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 $f_K : \Lambda_k \rightarrow \Lambda_{k+1}$, (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)
  ...
```

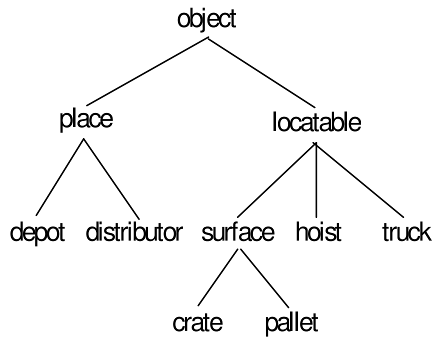

Fig. 1 - Type Hierarchy for the depot ground domain.

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.

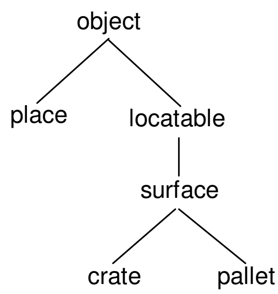

Fig. 2 - Type Hierarchy for depot abstract domain.

[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))
```

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))))
```

```
(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.

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.