DHG: A System for Generating Macro-Operators from Static Domain Analysis

Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu

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

ABSTRACT

The attempt of dealing with the complexity of planning tasks by resorting to abstraction techniques is a central issue in the field of automated planning. Although the generality of the approach has not been proved always useful on domains selected for benchmarking purposes, in our opinion it will play a central role as soon as the focus will move from artificial to real problems. In this case, it will be crucial to have a tool for automatically generating abstraction hierarchies from a domain description. This paper addresses the problem of how to identify macro-operators starting from a ground-level description of a domain, to be used for generating useful abstract-level descriptions. In particular, a preliminary release of a system devised to automatically generate abstraction hierarchies has been implemented. Compared to our previous work, this paper reports a step further, in the direction of fully automatizing the process, from both a conceptual and a pragmatic perspective. Conceptually, we refined the process of macro-operators extraction by dealing with the problem of parameters' unification through the exploitation of domain invariants, which can resolve ambiguities that may arise while performing abstraction. Pragmatically, we implemented a system that -given a description of the domain expressed in PDDL- outputs a set of macro-operators to be used as a starting point for defining abstract operators. Experimental results highlight the ability of the system to identify suitable macro-operators, used as starting point for populating the abstract level. Such macro-operators usually represent good alternatives to those extracted by a knowledge engineer after a thorough (and sometimes painful!) domain analysis.

KEY WORDS

Abstraction, Planning, Macro-Operators.

1 INTRODUCTION

Complex environments are difficult to handle by traditional planning methods, since the search space can be very large, even for relatively simple problems –e.g., benchmarks in the AIPS planning competitions ([20], [6], [19]). The issue of dealing with the increasing complexity of the problems is going to play a central role as soon as planners will be used to solve problems encountered in real-life applications. In the past, abstraction techniques have been used in a variety of planning systems (e.g., GPS [23], ABSTRIPS [24], ABTWEAK [26], PABLO [9], and PRODIGY [8]), and have proven to be effective when applied to problems of medium-high complexity [5]. Typically, they 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 abstraction 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 [18]. 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 [24], and on (b) reduced models [17], obtained by completely removing certain conditions from the problem space. In the third (e.g., [10]), problem and operators are organized into a set of tasks: a high-level task can be reduced to a set of partially ordered, lower-level, tasks. Reductions allow specifying how to obtain a detailed plan from an abstract one. In the fourth, abstract planning cases are automatically learned from given concrete cases, as done in the PARIS system [7], although the user must provide explicit refinement rules between adjacent levels of the hierarchy.

The performance of planners can also be improved by exploiting the knowledge about the domain, as shown by a number of researchers. In particular, the fact that state invariants can play an important role in "compiling" planning domains is widely acknowledged ([12], [15], [21], [16], [22]). A detailed discussion of the state invariants, as well as the description of the TIM system, which can automatically extract state invariants from a PDDL domain and problem description, can be found in [11].

This paper addresses the problem of how to identify macro-operators starting from a ground-level description of a domain, to be used for generating useful abstract-level descriptions. Compared to our previous work ([5], [2], and [3]), this paper reports a step further, in the direction of fully automatizing the process. The process of macro-operators extraction has been improved by the exploitation of state invariants, useful for solving the problem of parameters' unification.

2 RELATED WORK

The pionieristic work of Korf on macro-operators was not explicitly tailored for abstraction hierarchies –the adoption of macro-operators being limited to the ground level only. The approach preserves both the soundness and the completeness of the planner, since macro-operators represent legal sequences of ground operators and none of them are removed from the domain. Nevertheless, this technique negatively impacts on the average branching factor.

State-based techniques are mainly focused on removing predicates at different levels of granularity (either for preconditions only or for both preconditions and postconditions), while disregarding abstraction on operators. [1] Although they preserve the Upward Solution Property (USP) [25], its main drawback concerns the introduction of "false" solutions (i.e., not refinable solutions that anyway hold at the abstract level(s), due to the deletion of some constraints that apply to the ground level). Thus, the adoption of these techniques is strictly related to the actual ratio between "false" and "true" solutions [13], which must be kept reasonably low.

As for HTNs, in a sense they are a generalization of Korf's macro-operators, being aimed at supporting abstraction through the definition of suitable building blocks at different levels of granularity. Their main advantage is a great expressive power (due to their capability of actually defining an abstraction hierarchy), together with the ability of allowing partial ordering among operators. The main drawback appears to be its strict dependence from the domain engineer, which is responsible for defining a (possibly) sound and complete HTN network for the given domain / problem.

Case-based techniques are centered on a different perspective, assuming that a solution of a given problem can be found by adapting plans already found for similar problems. [2] Several different issues are very important in this framework: (i) how to define suitable metrics for measuring the similarity between problems, (ii) how to store and maintain a repository of "cases" encountered while solving problems, and (iii) which techniques and heuristics should be exploited to adapt a plan retrieved from the repository and deemed useful for solving the given problem. It is worth noting that the adoption of case-based planning is justified only agreeing with the conjecture that "repairing" an existing solution is computationally less costly than finding one from scratch, which is actually a very controversial issue.

[1] The overall technique can be classified as "a priori", abstractions being searched without resorting to information on solutions.

[2] The overall technique can be classified as "a posteriori", abstraction being rooted on the information elicited from solutions found while solving previous problems.

3 AUTOMATIC GENERATION OF MACRO-OPERATORS

Basically, a planning domain can be defined in terms of two kinds of entities: predicates and operators (a particular kind of unary predicates can also be taken into account, giving rise to a third kind of entities –i.e., types– possibly organized according to a suitable "is-a" hierarchy). Although, in principle, abstraction might be performed along both such dimensions, this paper is mainly concerned with abstraction on operators –in particular, with the automatic extraction of macro-operators.

Let us point out that the definition of abstract operators is strictly related with the definition of abstract predicates and vice versa. Keeping this in mind, our proposal can be positioned between action- and state-based techniques.

To make this point clearer, let us give some definitions first. For the sake of simplicity, let us consider only two abstraction layers, namely *ground* and *abstract* (the extension of the definitions to an N-level hierarchy being trivial).

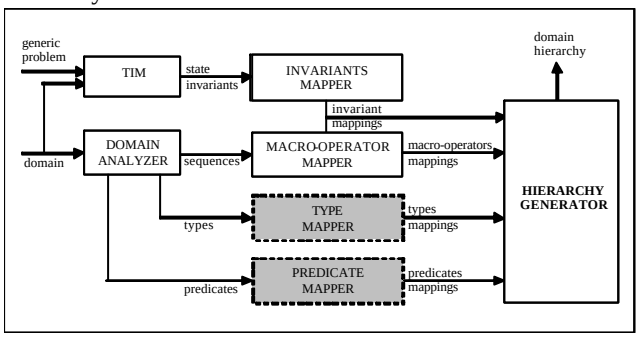
A deterministic ground operator is characterized by a name, a list of parameters, and the specification of its pre- and post-conditions given in terms of ground predicates. A ground operator can be instantiated by unifying each of its parameters with an object taken from the given domain. A macro-operator is any legal sequence of non-instantiated ground operators, together with the specification of its overall pre- and post-conditions. An abstract operator is characterized by a name, a list of parameters, and the specification of its overall pre- and post-conditions given in terms of abstract predicates.

Note that ground and abstract domains have the same form and are loosely related under the assumption that (most of the) abstract plans should be refinable at the ground level. To guarantee this desirable property, an abstract operator should be defined on top of several (at least one) *supporting* macro-operators, i.e., macro-operators whose pre- and post-conditions match the one defined for the corresponding abstract operator. On the other hand, a macro-operator can be obtained by uninstantiating any legal sequence of ground operators. It is worth noting that sequences deemed relevant can be obtained by resorting to both "a priori" and "a posteriori" analysis. The former is performed considering only the given domain (problem) (e.g., [5]), whereas the latter can be performed by taking into account (also) solutions previously found (see, for example, [1]).

To tackle a planning problem using abstraction, one (or more) abstract level(s) starting from the ground one should be defined. Abstracting a ground domain leads to the definition of an abstraction hierarchy, consisting of a set of predicates and operators, together with a mapping function devised to specify the mapping between ground and abstract level. In general, three kinds of mappings should be defined: (i) a set of types at the ground level can be

represented by a single type at the abstract level, [3] (ii) a single predicate at the ground level can be represented by a logical combination of predicates at the abstract level, and (iii) a set of macro-operators at the ground level can be combined into a single operator at the abstract level.

There is no predefined ordering in the abstraction process. [4] Nevertheless, as the paper is mainly concerned on automatically extracting macro-operators, let us adhere to the underlying assumption that our concerns about predicates (and types) play a secondary role, with respect to operators, in the process of defining an abstraction hierarchy. Figure 1 depicts the architecture of the system devised to automatically generate abstraction hierarchies. It has been called *DHG*, standing for Domain-oriented Hierarchy Generator.



The hierarchy generator module currently takes as inputs: (i) state invariants mappings (generated by the invariants mapper that processes the output produced by TIM [11]), and (ii) supporting macro-operators mappings (extracted from the sequences given by the domain analyzer – described in the following). *DHG* outputs a domain hierarchy, consisting of a ground and an abstract level. Currently, abstract operators and predicates are generated according to a simple strategy: for each supporting macro-operator a different abstract operator is generated, whose pre- and post-conditions are made coincident with the selected macro-operator. All predicates not involved in any pre- or post-condition are deleted from the abstract domain. [5]

The core of the whole process consists of finding a set of relevant sequences and then (possibly) promoting them to macro-operators. The basic steps for identifying such sequences are performed using a graph-oriented technique: first of all, a directed graph containing information about the dependencies between ground operators is built. Being G such graph, its nodes represent ground operators, and its 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, the corresponding edge is labelled with a pair of non-negative numbers, say <a b>. The pair accounts for how many predicates A can establish (a) and negate (b) that are also preconditions of B. From each acyclic path a relevant sequence of operators could be extracted. As considering all possible paths would end up to a large amount of macro-operators, a second step consists of pruning G –yielding the pruned graph G_p. The pruning activity is controlled by a set of domain-independent heuristics, which have been described in our previous work (see [5] for further details). A set of sequences (candidates to generate macro-operators) is then extracted from G_p. In particular, sequences whose post-conditions are represented by empty sets are disregarded for obvious reasons. The remaining sequences are considered relevant for generating macro-operators [3]. For each relevant sequence a corresponding macro-operator is generated, whose pre- and post-conditions are evaluated from pre- and post-conditions of the operators belonging to the sequence. Each extracted macro-operator is promoted to an abstract operator, defined –according to the *define action* statement of the standard PDDL notation– by its name, together with its parameters, its pre- and post-conditions.

Let us formally represent the process of promoting a sequence of ground operators to a macro-operator. In particular, let us assume that is a sequence of operators, whose application to the source state S1 leads to the destination state S2. Under this assumption, a corresponding macro-operator can be defined as follows:

where γ, η, α, and δ represent preconditions, effects, add-list, and delete-list of the resulting macro-operator, respectively.

$$\begin{cases} g_s = g_{s_1} \cup \left(g_{w_n} \setminus h_{s_1}\right) \\ a_s = \left(a_{s_1} \setminus d_{w_n}\right) \cup \left(a_{w_n} \setminus g_{s_1}\right) \\ d_s = \left(d_{w_n} \setminus a_{s_1}\right) \cup \left(d_{s_1} \setminus a_{w_n}\right) \end{cases} \quad (1)$$

The above formulas can be easily evaluated if all the actions belonging to σ are instantiated (i.e. all the involved parameters refer to a specified object). On the contrary, applying the formula in presence of variables could lead to semantic inconsistencies. A typical example that highlights this problem occurs when predicates that account for spatial relations are considered. For instance, while considering the predicate *(at ?o - object ?l - location)*, used in the *Logistics* domain to represent the position of an object, there cannot be two predicates stating that the same object is in two different locations. This condition can be expressed through the use of suitable state invariants. These are not explicitly stated in the domain description and can be retrieved using TIM. A detailed description about how to find state invariants is given in [11], where

[3] This is usually equivalent to defining an *exclusive or* in terms of unary predicates.

[4] In fact, one may start abstracting types, rather than predicates or operators –although any choice performed on one kind of mapping may impact on subsequent choices.

[5] The final system, consisting of additional modules devised to map also types and predicates (shadowed in the figure), will be able to perform abstraction along all the cited dimension –i.e., predicates, types, and operators.

four kinds of state invariants are defined: identity, state membership, uniqueness of state membership, and fixed resource. The information about the domain, enriched with invariants, allows to correctly unify macro-operators' parameters.

To automatically build the domain hierarchy, the *hierarchy generator* module requires a set of mapping functions that contain the translation rules (on types, predicates, operators, and invariants) between two adjacent levels of abstraction. These are expressed through a suitable extension of the hierarchy representation language. This information has been inserted (as a *:mapping* clause) into the *define hierarchy* statement, described in [4]. The proposed extension devised for dealing with invariants is:[6]

```
(<mapping-def>::=
  (:mapping (<src-domain> <dst-domain>)
    [:types <types-def>]
    [:predicates <predicates-def>]
    [:actions <actions-def>]
    [:invariants <invariants-def>])
```

Note that there is one *:invariant* statement for each mapping definition between two adjacent levels. In fact, in a *n-level* abstraction hierarchy, each mapping involves a specific set of invariants. The general form of the <*invariants-def*> is the following:

```
<invariants-def>::=
  ([:identity <identity-def>]
   [:statemembership <statemembership-def>]
   [:uniqueness <uniqueness-def>])

<identity-def>::=
  (and <typed-predicate> <typed-predicate>+)
  ((= <variable> <variable>)+)

<statemembership-def>::=
  (or <typed-predicate> <type-predicate>+)

<uniqueness-def>::=
  (not (and <typed-predicate>
            <typed-predicate>+))
```

The *:invariant* statement can be used to include the information about state invariants either by hand or automatically, as our abstraction system can also convert the output of TIM into the proposed notation. Given the mapping functions, abstract operators and predicates can be generated according to a simple strategy: for each macro-operator a suitable abstract operator is generated, whose pre- and post-conditions are made coincident with those of the selected macro-operator; predicates at the abstract level are the same of the ground level, except for those not involved in any pre- or post-condition of the abstract operators.

[6] With respect to the previous mapping definition, only the *:invariant* statement has been added.

4 EXPERIMENTAL RESULTS

To assess the functionality of the *DHG* system, we compared the automatically generated domain hierarchies with the corresponding domain hierarchies hand-coded by a knowledge engineer, and characterized by mapping on types, predicates, and operators. A set of benchmarking domains, taken from the planning competitions ([20], [6], [19]), has been selected to generate the abstraction hierarchies. The domain hierarchies have been used as input for the *HW[ ]* system (see [5]), devised to perform planning by abstraction. Let us briefly recall that, *HW[ ]* (which stands for *Hierarchical Wrapper*) can exploit any external PDDL-compliant planner to search for solutions at any required level of abstraction.

Experiments have been performed using FF ([14]) as external planner, being *HW[FF]* the resulting system. Let us point out that the planner chosen to be embedded into the system scarcely affects the relevance of the experimental results. In fact, only the relative performance between the automatic and the hand-coded versions of each domain hierarchy should be directly compared (for a description about the performance of abstraction mechanisms, see [5]). Experiments have been conducted on several domains including *Depots*, *Blocks-World* and *Elevator* (*simple-miconic*). For each domain, a set of problems has been selected to compare the performances of *HW[FF]* using the *DHG*'s domain hierarchies with those of *HW[FF]* using the hand-coded domain hierarchies.[7]

The abstract level found by *DHG* for the *Depots* domain is composed by four abstract operators, two of them (*lift* and *drop*) are identical to those defined at the ground level, while the others are obtained from the sequences *drive;load* and *drive;unload*. The hand-coded abstraction hierarchy defines two abstract-operators (obtained from the sequences *drive;unload;drop* and *drive;lift;load*), disregards the *lifting* predicate, and substitutes *depot* and *distributor* with the supertype *place* (this one being an example of abstraction on types).

The abstract level found by *DHG* for the *Elevator* domain is composed by four abstract operators: (obtained from the sequences *up;board*, *up;depart*, *down;board*, and *down;depart*). The corresponding hand-coded hierarchy defines two abstract operators (*oad* and *unload*) and disregards two predicates (*lift-at* and *above*).

The abstract level found by *DHG* for the *Blocks-World* domain is composed by two abstract operators: (obtained from the sequences *pick-up;stack* and *unstack;put-down*). The corresponding hand-coded hierarchy shows an abstract domain composed by the same operators, although

[7] For the sake of simplicity, since it is generally a demanding work to generate by hand abstraction hierarchies having more than two levels, the experiments have been made using two-level abstraction hierarchies.

	Hand-Coded				Automatic			
Problem	***abs***	***refs***	***tot***	***steps***	***abs***	***refs***	***tot***	***steps***
Depot1	28	73	106	12	23	120	147	11
Depot2	54	128	187	17	33	207	245	17
Depot3	488	340	841	38	69	532	609	36
Depot4	292	416	717	43	389	581	982	31
Depot5	845	100	950	71	-	-	-	-
Elevator1	10	51	63	8	19	57	78	8
Elevator2	17	142	163	15	20	145	170	16
Elevator3	18	226	248	4	11	28	40	4
Elevator4	18	359	383	23	23	396	427	26
Elevator5	19	740	767	28	26	566	603	28
Blocks1	11	41	54	6	11	41	54	6
Blocks2	18	104	125	14	19	107	129	14
Blocks3	40	450	497	44	41	463	513	44
Blocks4	45	479	532	48	46	471	524	48
Blocks5	55	501	564	48	58	472	538	48

Table 1. Hand-coded vs automatically generated hierarchy performance comparison using HW[FF].

the predicates *handempty* and *holding* have been disregarded.

Table 1 summarizes the results obtained for the selected domains. The results obtained using the planner without abstraction (FF, in this case) are not reported, since in this work we are not concerned on comparing the performance between a planning algorithm and its hierarchical counterpart. The columns labelled *abs* and *refs* report the time (expressed in milliseconds) needed to find the solution at the abstract level and the time needed to refine it, respectively. The column labelled *tot* reports the total time spent by *HW[FF]* to solve the problem, including disk usage, conversion to/from PDDL, etc. The column labelled *steps* is reported to compare the quality of plans (in terms of the steps required to reach the goal state) between the two counterparts.

Experiments show that, for the *Depots* domain, the performances of *HW[FF]* using the hierarchy found by *DHG* are in general slight worse (the difference is about 25%) than those of *HW[FF]* fed with the hand-coded hierarchy. In our opinion, the reason lies in the fact that automatically extracted hierarchy does not include abstraction on types and/or predicates, whereas the corresponding hand-coded hierarchy introduces types and predicates mappings. As for the *Elevator* domain the performance measured while feeding *HW[FF]* with the hierarchy found by DHG is about 20% worse than the one obtained by running *HW[FF]* with the hand-coded hierarchy. Also in this case the automatic hierarchy (being pure macro-operator based) lacks of mappings on types and/or predicates. In the *Blocks-World* domain, time intervals are approximately the same, since the hierarchy obtained from *DHG* is nearly identical to the one coded by hand. In fact, both of them define the abstract domain by two operators without abstracting types. The hand-coded hierarchy disregards two predicates, *(holding ?x - block)* and *(handempty)*, but this clearly does not introduce a substantial improvement, since *holding* does not appear in the preconditions and the effects of the macro-operators, and there is no macro-operator that negates the *handempty* predicate. In conclusion, the performances obtained by the automatic abstraction hierarchies should definitely be considered satisfactory. In fact, the required effort to make abstraction hierarchies by hand does not pay for the light advantages in terms of saved time.

5 CONCLUSIONS AND FUTURE WORK

The automatic definition of macro-operators is one of the most important steps in the task of abstracting a planning domain. In this paper, a technique devised to tackle this problem is briefly described, its implementation yielding a system called *DHG* (standing for Domain-oriented Hierarchy Generator). Experimental results –obtained comparing the performances of automatically-generated vs. hand-coded abstraction hierarchies– are encouraging and demonstrate the validity of the approach. In particular, the slightly negative impact on performances obtained by resorting to the automatic generation of abstraction hierarchies is more than counterbalanced by the fact that a negligible effort is required to the knowledge engineer in order to obtain suitable abstractions. The environment used to perform the experiments combines *DHG* with *HW[FF]*. The latter is a (parametric) hierarchical planning environment able to embed and run an external planner –in this case *FF*– at different levels of granularity.

As for the future work, we are currently dealing with the problem of combining predicate and operator abstractions. Furthermore, suitable heuristics for building an abstraction hierarchy able to ensure the USP are currently under study.

REFERENCES

[1] G. Armano, and E. Vargiu, An adaptive Approach for Planning in Dynamic Environments. Proceedings of the International Conference on Artificial Intelligence (IC-AI 2001), Las Vegas (Nevada) (2001) 987–993

[2] G. Armano, G. Cherchi, and E. Vargiu, Experimenting the Performance of Abstractions Mechanisms through a Parametric Hierarchical Planner. Proceedings of IASTED International Conference on Artificial Intelligence and Applications(AIA 2003), Innsbruck, Austria (2003) 399–404.

[3] G. Armano, G. Cherchi, and E. Vargiu, Generating Abstractions from Static Domain Analysis. Workshop dagli Oggetti agli Agenti - Sistemi Intelligenti e Computazione Pervasiva (WOA'03), Cagliari, Italy (2003).

[4] G. Armano, G. Cherchi, and E. Vargiu, An Extension to PDDL for Hierarchical Planning. Workshop on PDDL (ICAPS'03), Trento, Italy (2003).

[5] G. Armano, G. Cherchi, and E. Vargiu, A Parametric Hierarchical Planner for Experimenting Abstraction Techniques. Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03), Acapulco, Mexico (2003) 936–941,.

[6] F. Bacchus, Results of the aips 2000 planning competition (2000).

[7] R. Bergmann, W. Wilke, Building and refining abstract planning cases by change of representation language. Journal of Artificial Intelligence Research (JAIR), Vol. 3 (1995) 53–118.

[8] J.C. Carbonell, C.A. Knoblock, and S. Minton, PRODIGY: An Integrated Architecture for Planning and Learning. In D. Paul Benjiamin (ed.) Change of Representation and Inductive Bias, Kluwer Academic Publisher (1990) 125–146.

[9] J. Christensen, Automatic Abstraction in Planning. PhD Thesis, Department of Computer Science, Standford University (1991).

[10] K. Erol, J. Hendler, and D.S. Nau, HTN Planning: Complexity and Expressivity. Proceedings of the Twelveth National Conference on Artificial Intelligence (AAAI-94), AAAI Press / MT Press, Seattle, WA (1994) 1123–1128.

[11] M. Fox, and D. Long, The automatic inference of state invariants in tim. Journal of Artificial Intelligence Research (JAIR), Vol. 9 (1998) 367—421.

[12] A. Gerevini, and L. Schubert, Accelerating partial order planners: Some techniques for effective search control and pruning. Journal of Artificial Intelligence Research (JAIR), Vol. 5 (1996) 95—137.

[13] F. Giunchiglia, and T. Walsh, A theory of abstraction. Technical Report 9001-14, IRST, Trento, Italy (1990).

[14] J. Hoffmann, and B. Nebel, The ff planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research (JAIR)}, Vol. 14, (2001) 253–302.

[15] H. Kautz, and B. Selman, The Role of Domain Specific Knowledge in the Planning as Satisfiability Framework. Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98) (1998) 181–189.

[16] G. Kelleher, and A. Cohn, Automatically Synthesising Domain Constraints from Operator Descriptions. Proceedings of ECAI 1992 (1992) 653–655.

[17] C.A. Knoblock, Automatically generating abstractions for planning. Artificial Intelligence, Vol. 68(2) (1994) 243–302.

[18] R.E. Korf, Planning as search: A quantitative approach. Artificial Intelligence, Vol. 33(1), (1987) 65–88.

[19] D. Long, The aips-98 planning competition. AI Magazine, Vol. 21(2) (1998) 13–33.

[20] D. Long, Results of the aips 2002 planning competition, 2002. Url: http://www.dur.ac.uk/d.p.long/competition.html.

[21] T.L. McCluskey, and J. Porteous, Engineering and compiling planning domain models to promote validity and efficiency. Artificial Intelligence, Vol. 95(1) (1997) 1–65.

[22] P. Morris, and R. Feldman, Automatically Derived Heuristics for Planning Search. Proceedings of the Second Irish Conference on Artificial Intelligence and Cognitive Science, School of Computer Applications, Dublin City University (1989).

[23] A. Newell, and H.A. Simon, Human Problem Solving. Prentice-HAll, Englewood Cliffs, NJ (1972).

[24] E.D. Sacerdoti, Planning in a hierarchy of abstraction spaces. Artificial Intelligence, Vol. 5, (1974) 115–135.

[25] J.D. Tenenberg, Abstraction in Planning. PhD Thesis, Computer Science Department, University of Rochester (1988).

[26] Q. Yang, and J. Tenenberg, Abtweak: Abstracting a Nonlinear, Least-Commitment Planner. Proceedings of the Eight National Conference on Artificial Intelligence, Boston, MA (1990) 204–209.