Generating Abstractions from Static Domain Analysis

G. Armano, G. Cherchi, and E. Vargiu
DIEE Department of Electrical and Electronic Engineering
University of Cagliari
Piazza d'Armi I-09123 Cagliari
{armano, cherchi, vargiu}@diee.unica.it

Abstract— **This paper addresses the problem of how to implement a proactive behavior according to a two-tiered (i.e., both theoretical and pragmatic) perspective. Theoretically, we claim that abstraction must be used to render agents able to solve complex problems. Pragmatically, we illustrate a technique devised to generate abstract spaces starting from a "ground" description of the domain being modeled.**

I. Introduction

A complex environment is intractable using traditional planning methods, since the search space can be very large, even for relatively simple tasks. As intelligent agents are used in increasingly complex domains, their ability to decompose problems and to identify abstract aspects becomes a critical issue. An effective approach used for dealing with the complexity of planning tasks may consist of building a set of abstractions for controlling the search.

Typically, abstraction techniques require the original search space to be mapped into abstract spaces in which irrelevant details are disregarded at different levels of granularity. Let us briefly recall some relevant techniques proposed in the literature: (i) action-based, (ii) state-based, (iii) Hierarchical Task Networks, and (iv) case-based.

The first combines a group of actions to form macro-operators [1]. The second exploits representations of the world given at a lower level of detail; its most significant forms rely on (a) relaxed models, obtained by dropping operators' applicability conditions [2], and on (b) reduced models [3], obtained by completely removing certain conditions from the problem space. Both models, while preserving the provability of plans that hold at the ground level, perform a "weakening" of the original problem space, thus suffering from the drawback of introducing "false" (i.e., not refinable) solutions at the abstract levels [4]. In the third (e.g., [5]), problem and operators are organized into a set of tasks. A high-level task can be reduced to a set of ordered lower-level tasks, and a task can be reduced in several ways. Reductions allow specifying how to obtain a detailed plan from an abstract one. In the fourth [6], abstract planning cases are automatically learned from given concrete cases, as done in the PARIS system, although the user must provide explicit refinement rules between adjacent levels in the hierarchy.

In the past, abstraction has been widely used to reduce search in a variety of planning systems (e.g., GPS [7], ABSTRIPS [2], ABTWEAK [8], PABLO [9], and PRODIGY [10]). 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 and HTN-based systems are examples of operator-based abstraction. In [11] experiments on abstraction on all these dimensions are shown.

In this paper, a semi-automatic technique for generating abstract spaces is presented. Abstraction arises from a static analysis of relevancy relationships among ground operators, through the pruning of a suitable graph. It is worth pointing out that studying abstraction may have a great impact on the "internals" of intelligent agents, since –by definition– an agent must be able to generate plans in arbitrarily complex domains, no matter which environment, real (e.g., robotic applications) or virtual (e.g., Internet, computer games), is being considered.

This paper is organized as follows: in section 1, abstraction hierarchies are briefly framed according to a theoretical perspective. In section 2, a syntactic notation, devised to support abstraction hierarchies, is summarized. In section 3, the semi-automatic technique developed for generating abstract spaces is illustrated. Section 4 illustrates a sample of a complex domain (suitably devised to stress the proactive capabilities of intelligent agents) together with the corresponding abstraction. Finally, conclusions are drawn and future work is outlined.

II. Abstraction Hierarchies

According to [4], 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.

To deal with different levels of abstraction, a suitable representation language must be adopted. Nowadays, several planners adopt the PDDL standard notation [12] as target language for expressing each Λ_k (k=0, 1, …, n-1). As for the mapping functions f_κ between abstraction levels, to our knowledge existing systems adopt their specific notation without following any standard. Our choice has been to devise and adopt an extension to PDDL suitable tailored for hierarchical planning [13].

III. An Extension to PDDL for Dealing with Abstraction Hierarchies

Let us assume that a problem and its corresponding domain are described in accordance with the standard PDDL 1.2 syntax; i.e., using the "*define problem*" and "*define domain*" statements, respectively. To describe how bi-directional communication occurs between adjacent levels, the syntactic construct *define hierarchy* has been introduced. It encapsulates an ordered set of domains, together with a corresponding set of mappings between adjacent levels of abstraction. The general form of the construct is:

```
(define (hierarchy <name>)
  (:domains <domain-name>+)
  (:mapping (<src-domain> <dst-domain>)
   [:types <types-def>]
   [:predicates <predicates-def>]
   [:actions <actions-def>])*)
```

The `:types` field represents how types are mapped between adjacent levels; it embodies a list of clauses, according to the following notation:

```
(abstract-type ground-type)
```

Any such clause specifies that `ground-type` becomes `abstract-type` while performing upward translations. To disregard a `ground-type`, the following notation is used:

```
(nil ground-type)
```

The `:predicates` field represents how predicates are mapped between adjacent levels; it embodies a list of clauses, whose general form is:

```
((abstract-predicate ?v1 ?v2 …)
 (ground-predicate ?v1 – t1 ?v2 – t2 …))
```

Table 1 - Heuristics for pruning the operators' graph.

Type	Node relationships	Supporting Evidence	Action
1	A → B: <a b>; B → A: <c d>	(i) if a = c and b = d there is no supporting evidence for assuming that A usually precedes B in a plan, and vice-versa;	remove both edges.
		(ii) if a > c there is a high likelihood that A precedes B;	remove top edge.
		(iii) if c > a, there is a high likelihood that B precedes A.	remove bottom edge.
2	A → B: <a 0>; B → A: <c 0>	(i) if a > c there is a high likelihood that A precedes B;	remove top edge.
		(ii) if c > a, there is a high likelihood that B precedes A.	remove bottom edge.
3	A → B: <0 b>; B → A: <0 d>	A (B) negates one or more preconditions required by B (A).	remove both edges.
4	A → B: <a 0>; B → A: <0 d>	B negates one or more preconditions required by A.	remove bottom edge.
5	A → B: <a b>; B → A: <0 d>	B negates one or more preconditions required by A.	remove bottom edge.
6	A → B: <a b>; B → A: <c 0>	A and B are usually complementary or loosely-coupled actions.	remove both edges.
7	A → B: <a 0>; B → A: <0 0>	A precedes B with high likelihood.	remove top edge.
8	A → B: <0 b>; B → A: <0 0>	A negates one or more preconditions required by B.	remove the top edge.

Any such clause specifies how the `ground-predicate` becomes `abstract-predicate` while performing upward translations and vice-versa. [1] Note that `abstract-predicate`'s parameters could be a proper subset of `ground-predicate`'s parameters.

To disregard a predicate while performing upward translations, the following notation is used:

```
(nil (ground-predicate ?v1 – t1 ?v2 – t2 …))
```

It specifies that `ground-predicate` is not translated into any abstract-level predicate.

In addition, `abstract-predicate` can be expressed as a logical combination of ground level predicates. For example:

```
((abs-pred ?v11 ?v21 ?v12 ?v22 …)
  (and (gnd-pred1 ?v11 – t11 ?v21 – t21 …)
       (gnd-pred2 ?v12 – t12 ?v22 – t22 …)))
```

It specifies that `abs-pred` is the conjunction of `gnd-pred1` and `gnd-pred2`.

Similarly, to describe how to build a set of operators for the abstract domain, in the `:actions` field four kinds of mapping can be expressed:

1. an action is removed;
2. an action is expressed as a combination of ground domain actions;
3. an action remains unchanged or some of its parameters are disregarded;
4. a new operator is defined from scratch.

IV. Generating Abstractions

To facilitate the setting of abstract spaces, as an alternative to the hand-coded approach used in [14], a novel semi-automatic technique for generating abstraction hierarchies starting from ground-level domain descriptions has been devised.

From our particular perspective, performing abstraction basically involves executing two steps: (i) searching for macro-operator schemata through a priori or a posteriori analysis, (ii) selecting some of the schemata evidenced so far and translating them into abstract operators.

In particular, we concentrate on the task of finding macro-operator schemata throughout an a-priori analysis performed on the given domain and problem, rather than adopting the a-posteriori technique illustrated in [15], headed at finding macro-operator schemata according to a "post-mortem" analysis performed on plan "chunks".

Step (i) is performed by an algorithm for building and then pruning a directed graph, whose nodes represent operators and whose edges represent relations between effects of the source node and preconditions of the destination node. In particular, for each source node A and for each destination node B, representing operators defined in the given domain, the corresponding edge is labeled with a pair of non-negative numbers, denoted by <a b>. The pair accounts for how many predicates A can establish (a) and negate (b) that are also preconditions of B. It is worth noting that source and destination node may coincide, thus giving rise to a self-reference. Of course, edges labeled <0 0>, which link together completely independent operators, are not taken into account.

Pruning is performed according to the domain-independent heuristics reported in Table 1.

Step (ii) is performed by selecting the most promising macro-operator schemata from the pruned graph and by translating them into abstract operators. In the current implementation of the system, this step has not yet been completely automated.

As for the most promising macro-operator schemata, they can be easily extracted from the pruned graph, each path being related with a candidate macro-operator. Among all existing paths, only those containing a single occurrence of each operator are selected.

The simplest way of generating an abstract operator consists of defining its preconditions and effects as the preconditions and the effects of the corresponding macro-operator schema, according to the following definitions:

$$\begin{cases} \gamma_\sigma = \gamma_{\sigma_1} \cup \left(\gamma_{\omega_n} \setminus \eta_{\sigma_1}\right) \\ \alpha_\sigma = \left(\alpha_{\sigma_1} \setminus \delta_{\sigma_1}\right) \cup \alpha_{\omega_n} \\ \delta_\sigma = \left(\delta_{\sigma_1} \setminus \alpha_{\sigma_1}\right) \cup \delta_{\omega_n} \end{cases}$$

where σ is a macro-operator (expressed as a sequence of operators ω; i.e., $\sigma = \omega_1\omega_2....\omega_n = \sigma_1\omega_n$), whereas γ, η, α, and δ are preconditions, effects, add-list, and delete-list, [2] respectively.

It is worth pointing out that several macro-operator could actually lead to the same abstract-operator (Σ_a), provided that each macro-operator is entailed by Σ_a.

To perform simplification, all predicates that do not occur among preconditions or effects of any abstract-operator obtained by inspection of the pruned graph should be deleted from the abstract level. This process influences (and is influenced by) the translation rules that apply to both types and predicates.

The approach described above can be used also for generating abstractions tailored to a given problem, by simply adding a dummy operator representing the goal(s) of the problem itself. This "goal-oriented" operator has only preconditions (its set of effects being empty), representing a logic conjunct of predicates that characterize the goal of the input problem. In this way, all sequences deemed relevant to solve the problem are easily put into evidence (as they end with the "goal-oriented" operator).

[1] If no differences exist in mapping a predicate between adjacent levels, the corresponding clause can be omitted.

[2] Let us recall that effects can be split into add-list and delete list, representing positive and negative effects, respectively.

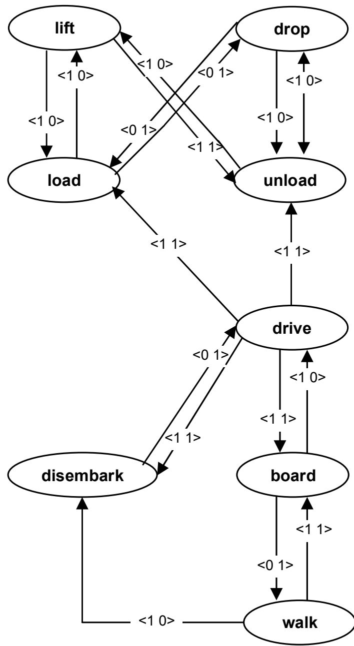

Fig. 2 - The directed graph (before pruning), representing static relationships among operators of the *driverdepot* domain.

V. EXPERIMENTS WITH THE DRIVERDEPOT DOMAIN

Experimental results have shown that abstraction is more effective when the complexity of planning problems increases.

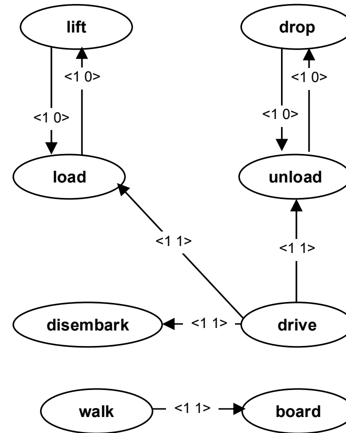

Fig. 1 - The directed graph (after pruning), representing static relations between operators of the *driverdepot* domain.

To assess the advantages of using this approach, we devised a suitable complex domain by extending the depot domain taken from the AIPS 2002 planning competition [16] (see the Appendix for more details). Fig. 2 shows the graph corresponding to the *driverdepot* domain, obtained by applying the approach described in the previous section.

Bearing in mind that the same mechanism has been applied to all operators' pairs, let us concentrate –for instance– on the

Table 2. Selected macro-operator schemata

Macro-Operator Schema	Ground Sequence	Preconditions	Effects
(DriveUnloadDrop ?h - hoist ?t - truck ?p1 ?p2 - place ?d - driver ?c - crate ?s - surface)	drive; unload; drop	(at ?t ?p1) (driving ?d ?t) (in ?c ?t) (at ?s ?p2) (clear ?s) (at ?h ?p2) (available ?h)	(not (at ?t ?p1)) (at ?t ?p2) (not (in ?c ?t)) (at ?c ?p2) (not (clear ?s)) (clear ?c) (on ?c ?s)
(UnloadDrop ?h - hoist ?t - truck ?p - place ?d - driver ?c - crate ?s - surface)	unload; drop	(at ?t ?p) (in ?c ?t) (at ?s ?p) (clear ?s) (at ?h p) (available ?h)	(not (in ?c ?t)) (at ?c ?p) (not (clear ?s)) (clear ?c) (on ?c ?s)
(WalkBoard ?d - driver ?p1 ?p2 place ?t - truck)	walk; board	(at ?d ?p1) (at ?t ?p2) (empty ?t)	(not (at ?d ?p1)) (driving ?d ?t) (not (empty ?t))
(DriveDisembark ?d - driver ?t - truck ?p1 ?p2 - place)	drive; disembark	(at ?t ?p1) (driving ?d ?t)	(not (driving ?d ?t)) (at ?d ?p2) (empty ?t) (at ?t ?p2) (not (at ?t ?p1))
(LiftLoad ?h - hoist ?c - crate ?t - truck ?s - surface ?p - place)	lift; load	(at ?h ?p) (available ?h) (at ?c ?p) (on ?c ?s) (clear ?c) (at ?t ?p)	(not (at ?c ?p)) (not (clear ?c)) (clear ?s) (in ?c ?t) (not (on ?c ?s))

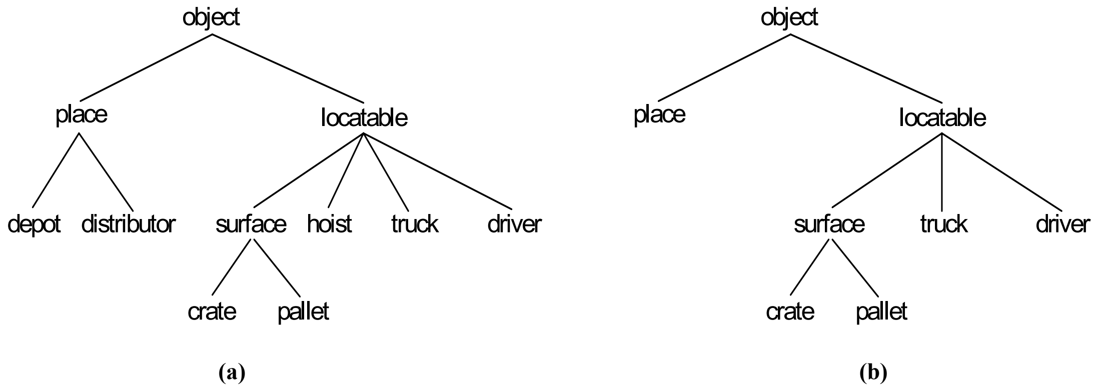

Fig. 3 - Type Hierarchy for the *driverdepot* ground (a) and (b) abstract domain.

relation that holds between *drive* (source node) and *board* (destination node).

Considering that the effects of the *drive* operator are:

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

and that the preconditions of the *board* operator are:

```
(at ?t ?p) (at ?d ?p) (empty ?t)
```

we label the corresponding edge with the pair <1 1>. In fact, it is apparent that *drive* establishes one precondition for *board*, while negating another.

Fig. 1 shows the resulting graph for the *driverdepot* domain after the pruning activity. [3] The resulting macro-operator schemata are (";" being used for concatenation): *drive;unload;drop*, *drive;load;lift*, *drive;disembark*, *lift;load*, *drop;unload*, *load;lift*, *unload;drop*, and *walk;board*.

Among these, *load;lift*, *drive;load;lift*, and *drop;unload* have been disregarded since they become meaningless when applied to the same object. For instance, loading a *truck* with a *crate* C and then lifting C back does not alter the state of the world.

Hence, *drive;unload;drop*, *unload;drop*, *walk;board*, *drive;disembark*, and *lift;load* (see Table 2) are the selected macro-operator schemata. As for the generation of abstract-operators, let us note that *drive;unload;drop* and *unload;drop*, can be considered alternative refinements of the same abstract-operator. Furthermore, let us stress that the *lifting* predicate does not appear as precondition or effect in any abstract operator; hence, it can be removed at the abstract level.

Since we are interested in abstracting the domain on types, predicates, and operators, the type hierarchy could be simplified by deleting –for example- the *hoist* type. This choice is feasible because *hoists* are always available, in every *place* (consequently, the *available* predicate can also be removed). Moreover, the type hierarchy can be further reduced by considering both *distributors* and *depots* as generic *places* (see Fig. 3 for the type hierarchy illustrating both ground and abstract level).

In the Appendix the hierarchy for the *driverdepot* domain and the corresponding abstract domain are shown.

VI. Conclusions and Future Work

In this paper, a semi-automatic technique for generating abstract spaces has been proposed. Abstraction arises from a static analysis of relevancy relationships among ground operators, through the pruning of a suitable graph. After briefly discussing the need of resorting to abstraction for solving complex problems, we illustrated an algorithm able to generate abstract spaces (i.e., abstractions on operators, predicates and types) starting from a ground-level description of the domain.

As for the future work, we are currently addressing the problem of rendering the generation of abstract spaces fully automatic.

Appendix

The *depot* domain joins 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. The proposed domain extends the *depot* domain adding to it a *driver* able to move trucks among places (see Fig. 4). Note that the *driver* could simulate the behavior of an agent able to deliver objects by driving trucks from a location to another. Fig. 6 shows the hierarchy for the *driverdepot* domain, whereas Fig. 5 shows the abstract domain, obtained by applying the mapping rules in Figure 5 to the ground domain.

[3] Since we are interested in finding macro-operators, we do not take into account self-references.

```
(define (domain DriverDepot-ground)

  (:requirements :strips :typing)

  (:types
    place locatable - object
    depot distributor - place
    driver truck hoist surface - locatable
    pallet crate - surface)

  (:predicates
    (at ?l - locatable ?p - place) (on ?c - crate ?s - surface)
    (in ?c - crate ?t - truck) (lifting ?h - hoist ?c - crate)
    (available ?h - hoist) (clear ?s - surface)
    (driving ?d - driver ?t - truck) (empty ?t - truck))

  (:action Drive
    :parameters (?t - truck ?p1 ?p2 – place ?d - driver)
    :precondition (and (at ?t ?p1) (driving ?d ?t))
    :effect (and (not (at ?t ?p1)) (at ?t ?p2)))

  (:action Lift
     :parameters (?h - hoist ?p – place ?c - crate ?s - surface)
     :precondition (and (at ?h ?p) (available ?h) (at ?c ?p) (on ?c ?s) (clear ?c))
     :effect (and (not (at ?c ?p)) (clear ?s) (lifting ?h ?c)
                  (not (clear ?c)) (not (available ?h)) (not (on ?c ?s))))

  (:action Drop
    :parameters (?h - hoist ?c – crate ?s - surface ?p - place)
    :precondition (and (at ?h ?p) (at ?s ?p) (clear ?s) (lifting ?h ?c))
    :effect (and (available ?h) (at ?c ?p)
                 (not (lifting ?h ?c)) (not (clear ?s))(clear ?c)(on ?c ?s)))

  (:action Load
    :parameters (?h - hoist ?c - crate ?t – truck p - place)
    :precondition (and (at ?h ?p) (at ?t ?p) (lifting ?h ?c))
    :effect (and (not (lifting ?h ?c)) (in ?c ?t) (available ?h)))

  (:action Unload
    :parameters (?h - hoist ?c - crate ?t - truck ?p - place)
    :precondition (and (at ?h ?p) (at ?t ?p) (available ?h) (in ?c ?t))
    :effect (and (not (in ?c ?t)) (not (available ?h)) (lifting ?h ?c)))

  (:action Board
    :parameters (?d - driver ?t - truck ?p - place)
    :precondition (and (at ?t ?p) (at ?d ?p) (empty ?t))
    :effect (and (not (at ?d ?p)) (driving ?d ?t) (not (empty ?t))))

  (:action Disembark
    :parameters (?d - driver ?t - truck ?p - place)
    :precondition (and (at ?t ?p) (driving ?d ?t))
    :effect (and (not (driving ?d ?t)) (at ?d ?p) (empty ?t)))

  (:action Walk
    :parameters (?d - driver ?p1 ?p2 - place)
    :precondition (and (at ?d ?p1))
    :effect (and (not (at ?d ?p1)) (at ?d ?p2))))
```

Fig. 4 - The *driverdepot* domain

```
(define (hierarchy DriverDepot)
  (:domains
    DriverDepot-ground
    DriverDepot-abstract)
  (:mapping
    (DriverDepot-ground
     DriverDepot-abstract)
    :types
      ((place depot)
       (place distributor)
       (nil hoist))
    :predicates
      ((nil
         (lifting ?h - hoist ?c - crate))
       (nil
         (available ?h - hoist))
       (nil
         (at ?h - hoist ?p - place)))
    :actions
      ((nil (load ?h ?c ?t ?p))
       (nil (unload ?h ?c ?t ?p)
       (nil (lift ?h ?c ?s ?p))
       (nil (drop ?h ?c ?s ?p))
       (nil (walk ?d ?p1 ?p2))
       (nil (board ?d ?t ?p))
       (nil (disembark ?d ?t ?p))
       (drive-unload-drop
             ?t ?p1 ?p2 ?d ?c ?s)
          (and (drive ?t ?p1 ?p2 ?d)
               (unload ?h ?c ?t ?p2)
               (drop ?h ?c ?s ?p2)))
       ((walk-board ?d ?p1 ?p2 ?t)
          (and (walk ?d ?p1 ?p2)
               (board ?d ?t ?p2)))
       ((drive-disembark ?d ?t ?p1 ?p2)
          (and (drive ?t ?p1 ?p2 ?d)
               (disembark ?d ?t ?p2)))
       ((lift-load ?c ?t ?s ?p)
          (and (lift ?h ?p ?c ?s)
               (load ?h ?c ?t ?p)))))
```

Fig. 6 - Hierarchy definition for the *driverdepot* domain.

```
(define (domain DriverDepot-abstract)
  (:requirements :strips :typing)
  (:types place locatable - object
          driver truck surface - locatable
          pallet crate - surface)
  (:predicates (at ?l - locatable ?p - place)
               (on ?c - crate ?s - surface)
               (in ?c - crate ?t - truck)
               (clear ?s - surface)
               (driving ?d - driver ?t - truck)
               (empty ?t - truck))
  (:action DriveUnloadDrop
    :parameters (?t - truck ?p1 ?p2 - place ?d - driver ?c - crate ?s - surface)
    :precondition (and (at ?t ?p1) (driving ?d ?t) (in ?c ?t) (at ?s ?p2) (clear ?s))
    :effect
      (and (not (at ?t ?p1)) (at ?t ?p2) (not (in ?c ?t)) (at ?c ?p2) (not (clear ?s))
           (clear ?c) (on ?c ?s)))
  (:action LiftLoad
    :parameters (?c - crate ?t - truck ?s - surface ?p - place)
    :precondition (and (at ?c ?p) (on ?c ?s) (clear ?c) (at ?t ?p))
    :effect
      (and (not (at ?c ?p)) (not (clear ?c)) (clear ?s) (in ?c ?t) (not (on ?c ?s))))
  (:action WalkBoard
    :parameters (?d - driver ?t - truck ?p1 ?p2 - place)
    :precondition (and (at ?t ?p2) (at ?d ?p1) (empty ?t))
    :effect (and (not (at ?d ?p1)) (driving ?d ?t) (not (empty ?t))))
  (:action DriveDisembark
    :parameters (?d - driver ?t - truck ?p1 ?p2 - place)
    :precondition (and (at ?t ?p1) (driving ?d ?t))
    :effect
      (and (not (driving ?d ?t)) (at ?d ?p2) (empty ?t) (not (at ?t p1)) (at ?t ?p2))
  (:action Drive
    :parameters (?t - truck ?p1 ?p2 - place ?d - driver)
    :precondition (and (at ?t ?p1) (driving ?d ?t))
    :effect (and (not (at ?t ?p1)) (at ?t ?p2)))))
```

Fig. 5 - The *driverdepot-abstract* domain.

REFERENCES

[1] R.E. Korf, “Planning as Search: A Quantitative Approach”, *Artificial Intelligence*, *33*(1), 1987, 65–88.

[2] E.D. Sacerdoti “Planning in a Hierarchy of Abstraction Spaces”. *Artificial Intelligence*, 5:115--135, 1974.

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

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

[5] K. Erol, J. Hendler, and D.S. Nau. „HTN Planning: Complexity and Expressivity”. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, 1123--1128, AAAI Press / MIT Press, Seattle, WA, 1994.

[6] R. Bergmann, and W. Wilke, “Building and Refining Abstract Planning Cases by Change of Representation Language”, *Journal of Artificial Intelligence Research (JAIR)*, 1995, 3:53-118.

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

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

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

[10] J. Carbonell, C.A. Knoblock, and S. Minton. “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.

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

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

[13] G. Armano, G. Cherchi, and E., “An Extension to PDDL for Hierarchical Planning”. *Workshop on PDDL (ICAPS’03)*, Trento, Italy, June 2003.

[14] G. Armano, G. Cherchi, and E. Vargiu, “Experimenting the Performance of Abstraction Mechanisms through a Parametric Hierarchical Planner”. *Proceedings of IASTED International Conference on Artificial Intelligence and Applications (AIA’2003)*, Innsbruck, Austria, Febbraio 2003.

[15] G. Armano and E. Vargiu, “An Adaptive Approach for Planning in Dynamic Environments”. *Proceedings of the International Conference on Artificial Intelligence (IC-AI 2001), Special Session on Learning and Adapting in AI Planning*, pages 987--993, Las Vegas, Nevada, June 2001.

[16] Long, D. *Results of the AIPS 2002 planning competition*, 2002. [Online]. Available: http://www.dur.ac.uk/d.p.long/competition.html.