Edinburgh
Appleton Tower, Crichton Street, Edinburgh EH8 9LE, UK
t.e.r.mccallum@sms.ed.ac.uk

Abstract

For 5 years now PDDL has been growing in complexity and size. With the languages success the field of Planning and Scheduling (P&S) has grown in the light of the International Planning Competition (IPC), in both complexity and popularity. It is proposed that now is the time to allow the community, as a whole, to drive the progress of PDDL and to create a community outside that of the competition. With this in mind some prerequisites are outlined that are deemed necessary to achieve success in this aim, with both some minor extensions to the language but also with greater development infrastructure.

Introduction

This paper sets out some answers to the issues raised as discussion topics for the ICAPS-03 workshop. It focuses on the PDDL infrastructure and the need to secure greater links with a wider community for the development of the language. There are a number of areas where Planning and Scheduling could greatly increase its desirability to the industrial community starting with some improvements in the way languages and resources are managed.

In the first part of the paper the various questions pointed to by the initial call for papers are discussed. Following this a number of infrastructure changes are proposed, ending up in some suggestions for small extensions to the language to enable more flexibility and knowledge sharing.

Infrastructure

What is the ambition for PDDL?

Although PDDL was initially conceived for the IPC in 1998 by Drew McDermott, it has since become the main language for the Planning and Scheduling community as a whole. As such it is no longer just a 'toy' for use with the competition and is becoming a popular research, and in the near future, industrial standard. These ambitions are necessary to ensure that a sensible, community driven language, that addresses all the issues that come with such a standard is created. Anything less than this and the continuing multitude of small specialist languages will continue.

*I would like to thank the reviewers and John Levine for their helpful comments.

There are many roles for languages in planning whether it be to capture heuristics, to communicate knowledge about a particular domain or to help knowledge acquisition, and as such it is for a standards committee to set the main objectives of the language, guided by what the community are using the language for, rather than one person's ideals, as it is this that will allow the language to continue to become more popular and useful in applied planning. The main feature of PDDL that is important to all these languages is the format and tools for dealing with it. By having the ambition to merge the various syntax and increasing PDDL's flexibility it will be possible to cut the learning and development curve of these language aspects down to a minimum, therefore aiding PDDL's overall appeal. This is in contrast to the belief that there is a need for individual languages for specific planning purposes, which hinders development of complimentary technologies.

There are those that would have PDDL as a language only available to experts rather than general users, but as in other modelling languages such as HTML and VRML, their popularity has been greatly aided by their broad appeal. Users could program PDDL by hand but by making the language available to the community in terms of libraries a number of utilities such as PDDL editors could be written allowing normal users to investigate their own individual planning problems. There are no gains to be had by keeping the language at a research/expert level only therefore it should be an ambition, as the whole premise of planning is to apply to problems in the real-world, to make the language accessible to 'real-world' users.

Are we ready for such a standard?

Even though there are a number of areas that can be improved upon in the language and also a number of pressures to consider in its later development from both research and industrial backgrounds, it is the position of this paper that the community is both ready and willing to accept a common standard. For years there has been development of various specialist languages that confuse and obscure the more general landscape of the field. This is one of the reasons why it is so hard to sell Planning and Scheduling as a serious industrial tool. If a common language was adopted with the ability for the community to make specialist changes for their own particular product it would enable and create a

more uniform and pervasive image to emerge.

Should there be a formal standard committee?

As in other languages such as HTML, C and Java there needs to be a lead direction for a language to go in. It is therefore both right and necessary that a central committee be formed to maintain the coding and theoretical standards of the language. It should be comprised of all the areas of influence, such as research, industrial and commercial. In this way the language will be able to accommodate and present itself to all interested parties. But having made the above points it should also be open to the community to add and update the language for there own particular use. With the right infrastructure a similar effect to the Mozilla project (mozilla 2003) could be achieved. This would enable specialist languages and tools to be written quickly and made available to the community but for them to be made part of the official standard the committee would have to vote on the quality of the change and the real value to the community as a whole.

Should developments be allowed to branch as expressive power increases?

This question rests on the assumption that one community base would be too restrictive to encompass all the needs of a increasing language complexity. With a flexible infrastructure such as the one proposed above there would be little need to branch out. That is not to say though that should a need arise to split the language that it should never be considered. In this early stage though it would be unwise and lead to the continuing complex image of the field. With the minor extensions proposed below it should be possible to keep logs of the additions and changes to a standard for the purpose of specialist areas and to allow the community access to it all. The most useful should then naturally be chosen to become part of the overall standard.

Infrastructure Changes

Due to the global nature of the subject the Internet is the most obvious way to promote and develop PDDL. As such an interface that allows users to submit their particular versions of the PDDL language should be made available with the ability to detail changes that they have made. These archives can then be made publicly accessible and if deemed important, additions to the language can be added to the PDDL standard. This common gateway could be used also to develop new tools and foster new discussion on the direction of the language currently only possible via these proceedings which occur too infrequently to drive the language forward.

An example of where a language has had huge success through a community-based approach is the growth of HTML as the standard markup language for the web. The W3C(W3C 2003) organisation is the main body in charge of setting up the standards used on the web but anyone can upload their own tools and updates to the HTML standard to be scrutinised and used by the community. By having mailing lists and newsgroups for people to air their views the whole community feels included in the overall direction of the language. Another advantage of this community-based approach is that documentation becomes more reliable and examples more prolific. This includes the translation of documents into various different languages, there by in itself enlarging the popularity of PDDL.

Another interesting case study is that of the C language(c 2003) that now has a number of working groups. The interesting point here is how it has adapted itself to new technologies such as embedded systems with ease due to its community-based approach. Although there are numerous libraries and extensions available in the public domain, it is still the responsibility of the ISO and IEC to set the C standard. Thereby giving a strict standard to adhere to but also giving maximum flexibility.

There are disadvantages though with the community-based approach. For instance if there is a disagreement about a particular implementation for the standard then this may cause a split in the community. At the moment as there is only a very small number of people who decide on how PDDL grows this is not a problem. A solution may be that some sort of version split may occur but the differences could be allowed to be switched on/off depending on a requirement flag as happens at the moment. Therefore in a way the community will produce its own solution to the problem in its flexibility. Another disadvantage is that there will be a lot more documentation to be done and laziness may creep in to certain versions. This though has also been overcome by other communities by allowing the less competent programmers to debug and document any new versions of tools and the language that come out. Therefore the scale of the community is what keeps these factors in check.

In order to properly archive the expected mass of versions it would be necessary to create add some meta data to the current language. These extensions, proposed in the next section, would allow a systematic library to be created, detailing standard and complexity of the language in use.

Extensions

With the above infrastructure ideas in mind a number of extensions to the language would seem necessary to be able to maintain integrity of the language. These extensions would also help the ability to share knowledge of domains amongst various Planning and Scheduling software.

The first extension is that of language version. With the formation of a committee one of the first things necessary would be to create a numbering system that reflected the distance of the new language concept from that of the standard. An example might be as follows:

[standard version e.g. year.month.day of release] - [specific version of change]

This particular example allows the programmer reading this to determine how current the language version is in terms of the PDDL standards but also a particular version number for the specialisation which would be maintainable by the author.

The next few extensions area also to do with authoring of the language. An indication of the maximum PDDL level reached would be useful so that programs written for a specific level do not try and read above that level which could

cause problems in an automated environment. Finally on this theme an indication of where the language originated and a contact email would be helpful for extra development purposes. These fields would be optional as in HTML but authors would be wise to use them to help third-party users adoption of their specialist modifications.

Therefore a new meta keyword might look as follows:

(:meta 02.11.14-1.11 3 'University Of Edinburgh' 'author@ed.ac.uk')

Translated as the community standard set on the 14-11-2002 altered by University of Edinburgh, released again at version 1.11 which goes up to PDDL level 3. The person to contact for any questions is author@ed.ac.uk.

Just as in HTML this sort of information will allow programs to inter-operate and programmers will be able to write their software to any degree of specialisation that they desire but still be able to contribute to the community as a whole. This sort of information might also generate greater interest in the generation of a hybrid between PDDL and XML which can handle this sort of meta-information in a predictable manner.

Conclusion

In this paper it has been outlined that changes to the infrastructure plus a few minor extensions would enable the language to present itself in as both a research and industrial standard. It has been argued that this would increase interest from the industrial and commercial communities by simplifying the overall landscape of Planning and Scheduling and allowing greater knowledge sharing between current software. Examples have been suggested from the Internet community where tools and languages have been allowed to grow via a large community making small suggestions and then an overall committee from all areas strengthen the language by using the most popular alterations to set a common standard. This community-based approach does need more planning but it is the way forward if PDDL wants to become a true language standard of the Planning and Scheduling community.

References

2003. Iso/iec c - approved standards. http://anubis.dkuug.dk/JTC1/SC22/WG14/.

2003. The mozilla organisation. http://www.mozilla.org.

2003. World wide web consortium. http://www.w3c.org.

PDDL: A Language with a Purpose?

T. L. McCluskey
Department of Computing and Mathematical Science,
School of Computing and Engineering,
University of Huddersfield, UK
email: lee@zeus.hud.ac.uk

Abstract

In order to make planning technology more accessible and usable the planning community may have to adopt standard notations for embodying symbolic models of planning domains. In this paper it is argued that before we design such languages for planning we must be able to evaluate their quality. In other words, we must clear for what *purpose* the languages are to be used, and by what criteria the languages' effectiveness are to be judged. Here some criteria are set down for languages used for theoretical and practical purposes respectively. PDDL is evaluated with respect to them, with differing results depending on whether PDDL's purpose is to be a theoretical or practical language. From the results of these evaluations some conclusions are drawn for the development of standard languages for AI planning.

Introduction

Good planning algorithms are hard to devise, but fairly easy to evaluate; on the other hand, modelling languages are fairly easy to devise, but hard to evaluate. Language extension is similar: it is relatively easy to add arbitrary features to a language, but adding the tools to manipulate the enhanced language, or perfecting a semantic definition of the extension, is much more difficult. Having devised a language[1], how can we evaluate it's quality? One way is to use *practical* methods. Experiments can be set up to test the effectiveness of a language, using engineers in a controlled environment. This is a time consuming and costly business, however, and the tests are prone to extraneous variables as people act differently when on their own to when they are being experimented on.

For reasons such as these, more analytical methods of evaluating languages are popular. This involves generating a list of criteria, usually called *design criteria*, that have been devised when considering the *purpose* of the language. Sometimes these criteria are well developed a priori, and sometimes old languages are subject to being evaluated with new criteria. A well-used language does not necessarily mean it will score highly on a desired set of criteria; it may be that one feature of the language makes it uniquely usable by a community. That feature may be that it is similar to a set of languages it was designed to replace, making it easy to migrate to. Or as another example, consider the old language FORTRAN IV. It was well respected by engineers of mathematical applications because of its compilers' efficiency and its wealth of mathematical primitives. But given it should embody desirable software engineering criteria such as *strong typing* and *structured programming* then it was quite obvious that it scored poorly. Thus languages like FORTRAN were either re-invented (hence environments such as 'MatLab') or they evolved to score higher against the new criteria (hence FORTRAN 77 with its structured control constructs).

In this paper I discuss the kinds of criteria against which an AI planning language might be judged, making a distinction between them depending on the purpose of the language. I apply them to version 1.2 of PDDL, and draw some conclusions for the future development of planning language standards.

Criteria for Evaluating Languages

The study of languages for machine as well as human consumption (ie ones that people have to manipulate or understand in some way) encompasses three aspects: syntax, semantics and pragmatics. A fundamental question about a language arises when considering these three aspects: *is it going to be used theoretically or is it going to be used generally by people to encode complex algorithms or knowledge?*

Theoretical formal languages: Considering theoretical languages, in computer science we have the Lambda Calculus, the Pi-Calculus, the Turing Machine, first order logics etc. They are often used to theorise about concepts (e.g. sequential or concurrent computation), or are used as the meaning domain for the semantical definition of practical languages. Considering the well-known languages which are used in theoretical research, the intrinsic criteria that underlie their success appear to be the following:

- (1) *simple, clear, precise syntax and well-researched semantics*

  For example, in Lambda Calculus the syntax is defined in a few BNF rules, with syntactic sugar being added when needed. The semantics have been studied in depth: for example, recursive functions in Lambda Calculus have a

[1] It is assumed in this paper that the languages considered are for domain models *input* to a planner, rather than 'plan' languages used to represent the output of a planner.

clear and precise operational semantics (using conversion rules and normal order reduction) and fixed point semantics. Research has showed that these two kinds of semantics co-inside.

- (2) *adequate expressiveness*

  Can the language adequately represent the range of its targeted application domains? For Lambda Calculus this is the domain of computable functions, and it is a well known (though unproven) conjecture that it is adequate for this.

- (3) *clear mechanisms for reasoning*

  Can a user (perhaps with tool support) reason with parts of a formula in the language? In Lambda Calculus one uses the conversion rules to transform one expression into another, equivalent expression.

Applied formal languages: Theoretical languages, however, tend to have *little or no* pragmatic features. At the other extreme are formal languages which have complex syntax which support many useful pragmatic features. For example we have Java in the field of programming, Z in formal specification of software, RML in requirements modelling (Greenspan *et al.* 1994) or $(ML)^2$ in knowledge-based systems (van Harmelen *et al.* 1996). Often, pragmatic features are present at the expense of clarity. For example, the amount of extra syntactic baggage employed by JAVA tends make it much less clear that the older, simpler PASCAL programming language. In AI planning there are a spectrum of languages between these two extremes. Some planning systems require complex practical-oriented features in their input languages, such as hierarchically structured objects and operators (McCluskey 2000), or Condition Types (Tate *et al.* 1994); some researchers need to use an input language that minimally models the dynamics of the domain, for example when exploring the theoretical complexity of planning (e.g. (Bylander 1991)).

I now consider some criteria that have been found useful for evaluating the pragmatic aspects of formal languages. A quite general framework for the evaluation of languages and their environments is Green's Cognitive Dimensions (Green 2000). This involves using a set of criteria as 'discussion points' to focus on the various dimensions of a language, and may result in an informal evaluation (Green admits his method in not analytic, and the dimensions are not mutually independent). He devised fourteen criteria which have been used to evaluate various types of language and environments, including theorem proving assistants, UML and programming languages. Although these criteria have been quite widely used, they have been successful for languages which are embedded in an environment rather than a language itself. Some of these criteria are aimed at the visual aspects of environments in which the language is embedded. Thus they would be better applied to a planning knowledge acquisition environment than the language used to represent the knowledge only. However, I have extracted and enhanced three criteria which are particularly related to the language itself, and have been used elsewhere in the literature:

- (4) *maintenance* (also referred to as *hidden dependencies* or *locality of change*)

  After changing one part of the notation, will this have any invisible knock-on effects on other parts? Do changes to a part of a model just have a local effect, or will they have global connotations? Can the model be easily and consistently updated to reflect changes? (from the viewpoint of maintenance, it is desirable that all changes have minimal global effects).

- (5) *closeness of mapping / customisation*

  How natural is the mapping between the domain and the model? how small is the 'semantic gap'? Is the language customisable in some sense so that it can fit in well with applications?

Since there is a whole range of assumptions involved in planning which may or may not hold in an application (for example to do with action duration, resources, closed world) it may be that the modelling language will have "variants" to deal with different assumptions. Related to this is the need to have 'hooks' in the language to allow extension: if the scope or depth of requirements of the domain are increased, can the formalism be likewise extended?

- (6) *error-proneness*:

  does the design of the language discourage errors, or are there any parts where it is hard to avoid errors? Is the construction of domain models error prone in a particular way?

Criteria (4) - (6) are analogous to those used to evaluate programming languages: (4) reflects the idea that languages should embody structures to promote loose coupling between sub-parts, and strong coherence. The 'object' in object-oriented programming scores highly in this respect, as implementations of object behaviour are insulated from other parts via the object interface. (5) reflects the dominance of 'high-level' languages - those that are more problem-oriented than machine oriented, and are equipped with user-defined structures for customisation. Finally, (3) has influenced programming language design in order to eliminate common errors; for example, languages which are *not* strongly typed are particularly prone to errors resulting from variable misuse and misspelling.

To investigate more criteria we use Van Harmelen et al's evaluation of $(ML)^2$, a formal KBS specification language, and hence relevant to AI planning languages. They use six criteria to evaluate this formal language used for formalising KADS expertise models (van Harmelen *et al.* 1996). Although objectiveness may be compromised when a group sets out their own criteria for evaluating their own product, the criteria they use are clearly worked out in response to considering the purpose of the language. They use criteria similar to those above (in particular (1), (4) and (6)), as well as the following:

- (7) *reusability*

  Can models or parts of models be easily reused to construct models for new domains?

- (8) *guidelines and tool support*

Is there a useful method to follow to build up a model, and are there tools to support this process?

With respect to the last point, in all areas in computer science involving some kind of non-trivial knowledge capture, methods have been developed to support this. For example in formal specification of software there is the B method(Schneider 2001), or in the acquisition of knowledge for KBS there is the KADS method (Wielinga *et al.* 1992), and some methods have also been developed for acquiring AI planning knowledge (Tate *et al.* 1998; McCluskey & Porteous 1997). The method will give a set of ordered steps to be carried out in order to capture and debug the domain model, thus guiding the knowledge engineer throughout the process. Ideally, the tools will be available in an integrated environment, and will support the steps in the method. Using the structure of the model language, the tools should be able to provide powerful support for statically validating, analysing and operationalising the model.

Finally, I draw on the guidelines for the design of domain model languages as recorded in the Knowledge Engineering for AI Planning Roadmap (McCluskey *et al.* 2003). This was written in the context of planning domain modelling, with the purpose of the language being to assist the process of knowledge acquisition and domain model validation. The criteria included several similar to those discussed above (in particular (1), (2), (5), (8)), and additionally the following:

- (9) *structure*

  It should provide mechanisms that allow complex actions, complex states and complex objects to be broken down into manageable and maintainable units. For example, the dynamic state of a planning application could be broken down into the dynamic state associated with each object. On this structure can then be hung ways of checking the model for internal consistency and completeness.

- (10) *support for operational aspects*

  The language's framework should include a set of properties and metrics which can be evaluated to assess a model's operationality and likely efficiency. It should be possible to predict whether the model can be translated to an efficient application, and what kind of planner should be used with the model.

To sum up, the criteria for practical formal languages are based around the idea that the structure of the language should support initial model acquisition and debugging, and subsequent model maintenance and re-use. Also, although criteria (1) - (3) are aimed specifically for theoretical languages they are often thought desirable for practical languages also.

Design Criteria for a Planning Language

What are the design criteria for an AI planning language? As mentioned above, it depends for what *purpose* the language is set, and a particular concern is whether the language is for theoretical or practical use. In the case of PDDL, this 'purpose' seems to have grown and changed as the language is used more widely. From the initial PDDL report(AIPS-98 Planning Competition Committee 1998), it appears that the language was designed to represent the syntax and semantics of domain models that were currently available to the authors, and that were used as input languages to many of the published planners of the time. Not all planners were expected to use all PDDL's features, and on the other hand planners were expected to have requirements that would mean a user extending PDDL in a controlled way. Its initial purpose, therefore, appears to have been as a communication language - a basic common denominator for planners of the STRIPS-tradition at the time so that (a) they could be compared in competition and (b) problem sets could be shared and planning algorithms independently validated. In this respect, as a communication language it has clearly been successful.

Nowadays the Planning Domain *Definition Language* is sometimes described as a 'modelling language', which has quite different ramifications than its originally expressed purpose. If its purpose is to support theoretical study, eg to help compare the capabilities of new planning algorithms, then it should be evaluated with respect to a restricted set of criteria such as (1) - (3). If its purpose (now) is to be a practical language, to help an engineer accurately and efficiently encode an application domain into a planning domain model then additionally it should be subject to evaluation by a range of the criteria such as (3) - (10).

Evaluation of PDDL with respect to stated criteria

In PDDL we have a family of languages to suit planners with different capabilities. The basic requirement in PDDL is ':strips' which indicates the underlying semantics of the language worlds are considered as sets of situations (states), where each state is specified by stating a list of all predicates that are true. Firstly, I evaluate the language using criteria (1) - (3) given above. I concentrate here on version 1.2 of the language, and remark on the extensions later.

Clear syntax and semantics: The syntax is clear and precisely defined within the manual, and parsing tools that embody this definition are publically available. The semantics of PDDL version 1.2, however, are informal and appear to be distributed among the manual itself, the pre-existing languages/systems that PDDL replaced (eg ucpop), PDDL's language processors, and the LISP interpreter. Although the fact that PDDL's syntax is LISP-like appears a superficial observation, the meaning of several of the primitive functions is given in terms of LISP functions. For example, the manual often relies on the reader using his intuition (p9: 'Hopefully, the semantics of these expressions is obvious'). As the language becomes more complex, then the natural language semantics are less obvious (for example, consider the meaning of domain axioms and their relationship with action definitions on page 13 of the manual). These extensions need to be defined precisely, as if two systems use these extensions, then they ought to do so in a uniform way, otherwise the standard is not preserved.

Adequate expressiveness: That PDDL a very expressive language for a range of planning applications has been

shown by the range of problem domains used in competitions and in benchmark sets. Further, the ability to change some of the environmental assumptions is also present, although the semantics of some of these extensions is not clear.

Clear mechanisms for reasoning: A domain definition in PDDL is a 'model' in the sense that we have a representation that can be used to perform operations in the same manner that occur in the domain; and that there is a well-known operational semantics for constructs in the model. The declarative features of the notation - pre- and post-conditions, logic expressions, and named objects within the model which correspond directly to named objects in the domain, make reasoning about the notation feasible. However, problems to do with semantics, particularly to do with its extensions, restrict the success of this language with respect to the criterion.

Summary

In terms of the criteria for a language used for theoretical purposes, PDDL scores well on some aspects. There are problems with the lack of a clear semantics but these tend to be more to do with the non-basic parts such as the domain axioms. Also, the temporal and resource extensions of version 2.1 seem to have addressed the semantic issues more thoroughly (Fox & D.Long 2001).

PDDL: a modelling language?

Here I briefly evaluate PDDL using (3) - (10), the criteria reserved for languages aimed at practical application.

structure and error-proneness: PDDL has features such as ':timeless' - which allow the statement of static knowledge, and ':domain-axioms' which allow left-to-right rules that form invariants on situations. A domain definition is structured into components by Keywords e.g. :constants :actions etc. A special keyword is :requirements which tells a process which blend of PDDL features are used in the domain definition. Further, the manual devotes several pages to a hierarchical action notation; unfortunately, perhaps related to the fact that it was not subsequently used, version 2.1 of PDDL excludes this. On the negative side, whereas PDDL (v1.2) has features for hierarchically structuring actions, it does not have sufficient features for giving structure to objects or states. Further, the language lacks structures for setting up internal consistency criteria such as the completeness or validity of world states or actions.

maintenance and re-usability: PDDL's declarative form makes adding and changing operators a local task, and reusing operators in new domains feasible. The ':extends' feature allows a form of modularisation - one can import previously written components into a new model. However, no help is given in dealing with the natural dependency of actions on each other: the requirement that pre-conditions should be achievable by the execution of other actions or the initial state causes global interference and is the cause of many errors when defining domains.

guidelines and tool support: there are parsers, solution checkers and domain analysis tools available publically, but PDDL was not designed to be associated with a method for model building. This one point alone seems to make it currently ineffective as a practical 'modelling language' for complex applications.

closeness of mapping / customisation: Clearly PDDL's encodings shares the same 'high level' aspects as do propositional encodings in general. Also, one can pose domain axioms to model invariants in the domain. Reflecting domain *structure* (as mentioned above) by for example creating composite objects is not possible. Customisation does appear to be addressed in PDDL with features such as ':requirements' where fundamental assumptions about the model of the domain can be set.

support for operational aspects: The PDDL manual makes it clear that this area is not one that fits in with PDDL's aim - to model the physics of a domain. It does recommend a convention by which such extensions can be made in a controlled way, such that the model with the extensions stripped away will make sense to a pure PDDL interpreter.

Summary

For a language whose initial purpose was one of domain model communication, and which aspired to include only feature which capture dynamics, PDDL has in fact several features to help domain builders. These include HTN operators, domain axioms, modularisation through the 'extends' keyword etc. On the other hand, it fails to meet the criteria is in not being associated with a model building method, and in its lack of structure for objects, predicates and states. Structuring devices are present in several modelling languages (e.g. DDL.1 (Cesta & Oddi 1996; McCluskey & Porteous 1997)); these allow the state-space of objects to be modelled independently of the actions, and hence are useful in removing errors from action representations.

Conclusions

Both to help the Planning field mature, *and* to help engineers apply the technology, language conventions have to be achieved. The requirements of the future Semantic Web in particular will demand a common model for planning knowledge. This paper has argued that before conventions are devised there must first be an agreement on the purpose of a language, and secondly a set of criteria to be used to help form and develop the language.

Two broad purposes for an AI planning domain language were outlined - one as a theoretical device, to be used for exploring the properties of planning algorithms, and one as a practical language, to be used to help an engineer efficiently and accurately encode an application domain.

I performed an initial evaluation of PDDL with respect to the criteria formed from both purposes, with mixed results. The evaluation leads me to the following conclusions:

- in standardising a form of PDDL for theoretical purposes,

more attention needs to be devoted to precisely defining its semantics, and that of any of its extensions;

- in standardising a form of PDDL for practical domain model building, then more structure, guidelines and tool support is required.

For the future, I feel that the community needs to settle on the purpose of PDDL, decide on the criteria that can be used to evaluate PDDL's quality, and perform a thorough evaluation using the language's most recent version. This will lead, I believe, to a sound path for its future development.

References

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

Bylander, T. 1991. Complexity Results for Planning. In *Proc. IJCAI'91*.

Cesta, A., and Oddi, A. 1996. DDL.1: A Formal Description of a Constraint Representation Language for Physical Domains. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*. IOS Press. 341–352.

Fox, M., and D.Long. 2001. PDDL2.1: An extension to PDDL for expressing temporal planning domains . In *Technical Report, Dept of Computer Science, University of Durham*.

Green, T. 2000. Instructions and descriptions: some cognitive aspects of programming and similar activities. In *Advanced Visual Interfaces*, 21–28.

Greenspan, S.; Mylopoulos, J.; and Borgida, A. 1994. On formal requirements modeling languages: RML revisited. In *Proceedings of the 16th International Conference on Software Engineering*, 135 – 148. IEEE Computer Science Press.

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

McCluskey, T. L.; Aler, R.; Borrajo, D.; Haslum, P.; Jarvis, P.; I.Refanidis; and Scholz, U. 2003. Knowledge Engineering for Planning Roadmap. *http://scom.hud.ac.uk/planet/*.

McCluskey, T. L. 2000. Object Transition Sequences: A New Form of Abstraction for HTN Planners. In *The Fifth International Conference on Artificial Intelligence Planning Systems*.

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

Tate, A.; Drabble, B.; and Levine, J. 1994. The Use of Condition Types to Restrict Search in an AI Planner. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*.

Tate, A.; Polyak, S. T.; and Jarvis, P. 1998. TF Method: An Initial Framework for Modelling and Analysing Planning Domains. Technical report, University of Edinburgh.

van Harmelen, F.; Aben, M.; Ruiz, F.; and van de Plassche, J. 1996. Evaluating a formal KBS specification language. *IEEE Expert* 11(1):56–62.

Wielinga, B. J.; Schrieber, A. T.; and Breuker, J. 1992. KADS - a modelling approach to knowledge engineering. *Knowledge Acquisition* 4(1):5 – 53.

The Formal Semantics of Processes in PDDL

Drew V. McDermott
Yale University
Computer Science Department
drew.mcdermott@yale.edu

Abstract

One reason why autonomous processes have not been officially incorporated into PDDL is that there is no agreed-upon semantics for them. I propose a semantics with interpretations grounded on a branching time structure. A model for a PDDL domain is then simply an interpretation that makes all its axioms, action definitions, and process definitions true. In particular, a process definition is true if and only if over every interval in which its condition is true, the process is active and all its effects occur at the right times.

The Semantics Problem for PDDL

In (Fox & Long 2001a), Maria Fox and Derek Long proposed extensions to PDDL for describing *processes*, that is, activities that could go on independent of what the executor of plans did. They also described a semantics for processes in terms of hybrid automata (Henzinger 1996). Their proposal was not adopted by the rules committee for the third International Planning Competition, mainly because most of the members preferred to focus on the narrower extension to "durative actions," actions that took a nonzero amount of time. Another potential reason is that the semantics of processes were so complex that no one really understood them.

A semantics for all of PDDL is hard to lay out, for a couple of reasons. One is that the requirement-flag system turns it into a family of languages with quite different behavior. For instance, the `:open-world` flag transforms the language from one in which `not` is handled via "negation as failure" into god-knows-what. To sidestep this complexity, I will make a variety of simplifying omissions and deliberate oversights:

- ◇ I will ignore the possibility of action `:expansion`s.
- ◇ There will be no formalization of the fact that propositions persist in truth value until some event changes them.
- ◇ `:vars` fields in action and process definitions will be left out.
- ◇ I will assume effects are simple conjunctions of literals. There are no secondary preconditions ("`:when`s") or universally quantified effects.
- ◇ I won't discuss durative actions. They can be defined in terms of processes, as discussed by (Fox & Long 2001b) and (McDermott 2003).
- ◇ Fox and Long (Fox & Long 2001b) allow for "events," which occur autonomously, like processes, but are instantaneous, like primitive actions. I won't discuss these, although they present no particular problem.
- ◇ Except for the presence of autonomous processes, we'll stay in the "classical milieu," and in particular assume that the planner knows everything about the initial situation and the consequences of events and processes. There's no nondeterminism, and no reason to use sensors to query the world.

The Structure of Time

It's fairly traditional (McDermott 1982; 1985) to think of an action term as denoting a set of intervals; intuitively, these are the intervals over which the action occurs. To make this precise, we have to specify what we mean by "interval." A *date* is a pair $\langle r, i\rangle$, where r is a nonnegative real number and i is a natural number. A *situation* is a mapping from propositions to truth values. A *date range function* is a function from real numbers to natural numbers. A *situation continuum* is a pair $\langle C, h\rangle$, where h is a date range function and C is a *timeline*, that is, a mapping from

$$\{\langle r, i\rangle \mid r \in \textit{nonnegative reals and } 0 \leq i < h(r)\}$$

to situations. If $d_1 = \langle r_1, i_1\rangle$ and $d_2 = \langle r_2, i_2\rangle$, then $d_1 < d_2$ if $r_1 < r_2$ or $r_1 = r_2$ and $i_1 < i_2$. A date $\langle r, d\rangle$ is *in* $\langle C, h\rangle$ if $r \geq 0$ and $0 \leq d < h(r)$. A situation s is *in* $\langle C, h\rangle$ if there exists a date d in $\langle C, h\rangle$ such that $C(d) = s$.

The intuitive meaning of these definitions is that at any point r in time there can be a series of zero or more actions taken by the *target agent* (i.e., the hypothetical agent that executes plans). The range function says how many actions are actually taken at point r in that continuum. Each action takes an infinitesimal amount of time, so that "just after" time r there can be an arbitrary number $h(r)$ of actions that precede all time points with times $r' > r$. (This picture is reminiscent of nonstandard analysis (Robinson 1979).)

Using this framework, a purely classical plan might be specified by giving an action $A(i)$ for all dates $\langle 0, i\rangle$ where $i < h(0)$, and $h(0)$ is the length of the plan. The plan is feasible if $A(0)$ is feasible in the initial situation, and $A(i+1)$ is feasible in $C(0, i)$. The plan achieves a goal G if G is true in $C(0, h(0))$. It may sound odd to imagine the entire

plan being executed in zero time, but time is not a factor in classical planning, so a plan might as well take no time at all to execute, or $h(0) \times dt$ if you prefer. In what follows, I will use the dt notation to denote an infinitesimal time interval, i.e., the time from $\langle r, i\rangle$ to $\langle r, i+1\rangle$. If date $d = \langle r, i\rangle$, then $d + dt = \langle r, i+1\rangle$.

If we broaden our ontology to allow for autonomous processes, then plans can have steps such as "turn on the faucet," and "wait until the bucket is full." We can use situation continua to model bursts of instantaneous "classical" actions separated by periods of waiting while processes run their course. A plan may even call for the target agent to do nothing, but simply wait for processes that are active in the initial situation to solve a problem. In that case $h(r) = 0$ for all r.

A *closed situation interval* in continuum $\langle C, h\rangle$ is a pair of dates $[d_1, d_2]_{\langle C,h\rangle}$ with d_1 and d_2 both in $\langle C, h\rangle$. (The subscript will be omitted when it is obvious which continuum we're talking about.) Informally, the interval denotes the set of all situations in $\langle C, h\rangle$ with dates d such that $d_1 \leq d \leq d_2$. The interval is *nontrivial* if $d_1 \leq d_2$. We also have *open* and *half-open* situation intervals defined and written in exact analogy to the usual concepts of open and half-open intervals on the reals.

A Notation for Processes

Fox and Long, in (Fox & Long 2001a), suggested a notation for autonomous processes. I propose a slightly different one, which I believe is clearer and somewhat easier to formalize. A process is declared using a syntax similar to that for actions. (This notation is part of an overall reform and extension of PDDL called Opt. The Opt Manual (McDermott 2003) lays out the whole language and describes other features of processes that I've omitted in this discussion.)

```
<process-def>
         ::= (:process <name>
         :parameters
             <typed list (variable)>
         <process-def body>)
<process-def body>
         ::=
         [:condition <goal proposition>]
         [:start-effect <effect proposition>]
         [:effect <effect proposition>]
         [:stop-effect <effect proposition>]
```

The syntax `<typed list (variable)>` refers to the PDDL notation exemplified by `(x y - Location r - Truck)`, although here again the full Opt language allows several extensions.

The informal semantics of a process are simple: Whenever the `:condition` of the process is true, the process becomes active. It has immediate effects spelled out by its `:start-effect` field and the `:effect` field. These take time dt. The start-effects occur once, but the *through-effects*, i.e., those specified by the `:effect` field, remain true as time passes. As soon as the process's condition becomes false, the through-effects become false and the stop-effects (described by the `:stop-effects` field) occur, again taking time dt.

The contents of `:effect` field of a process definition is a conjunction of one or more propositions of the form

```
(derivative $x$ $d$)
```

where x and d are numerical fluents, that is, objects of type `(Fluent Number)`.[1] So for a process's effects to remain true is for various quantities to change at the given rates. The rates are not constants in general, so we can express any set of differential equations in this notation.

A formal specification of the semantics of processes is given by detailing the *truth conditions* of a process definition, that is, the constraints mandated by the definition on situation intervals over which the process is *active*. This specification must fit with an overall semantics for plans and facts.

I said above that a plan might be described by listing the times when actions occur, but I'm actually going to describe plans in terms of the method language of Opt. The full method language is rather different from PDDL's, but I'm going to stick to a subset that is close to what PDDL allows. The grammar of plans is simply

```
P ::= A
      | (seq P1 ...Pn)
      | (parallel P1 ...Pn)
A ::= action term
      | (wait-while p)
      | (wait-for q p)
```

where p is a process term and q is an inequality that p presumably affects. An example plan is

```
Plan A =
   (seq (parallel (turn-on faucet1)
                  (plug-up outlet1))
        (wait-for (>= (level tub1)
                      (cm 30))
                  (filling tub1)))
```

Here `filling` is defined as a process thus:

```
(:process filling
    :parameters (b - Tub)
    :vars (f - Faucet l - Outlet)
    :condition (and (faucet-of f b)
                    (outlet-of l b)
                    (faucet-on f))
    :effect
       (and (when (plugged-up l)
                  (derivative
                     (level b)
                     (constant (cm/sec 2))))
            (when (not (plugged-up l))
                  (derivative
                     (level b)
                     (constant (cm/sec 1))))))
```

and the actions are defined thus:

[1] I capitalize types such as `Number` and type functions such as `Fluent`.

```
(:action turn-on
    :parameters (f - Faucet)
    :effect (faucet-on f))

(:action plug-up
    :parameters (l - outlet)
    :effect (plugged-up l))
```

Obviously, these are all simplified for expository purposes. One apparent simplification is that we provide no method for the target agent to test whether the level in `tub1` has reached 30 cm. But in our classical framework, there is no need for such a method; the agent knows exactly when the level will get to that point.

What we want our formal semantics to tell us is that Plan A is executed over any situation interval $[\langle r_1, i_1\rangle, \langle r_2, 0\rangle]$ in which `(turn-on faucet1)` and `(plug-up outlet1)` occur at times $\langle r_1, i_1\rangle$ and $\langle r_1, i_1 + 1\rangle$ (in either order), and the level of `tub1` is < 30 cm at time r_1 and reaches 30 cm at time r_2. It is also executed over any situation interval $[\langle r_1, i_1\rangle, \langle r_1, i_1 + 2\rangle]$, where the `turn-on` and `plug-up` actions are executed as before, and where `(level tub1)` $\geq$ 30 cm at r_1.

The Formal Semantics

Interpretations of Domains with Processes

An *interpretation* of a domain D is a tuple $\langle U_0, T, I_0\rangle$, where U_0 is a function from type symbols in D to sets of objects called *subuniverses*; T is a set of situation continua; and I_0 is a function from the non-type symbols of D to objects in the subuniverses. If $I_0(s) = v$, and s has type τ, then it must be the case that $v \in U_0(\tau)$.

U_0 must obey the following constraints:

$U_0(\texttt{Void}) = \emptyset$

$U_0(\texttt{Boolean}) = \{\textit{true}, \textit{false}\}$

$U_0(\texttt{Integer})$ = the set of all integers

$U_0(\texttt{Number})$ = the set of all real numbers. I treat integers as a subset of the reals, not an alternative data type

$U_0(\texttt{Situation}) =$ $\{s \mid$ *For some* $\langle C, h\rangle \in T$ *and date* $d, C(d) = s\}$

We want to extend U_0 to a function U that gives the meaning of all type expressions, and I_0 to a function I giving the meaning of all formulas and terms. To extend U_0, we need the notion of a *tuple*, of which Opt distinguishes two varieties, `Tup`-tuples and `Arg`-tuples. The former are like lists in Lisp, the latter like ordered n-tuples in mathematics. The difference is that `Tup`-tuples can be of any length, including 0 and 1, while `Arg`-tuples have to have length at least 2. An `Arg`-tuple $\langle x\rangle$ of length 1 is identical to x. An empty `Arg`-tuple is impossible, so we identify the type `(Arg)` with `Void`, the empty type. In this paper we need only `Arg`-tuples, so I concentrate on those.

Both kinds of tuples have designators with named fields, as in

```
(Arg num - Integer &rest strings - String)
```

but in the `Arg` case the names are there only because an `Arg` expression often does double duty as defining a tuple and declaring local variables in an action or process. The subuniverse denoted by `(Arg num - Integer &rest strings - String)` is

$\{\langle i, s_1, \ldots, s_n\rangle \mid i$ *is an integer and each* s_j *is a string*$\}$

which, not surprisingly, is the type of arguments to a an action declared thus:

```
(:action name
    :parameters (num -  Integer
                 &rest strings - String)
    ...)
```

The labels are simply ignored when determining the denotation of the type. We can replace each label with a "don't care" symbol ("_") or omit them entirely. Note that if τ is a type, $U(\texttt{(Arg } \tau\texttt{)}) = U(\tau)$. (I will use the term `Arg`-type for expressions such as the one following the keyword `:parameters` even though the `Arg` flag is missing.)

We extend U to tuples by making $U(\texttt{(Tup ...)})$ denote a `Tup`-tuple and $U(\texttt{(Arg ...)})$ denote an `Arg`-tuple as suggested by these examples. I won't try to fill in the messy details.

Using `Arg`-tuples, we can give a meaning to the type notation `(Fun` τ_r `<-` τ_d`)` of functions from domain type τ_d to range type τ_r. The domain type τ_d is in general an `Arg`-tuple, and $U(\texttt{(Fun } \tau_r \texttt{ <- } \tau_d\texttt{)}) = U(\tau_d) \otimes U(\tau_r)$.

The type `(Fluent` τ`)` is an abbreviation for `(Fun` τ `<- Situation)`. `Prop`, for "proposition," is an abbreviation for `(Fluent Boolean)`. Predicates have types of the form `(Fun Prop <- ...)`.

Recall our intuition that action and process terms denote sets of intervals. For this to be the case, the *subuniverse* that the denotation resides in must be the *powerset* of a set of intervals, written *pow*(S).

There are four kinds of event type, and hence four types of sets of interval sets to contemplate: no-ops, skips, hops, and slides:[2]

1. `Skip`: The type of all actions that take one infinitesimal time unit:

$$U(\texttt{Skip}) = \textit{pow}(\{[\langle r, d\rangle, \langle r, d+1\rangle]_{\langle C,h\rangle} \mid \langle C, h\rangle \in T \textit{ and } d + 1 \leq h(r)\}) = \textit{pow}(\{[d, d + dt]_{\langle C,h\rangle} \mid \langle C, h\rangle \in T\})$$

2. `Hop`: The type of all actions that take more time than a `Skip`:

$$U(\texttt{Hop}) = \textit{pow}(\{[\langle d_1, d_2\rangle]_{\langle C,h\rangle} \mid \langle C, h\rangle \in T,\ d_1, d_2 \textit{ are in } \langle C, h\rangle, \textit{ and there is a } d \textit{ in } \langle C, h\rangle \textit{ such that } d_1 < d < d_2\})$$

[2] In the full Opt implementation, events can have values as well as effects, so we can distinguish, say, `(Skip Integer)` from `(Skip String)`. What I write as `Skip` in this paper would actually be written `(Skip Void)` in Opt, meaning a `Skip` that returns no value.

3. `no-op`: A constant whose value is the only element of subuniverse

   $$U(\texttt{(Con no-op)}) = \{\{[\texttt{d},\texttt{d}]_{\langle \texttt{C},\texttt{h}\rangle} \mid \texttt{d} \textit{ is in } \langle \texttt{C},\texttt{h}\rangle \in \texttt{T}\}\}$$

   that is, the singleton set whose only element is the set of all zero-duration closed situation intervals in T. The name of this type is `(Con no-op)`. Unlike the others, this subuniverse is not the powerset of anything.

   I define $U(\texttt{Step})$ to be $U(\texttt{(Con no-op)}) \cup U(\texttt{Skip}) \cup U(\texttt{Hop})$.

4. `Slide`: The type of autonomous processes. A process must take noninfinitesimal time, so

   $$U(\texttt{Slide}) = \textit{pow}(\{[\langle r_1, i_1\rangle, \langle r_2, i_2\rangle]_{\langle C,h\rangle} \mid \langle C,h\rangle \in T \textit{ and } r_1 < r_2\})$$

We also assume that every expression in Opt is typed. In all our examples imagine a superscript giving the type of the expression, with the proviso that all the type labels are consistent. For instance, the formula

```
(parallel (turn-on faucet1)
          (plug-up outlet1))
```

with type annotations would be

```
(parallel(Fun Action <- (Arg &rest Action))
    (turn-on(Fun Action <- Faucet)
        faucet1Faucet)Action
    (plug-up(Fun Action <- Outlet)
        outlet1Outlet)Action)Action
```

The annotations in this example are consistent because whenever f is labeled `(Fun` τ `<-` α`)`, then in `(`f a`)` a is of type α and `(`f a`)` is of type τ. The proper-typing requirementavoids absurd formulas like `(if 3 (= (not "a")))`. In the implementation, finding a consistent typing is a mostly automatic process, which is not relevant to this paper.

We can now state very simply how to extend I_0 to I, which assigns a denotation to every (properly typed) term of the language. I takes two arguments, a term and an *environment*, which is a total function from variables to ordered pairs $\langle v, y\rangle$, where y is a subuniverse.

- If s is a symbol, $I(s^\tau, E) = I_0(s) \in U(\tau)$.
- If x is a variable, $I(x^\tau, E) = v$ iff $E(x) = \langle v, U(\tau)\rangle$.
- For a functional term,

  $I((f^{\texttt{(Fun } \tau \texttt{ <- } \alpha\texttt{)}} a_1^{\alpha_1} \ldots a_n^{\alpha_n})^\tau, E) = v$
  iff
  $\langle\langle I(a_1^{\alpha_1}, E), \ldots I(a_n^{\alpha_n}, E)\rangle, v\rangle$
  $\in I(f^{\texttt{(Fun } \tau \texttt{ <- } \alpha\texttt{)}}, E)$
  (which is possible only if
  $U(\alpha_1) \otimes \ldots \otimes U(\alpha_n) \subseteq U(\alpha)$)

Because f can be an arbitrary function symbol, we don't need special rules to give the meanings of `(if` p q`)`, `(=` a b`)`, and such. We just need to stipulate that some symbols have the same meaning in all interpretations:

- $I_0(\texttt{if}) = (\lambda\,(x,y)(\lambda\,(s)\; x(s) = \textit{false or } y(s) = \textit{true}))$
- $I_0(\texttt{=}) = (\lambda\,(x,y)(\lambda(s)\; x = y))$
- ... and so forth

In general, the type of a predicate symbol P is `(Fun Prop <-` α`)` for some α, recalling that `Prop` is the type of functions from situations to booleans. We can take `if` to be just another predicate, whose argument type is `(Arg Prop Prop)`. $I(\texttt{if})$ must then be a function from situations to Booleans, which yields true when its first argument yields *false* or its second yields *true* in that situation. Although I use λ here, it's just a shorthand for a set of ordered pairs. I could have said $\{\langle\langle x, y\rangle, \{\langle s, b\rangle \mid b = \textit{true iff } x(s) = \textit{false or } y(s) = \textit{true}\}\rangle\}$

The denotation of "`=`" is a *constant* function on situations; two objects are equal only if they are always equal. The equality tester that tests whether two fluents have the same value in a situation is called `fl=`, with denotation

- $I_0(\texttt{fl=}) = (\lambda\,(f_1, f_2)(\lambda\,(s)\; f_1(s) = f_2(s)))$

The denotation of an action term or a process term must be a set of intervals:

For every primitive action function f, that is, every symbol f defined by an `:action` definition $I_0(f)$ must be of type `(Fun Skip <-` α`)`.
For every process function f, that is, every symbol f defined by a `:process` definition, $I_0(f)$ must be of type `(Fun Slide <-` α`)`.

In both cases, α is the `Arg` type from the `:parameters` of the definition.

I'll deal with domain-dependent functions in a later section. The meanings of `seq` and `parallel` are defined in every domain thus:

- $I_0(\texttt{seq})$

  $= (\lambda\,(a_1, \ldots, a_n)$
  $\{[d_0, d_e]_{\langle C,h\rangle}$
  $\mid \langle C,h\rangle \in T$
  and there exist dates $d_1, \ldots, d_n$
  such that for $j = 1, \ldots, n$,
  $[d_{j-1}, d_j] \in a_j$
  and $d_n = d_e\})$

- $I_0(\texttt{parallel})$

  $= (\lambda\,(a_1, \ldots, a_n)$
  $\{[d_b, d_e]_{\langle C,h\rangle}$
  $\mid \langle C,h\rangle \in T$
  there is a function $s : [1, \ldots, n]$
  $\rightarrow$ *situation intervals*,
  such that for $j = 1, \ldots, n$,
  $s(j) = [d_{j1}, d_{j2}]_{\langle C,h\rangle} \in a_j$
  and $d_b \le d_{j1} \le d_{j2} \le d_e$
  and for some $j_b, j_e \in [1, \ldots, n]$,
  $s(j_b) = [d_b, d_{j_b 2}]$
  and $s(j_e) = [d_{j_e 1}, d_e]$)

Another symbol that must have a standard meaning is `derivative`:

- $U_0(\texttt{derivative})$

  $= U(\texttt{(Fun (Fluent Number) <- (Fluent Number))})$

- I_0(`derivative`) =

  $(\lambda\,(f)$
  $\quad(\lambda\,(s)$ *if there is a* d *such that*
  $\qquad$(*for all* (r, i, C, h)
  $\qquad\quad$ *if* $\langle C, h\rangle \in T,\ s = C(r, i),\ h(r) = i$
  $\qquad\qquad$ *and there is an* r'
  $\qquad\qquad\quad$ *such that* $r < r'$
  $\qquad\qquad\qquad$ *and for all* $r'',\ r < r'' < r'$
  $\qquad\qquad\qquad\quad h(r'') = 0$
  $\qquad\qquad$ *then* $f^+(C)(r) = d$),
  $\qquad$ *then* d
  $\qquad$ *else* 0))

  where $f^+(C)$ is the "right derivative" of f in timeline C measured at time r, that is, the function of time whose value at t is the limit as $\Delta t \to 0$ of

$$\frac{f(C)(r + \Delta t) - f(C)(r)}{\Delta t}$$

This will require a bit of explanation. The same situation can occur in more than one continuum of T. So we first define the derivative at a "situation occurrence," that is, a date $\langle r, i\rangle_{\langle C,h\rangle}$. The derivative can be meaningfully defined only if there is an open interval after $\langle r, i\rangle$ such that no discrete events occur during that interval — i.e., $h(r) = i$ and $h(r'') = 0$ for all times r'' in that interval. In that case $(\lambda\,(t)\ f(C(t, 0)))$ is an ordinary function of time over that interval, which may have a derivative. The derivative of that function we denote by $f^+(C)$.

The remaining detail to fill in is to switch from situation occurrences to situations by requiring $f^+(C)(r)$ to have the same value for all occurrences of $C(r, i)$. Actually, it might be reasonable simply to require that this be the case, because it follows from the *Markov property*, that what happens starting in a situation depends only on what's true in that situation, assuming the target agent doesn't interfere. For now, I don't impose that requirement, but it will usually follow trivially from the axioms of a domain.

Important note: Even though the numerical value of the derivative in s is defined in terms of timelines in which no actions happen, the derivative still has a value at a date $\langle r, i\rangle$ in a timeline $\langle C, h\rangle$ in which an action, or even a series of actions, occurs at $\langle r, i\rangle$, so long as $C(r, i) = s$. You can think of (`derivative` f) as the rate at which the quantity f would change starting at s if it were undisturbed. In some timelines, the disturbance may be sufficient to cause the derivative to disappear Δt after the current situation, that is, at $C(r, i + 1)$, but it is still well defined at s.

Truth and Models

As usual, we want to define a model to be an interpretation of a domain's language that makes all its axioms true. Because propositions change truth value from situation to situation, we must amend that to: A *model* of a PDDL domain is an interpretation $\langle U_0, T, I_0\rangle$ that makes all axioms true in all situations and environments.

For ordinary axioms, we simply translate an axiom A from the `:axiom` syntax of PDDL to the traditional syntax,[3] yielding A', and then test whether $I(A', E)(s)$ is true for all environments E and situtations s in $\langle C, h\rangle$. We use the traditional specification of the interpretation of quantified formulas:

- I((`forall` $(x$ `-` $\alpha)\ P^{\texttt{Prop}}), E)$ = p, an object from subuniverse

  situations $(T) \otimes \{$*true*,*false*$\}$

  such that $p(s)$ = *true* if and only if $I(P, E')(s)$ = *true* for every environment E' that differs from E, if at all, only in assigning a different value, drawn from $U(\alpha)$, to variable x. (That is, $E'(x^\alpha) = \langle v, U(\alpha)\rangle$ for some $v \in U(\alpha)$.)

- Dual formula for `exists` left as an exercise for the reader.

In addition to axioms, we must require that an interpretation make action and process definitions true. I is extended to an action definition thus:

- I((`:action` $a^{\texttt{Fun Skip <-}\ \alpha}$
  `:parameters` r^α
  `:precondition` $p^{\texttt{Prop}}$
  `:effect` $e^{\texttt{Prop}}$),
  E)
  = *true*
  if and only if
  for all $\langle C, h\rangle \in T$,
  and every E' that differs from E,
  if at all, only in the assignments
  of the variables in r,
  and all $d_1 = \langle r, i\rangle$ and $d_2 = \langle r, i + 1\rangle$ in $\langle C, h\rangle$,
  if $I(p^{\texttt{Prop}}, E')(C(r, i))$ = *true*
  and $\langle I(r^\alpha, E'), [d_1, d_2]_{\langle C,h\rangle}\rangle \in I_0(a)$
  then $I(e^{\texttt{Prop}}, E')(C(r, i + 1))$ = *true*

In other words, the action definition is true if whenever I says the action occurs and that its preconditions are true, the effect is true. In this definition, α is an `Arg` type, and $I(r^\alpha, E')$ refers to an instance of the `Arg`-tuple obtained by substituting its variables as specified by E'.

We also need the following condition

- For all pairs of action terms $a_1^{\texttt{Skip}} \neq a_2^{\texttt{Skip}}$, all timelines $\langle C, h\rangle \in T$, and all environments E_1 and E_2, if $[d_1, d_1 + dt]_{\langle C,h\rangle} \in I(a_1, E_1)$ and $[d_2, d_2 + dt]_{\langle C,h\rangle} \in I(a_2, E_2)$, then $d_1 \neq d_2$. In other words, no two actions occur over precisely the same infinitesimal ("skip") interval.

This condition enforces an *interleaving* interpretation of concurrency. Two actions can occur at the same time, but they must still be ordered.

The truth condition on process definitions relies on the following definition:

With respect to an interpretation $\langle U, T, I\rangle$
and an environment E,
an interval $[d_1, d_2]_{\langle C,h\rangle}$ is a
maximal slide in $\langle C, h\rangle$ *over which* p *is true*

[3] Opt allows you to use the traditional syntax for axioms, which is a lot less cumbersome that PDDL's.

if and only if

$[d_1, d_2]$ is a slide,
and for all d, $d_1 < d < d_2$
$I(p, E)(C(d))$ = *true*,
and there is a $d_0 < d_1$ such that
for all d, $d_0 < d < d_1$
$I(p, E)(C(d))$ = *false*,
and there is a $d_3 > d_2$ such that
for all d, $d_2 < d < d_3$
$I(p, E)(C(d))$ = *false*

To preserve flexibility, the definition doesn't say what the truth value of p is at d_1 or d_2. So the interval over which p is true can be open or closed.

Finally, figure 1 shows truth condition for processes. This condition may appear more complex than one would expect, because a conceptual "if and only if":

> A process occurs over an interval if and only if that interval is a maximal slide over which its `:condition` is true.

has had to be broken into an "if" clause and an "only if" clause, to allow the process's boundaries to differ infinitesimally from the boundaries of the maximal slide. The reason for this slop is to enforce another interleaved-concurrency constraint:

- For all pairs of process terms $p_1 \neq p_2$, all timelines $\langle C, h\rangle \in T$ and all environments E_1 and E_2, if $I(p_1, E) = [d_{b1}, d_{e1}]_{\langle C,h\rangle}$ and $I(p_2, E) = [d_{b2}, d_{e2}]_{\langle C,h\rangle}$ then $d_{b1} \neq d_{b2}$ and $d_{e1} \neq d_{e2}$. In other words, no two processes begin or end at precisely the same moment.

The interpretation of the "wait" actions can now be specified:

- $I_0($`wait-while`$) =$

  $(\lambda\,(p)\ \{[d_1, d_2]_{\langle C,h\rangle}$
  $\mid$ *either there is a slide* $[p_b, p_e] \in p$
  such that $p_b \leq d_1 < p_e$ *and* $d_2 = p_e$
  or there is no such slide and $d_1 = d_2\}$)

- $I_0($`wait-for`$) =$

  $(\lambda\,(q, p)\ \{[d_1, d_2]_{\langle C,h\rangle}$
  $\mid$ *there is a slide* $[p_b, p_e] \in p$
  such that $p_b \leq d_1 \leq d_2 < p_e$
  and d_2 *is the first date in* $[d_1, p_e]_{\langle C,h\rangle}$
  such that $q(C(d)) =$ *true*$\}$)

Although there are many details to be fleshed out, we can summarize with this definition:

> A *model* of a PDDL domain D is an interpretation $\langle U_0, T, I_0\rangle$ of the symbols in D such that for every variable-binding environment E
>
> - ⋆ For every axiom A and every date $\langle r, i\rangle$ in $\langle C, h\rangle \in T$, $I(A, E)(s) =$ *true*.
> - ⋆ and For every process or action definition P, $I(P, E)$ = true.
>
> where I is the extension of I_0 outlined above, respecting the interleaving constraints and the required definitions of symbols such as `derivative` and `seq`.

Assessment

I have been thinking about the logic of processes for a long time (McDermott 1982). The contents of this paper are basically a further refinement of those ideas.

The only other attempt I know of to add autonomous processes to PDDL is the proposal by Fox and Long (Fox & Long 2001a) for the AIPS 2002 Planning Competition. The syntax of their notation differs from mine only in detail.[4] The semantics they propose is very different. They base it on the theory of hybrid automata (Henzinger 1996), so that to every domain and initial situation there corresponds an automaton. Finding a plan is finding a path through the states of the automaton. The main virtue of their approach is also its main defect: They are determined to preserve finiteness properties exploited by many planning algorithms, especially that there are only a finite number of plan states, and that the branching factor at each plan state is finite. The states of the hybrid automaton are all possible sets of ground atomic formulas in the language. For the state set to be finite, atomic formulas with numeric arguments must be separated out and treated in a special way. The whole apparatus becomes quite unwieldy.[5]

I believe that maintaining finiteness properties required by some current planners should be rejected as a constraint on domain-description languages. If some problems cannot be solved by some planners, so be it. It would be better to make it the responsibility of any planning system to decide whether a problem is beyond its scope. Of course, requirements flags can provide a broad-brush portrait of what a planner must be able to handle in order to solve a problem, so it's important to keep the set of flags up to date as the language evolves. But it's asking too much for PDDL to fit the capabilities of some set of existing planners exactly. In the first planning competition, there were loud votes for a `:length` field to be included in every problem description, specifying a bound on the length of a solution, because several systems had to guess a length in order to get started. We reluctantly acceded to that demand, but the `:length` field has since been taken out on the grounds that it's an arbitrary hint to a special class of planners. Trying to base the semantics of PDDL processes on finite-state hybrid automata strikes me as an even worse accommodation to the needs of a "special-interest group."

There is an important issue raised by (Fox & Long 2001a), namely, how do you check the correctness of a problem solution when real numbers are involved? A complete answer is beyond the scope of this paper, but I believe the semantic framework laid out here lends itself to the idea of "approxi-

[4] Which means it will be the subject of bitter and unending debate before the next competition!

[5] I should acknowledge the regrettable fact that a specification of the formal semantics of anything is generally clear only to the person that wrote it, so my proposal is probably as opaque to Maria and Derek as theirs is to me.

- $I($(`:process` a `(Fun Slide <-` α`)`
  `:parameters` r^{α}
  `:condition` $p^{\texttt{Prop}}$
  `:start-effect` $b^{\texttt{Prop}}$
  `:effect` $e^{\texttt{Prop}}$
  `:stop-effect` $w^{\texttt{Prop}}$`)`,
  E)
  = *true*
  if and only if
  for all $\langle C, h\rangle \in T$,
  and every E' that differs from E,
  if at all, only in the assignments
  of the variables in r,
  and every interval $[d_1, d_2]$ in $\langle C, h\rangle$
  where $d_1 = \langle r_1, i_1\rangle$ and $d_2 = \langle r_2, i_2\rangle$,
  (a) If $\langle I(r^{\alpha}, E'), [d_1, d_2]_{\langle C,h\rangle}\rangle \in I_0(a)$
  then there is a maximal slide $[\langle r_1, i_{s1}\rangle, \langle r_2, i_{s2}\rangle]_{\langle C,h\rangle}$ over which p is true
  with respect to $\langle U, T, I\rangle$ and E'
  such that $i_{s1} \leq i_1$ and $i_{s2} \leq i_2$
  and $I(b^{\texttt{Prop}}, E')(C(r_1, i_1 + 1)) =$ *true*
  and $I(w^{\texttt{Prop}}, E')(C(r_2, i_2 + 1)) =$ *true*
  and
  (b) If $[d_1, d_2]$ is a maximal slide over which p is true
  with respect to $\langle U, T, I\rangle$ and E'
  then there are dates $d_1' = \langle r_1, i_{p1}\rangle$ and $d_2' = \langle r_2, i_{p2}\rangle$,
  such that $\langle I(r^{\alpha}, E'), [d_1', d_2']_{\langle C,h\rangle}\rangle \in I_0(a)$
  and $i_{p1} \geq i_1$ and $i_{p2} \geq i_2$
  and $\langle I(r^{\alpha}, E'), [d_1', d_2']\rangle \in I_0(a)$
  and $I(b^{\texttt{Prop}}, E')(C(r_1, i_{p1} + 1)) =$ *true*
  and $I(w^{\texttt{Prop}}, E')(C(r_2, i_{p2} + 1)) =$ *true*

Figure 1: Truth Condition for Process Definitions

mate simulation" of a plan. The exact dates at which events occur is not important as long as every step the planner takes is feasible when it takes it, and the plan simulator stops in a state where the goal condition is true. If a problem includes an objective function, we evaluate it in that final state. The value may be slightly different from the true mathematically attainable minimum, but all we require is that it be comparable to the results obtained by other planners. The only tricky part is how to deal with equalities in process specifications. If the `:condition` of a process includes a formula `(not (= x y))`, then the process stops when x equals y. Finding the precise instant when that occurs is usually impossible. To fix this problem, all the planner has to do is convert the "$=$" goal relationship to a "$\geq$" or "$\leq$" by observing whether $x < y$ or $x > y$ when the process starts, and do the best job it can of finding the earliest time when the new inequality becomes true. A similar trick will work for determining when the action `(wait-for (= x y) p)` ends.

There is also the easily overlooked issue of plan execution. The assumption that actions take infinitesimal time is of course absurd when applied to a physically possible action. Obviously, there must be some temporal grain size ϵ specified to the executor such that any action must take less than ϵ, and every process must take more than ϵ. Otherwise, the domain model is simply inappropriate.

Finally, it should be noted that, because action and process definitions have truth conditions, they can be "reverse engineered" to yield formulas with the same truth conditions. These would not be legal PDDL axioms, because they would have to mention multiple situations explicitly, whereas PDDL axioms refer implicitly to one and only one situation. However, to translate PDDL to Kif (**?**), which has no built-in action-definition syntax, these axioms could be very useful. There existence also makes it clear that PDDL is not "predicate calculus + actions"; it's just "predicate calculus + useful macros for writing action-definition axioms."

References

Fox, M., and Long, D. 2001a. Pddl. + level 5: An Extension to PDDL2.1 for Modeling Planning Domain with Continuous Time-dependent Effects available at `http://www.dur.ac.uk/d.p.long/pddllevel5.ps.gz`.

Fox, M., and Long, D. 2001b. Pddl. 2.1: An Extension to PDDL for Expressing Temporal Planning Domains available at `http://www.dur.ac.uk/d.p.long/pddl2.ps.gz`.

Henzinger, T. 1996. The theory of hybrid automata. In *Proc. Annual Symposium on Logic in Computer Science*, 278–292.

McDermott, D. 1982. A temporal logic for reasoning about processes and plans. *Cognitive Science* 6:101–155.

McDermott, D. 1985. Reasoning about plans. In Hobbs, J., and Moore, R. C., eds., *Formal Theories of the Commonsense World*. Ablex Publishing Corporation. 269–317.

McDermott, D. 2003. Draft opt manual. Available at `http://www.cs.yale.edu/~dvm/papers/opt-manual.ps`.

Robinson, A. 1979. *Selected papers of Abraham Robinson: Vol. II. Nonstandard analysis and philosophy*. Yale University Press.

Extending PDDL to Model Stochastic Decision Processes

Håkan L. S. Younes
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
lorens@cs.cmu.edu

Abstract

We present an extension of PDDL for modeling stochastic decision processes. Our domain description language allows the specification of actions with probabilistic effects, exogenous events, and actions and events with delayed effects. The result is a language that can be used to specify stochastic decision processes, both discrete-time and continuous-time, of varying complexity. We also propose the use of established logic formalisms, taken from the model checking community, for specifying probabilistic temporally extended goals.

Introduction

A standard domain description language, PDDL (McDermott 2000; Fox & Long 2002b), for deterministic planning domains and the biannual International Planning Competition, first held in 1998, have resulted in a large library of benchmark problems enabling direct comparisons of different deterministic planners. For the 4th International Planning Competition to be held in 2004, there are plans to include a track for probabilistic planners. This will require a domain description language for specifying probabilistic planning domains. In this paper, we propose such a domain description language that in many ways can be seen as an extension of PDDL.

We start by introducing a PDDL-like syntax for specifying actions with probabilistic effects, which allows us to define *Markov decision processes* (MDPs). We then go on to introduce exogenous events, as well as actions and events with random delay. The result is a domain description language that can be used for specifying a wide range of stochastic decision processes, from MDPs to *generalized semi-Markov decision processes* (GSMDPs). A GSMDP can be viewed as the composition of concurrent *semi-Markov decision processes* (SMDPs), and captures the essential dynamical structure of a *discrete event system* (Glynn 1989).

A discrete event system consists of a set of states S and a set events E. At any point in time, the system occupies some state $s \in S$. The system remains in state s until the occurrence of an event $e \in E$, at which point the system instantaneously transitions to a state s' (possibly the same state as s). Our domain description language can be used to specify both continuous-time and discrete-time discrete event systems. By including the process concept from level 5 of PDDL+ (Fox & Long 2002a), we could also specify stochastic hybrid systems, but that is beyond the scope of this paper.

As a formalism for specifying probabilistic goal conditions we propose PCTL (Hansson & Jonsson 1994) for discrete-time domains and CSL (Baier, Katoen, & Hermanns 1999) for continuous-time domains. This permits the specification of planning deadlines and maintenance and prevention goals, in addition to the traditional achievement goals. The benefit of using established logic formalisms for goal specification is that we can take advantage of recent developments in probabilistic model checking for efficient plan verification.

We leave the representation of plans open. The sole focus of this paper is the representation of probabilistic planning domains.

Actions with Probabilistic Effects

An important aspect of stochastic decision processes is that actions can have probabilistic effects. We adopt a model of stochastic actions that is a variation of *factored probabilistic STRIPS operators* proposed by Dearden & Boutilier (1997). A stochastic action a consists of a precondition ϕ and a consequence set $C = \{c_1, \ldots, c_n\}$. Each consequence c_i has a trigger condition ϕ_i with a corresponding effects list $\mathcal{E}_i = \langle p^i_1, E^i_1; \ldots; p^i_{k_i}, E^i_{k_i} \rangle$, where E^i_j is a set of literals and $p^i_j \in [0, 1]$ is a probability associated with the jth literal set. We require that $\sum_{j=1}^{k_i} p^i_j = 1$.

Semantics

In order for an action with precondition ϕ to be applicable in a state s, ϕ must hold in s. A state is a set of atoms that hold and bindings of functional expressions to rational values. We propose that executing an action a whose precondition is not satisfied be given the meaning that a has no effects (cf. Kushmerick, Hanks, & Weld 1995), instead of this being a violation as is the case for deterministic actions in PDDL. The precondition ϕ can in this way be viewed as a factored trigger condition common to all consequences in C. The semantics of stochastic actions can then be stated as follows.

When applying a stochastic action $a = \langle \phi, C \rangle$ to a state s, an effect set is selected for each consequence $c_i \in C$.

Let $x(i) \in [1, k_i]$ denote the index of the selected effect set for c_i, with $x(i)$ being a sample of a random variable X_i such that $\Pr[X_i = j] = p^i_j$. Let the *acting* effect set for a consequence c_i be

$$\tilde{E}^i_{x(i)} = \begin{cases} E^i_{x(i)} & \text{if } s \models \phi \wedge \phi_i \\ \emptyset & \text{otherwise} \end{cases} .$$

The acting effect set for action a is then the union of acting effect sets for the consequences:

$$\tilde{E} = \bigcup_{i=1}^{n} \tilde{E}^i_{x(i)}$$

We divide $\tilde{E}$ into disjoint sets: $\tilde{E}^+$ representing positive literals, $\tilde{E}^-$ representing negative literals, and $\tilde{E}^u$ representing effects updating value bindings for functional expressions. The result of applying the stochastic action a to the state s is a state s' with atoms $\big(atoms(s) \setminus \{p : \neg p \in \tilde{E}^-\}\big) \cup \tilde{E}^+$ and bindings of functional expressions to values updated in accordance with $\tilde{E}^u$.

We require that consequences with mutually consistent trigger conditions have *commutative* effects. This means that the successor state is the same after applying an action to a state s regardless of the order in which the acting effect sets of enabled consequences are applied to s.

Consequences in our stochastic action model is closely related to *action aspects* in the model of Dearden & Boutilier. Each consequence $c_i = \langle \phi_i, \mathcal{E}_i \rangle$ of an action with precondition ϕ corresponds to an action aspect with discriminant set $\{\phi \wedge \phi_i, \neg(\phi \wedge \phi_i)\}$ and effects lists $\mathcal{E}_i$ and $\langle 1, \emptyset \rangle$. The condition that mutually consistent discriminants taken from distinct aspects of an action have effects lists with no common atoms corresponds to our requirement of commutative effects for consequences with mutually consistent trigger conditions.

Syntax

Stochastic actions can be specified by extending the PDDL syntax for action effects with a probabilistic construct inspired by Bonet & Geffner (2001). Figure 1 shows the proposed extension. The syntax we propose does not allow nested probabilistic statements in effects lists or conditional effects inside probabilistic statements, which is in line with the design decision for PDDL2.1 to disallow nesting of conditional effects. Such language constructs would not add any expressiveness.

As it stands, there is a clear correspondence between the syntax and the representation of stochastic actions introduced above. An effects list is specified as

(probabilistic p^i_1 E^i_1 ... $p^i_{k_i}$ $E^i_{k_i}$).

The above statement also represents a consequence with a trigger condition $\phi_i = true$. Consequences with non-trivial trigger conditions are specified using conditional effects:

(when ϕ_i (probabilistic p^i_1 E^i_1 ... $p^i_{k_i}$ $E^i_{k_i}$))

Figure 2 gives a specification in the extended PDDL of the stochastic move action used by Dearden & Boutilier as an example. A statement such as

```
<effect>      ::= <d-effect>
<effect>      ::= (and <effect>*)
<effect>      ::= (forall (<typed list(variable)>) <effect>)
<effect>      ::= (when <GD> <d-effect>)
<d-effect>    ::= (probabilistic <prob-eff>+)
<d-effect>    ::= <a-effect>
<prob-eff>    ::= <probability> <a-effect>
<a-effect>    ::= (and <p-effect>*)
<a-effect>    ::= <p-effect>
<p-effect>    ::= (not <atomic formula(term)>)
<p-effect>    ::= <atomic formula(term)>
<p-effect>    ::= (<assign-op> <f-head> <f-exp>)
<probability> ::= Any rational number in the interval [0, 1].
```

Figure 1: PDDL extension for probabilistic effects.

```
(:action move
         :parameters ()
         :effect (and (when (office)
                         (probabilistic 0.9 (not (office))))
                      (when (not (office))
                         (probabilistic 0.9 (office)))
                      (when (and (rain) (not (umbrella)))
                         (probabilistic 0.9 (wet)))))
```

Figure 2: Specification of stochastic move action in probabilistic PDDL.

```
(probabilistic 0.9 (wet))
```

with the probabilities not adding up to 1 is meant as a syntactic sugar for

```
(probabilistic 0.9 (wet) 0.1 (and)),
```

where `(and)` represents an empty effect set.

Numeric effects can be used in combination with probabilistic effects, although this could result in a stochastic process with an infinite state space. We therefore propose the introduction of a bounded integer type, `(integer` *low high*`)`, in addition to the standard PDDL type, `number`, for functional expressions. This provides a straightforward way of ensuring a finite state space. For example,

```
(:functions (power ?x) - (integer 0 10))
```

effectively defines an integer state variable $power_x \in [0, 10]$ for each object x in the domain.

Expressiveness

For now, we assume a simple discrete model of time, where time is progressing in unit steps with each state transition (execution of an action). We can model discrete-time Markov decision processes (MDPs) using stochastic actions as defined in this section. Later on we will consider richer time and action models that will allow us to model more complex stochastic decision processes.

Exogenous Events

Boutilier, Dean, & Hanks (1999) make a distinction between *implicit-event models* where the effects of the environment are factored into the representation of stochastic actions, and *explicit-event models* where change caused by the environment is modeled separately from change caused by actions selected for execution by the decision maker. We have so far only provided the means for specifying implicit-event models. It is sometimes convenient, however, to model environmental effects separate from effects of consciously chosen

actions. For this purpose we introduce the notion of an *exogenous event*, $e = \langle \phi, C \rangle$, having the same structure as a stochastic action.

Semantics

An exogenous event e is given the same semantics as a stochastic action, except that the triggering of e is beyond the control of the decision maker. When in a state s, any action chosen for s and all events with a precondition ϕ that holds in s are applied to s, producing a successor state s' at the next time step.

With exogenous events in the domain model it becomes possible to have an action and one or more exogenous events triggering within the same unit time interval, and the successor state s' can then depend on the order in which the action and the events are applied to the current state s. We could require events and actions that can be enabled simultaneously to be commutative, meaning that the successor state is independent of the order in which the events and actions are applied to s, but it would be hard for a domain designer to adhere to this requirement when constructing a large domain with many exogenous events. Instead we choose to assign an equal probability to all orderings of enabled event and actions in a state. So, for example, if an action adding the atomic proposition a and an event deleting a is enabled in a state s, then there is a 0.5 probability of a holding in the next state.

Boutilier, Dean, & Hanks (1999) consider other ways of dealing with simultaneity, but these typically requiring additional information from the domain designer. The fact, though, is that simultaneity almost always is an artifact of using a discrete-time model for an inherently continuous-time stochastic process, and the probability of two events triggering at exactly the same time becomes zero if we work directly with a continuous-time model.

Syntax

We propose using the same syntax for specifying exogenous events as stochastic actions, except that the keyword `:event` is used instead of `:action`. The `:event` keyword was introduced in level 5 of PDDL+ (Fox & Long 2002a) for the specification of deterministic events, and our exogenous events can be viewed as stochastic extensions of PDDL+ level 5 events.

We can break the stochastic move action in Figure 2 into an action modeling intended effects and an exogenous event modeling environmentally triggered effects. Figure 3 shows how this would be done.

Expressiveness

The addition of exogenous events does not add to the expressiveness of our specification language: we can still only model discrete-time MDPs. At this point it is merely added for the convenience of the modeler, but once we consider more complex processes we will see that the effects of exogenous events cannot in a reasonable way be factored into the effects of actions.

```
(:action move
         :parameters ()
         :effect (and (when (office)
                        (probabilistic 0.9 (not (office))))
                      (when (not (office))
                        (probabilistic 0.9 (office)))))

(:event make-wet
        :parameters ()
        :precondition (and (rain) (not (umbrella)))
        :effect (probabilistic 0.9 (wet)))
```

Figure 3: Partial specification of explicit-event model in probabilistic PDDL.

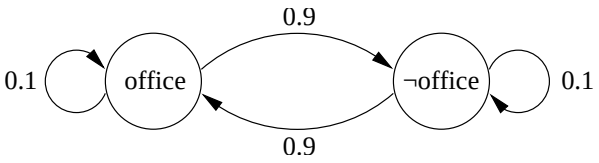


Figure 4: State-transition model for executing the stochastic move action of Figure 3.

Delayed Actions and Events

Consider the stochastic move action of Figure 3 in isolation. Figure 4 shows the state-transition model for executing this action both when in the office and when not in the office. The stochastic move action is executed at every time step and has a 0.9 probability of succeeding each time. The time spent in a state before the action succeeds is a random variable with a geometric distribution $G(0.9)$. Instead of thinking of the stochastic move action as being executed at every time step, we can think of it as being executed once in each state but with a delayed effect. The delay is in this case a random variable with distribution $G(0.9)$. Figure 5 shows the state-transition model for a delayed move action, where probability distributions instead of probabilities are associated with each state transition.

A delayed action a is a triple $\langle \phi, C, F(t) \rangle$, where ϕ is the enabling condition of a, C is the consequence set of a defined in the same way as for stochastic actions, and $F(t)$ is the cumulative distribution function (cdf) for the delay from when a is enabled until it triggers. We require that $F(0) = 0$, meaning that an action must trigger strictly after it becomes enabled. Delayed exogenous events are defined analogously. A regular stochastic action (exogenous event) can be viewed as a delayed action (event) with a cdf satisfying the condition $F(1) = 1$, i.e. it always triggers within one time unit from when it is enabled,but with no additional assumptions being made regarding the shape of $F(t)$.

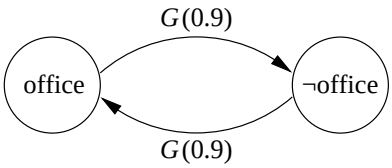


Figure 5: State-transition model for executing a delayed move action.

```
<delayed-action-def> ::= (:delayed-action <name>
                          :parameters (<typed list<variable>)
                          :delay <delay-distribution>
                          [:condition <GD>]
                          [:effect <effect>])
<delayed-event-def>  ::= (:delayed-event <name>
                          :parameters (<typed list<variable>)
                          :delay <delay-distribution>
                          [:condition <GD>]
                          [:effect <effect>])
<delay-distribution> ::= Any distribution s.t. Pr[delay ≤ 0] = 0.
```

Figure 6: PDDL extension for delayed actions and events.

Semantics

For now, let us assume that all delay distributions are *memoryless*. We will consider general delay distribution in the next section. The probability distribution of a random variable X is memoryless if $\Pr[X > t + \Delta t | X > t] = \Pr[X > \Delta t]$ for all $t, \Delta t \geq 0$. For an action a with a memoryless delay distribution this means that if a has been enabled for t time units without triggering, then the remaining delay has the same distribution as if a had just been enabled. The geometric distribution mentioned earlier in this section is a memoryless distribution, and so is its continuous analog: the exponential distribution. The semantics of delayed actions and events with memoryless delay distributions is as follows.

Assume we are entering state s at time t. Any action chosen by the decision maker to be enabled in s and all events with a condition ϕ holding in s race to trigger first. Let e^* be the event or action with the shortest delay in s and let d^* be the delay of e^* in s. We then get the successor state s' at time $t + d^*$ by applying e^* to s. An action or event enabled in s may still be enabled in s', but this is inconsequential when all delay distributions are memoryless. If an action or event did not trigger in s and is not enabled in s', then that action or event is simply canceled. If multiple events or actions have minimum delay d^* in s, then all those events and actions are simultaneously applied to s to produce s' at time $t + d^*$.

Syntax

The proposed syntax for specifying delayed actions and events is given in Figure 6. Delayed actions can be viewed as a stochastic variation of the deterministic durative actions available in PDDL+. A delayed action with a deterministic delay distribution $D(x)$ and enabling condition ϕ corresponds to a durative action with duration x, invariant condition ϕ, and effects associated with the end of the durative action.

Figure 7 shows the partial specification of an explicit-event model with delayed actions and events.

Expressiveness

By just considering memoryless delay distributions, we are nevertheless only able to model MDPs. With geometric delay distributions we have a discrete-time MDP, while with exponential delay distributions we have a continuous-time MDP. Moreover, exogenous events still do not add expressiveness as they can easily be factored into the representation

```
(:delayed-action move
                 :parameters ()
                 :delay (geometric 0.9)
                 :effect (and (when (office)
                                 (not (office)))
                              (when (not (office))
                                 (office))))

(:delayed-event make-wet
                :parameters ()
                :delay (geometric 0.9)
                :precondition (and (rain) (not (umbrella)))
                :effect (wet))
```

Figure 7: Partial specification of explicit-event model with delayed actions and events.

```
(:delayed-action move
                :parameters ()
                :delay (geometric 0.99)
                :effect (and (when (office)
                                (probabilistic 10/11
                                               (not (office))))
                             (when (not (office))
                                (probabilistic 10/11 (office)))
                             (when (and (rain) (not (umbrella)))
                                (probabilistic 10/11 (wet))
```

Figure 8: Partial specification of implicit-event model with delayed move action.

of actions. We can combine all actions and events enabled in a state s into one single action.

For a discrete-time model, let $G(p_i)$ be the distribution of the ith action or event enabled in a state s. The probability of at least one action or event triggering in s after one time unit is $p = 1 - \prod_i (1 - p_i)$, and the probability of the ith action or event triggering after one time unit given that something triggers is p_i/p. We can therefore represent all actions and events enabled in s with a single action having delay distribution $G(p)$, and with the effects of the ith action or event weighted by p_i/p. Figure 8 shows the implicit-event representation of the action and event in Figure 7.

A similar transformation of an explicit-event model to an implicit-event model can be made for continuous-time models. Let $E(\lambda_i)$ be the delay distribution of the ith action or event enabled in a state s, with λ_i being the *rate* of the exponential delay distribution. The rate at which some action or event triggers in s is $\lambda = \sum_i \lambda_i$, and the probability of the ith action or event triggering before any other action or event is λ_i/λ. We can use this information to construct a single action representing all actions and events enabled in s. Part of a continuous-time explicit-event model is given in Figure 9 and the corresponding implicit-event model is given in Figure 10.

General Delay Distributions

Although memoryless distributions can be used to adequately model many real-world phenomena, they are many times insufficient for accurately capturing certain aspects of stochastic processes. Hardware failure, for example, is often more accurately modeled using a Weibull distribution rather than an exponential distribution (Nelson 1985).

Figure 11 shows an example domain with general delay distributions for actions and events. Here, we have associated a uniform delay distribution with the move action and a Weibull distribution with the make-wet event.

```
(:delayed-action move
                 :parameters ()
                 :delay (exponential 3)
                 :effect (and (when (office)
                                 (not (office)))
                              (when (not (office))
                                 (office))))

(:delayed-event make-wet
                :parameters ()
                :delay (exponential 2)
                :precondition (and (rain) (not (umbrella)))
                :effect (wet))
```

Figure 9: Partial specification of continuous-time explicit-event model with delayed actions and events.

```
(:delayed-action move
                :parameters ()
                :delay (exponential 5)
                :effect (and (when (and (office)
                                        (rain) (not (umbrella)))
                                   (probabilistic 0.6 (not (office))
                                                  0.4 (wet)))
                             (when (and (office)
                                        (or (not (rain))
                                            (umbrella)))
                                   (probabilistic 0.6
                                                  (not (office))))
                             (when (and (not (office))
                                        (rain) (not (umbrella)))
                                   (probabilistic 0.6 (office)
                                                  0.4 (wet)))
                             (when (and (not (office))
                                        (or (not (rain))
                                            (umbrella)))
                                   (probabilistic 0.6 (office)))))
```

Figure 10: Partial specification of continuous-time implicit-event model with delayed move action.

The state-transition model for the action and event considered separately is depicted in Figure 12. Each state-transition model corresponds to a *semi-Markov process* (SMP; Howard 1971). However, when viewing the domain model as a whole—as the composition of concurrent SMPs (Figure 13)—we have what is called a *generalized semi-Markov process* (GSMP; Glynn 1989).

GSMPs differ from SMPs in that the delay distribution of an enabled event can depend not only on the current state but on the entire path taken to that state. Consider the state-transition model in Figure 13. Assume that we start out not being in the office and not being wet (upper right state). The move action and the make-wet event are both enabled. Say that the make-wet event happens to trigger before the move action after having been in the current state for 3 time units, causing a transition to the lower right state where the move

```
(:delayed-action move
                 :parameters ()
                 :delay (uniform 0 6)
                 :effect (and (when (office)
                                 (not (office)))
                              (when (not (office))
                                 (office))))

(:delayed-event make-wet
                :parameters ()
                :delay (weibull 2)
                :precondition (and (rain) (not (umbrella)))
                :effect (wet))
```

Figure 11: Partial specification of continuous-time explicit-event model with general delay distributions.

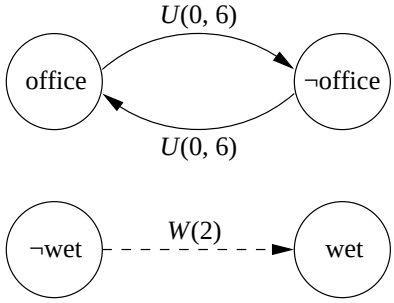


Figure 12: State-transition model for move action (above) and make-wet event (below).

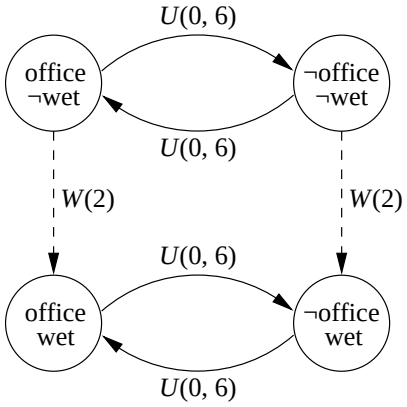


Figure 13: Composite state-transition model for move action and make-wet event.

action is still enabled. Because the move action already has been enabled for 3 time units in the previous state without triggering, the delay distribution for the move action in the new state is in effect $U(0, 3)$. If, on the other hand, we had entered the lower right state by executing the move action when being wet in the office (lower left state), then the delay distribution of the move action would have been $U(0, 6)$. This history dependence occurs because we are not using memoryless distributions.

Generalized Semi-Markov Processes

GSMPs, first introduced by Matthes (1962), have the following components:

- A set S of states.
- A set E of events.
- For each state $s \in S$, a set $E(s) \subset E$ of events enabled in s.
- For each pair $\langle s, e \rangle \in S \times E(s)$, a probability distribution $p(s'; s, e)$ over S giving the probability of the next state being s' if event e triggers in state s.
- For each event e, a cdf $F(t; e)$, s.t. $F(0; e) = 0$, giving the probability that e has triggered t time units after it was last enabled.

A generalized semi-Markov *decision* process (GSMDP) is a GSMP with a subset of E being controllable events (i.e. actions).

By allowing general delay distributions, we can specify GSMDPs using the proposed extensions of PDDL. The preconditions of actions and events determine the sets $E(s)$, the probability distributions $p(s'; s, e)$ can be derived from conditional probabilistic effects, and the distributions $F(t; e)$ correspond directly to the delay distributions of actions and events.

The semantics of a GSM(D)P is best described in terms of discrete event simulation (Shedler 1993). We associate a real-valued clock $c(e)$ with each event $e \in E$ that indicates the time remaining until e is scheduled to occur. The system starts in some initial state s with events $E(s)$ enabled. For each enabled event $e \in E(s)$, we sample a duration according to the cdf $F(t; e)$ and set $c(e)$ to the sampled value. Let e^* be the event in $E(s)$ with the shortest duration, and let $c^* = c(e^*)$. The event e^* becomes the triggering event in s. When e^* triggers, we sample a next state s' according to the probability distribution $p(s'; s, e^*)$ and update the clock for each event $e \in E(s')$ enabled in the next state as follows:

- If $e \in E(s) \setminus \{e^*\}$, then subtract c^* from $c(e)$.
- If $e \notin E(s) \setminus \{e^*\}$, then sample a new duration according to the cdf $F(t; e)$ and set $c(e)$ to the sampled value.

An event enabled in s but not in s' will have its clock reset next time it becomes enabled. The first condition above highlights the fact that GSM(D)Ps are non-Markovian, as the durations for events are not independent of the history. The system evolves by repeating the process of finding the triggering event in the current state, and updating clock values according to the scheme specified above.

Summary of Expressiveness

Figure 14 shows the hierarchy of stochastic decision processes that can be specified using our proposed probabilistic extension of PDDL. The most general class is GSMDPs, which allow for concurrency, general delay distributions, and probabilistic effects. A GSMDP is an SMDP if no action or event can be enabled in consecutive states without triggering, and it is an MDP if all delay distributions are memoryless (Glynn 1989).

With general delay distributions, there is no longer an easy way to factor the effects of exogenous events into the effects of actions. The delay distribution of the combined action would be the distribution of the minimum of the individual delay distributions. The minimum of exponential distributions with rates λ_i is simply an exponential distribution with rate $\sum_i \lambda_i$, but for general distributions there is typically no simple distribution for the minimum. Neither is it in general possible to obtain a closed-form expression for the probability of a specific event triggering first. Exogenous events is therefore more than just a modeling convenience once we allow general delay distribution.

A GSMDP can be approximated with an MDP by approximating each general delay distribution with a phase-type distribution. Figure 15 shows two commonly used phase-type distributions. The phase of each approximating distribution becomes part of the state-space for the MDP, which potentially can lead to state-space explosion. It is therefore desirable to approximate a general distribution with a phase-type distribution having few phases, while still matching at least the first three moments of the general distribution (Osogami & Harchol-Balter 2003).

We can model both discrete-time and continuous-time stochastic decision processes. German (2000) discusses techniques for approximating a continuous-time Markov process with a discrete-time Markov process that can be used in case a discrete-time model is preferred.

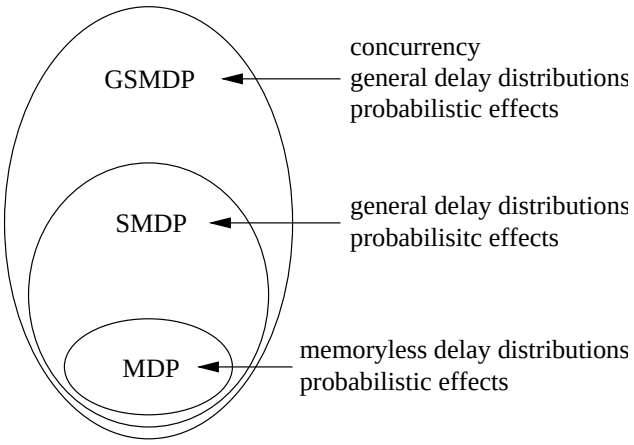
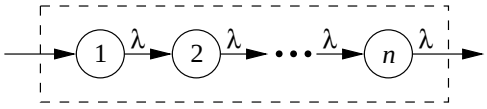
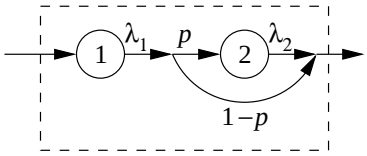


Figure 14: A hierarchy of stochastic decision processes.



(a) An n-phase Erlang distribution.



(b) A two-phase Coxian distribution.

Figure 15: Examples of phase-type distributions.

Probabilistic Planning Problems

A probabilistic planning problem is commonly specified as an initial state s_0 (or initial distribution over states), a set of goal states G, and a probability threshold p. A plan is considered a solution to a problem if the set of paths from s_0 to states in G has probability $p' \geq p$ (Farley 1983; Blythe 1994; Goldman & Boddy 1994; Kushmerick, Hanks, & Weld 1995; Lesh, Martin, & Allen 1998). Drummond &

Bresina (1990) suggest the need for maintenance and prevention goals, in addition to goals of achievement traditionally considered in probabilistic planning, and propose a branching temporal logic for specifying temporally extended goals. Dean *et al.* (1995) embrace a similar view, but take a decision-theoretic approach with goals encoded using utility functions.

We propose a definition of probabilistic planning problems closely related to that of Drummond & Bresina, adopting PCTL (Hansson & Jonsson 1994) and its continuous-time analog CSL (Baier, Katoen, & Hermanns 1999) as a logic for specifying probabilistic temporally extended goals. We define a planning problem as an initial state s_0 and a CSL (PCTL) formula ϕ, and a solution is a plan that makes ϕ true in s_0.

Probabilistic Temporally Extended Goals

The syntax of CSL (PCTL) is defined as

$$\phi ::= true \left| a \right| \phi \wedge \phi \left| \neg\phi \right| \mathcal{P}_{\bowtie p}\left(\phi \, \mathcal{U}^{\leq t} \, \phi\right) \left| \mathcal{P}_{\bowtie p}\left(\phi \, \mathcal{U} \, \phi\right), \right.$$

where a is an atomic proposition, $p \in [0,1]$, $t \in \mathbb{R}_{\geq 0}$ ($t \in \mathbb{Z}_{\geq 0}$ for PCTL), and $\bowtie \in \{\geq, >\}$.

Regular logic operators have their usual semantics. A probabilistic formula $\mathcal{P}_{\bowtie p}(\rho)$ holds in a state s if and only if the set of paths starting in s and satisfying the path formula ρ is p' and $p' \bowtie p$. A path of a stochastic process is a sequence of states and holding times:

$$\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots$$

A path formula $\phi_1 \; \mathcal{U}^{\leq t} \, \phi_2$ ("time-bounded until") holds over a path σ if and only if ϕ_2 holds in some state s_i such that $\sum_{j=0}^{i-1} t_j \leq t$ and ϕ_1 holds in all state s_j for $j < i$. The formula $\phi_1 \, \mathcal{U} \, \phi_2$ ("until") holds over a path σ if and only if $\phi_1 \, \mathcal{U}^{\leq t} \, \phi_2$ holds over σ for some $t \geq 0$.

We can derive other common logic and path operators, for example:

$$\begin{aligned}
false \equiv & \neg true \\
\phi_1 \vee \phi_2 \equiv & \neg(\neg\phi_1 \wedge \neg\phi_2) \\
\mathcal{P}_{\bowtie p}\left(\Diamond^{\leq t} \, \phi\right) \equiv & \mathcal{P}_{\bowtie p}\left(true \, \mathcal{U}^{\leq t} \, \phi\right) \\
\mathcal{P}_{\geq p}\left(\phi_1 \, \mathcal{W}^{\leq t} \, \phi_2\right) \equiv & \neg\mathcal{P}_{> 1-p}\left(\neg\phi_2 \, \mathcal{U}^{\leq t} \, \neg(\phi_1 \vee \phi_2)\right) \\
\mathcal{P}_{> p}\left(\phi_1 \, \mathcal{W}^{\leq t} \, \phi_2\right) \equiv & \neg\mathcal{P}_{\geq 1-p}\left(\neg\phi_2 \, \mathcal{U}^{\leq t} \, \neg(\phi_1 \vee \phi_2)\right) \\
\mathcal{P}_{\bowtie p}\left(\Box^{\leq t} \, \phi\right) \equiv & \mathcal{P}_{\bowtie p}\left(\phi \, \mathcal{W}^{\leq t} \, false\right)
\end{aligned}$$

Intuitively, $\Diamond^{\leq t} \, \phi$ ("eventually") holds if ϕ becomes true within t time units, $\phi_1 \, \mathcal{W}^{\leq t} \, \phi_2$ ("weak until") holds if either ϕ_1 remains true for t time units or ϕ_2 becomes true within t time units with ϕ_1 holding until then, and $\Box^{\leq t} \, \phi$ ("continuously") holds if ϕ continuously holds for t time units. These path operators can be defined without a time-bound in an analogous way.

Table 1 gives a few examples of goals that we can express using PCTL/CSL. In addition to regular achievement goals, we are able to specify goals with deadlines, safety constraints over execution paths, maintenance goals, and prevention goals.

Relation to Decision-Theoretic Planning

We can encode a probabilistic goal $\mathcal{P}_{\bowtie p}(\phi_1 \, \mathcal{U} \, \phi_2)$ for a decision process M as a Markovian reward function for a modified decision process M'. This is done by making every state for which $\neg\phi_1 \vee \phi_2$ holds in M absorbing in M' and assigning reward one in M' to those states that satisfy ϕ_2 in M. All other states are assigned reward zero. The expected total undiscounted reward for M' then equals the probability of $\phi_1 \, \mathcal{U} \, \phi_2$ holding in M. For a time-bounded formula $\mathcal{P}_{\bowtie p}\left(\phi_1 \, \mathcal{U}^{\leq t} \, \phi_2\right)$, the time-bound serves as a planning horizon.

Bacchus, Boutilier, & Grove (1996; 1997) use PLTL, a past-tense variation of the linear temporal logic (LTL; Emerson *et al.* 1990), to specify desired plan behavior, and associate rewards with PLTL formulas. This enables the specification of non-Markovian rewards. A similar approach is suggested by Thiébaux, Kabanza, & Slaney (2002), who present a formalism for expressing non-Markovian rewards adapted to anytime solution methods.

Decision Epochs

For discrete-time MDPs decision epochs occur at every time point, meaning that the decision maker is allowed to select an action for execution at regular intervals of unit length. If we count movements from a state to itself as state transitions (caused by the triggering of an event with no effects on the current state), then we can say that decision epochs for discrete-time MDPs occur after every state transition.

A natural extension of this decision model to continuous-time and non-Markovian decision processes is to have decision epochs occur only at state transitions. This is the typical decision model for SMDPs (Howard 1971), and is also the model used by Younes, Musliner, & Simmons (2003) for GSMDPs.

Alternatively, we could allow policies that associate with each state s an action selection function $\pi(s,t)$, with the currently enabled action being a function of the time since the last state transition. This decision model is, for example, used by Doshi (1979). Decision epochs occur at every point in time with this model, making the number of decision epochs uncountable for continuous-time decision processes.

Plan Verification

Given a plan π, a stochastic domain model $\mathcal{M}$, and a planning problem $\langle s_0, \phi \rangle$, we accept π as a solution if ϕ holds in s_0 for the stochastic process $\mathcal{M}$ controlled by π. We can take advantage of recent developments in probabilistic verification to verify if a plan is a solution to a planning problem.

Interest in verification of probabilistic systems has been on the rise in the last ten years. Symbolic methods for probabilistic verification of discrete-time (Hansson & Jonsson 1994) and continuous-time (Baier, Katoen, & Hermanns 1999; Baier *et al.* 2000; Katoen *et al.* 2001) Markov processes have been proposed. PRISM (Kwiatkowska, Norman, & Parker 2002a; 2002b) is a fast symbolic probabilistic model checker for both PCTL and CSL formulae.

Goal description	Formula
reach office with probability at least 0.9	$\mathcal{P}_{\geq 0.9}\left(\Diamond \text{ office}\right)$
reach office within 5 time units with probability at least 0.9	$\mathcal{P}_{\geq 0.9}\left(\Diamond^{\leq 5} \text{ office}\right)$
reach office with probability at least 0.9 along paths where the recharging station can be reached within 17 time units with probability at least 0.5	$\mathcal{P}_{\geq 0.9}\left(\mathcal{P}_{\geq 0.5}\left(\Diamond^{\leq 17} \text{ recharging}\right) \mathcal{U} \text{ office}\right)$
maintain stability for at least 8.2 time units with probability at least 0.7	$\mathcal{P}_{\geq 0.7}\left(\Box^{\leq 8.2} \text{ stable}\right)$
avoid becoming wet with probability at least 0.8	$\mathcal{P}_{\geq 0.8}\left(\Box \neg \text{wet}\right)$

Table 1: Probabilistic goals expressible in CSL and PCTL.

Model checking algorithms for more complex models have been proposed by Infante López, Hermanns, & Katoen (2001) (SMPs) and Kwiatkowska *et al.* (2000) (stochastic timed automata with clocks governed by general distributions). While verification of CSL properties without a time-bound is no harder for SMPs than for Markov processes, the proposed symbolic methods for verifying time-bounded properties of more general processes rely on techniques that are prohibitively complex, and for GSMPs no symbolic methods exist at all.

Younes & Simmons (2002) have developed an efficient sampling-based approach to verifying time-bounded probabilistic properties of general discrete event systems. For GSMPs without any restrictions on the class of delay distributions that can be used, there are currently no alternatives to sampling-based approaches. Without exhaustive sampling, we can never be certain that the result returned by a sampling-based approach is correct, but Younes & Simmons use statistical hypothesis testing techniques to bound the probability of an incorrect verification result.

Discussion

We have presented an extension of PDDL for modeling stochastic decision processes of varying complexity. Our representation of actions with probabilistic effects is in essence the same as that of Dearden & Boutilier (1997). In this paper we have tied this representation to a PDDL-like syntax. We have also extended the representation of stochastic actions to include actions with random delay, which allows us to specify SMDPs and GSMDPs.

We have extended the classical representation of probabilistic planning problems by using PCTL and CSL for specifying goals. This allows us to express, for example, deadlines and maintenance and prevention goals in addition to the traditional achievement goals. We have discussed work in probabilistic model checking that could be utilized for efficient plan verification.

The focus of this paper has been on the representation of planning problems and not on the representation or generation of plans. Much of the work in probabilistic and decision theoretic planning assumes an MDP model. Boutilier, Dean, & Hanks (1999) provides an excellent overview of planning with MDP models. Howard (1971) provides a good introduction to SMPs and SMDPs, and presents dynamic programming algorithms for solving decision-theoretic SMDP planning problems. Probabilistic planning with GSMDP domain models have been considered recently by Younes, Musliner, & Simmons (2003) who propose a sampling-based algorithm for generating stationary policies for GSMDPs, and decision-theoretic extensions of this work is discussed by Ha & Musliner (2002).

Acknowledgments. The author acknowledges Reid Simmons for helping him clarify his thoughts on issues discussed in this paper through many long and fruitful discussions. The author also thanks Michael Littman and the anonymous reviewers for helpful comments.

This material is based upon work supported by DARPA and ARO under contract no. DAAD19–01–1–0485, and a grant from the Royal Swedish Academy of Engineering Sciences (IVA). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsors.

References

Bacchus, F.; Boutilier, C.; and Grove, A. 1996. Rewarding behaviors. In *Proc. Thirteenth National Conference on Artificial Intelligence*, 1160–1167. AAAI Press.

Bacchus, F.; Boutilier, C.; and Grove, A. 1997. Structured solution methods for non-Markovian decision processes. In *Proc. Fourteenth National Conference on Artificial Intelligence*, 112–117. AAAI Press.

Baier, C.; Haverkort, B. R.; Hermanns, H.; and Katoen, J.-P. 2000. Model checking continuous-time Markov chains by transient analysis. In *Proc. 12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, 358–372. Springer.

Baier, C.; Katoen, J.-P.; and Hermanns, H. 1999. Approximate symbolic model checking of continuous-time Markov chains. In *Proc. 10th International Conference on Concurrency Theory*, volume 1664 of *LNCS*, 146–161. Springer.

Blythe, J. 1994. Planning with external events. In *Proc. Tenth Conference on Uncertainty in Artificial Intelligence*, 94–101. Morgan Kaufmann Publishers.

Bonet, B., and Geffner, H. 2001. GPT: A tool for planning with uncertainty and partial information. In *Proc. IJCAI-01 Workshop on Planning with Uncertainty and Incomplete Information*, 82–87.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computa-

tional leverage. *Journal of Artificial Intelligence Research* 11:1–94.

Dean, T.; Kaelbling, L. P.; Kirman, J.; and Nicholson, A. 1995. Planning under time constraints in stochastic domains. *Artificial Intelligence* 76(1–2):35–74.

Dearden, R., and Boutilier, C. 1997. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence* 89(1–2):219–283.

Doshi, B. T. 1979. Generalized semi-Markov decision processes. *Journal of Applied Probability* 16(3):618–630.

Drummond, M., and Bresina, J. 1990. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proc. Eighth National Conference on Artificial Intelligence*, 138–144. AAAI Press.

Emerson, E. A.; Mok, A. K.; Sistla, A. P.; and Srinivasan, J. 1990. Quantitative temporal reasoning. In *Proc. 2nd International Conference on Computer Aided Verification*, volume 531 of *LNCS*, 136–145. Springer.

Farley, A. M. 1983. A probabilistic model for uncertain problem solving. *IEEE Transactions on Systems, Man, and Cybernetics* 13(4):568–579.

Fox, M., and Long, D. 2002a. PDDL+: Modelling continuous time-dependent effects. In *Proc. 3rd International NASA Workshop on Planning and Scheduling for Space*.

Fox, M., and Long, D. 2002b. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Technical Report 20/02, University of Durham, Durham, UK.

German, R. 2000. *Performance Analysis of Communication Systems: Modeling with Non-Markovian Stochastic Petri Nets*. New York, NY: John Wiley & Sons.

Glynn, P. W. 1989. A GSMP formalism for discrete event systems. *Proceedings of the IEEE* 77(1):14–23.

Goldman, R. P., and Boddy, M. S. 1994. Epsilon-safe planning. In *Proc. Tenth Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann Publishers.

Ha, V., and Musliner, D. J. 2002. Toward decision-theoretic CIRCA with application to real-time computer security control. In *Papers from the AAAI Workshop on Real-Time Decision Support and Diagnosis Systems*, 89–90. AAAI Press. Technical Report WS-02-15.

Hansson, H., and Jonsson, B. 1994. A logic for reasoning about time and reliability. *Formal Aspects of Computing* 6(5):512–535.

Howard, R. A. 1971. *Dynamic Probabilistic Systems*, volume II. New York, NY: John Wiley & Sons.

Infante López, G. G.; Hermanns, H.; and Katoen, J.-P. 2001. Beyond memoryless distributions: Model checking semi-Markov chains. In *Proc. 1st Joint International PAPM-PROBMIV Workshop*, volume 2165 of *LNCS*, 57–70. Springer.

Katoen, J.-P.; Kwiatkowska, M.; Norman, G.; and Parker, D. 2001. Faster and symbolic CTMC model checking. In *Proc. 1st Joint International PAPM-PROBMIV Workshop*, volume 2165 of *LNCS*, 23–38. Springer.

Kushmerick, N.; Hanks, S.; and Weld, D. S. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76(1–2):239–286.

Kwiatkowska, M.; Norman, G.; Segala, R.; and Sproston, J. 2000. Verifying quantitative properties of continuous probabilistic timed automata. In *Proc. 11th International Conference on Concurrency Theory*, volume 1877 of *LNCS*, 123–137. Springer.

Kwiatkowska, M.; Norman, G.; and Parker, D. 2002a. PRISM: Probabilistic symbolic model checker. In *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 2324 of *LNCS*, 200–204. Springer.

Kwiatkowska, M.; Norman, G.; and Parker, D. 2002b. Probabilistic symbolic model checking with PRISM: A hybrid approach. In *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, 52–66. Springer.

Lesh, N.; Martin, N.; and Allen, J. 1998. Improving big plans. In *Proc. Fifteenth National Conference on Artificial Intelligence*, 860–867. AAAI Press.

Matthes, K. 1962. Zur Theorie der Bedienungsprozesse. In *Transactions of the Third Prague Conference on Information Theory, Statistical Decision Functions, Random Processes*, 513–528. Publishing House of the Czechoslovak Academy of Sciences.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

Nelson, W. 1985. Weibull analysis of reliability data with few or no failures. *Journal of Quality Technology* 17(3):140–146.

Osogami, T., and Harchol-Balter, M. 2003. A closed-form solution for mapping general distributions to minimal PH distributions. Technical Report CMU-CS-03-114, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Shedler, G. S. 1993. *Regenerative Stochastic Simulation*. Boston, MA: Academic Press.

Thiébaux, S.; Kabanza, F.; and Slaney, J. 2002. Anytime state-based solution methods for decision processes with non-markovian rewards. In *Proc. Eighteenth Conference on Uncertainty in Artificial Intelligence*, 501–510. Morgan Kaufmann Publishers.

Younes, H. L. S., and Simmons, R. G. 2002. Probabilistic verification of discrete event systems using acceptance sampling. In *Proc. 14th International Conference on Computer Aided Verification*, volume 2404 of *LNCS*, 223–235. Springer.

Younes, H. L. S.; Musliner, D. J.; and Simmons, R. G. 2003. A framework for planning in continuous-time stochastic domains. In *Proc. 13th International Conference on Automated Planning and Scheduling*. AAAI Press. Forthcoming.

Author Index