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.
2 RELATED WORK

The pioneristic 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.

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

---

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.
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 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.

\[
\begin{align*}
\gamma_0 &= \gamma_1 \cup \eta_1 \\
\eta_0 &= \eta_1 \setminus \delta_1 \\
\alpha_0 &= \alpha_1 \setminus \gamma_1 \\
\delta_0 &= \delta_1 \cup \alpha_1 
\end{align*}
\]

The above formulas can be easily evaluated if all the actions belonging to 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
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:

```
<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.

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.

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 (load 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

---

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

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.
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 Depot 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.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Hand-Coded</th>
<th></th>
<th>Automatic</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>refs</td>
<td>tot</td>
<td>steps</td>
<td>abs</td>
</tr>
<tr>
<td>Depot1</td>
<td>28</td>
<td>73</td>
<td>106</td>
<td>12</td>
</tr>
<tr>
<td>Depot2</td>
<td>54</td>
<td>128</td>
<td>187</td>
<td>17</td>
</tr>
<tr>
<td>Depot3</td>
<td>488</td>
<td>340</td>
<td>841</td>
<td>38</td>
</tr>
<tr>
<td>Depot4</td>
<td>292</td>
<td>416</td>
<td>717</td>
<td>43</td>
</tr>
<tr>
<td>Depot5</td>
<td>845</td>
<td>100</td>
<td>950</td>
<td>71</td>
</tr>
<tr>
<td>Elevator1</td>
<td>10</td>
<td>51</td>
<td>63</td>
<td>8</td>
</tr>
<tr>
<td>Elevator2</td>
<td>17</td>
<td>142</td>
<td>163</td>
<td>15</td>
</tr>
<tr>
<td>Elevator3</td>
<td>18</td>
<td>226</td>
<td>248</td>
<td>4</td>
</tr>
<tr>
<td>Elevator4</td>
<td>18</td>
<td>359</td>
<td>383</td>
<td>23</td>
</tr>
<tr>
<td>Elevator5</td>
<td>19</td>
<td>740</td>
<td>767</td>
<td>28</td>
</tr>
<tr>
<td>Blocks1</td>
<td>11</td>
<td>41</td>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>Blocks2</td>
<td>18</td>
<td>104</td>
<td>125</td>
<td>14</td>
</tr>
<tr>
<td>Blocks3</td>
<td>40</td>
<td>450</td>
<td>497</td>
<td>44</td>
</tr>
<tr>
<td>Blocks4</td>
<td>45</td>
<td>479</td>
<td>532</td>
<td>48</td>
</tr>
<tr>
<td>Blocks5</td>
<td>55</td>
<td>301</td>
<td>364</td>
<td>48</td>
</tr>
</tbody>
</table>

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


