# Using Component Abstraction for Automatic Generation of Macro-Actions

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

### Abstract

Despite major progress in AI planning over the last few years, many interesting domains remain challenging for current planners. This paper presents component abstraction, an automatic and generic technique that can reduce the complexity of an important class of planning problems. Component abstraction uses static facts in a problem definition to decompose the problem into linked abstract components. A local analysis of each component is performed to speed up planning at the component level. Our implementation uses this analysis to statically build macro operators specific to each component. A dynamic filtering process keeps for future use only the most useful macro operators. We demonstrate our ideas in Depots, Satellite, and Rovers, three standard domains used in the third AI planning competition. Our results show an impressive potential for macro operators to reduce the search complexity and achieve more stable performance.

## Introduction

AI planning has recently achieved significant progress, both in theory and in practice. The last few years have seen major advances in the performance of planning systems, in part stimulated by the planning competitions held as part of the AIPS series of conferences (McDermott 2000; Bacchus 2001; Long & Fox 2003). However, many hard domains still remain a great challenge for the current capabilities of planning systems.

In this paper we present component abstraction, a technique for reducing planning complexity by decomposing a problem into linked components. Our method is automatic, uses no domain-specific knowledge, and can be applied to domains that use static facts for problem definition. A fact is static if it is true in all states of the problem search space. The problem decomposition uses static facts to define abstract components. Components with equivalent structure are assigned to the same abstract type.

For each abstract type we create macro operators that can speed up planning at the component level. A macro operator has the same formal definition as a normal operator, being characterized by a set of variables (parameters), a set of preconditions, a set of add effects, and a set of delete effects.



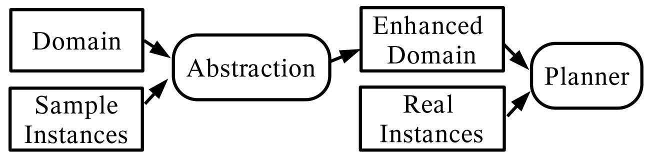


Figure 1: Integrating our abstraction approach into a standard planning framework. Abstraction includes component abstraction and macro generation.

We build a macro operator as an ordered sequence of operators linked through a mapping of the operators' variables. Applying a macro operator is semantically equivalent to applying all operators that compose the macro, respecting the macro's variable mapping. The preconditions and the effects of a macro operator are obtained using a straightforward set of rules that we describe in the fourth section. We generate the macros using a forward search in the space of macro operators. A set of heuristic rules is used to prune the search space and generate only macros that are likely to be useful. For best performance, we dynamically filter the initial set of macros and keep only the most effective ones for future use.

Figure 1 shows how our method can be integrated into a standard planning system. The resulting framework is planner independent and uses standard PDDL, with no need for language extensions. We use the domain definition and one or more problem instances as input for component abstraction and macro generation. The best macro operators that our method generates are added as new operators to the initial PDDL domain formulation, enhancing the initial set of operators. Once the enhanced domain formulation is available in standard PDDL, any planner could be used to solve problem instances.

In contrast, using a standard framework might result in reduced efficiency. If better performance is sought, the usage of component abstraction and macro operators could be encoded inside the search algorithm of a certain planner, for the price of increased engineering effort.

## Motivation

**Using Static Facts for Component Abstraction.** Complex real life domains often have static relationships between features present in the domain definition. Humans can abstract features connected through static relationships in one more complex functional unit. For example, a robot that carries a big hammer could be considered a single component, which has mobility as well as maintainance skills. Such a component can become a permanent object in our representation of the world, provided that no action can invalidate the static relation between the robot and the hammer (at the risk of misrepresenting reality, we assume the robot never considers the action of hammering itself).

The early standard planning domains did not make extensive use of static facts. Since such domains were strongly simplified representations of the real world, many specific constraints, including static relationships between domain features, are abstracted away from the domain definition. More recent planning benchmarks exhibit increased complexity, and static facts are used as part of their formulation. Consider *Depots*, a domain created as a combination of *Blocks* and *Logistics*. In *Depots*, a collection of low-level features such as a depot, a hoist, and a pallet can act like a single permanent component with multiple capabilities such as loading, unloading, or storing crates. The goal of component abstraction is to automatically identify such permanent components, treat them as functional units, and simplify planning through a component analysis process.

**Using Macro-Actions.** When AI planning is seen as heuristic search, a search space that originates from the initial state of a problem can be defined. Given a state in the search space, its successors are generated considering all actions that can be applied in the current state. A simple but useful standard model measures the size of a search space by two parameters: the branching factor $B$ and the search depth $D$. In this model, the size grows exponentially with $D$, and if $B > 2$, most of the search effort is spent on the deepest level achieved. The goal of using macro actions is to reduce $D$ for the price of slightly increasing $B$, obtaining a significant overall reduction of the search space.

## Contributions

This sub-section briefly summarizes the main contributions of the paper.

- We present a new type of automatic abstraction for AI planning, based on static relationships that link atomic problem constants.
- We use component abstraction to automatically generate macro actions that speed up planning at the component level. We present techniques for both building and filtering macros.
- We provide a performance analysis of our technique based on experiments in the standard domains *Depots*, *Rovers*, and *Satellite*. We show that using a small number of macros can greatly simplify the search for solutions of hard problem instances.

The rest of this paper is structured as follows: The next section reviews related work. The third section presents the component abstraction, the fourth presents the domain enhancement using macro operators, and the fifth section discusses our experimental results. The last section contains conclusions and ideas for further work.

# Related Work

Abstraction is a frequently used technique to reduce problem complexity in AI planning. Automatically abstracting planning domains has been explored by (Knoblock 1994). This approach builds a hierarchy of abstractions by dropping literals from the problem definition at the previous abstraction level. (Bacchus & Yang 1994) define a theoretical probabilistic framework to analyze the search complexity in hierarchical models. They also use some concepts of that model to improve Knoblock's abstraction algorithm. In this work, the abstraction consists of problem relaxation. In our approach, abstraction is achieved by identifying closely related atomic features and group them into abstract components.

(Long, Fox, & Hamdi 2002) use *generic types* and *active preconditions* to reformulate and abstract planning problems. As a result of the reformulation, sub-problems of the initial problem are identified and solved by using specialized solvers. Our approach has similarities with this work. Both formalisms try to cope with domain-specific features at the local level, reducing the complexity of the global problem. The difference is that we reformulate problems as a result of component abstraction, whereas in the cited work reformulation is obtained by identifying various generic types of behavior and objects such as *mobile objects*.

Component abstraction has similarities with topological abstraction. The first paradigm uses several types of static facts for problem decomposition, whereas the second uses only one class of static facts, corresponding to the predicate that models topological relationships in the problem space. As we show in the third section, these methods are also different in a significant way, using different types of static predicates for abstraction. (Botea, Müller, & Schaeffer 2003) use topological abstraction as a basis for hierarchical planning and propose a PDDL extension for supporting this. Using topological abstraction to speed up planning in a reinforcement learning framework has been proposed in (Precup, Sutton, & Singh 1997). In this work, the authors define macro actions as *offset-casual* policies. In such a policy, the probability of an atomic action depends not only on the current state, but also on the previous states and atomic actions of the policy. Learning macro actions in a grid robot planning domain induces a topological abstraction of the problem space.

In single-agent search, macro-moves can be considered as simple plans and are arguably the most successful planning idea to make its way into games/puzzle practice. Macro moves have successfully been used in the sliding-tile puzzle (Korf 1985). Two of the most effective concepts used in the Sokoban solver *Rolling Stone*, tunnel and goal macros, are applications of this idea (Junghanns 1999). Hernádvölgyi uses macro-moves for solving Rubik's Cube puzzles (Hernádvölgyi 2001). While these methods are application-specific, our approach is generic, building macros with no prior domain-specific knowledge.

| | |
|---|---|
| (AT PALLET0 DEPOT0) | (CLEAR CRATE1) |
| (AT HOIST0 DEPOT0) | (CLEAR CRATE0) |
| (AT PALLET1 DISTRIBUTOR0) | (CLEAR PALLET2) |
| (AT HOIST1 DISTRIBUTOR0) | (AT TRUCK0 DISTRIBUTOR1) |
| (AT PALLET2 DISTRIBUTOR1) | (AT TRUCK1 DEPOT0) |
| (AT HOIST2 DISTRIBUTOR1) | (AVAILABLE HOIST0) |
| | (AVAILABLE HOIST1) |
| | (AVAILABLE HOIST2) |
| | (AT CRATE0 DISTRIBUTOR0) |
| | (ON CRATE0 PALLET1) |
| | (AT CRATE1 DEPOT0) |
| | (ON CRATE1 PALLET0) |

Figure 2: Initial state of a Depots problem.

## Component Abstraction in Planning

Component abstraction is a generic technique that decomposes a planning problem into linked components, based on PDDL formulations of the problem and the corresponding domain. For a domain, abstracting different problems may produce different components and abstract types, according to the size and the structure of each problem. Local analysis of components can be used to reduce the complexity of the initial problem.

Component abstraction is a two-step procedure. First, we identify *static* facts in the problem definition. A fact is an instantiation of a domain predicate, i.e., a predicate whose parameters have been instantiated to concrete problem constants. A fact $f$ is static if $f$ is part of the initial state of the problem and no operator can delete it. Second, we use static facts to build the problem components. An abstract component contains problem constants linked by static facts.

We use problem 1 in the Depots test suite used in the third planning competition (Long & Fox 2003) as a running example. Figure 2 shows the initial state of the problem. In Depots, stacks of *crates* can be built on top of *pallets* using *hoists* that are located at the same *place* as the pallets. A place can be either a *depot* or a *distributor*. *Trucks* can transport crates from one place to another. For more information on the competition, including the complete definition of the domains cited in this paper, see (Long & Fox 2003) or visit the url `http://www.cis.strath.ac.uk/~derek/competition.html`.

### Identifying Static Facts

We use the set of the domain operators $\mathcal{O}$ to partition the predicate set $\mathcal{P}$ into two disjoint sets, $\mathcal{P} = \mathcal{P}_F \cup \mathcal{P}_S$, corresponding to *fluent* and *static* predicates. Assume that we represent an operator $o \in \mathcal{O}$ as a structure

$$o = (V(o), P(o), A(o), D(o)),$$

where $V(o)$ is the variable set, $P(o)$ is the precondition set, $A(o)$ is the set of add effects, and $D(o)$ is the set of delete effects. A predicate $p$ is fluent if $p$ is part of an operator's effects (either positive or negative):

$$p \in \mathcal{P}_F \Leftrightarrow \exists o \in \mathcal{O} : p \in A(o) \cup D(o).$$

Otherwise, we say that $p$ is static ($p \in \mathcal{P}_S$).

Before we determine fluent and static predicates, we have to address the issue of hierarchical types. In a domain with hierarchical types, instances of the same predicate can be both static and fluent. Consider again the Depots domain, which uses such a type hierarchy. Type LOCATABLE has four atomic sub-types: PALLET, HOIST, TRUCK, and CRATE. Type PLACE has two atomic sub-types: DEPOT and DISTRIBUTOR. Predicate (AT ?L - LOCATABLE ?P - PLACE), which indicates that object ?L is located at place ?P, corresponds to eight specialized predicates at the atomic type level. Here we show two such predicates, one static and one fluent. Predicate (AT ?P - PALLET ?D - DEPOT) is static, as there is no operator that adds, deletes, or moves a pallet. Predicate (AT ?C - CRATE ?D - DEPOT) is fluent. For instance, operator LIFT deletes a fact corresponding to this predicate.

To address the issue of hierarchical types, we use a *ground* domain formulation where all types are *ground types* at the lowest level in the hierarchy. We expand each predicate into a set of ground predicates whose arguments have ground types. Similarly, *ground operators* have variable types from the lowest hierarchy level. Component abstraction and macro generation are done at the ground level. After building the macros, we restore the type hierarchy of the domain. Similar macro operators with ground types are merged into one macro operator with hierarchical types, achieving a compact macro representation.

After determining fluent and static predicates, all facts corresponding to static predicates are considered static facts. Figure 2 shows the static facts of our Depots example in the left side of the picture, and the fluent facts in the right side. In this example, all static facts model the relationship between either a hoist or a pallet, and its location.

In our current implementation we ignore static predicates that are unary or have variables of the same type. The latter can model topological relationships and lead to large components. See the next subsection for a discussion.

### Building Abstract Components

Abstract components contain constants linked by static facts. Table 1 shows the abstract components of our example. We obtain three abstract components, each containing a pallet, a hoist, and either a depot or a distributor. In this example, the decomposition is straightforward, since the components are not linked by static facts. For instance, there are no static facts that place the same hoist at two different locations.

However, in general the graph of constants linked by all static facts can be connected. This often happens in domains such as Satellite or Rovers. Consider the Rovers domain, where predicates (STORE_OF ?S - STORE ?R - ROVER), (ON_BOARD ?C - CAMERA ?R - ROVER), (SUPPORTS ?C - CAMERA ?M - MODE), (CALIBRATION_TARGET ?C - CAMERA ?O - OBJECTIVE), and (VISIBLE_FROM ?O - OBJECTIVE ?W - WAYPOINT) are static. Assume that we want to build the components of the Rovers problem partially shown in Figure 3. If we use all static facts to create the components, we end up with one big component. To avoid this,

| Comp. | Constants | Facts |
|---|---|---|
| C0 | DEPOT0<br>HOIST0<br>PALLET0 | (AT PALLET0 DEPOT0)<br>(AT HOIST0 DEPOT0) |
| C1 | DISTRIBUTOR0<br>HOIST1<br>PALLET1 | (AT PALLET1 DISTRIBUTOR0)<br>(AT HOIST1 DISTRIBUTOR0) |
| C2 | DISTRIBUTOR1<br>HOIST2<br>PALLET2 | (AT PALLET2 DISTRIBUTOR1)<br>(AT HOIST2 DISTRIBUTOR1) |

Table 1: Abstract components built for the Depots example.

| | |
|---|---|
| (STORE_OF STORE0 ROVER0) | (VISIBLE_FROM OBJ0 POINT0) |
| (STOREI_OF STORE1 ROVER1) | (VISIBLE_FROM OBJ0 POINT1) |
| (ON_BOARD CAM0 ROVER0) | (VISIBLE_FROM OBJ0 POINT2) |
| (ON_BOARD CAM1 ROVER1) | (VISIBLE_FROM OBJ0 POINT3) |
| (SUPPORTS CAM0 COLOUR) | (VISIBLE_FROM OBJ1 POINT0) |
| (SUPPORTS CAM0 HIGH_RES) | (VISIBLE_FROM OBJ1 POINT1) |
| (SUPPORTS CAM1 COLOUR) | (VISIBLE_FROM OBJ1 POINT2) |
| (SUPPORTS CAM1 HIGH_RES) | (VISIBLE_FROM OBJ1 POINT3) |
| (CALIBRATION_TARGET CAM0 OBJ1) | |
| (CALIBRATION_TARGET CAM1 OBJ1) | |

Figure 3: Partial initial state of a Rovers problem. We show only the static facts that can be used for component abstraction.

we use a more general method for problem decomposition, which we describe below. First we show how the method works in the Rovers sample problem. Next we provide the formal description, including pseudo-code.

**Detailed Example.** Table 2 shows how component abstraction works in the sample Rovers problem. The method starts building components using a randomly chosen domain type, which in our example is CAMERA. The steps summarized in the table correspond to the following actions:

- Step 0. We create one abstract component for each constant of type CAMERA: COMPONENT0 contains CAM0, and COMPONENT1 contains CAM1. Next we iteratively extend the components created at Step 0. One extension step uses a static predicate that has at least one variable type already encoded into the components.
- Step 1. We choose predicate (SUPPORTS ?C - CAMERA ?M - MODE), which has a variable of type camera. First we check if static facts based on this predicate keep the existing components separated. These static facts are (SUPPORTS CAM0 COLOR), (SUPPORTS CAM0 HIGH_RES), (SUPPORTS CAM1 COLOR), and (SUPPORTS CAM1 HIGH_RES). The test fails, as constants COLOUR and HIGH_RES would be part of both components. We therefore do not use this predicate for component extension (we say we invalidate the predicate).
- Step 2. Similarly, we invalidate predicate (CALIBRATION_TARGET ?C - CAMERA ?O - OBJECTIVE), which would add constant OBJ1 to both components.
- Step 3. We analyse predicate (ON_BOARD ?C - CAMERA ?R - ROVER) and use it for component extension. The components are expanded as shown in Table 2, Step 3.
- Step 4. We consider predicate (STORE_OF ?S - STORE ?R - ROVER), whose type ROVER has previously been encoded into the components. This predicate extends the components as presented in Table 2, Step 4.

After Step 4 is completed, no further component extension can be performed. There are no other static predicates using at least one of the component types to be tried for further extension. At this moment we evaluate the quality of the decomposition. In this example the decomposition is good (see discussion below) and the process terminates. Otherwise, the decomposition process restarts with another domain type.

**Algorithm.** Figure 4 shows our component abstraction method in pseudo-code. The procedure iteratively tries to build the components starting from a domain type $t$ randomly chosen. At the beginning, each constant of type $t$ becomes the seed of an abstract component. The components are greedily extended by adding new facts and constants, so that no constant is part of two distinct components. If a good decomposition is found starting from $t$, the procedure returns. Otherwise, we reset all the internal data structures (e.g., Open, Closed, the validation flag for predicates, and the abstract components) and restart the process using another randomly picked initial type.

Method $extendComponents(p)$ extends the components using static facts based on predicate $p$. Each fact $f$ based on $p$ becomes part of a component. Assume $f$ uses constants $c_1$ and $c_2$. If $c_1$ is part of component $C$ and $c_2$ is not assigned to a component yet, then $c_2$ and $f$ become part of $C$ too. If neither $c_1$ nor $c_2$ are part of a previously built component, a new component that contains $f$, $c_1$, and $c_2$ is created.

We evaluate the quality of a decomposition according to the size of the built components. We measure the size as the number of ground types used in a component. In our experiments we limited the size range of components between 2 and 4. The lower limit is trivial, since an abstract component should put together at least two ground types connected by a static predicate. The upper limit was heuristically set so that the abstraction does not end-up building one large component. These relatively small values are also consistent to our goal of limiting the size and the number of generated macro operators. We discuss this issue in more detail in the next section.

**Component Abstraction vs Topological Abstraction.** Our decomposition method can consider only a subset of static predicates to participate in the process of building abstract components. Given a static predicate $p$, we use the same validation rule for all facts based on $p$. If $p$ is considered for abstraction, then each static fact based on $p$ will be part of an abstract component. If $p$ is ignored, then no static fact based on $p$ can be part of an abstract component.

This choice is useful for building components that con-

| Step | Current Predicate | Validated Predicate | COMPONENT0 Constants | COMPONENT0 Facts | COMPONENT1 Constants | COMPONENT1 Facts |
|---|---|---|---|---|---|---|
| 0 | | | CAM0 | | CAM1 | |
| 1 | (SUPPORTS ?C - CAMERA ?M - MODE) | NO | CAM0 | | CAM1 | |
| 2 | (CALIBRATION_TARGET ?C - CAMERA ?O - OBJECTIVE) | NO | CAM0 | | CAM1 | |
| 3 | (ON_BOARD ?C - CAMERA ?R - ROVER) | YES | CAM0 ROVER0 | (ON_BOARD CAM0 ROVER0) | CAM1 ROVER1 | (ON_BOARD CAM1 ROVER1) |
| 4 | (STORE_OF ?S - STORE ?R - ROVER) | YES | CAM0 ROVER0 STORE0 | (ON_BOARD CAM0 ROVER0) (STORE_OF STORE0 ROVER0) | CAM1 ROVER1 STORE1 | (ON_BOARD CAM1 ROVER1) (STORE_OF STORE1 ROVER1) |

Table 2: Building abstract components for the Rovers example.

```
bool componentAbstraction() {
  for (each type t chosen in random order) {
    resetAllStructures();
    pushToQueue(Open, t);
    for (each constant c_i with type t)
      C_i = createComponent(c_i);
    while (!emptyQueue(Open)) {
      t_1 = popFromQueue(Open);
      pushToQueue(Closed, t_1);
      for (each static predicate p that uses t_1)
        if (predConnectsComponents(p)) {
          setPredicate(p, INVALID);
          continue;
        }
        else{
          setPredicate(p, VALID);
          extendComponents(p);
          for (each type t_2 used in p)
            if (!(t_2 ∈ Open ∪ Closed))
              pushToQueue(Open, t_2);
        }
    }
    if (evaluateDecomposition() == OK)
      return true;
  }
  return false;
}
```

Figure 4: Component abstraction in pseudo-code.

tain constants of different types (e.g., a place, a pallet, and a hoist). In contrast, this rule does not work if we want to cluster constants modelling the topology of a problem. In topological abstraction, the goal is to cluster a set of similar constants, representing locations. Locations are connected by symmetrical facts corresponding to a predicate $p$ that models the neighborhood relationship. Topological clustering would consider some of these facts for building the components and ignore others. We would not apply the same validation rule to all facts corresponding to $p$.

**Abstract Types.** After building components, we identify components with identical structure and assign them to the same abstract type. Consider a component $c = (C(c), F(c))$, where $C(c)$ is the set of constants and $F(c)$ is the set of static facts of $c$. Note that a fact $f \in F(c)$ is a predicate whose variables have been instantiated to constants from $C(c)$: $f(c^1, ..., c^k) \in F(c), c^i \in C(c)$.

We say that two components $c_1$ and $c_2$ have identical structure if:

1. $|C(c_1)| = |C(c_2)|$; and
2. $|F(c_1)| = |F(c_2)|$; and
3. there is a permutation $p : C(c_1) \rightarrow C(c_2)$ such that
   - $\forall f(c_1^1, ..., c_1^k) \in F(c_1) : f(p(c_1^1), ..., p(c_1^k)) \in F(c_2)$;
   - $\forall f(c_2^1, ..., c_2^k) \in F(c_2) : f(p^{-1}(c_2^1), ..., p^{-1}(c_2^k)) \in F(c_1)$;

The abstract type of a component is obtained from the component structure by replacing each constant with a generic variable having the same type as the constant. In the Rovers example, both components belong to the same abstract type. In the Depots example shown in Table 1, we define two abstract types: one for C0, and one for both C1 and C2. For an abstract type we perform a local analysis to reduce the problem complexity. In this paper we show how the local analysis can be used to generate macro operators. This is only one possible way to exploit component abstraction. Other ideas will be discussed briefly in the Future Work section. Generating macro operators is discussed in detail in the next section.

## Creating and Using Macro-Operators

A macro-operator $m$ is formally equivalent to a normal operator: it has a set of variables $V(m)$, a set of preconditions $P(m)$, a set of add effects $A(m)$, and a set of delete effects $D(m)$. We enhance the initial domain formulation by adding macro-operators to the initial operator set.

A new macro-operator is built as a linear sequence of operators. The variable set $V(m)$ is obtained from the variable sets of the contained operators together with a variable mapping showing how the initial sets overlap. The operator

```
bool addOperatorToMacro(o, m, vm) {
  for (each precondition p ∈ P(o)) {
    if (p ∈ D(m))
      return false;
    if (not p ∈ A(m) ∪ P(m))
      P(m) = P(m) ∪ {p};
  }
  for (each delete effect d ∈ D(o)) {
    if (d ∈ A(m))
      A(m) = A(m) − {d};
    D(m) = D(m) ∪ {d};
  }
  for (each add effect a ∈ A(o)) {
    if (a ∈ D(m))
      D(m) = D(m) − {a};
    A(m) = A(m) ∪ {a};
  }
  return true;
}
```

Figure 5: Adding operators to a macro.

sequence and the variable mapping completely determine a macro. Knowing what variables are common to two operators further determines what predicates are common in the operators' precondition and effect sets.

The macro precondition and effect sets are initially empty. Adding a new operator $o$ to a macro $m$ modifies $P(m)$, $A(m)$, and $D(m)$ as shown in Figure 5. Parameter $o$ is an operator, $m$ is a macro, and $vm$ is a variable mapping. The variable mapping is used to check the identity between operator's predicates and macro's predicates. We assume that the decision whether the operator should be added to the macro is made before calling this function. The function shown in Figure 5 rejects (i.e., returns false) only operators that try to use as precondition a false predicate. See the next subsection for more details on selecting an operator to be added to a macro. In Figure 6 we show the complete definition of the macro operator UNLOAD_DROP, from Depots and the operators that it contains.

Macro operators are obtained in a two-step process. First, an extended set of macros is built and next the macros are filtered in a quick training process. Since analysis based on empirical evidence shows that the extra information added to a domain definition should be quite small, the methods that we describe next tend to minimize the number of macros and their "size" (i.e., number of variables, preconditions and effects). The static macro generation uses many constraints for pruning the space of macro operators, and discards "large" macros. Furthermore, the dynamic filtering keeps only two macros for solving future problems.

## Macro Generation

We build macro operators for an abstract type by performing a forward search in the space of macro operators. Macro operators built for an abstract type $t$ should perform local processing for components of type $t$. We build such an operator $m$ based on the structure of $t$: $m$ uses at least one static predicate of $t$ as precondition.

```
(:action UNLOAD_DROP
  :parameters
    (?h - hoist ?c - crate ?t - truck ?p - place ?s - surface)
  :precondition
    (and (at ?h ?p) (in ?c ?t) (available ?h)
    (at ?t ?p) (clear ?s) (at ?s ?p))
  :effect
    (and (not (in ?c ?t)) (not (clear ?s))
    (at ?c ?p) (clear ?c) (on ?c ?s))
)
(:action UNLOAD
  :parameters
    (?x - hoist ?y - crate ?t - truck ?p - place)
  :precondition
    (and (in ?y ?t) (available ?x) (at ?t ?p) (at ?x ?p))
  :effect
    (and (not (in ?y ?t)) (not (available ?x)) (lifting ?x ?y))
)
(:action DROP
  :parameters
    (?x - hoist ?y - crate ?s - surface ?p - place)
  :precondition
    (and (lifting ?x ?y) (clear ?s) (at ?s ?p) (at ?x ?p))
  :effect
    (and (available ?x) (not (lifting ?x ?y)) (at ?y ?p)
    (not (clear ?s)) (clear ?y) (on ?y ?s))
)
```

Figure 6: PDDL definition of macro UNLOAD_DROP and the operators that it contains.

The root state of the search represents an empty macro with no operators. A search step appends an operator to the current macro, with a mapping between the operator variables and the macro variables. The search is selective, as it includes a set of rules for pruning the search tree and for validating a built macro operator. Validated macros can be seen as goal states in our search space. The purpose of the search is to enumerate all valid macro operators.

Pruning is performed according to the following rules:

- The *negated precondition rule* prunes operators with a precondition that matches one of the current delete effects of the macro operator. This rule avoids building incorrect macros where a predicate should be both true and false.
- The *repetition rule* requires that operators that generate cycles cannot be added to a macro. A macro with cycle is either useless, when an empty effect set is produced or it can be written in a shorter form, eliminating the cycle. A cycle in a macro is detected when the effects of the first $k_1$ operators are the same as for the first $k_2$ operators, with $k_1 < k_2$. In particular, if $k_1 = 0$ then the first $k_2$ operators have no effect.
- The *chaining rule* states that, if operators $o_1$ and $o_2$ are consecutive in a macro, $o_2$ should use as precondition a positive effect of $o_1$. This is motivated by the idea that the action sequence of a macro should have a coherent meaning.

- The *locality rule* states that a macro action cannot change two distinct abstract components at the same time.
- Finally, we impose a *maximal length* for a macro.

Macro operators built in the search are evaluated according to the size rule. We discard "large" macros, i.e., macros with many preconditions, effects, and variables. Large macros are less likely to help with the search. First, a large macro can add a significant overhead per node in the planner's search. Second, a large number of elements is a hint that the macro might not be so useful, as the operators do not "chain" well.

### Macro Filtering

The goal of filtering is to reduce the number of macros and use only the most efficient ones for solving problems. Two main reasons support the need for a dynamic filtering algorithm. First, adding more operators to a domain increases the cost per node in the planner's search. Operators whose overhead is larger than the possible benefits should be discarded. Second, some of the generated macro operators might contain mutex predicate tuples as part of their preconditions or effects. If used in the domain formulation, macro operators containing mutexes are never instantiated as possible macro actions (moves), but increase the cost per node.

The problem of dynamic macro filtering was not hard, since we only wanted to obtain the top few elements from a relatively small set of macro operators. Therefore, we could use a method that was simple, fast to implement, and used no planner internal information. We count how often a macro operator is instantiated as an action in the problem solutions found by the planner. The more often a macro has been used in the past, the greater the chance that the macro will be useful in the future. The technique turned out to be efficient, since the filtering process quickly converges to a small set of useful macros. We spent no effort to find a "better" scoring heuristic or tune the values of method parameters before we ran the experiments reported in this paper.

Macro operators have weights that estimate their efficiency. Initially, all macro weights are set to $0$. Each time a macro is present in a plan, we increase its weight by the number of occurrences of the macro in the plan plus a bonus of $10$. We use the simplest problems in a domain for a training process. For these problems, we allow the domain to use all macro operators, giving each macro a chance to participate in a solution plan and increase its weight. After the training is over, we allow only the $2$ best macro operators to be part of the domain definition. Our experiments showed that using such a small number of macro operators balances well the benefits and the additional cost per node that macro operators generate. In the domains that we used, only one or two macros that our technique generates are helpful for reducing the search. However, *all* operators added to a domain generate additional cost per node in the planner.

Even if the method based on action counting worked well in our first experiments, designing a better algorithm for learning macro weights is one of our main interests for the future. To update the weight of a macro $m$, we would compare the search effort for solving a problem using the initial domain formulation to the search effort for solving the same problem with $m$ added to the domain operator set. We plan to use a comparison formula that should consider the variation from one domain formulation to the other for parameters such as number of expanded nodes, search time, or maximal search depth. This algorithm would use more CPU time for training, since we solve one training problem several times, once with no macros added to the domain formulation, and once for each macro considered for weight update.

## Experimental Results

### Experimental Setup

The implementation of our planning framework keeps the abstraction process separated from the planner. The result of abstraction is a new PDDL formulation of the domain, where the initial set of operators has been enhanced with the selected macro operators. The enhanced domain file can be used by a planner to solve problem instances, with no need for further problem abstraction.

We developed our tools for component abstraction and macro generation based on FF, version 2.3 (Hoffmann & Nebel 2001). This helped us perform quicker development, since we used the input parser and the internal data structures provided by FF. For solving planning problems, we used FF too. However, our method is planner independent and any general purpose planner could be used in our experiments.

We measured the performance of our technique on Depots, Satellite, and Rovers, three standard domains used in the third planning competition. These domains use static facts, making them suitable for our approach. Each domain exhibits interesting features: Depots uses hierarchical types, and Satellite and Rovers require a more general technique for component abstraction. The planning competition had several tracks (i.e., Numeric, Strips, etc.), each with the appropriate domain definition. In our experiments we used the Strips domain representation. We limited the experiments to Depots, Satellite, and Rovers because other competition domains were either not available in a Strips version, or not suitable for component abstraction. We consider that a domain is suitable for component abstraction if it uses static facts that do not model the domain topology.

For Depots we used the same test suite of $22$ problems as in the competition. This set includes problems which are difficult in the initial domain formulation, allowing us to show the advantages of using macro operators. For Rovers and Satellite, the problems used in the competition are easily solved by FF in the initial domain formulation, and there is not much room for performance improvement. For this reason, we extended the test set for each of these two domains with $20$ problems, obtaining test sets of $40$ problems each. We used the same problem generator as for the competition. The generator takes as parameters the number of objects of each type, the number of goals, and the value of the random seed.

In Satellite, each of the additional $20$ problems was generated with the same parameters as problem $20$ from the initial test-suite, except for the random seed parameter. In Rovers, problems generated with similar parameters as problem 20 are also easy. For this reason we generated additional problems on two difficulty levels. Problems in the range 21—30 have the same parameters as problem 20, except for the random seed. Problems 31—40 are more difficult. We increased the number of rovers, objectives, cameras, and goals to 15 each and preserved the initial value of 25 for the number of waypoints. In effect, we obtain larger data sets containing both easy and hard problems in the original domain definition, allowing us to make a more complete performance analysis. For each data set, the first 5 iterations run with all macros in use ("training mode"), while the rest of the problems were solved using a reduced number of macros ("solving mode").

## Analysis

Tables 4, 5 and 6 summarize the results for Depots, Rovers and Satellite respectively. We show the running time measured in seconds (Time), the number of expanded nodes (Nodes), and the solution length for each problem (Length). The timings were obtained on a machine with a 2 GHz AMD Athlon processor and 1 GByte of memory. The number of expanded nodes evaluates the search complexity in each domain formulation. For each main column (i.e., Time, Nodes, and Length), column $C$ shows the data corresponding to the classical domain formulation. For the columns Time and Nodes, column $M$ represents the results obtained when the macro enhancement domain formulation was used. The times reported for the enhanced formulation do not include the effort for component abstraction and macro generation. This processing is fast and can be amortized over many problems. For the macro enhanced formulation, we report two numbers for the solution length, each being relevant in a different way. $A$ counts each macro action as one step in the solution plan. The difference between $C$ and $A$ is a measure of how using macro operators reduces the search complexity. $G$ is the solution length at the ground level, where each macro is mapped to the corresponding sequence of actions. Comparing $G$ to $C$ is useful to evaluate how the solution quality is affected by our approach. It is interesting that our method sometimes finds shorter solutions.

In Satellite, problems 27 and 38 could not be solved using the original domain formulation with a time limit of 30 minutes. Our method produced the macros shown in Table 3. These macros have an important contribution to the search space reduction, except for CALIBRATE__TAKE_IMAGE. In Satellite, an instrument can be calibrated once, then used many times for taking pictures. For this reason, that macro is usually applied only once at the beginning of a plan.

The data show huge variations in problem difficulty when the original domain formulation is used, especially for Depots and Satellite. With the macro enhanced domain definition, the performance is much more stable. The difficulty level of hard problems can be reduced by several orders of magnitude. For example, using macro actions for problem 8 in Depots reduces the running time by a factor of 10,000 and expanded nodes by a factor of 1,000.

Next we focus our discussion in two directions: performance on hard problems and performance on easy problems. Because of the huge variation in terms of time and nodes between problems, it does not make sense to talk about "overall average" speed up or tree size. For a comprehensive theoretical and empirical analysis of the problem complexity in current benchmark domains for AI planning, see (Hoffmann 2001; 2002).

| Domain | Macro operators |
|---|---|
| Depots | UNLOAD__DROP<br>LIFT__LOAD |
| Rovers | SAMPLE_ROCK__DROP<br>SAMPLE_SOIL__DROP |
| Satellite | TURN_TO__TAKE_IMAGE<br>CALIBRATE__TAKE_IMAGE |

Table 3: Macro operators generated for our test domains (after dynamic filtering).

| | Time | | Nodes | | Length | | |
|---|---|---|---|---|---|---|---|
| | C | M | C | M | C | A | G |
| 1 | 0.00 | 0.00 | 20 | 12 | 10 | 7 | 11 |
| 2 | 0.01 | 0.01 | 33 | 25 | 15 | 10 | 16 |
| 3 | 0.03 | 0.05 | 318 | 123 | 37 | 18 | 30 |
| 4 | 648.42 | 0.54 | 173342 | 534 | 30 | 20 | 34 |
| 5 | 40.65 | 0.33 | 220433 | 403 | 72 | 37 | 59 |
| 6 | 620.98 | 10.05 | 789227 | 3848 | 91 | 48 | 81 |
| 7 | 0.02 | 0.03 | 148 | 77 | 27 | 17 | 25 |
| 8 | 1280.33 | 0.14 | 174031 | 142 | 44 | 26 | 45 |
| 9 | 1.63 | 0.84 | 2356 | 260 | 75 | 37 | 65 |
| 10 | 110.29 | 0.05 | 41784 | 45 | 29 | 15 | 25 |
| 11 | 0.36 | 1.95 | 574 | 716 | 63 | 43 | 67 |
| 12 | 10.04 | 6.11 | 5008 | 614 | 94 | 40 | 66 |
| 13 | 0.04 | 0.10 | 79 | 53 | 26 | 17 | 27 |
| 14 | 0.25 | 0.26 | 427 | 66 | 37 | 17 | 29 |
| 15 | 45.83 | 19.00 | 22421 | 2076 | 85 | 56 | 90 |
| 16 | 0.05 | 0.28 | 108 | 73 | 28 | 20 | 31 |
| 17 | 1.73 | 1.54 | 1600 | 178 | 38 | 19 | 29 |
| 18 | 1.83 | 5.23 | 533 | 199 | 60 | 42 | 65 |
| 19 | 0.42 | 0.68 | 430 | 85 | 47 | 25 | 40 |
| 20 | 29.92 | 14.58 | 6927 | 555 | 98 | 49 | 78 |
| 21 | 0.74 | 3.72 | 104 | 77 | 32 | 23 | 35 |
| 22 | 95.61 | 148.64 | 4524 | 1176 | 102 | 62 | 97 |

Table 4: Summary of results for Depots.

Our analysis for hard problems shows an impressive potential of macro actions for reducing problem complexity in terms of running time and expanded nodes. In this context, the main lesson that we have learned is that a very small number of macros can greatly simplify a hard problem.

For problems that FF solves easily there is little room for improvement. On average, our method reduces the number of expanded nodes, but the total running time can be greater. The explanation is that adding operators to a domain induces additional cost per node. Since for small problems the extra cost per node can exceed the node savings, reducing the overhead becomes important. We believe that most

| | Time | | Nodes | | Length | | |
|---|---|---|---|---|---|---|---|
| | C | M | C | M | C | A | G |
| 1 | 0.00 | 0.00 | 14 | 10 | 10 | 8 | 11 |
| 2 | 0.00 | 0.01 | 10 | 8 | 8 | 7 | 9 |
| 3 | 0.00 | 0.01 | 20 | 15 | 13 | 10 | 13 |
| 4 | 0.01 | 0.01 | 9 | 8 | 8 | 7 | 10 |
| 5 | 0.01 | 0.01 | 53 | 27 | 22 | 20 | 24 |
| 6 | 0.01 | 0.01 | 189 | 81 | 38 | 33 | 40 |
| 7 | 0.01 | 0.01 | 37 | 23 | 18 | 16 | 21 |
| 8 | 0.01 | 0.02 | 96 | 43 | 28 | 24 | 29 |
| 9 | 0.02 | 0.02 | 125 | 114 | 33 | 30 | 36 |
| 10 | 0.03 | 0.02 | 199 | 62 | 37 | 31 | 39 |
| 11 | 0.02 | 0.02 | 92 | 81 | 37 | 33 | 41 |
| 12 | 0.01 | 0.01 | 35 | 29 | 19 | 19 | 22 |
| 13 | 0.05 | 0.03 | 327 | 157 | 46 | 38 | 47 |
| 14 | 0.02 | 0.02 | 71 | 55 | 28 | 26 | 31 |
| 15 | 0.05 | 0.05 | 281 | 322 | 42 | 38 | 46 |
| 16 | 0.06 | 0.03 | 468 | 132 | 46 | 38 | 46 |
| 17 | 0.08 | 0.05 | 246 | 160 | 49 | 45 | 53 |
| 18 | 0.15 | 0.16 | 307 | 254 | 42 | 39 | 46 |
| 19 | 1.20 | 0.68 | 1144 | 607 | 74 | 65 | 74 |
| 20 | 3.83 | 1.51 | 2176 | 898 | 96 | 82 | 97 |
| 21 | 0.60 | 0.28 | 313 | 138 | 46 | 43 | 51 |
| 22 | 1.07 | 0.56 | 1275 | 521 | 88 | 80 | 93 |
| 23 | 0.39 | 0.26 | 327 | 229 | 60 | 55 | 63 |
| 24 | 0.80 | 0.36 | 481 | 198 | 61 | 55 | 66 |
| 25 | 1.03 | 0.46 | 639 | 302 | 58 | 55 | 64 |
| 26 | 0.99 | 0.61 | 828 | 465 | 71 | 59 | 71 |
| 27 | 0.71 | 0.47 | 756 | 409 | 53 | 51 | 59 |
| 28 | 0.80 | 0.58 | 398 | 281 | 64 | 58 | 68 |
| 29 | 1.65 | 0.56 | 1078 | 341 | 91 | 74 | 92 |
| 30 | 40.88 | 8.67 | 6788 | 1771 | 155 | 122 | 145 |
| 31 | 38.17 | 12.16 | 4404 | 1454 | 141 | 110 | 139 |
| 32 | 40.15 | 6.92 | 6615 | 1147 | 137 | 111 | 138 |
| 33 | 66.04 | 15.82 | 6841 | 2079 | 142 | 132 | 154 |
| 34 | 21.30 | 5.06 | 2604 | 655 | 114 | 94 | 117 |
| 35 | 23.20 | 6.14 | 1984 | 562 | 88 | 78 | 97 |
| 36 | 21.60 | 10.10 | 2441 | 1175 | 106 | 89 | 106 |
| 37 | 13.35 | 1.96 | 2124 | 375 | 110 | 88 | 110 |
| 38 | 11.02 | 4.59 | 1056 | 449 | 76 | 69 | 84 |
| 39 | 34.01 | 7.41 | 3899 | 961 | 113 | 102 | 122 |
| 40 | 47.91 | 11.64 | 5875 | 1514 | 127 | 110 | 129 |

Table 5: Summary of results for Rovers.

| | Time | | Nodes | | Length | | |
|---|---|---|---|---|---|---|---|
| | C | M | C | M | C | A | G |
| 1 | 0.01 | 0.00 | 15 | 15 | 9 | 9 | 9 |
| 2 | 0.00 | 0.00 | 24 | 24 | 13 | 13 | 14 |
| 3 | 0.00 | 0.01 | 19 | 19 | 11 | 11 | 12 |
| 4 | 0.01 | 0.00 | 27 | 27 | 18 | 18 | 19 |
| 5 | 0.00 | 0.01 | 28 | 28 | 16 | 16 | 17 |
| 6 | 0.01 | 0.01 | 47 | 47 | 20 | 20 | 22 |
| 7 | 0.02 | 0.02 | 54 | 54 | 22 | 22 | 24 |
| 8 | 0.00 | 0.02 | 54 | 54 | 28 | 28 | 30 |
| 9 | 0.03 | 0.03 | 73 | 73 | 35 | 35 | 38 |
| 10 | 0.05 | 0.05 | 87 | 87 | 35 | 35 | 39 |
| 11 | 0.08 | 0.08 | 91 | 91 | 34 | 34 | 37 |
| 12 | 0.16 | 0.17 | 91 | 91 | 43 | 43 | 45 |
| 13 | 0.72 | 1.81 | 243 | 141 | 61 | 37 | 66 |
| 14 | 0.23 | 1.30 | 84 | 82 | 42 | 27 | 46 |
| 15 | 0.74 | 2.18 | 182 | 150 | 52 | 35 | 56 |
| 16 | 1.02 | 3.60 | 180 | 100 | 53 | 36 | 59 |
| 17 | 0.99 | 1.59 | 152 | 59 | 48 | 33 | 53 |
| 18 | 0.14 | 0.38 | 75 | 25 | 35 | 22 | 38 |
| 19 | 0.83 | 2.85 | 365 | 124 | 73 | 42 | 73 |
| 20 | 14.41 | 3.74 | 5889 | 138 | 107 | 57 | 102 |
| 21 | 168.55 | 26.67 | 65387 | 961 | 119 | 77 | 131 |
| 22 | 1077.47 | 5.19 | 290657 | 111 | 89 | 53 | 93 |
| 23 | 59.15 | 8.65 | 26970 | 333 | 118 | 63 | 106 |
| 24 | 88.02 | 7.27 | 44890 | 407 | 115 | 75 | 131 |
| 25 | 0.94 | 5.05 | 517 | 324 | 105 | 73 | 132 |
| 26 | 13.07 | 8.77 | 4605 | 272 | 97 | 62 | 111 |
| 27 | – | 3.97 | – | 175 | – | 47 | 84 |
| 28 | 6.09 | 8.77 | 2734 | 332 | 118 | 65 | 111 |
| 29 | 62.49 | 26.70 | 25616 | 961 | 98 | 76 | 133 |
| 30 | 1.02 | 13.61 | 436 | 468 | 77 | 65 | 110 |
| 31 | 3.62 | 5.04 | 1350 | 169 | 86 | 54 | 96 |
| 32 | 494.15 | 5.81 | 169783 | 200 | 107 | 66 | 117 |
| 33 | 42.76 | 4.48 | 15057 | 160 | 112 | 61 | 107 |
| 34 | 88.47 | 9.22 | 28012 | 267 | 95 | 61 | 110 |
| 35 | 118.80 | 5.28 | 35330 | 133 | 66 | 48 | 83 |
| 36 | 2.28 | 6.87 | 733 | 216 | 90 | 57 | 99 |
| 37 | 45.83 | 4.69 | 17134 | 187 | 98 | 63 | 110 |
| 38 | – | 9.37 | – | 292 | – | 61 | 104 |
| 39 | 27.12 | 5.95 | 11981 | 314 | 133 | 70 | 118 |
| 40 | 1.40 | 2.53 | 671 | 148 | 77 | 50 | 86 |

Table 6: Summary of results for Satellite.

of the overhead comes from computing the heuristic node evaluation. FF computes the heuristic in an automatic and generic way, by solving a relaxed problem where operators do not have delete effects. This computation is performed in a GRAPHPLAN framework. We didn't explore how to reduce the overhead in GRAPHPLAN, but we believe that this could be possible, since a macro action and the actions that compose it encode similar information. The issue of extended cost per node may not be present for planners that use application specific heuristics, which are usually cheaper to compute.

Our experience also suggests that the cost per node quickly increases with the number and the "size" of macros added. We evaluate the size of a macro by the number of preconditions, effects, and variables. This is one of the reasons for using only a small number of macros in an enhanced domain formulation.

In Rovers, the results contain not only fewer expanded nodes, but also better times in the enhanced domain formulation for most of the problems, including the easy ones. The ratio between running time and number of expanded nodes remains about the same, suggesting that, in this domain, the macro operators do not generate significant extra cost per node.

## Conclusion and Future Work

We presented component abstraction, a generic and automatic technique for decomposing a planning problem into linked components. We used component abstraction to build macro operators that speed up planning at the component level. We explored our technique in standard planning domains, showing that a small set of macro operators added to a domain definition can help in reducing complexity of hard problem instances and achieving more stable performance.

We have many ideas to explore in the future. We plan to extend macro generation for domains without static predicates, exploiting the advantages of macro actions on more general classes of problems. We could use problem solutions as a basis for generating macro actions. A plan can be represented as a directed graph, where nodes are actions and edges model the relative order between actions in the solution. A macro action can be generated as a linear path in the solution graph.

We also want to extend our component analysis, aiming to obtain a better set of operators for a component. This means not only adding new operators, but also removing or changing existing operators. The model could either guarantee the completeness or use heuristic rules to minimize the failures caused by the incompleteness.

We are interested in building a tool for automatic reformulation of abstracted problems. The challenge is to express an abstracted problem in standard PDDL, with no need for language capabilities to support hierarchical planning. Several constants and static facts that compose an abstract component would be replaced in the problem formulation by one object corresponding to the abstract component. The initial operators would be replaced by new operators corresponding to abstract components, resulting in a simpler, more compact, and more scalable problem definition. This might also reduce the cost per node in the planner's search.

## References


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

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

Botea, A.; Müller, M.; and Schaeffer, J. 2003. Extending PDDL for Hierarchical Planning and Topological Abstraction. In *Proceedings of the ICAPS-03 Workshop on PDDL*, 25–32.

Hernádvölgyi, I. 2001. Searching for Macro-operators with Automatically Generated Heuristics. In *14th Canadian Conference on Artificial Intelligence, AI 2001*, 194–203.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J. 2001. Local search topology in planning benchmarks: An empirical analysis. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, 453–458.

Hoffmann, J. 2002. Local search topology in planning benchmarks: A theoretical analysis. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)*. 379-387.

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

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

Korf, R. E. 1985. Macro-operators: A weak method for learning. *Artificial Intelligence* 26(1):35–77.

Long, D., and Fox, M. 2003. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research* 20:1–59. Special Issue on the 3rd International Planning Competition.

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

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

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