Abstraction in Planning via Model Checking*

A. Cimatti
IRST, Povo, 38100 Trento, Italy
Mechanized Reasoning Group
cimatti@irst.itc.it

F. Giunchiglia
IRST, Povo, 38100 Trento, Italy
Mechanized Reasoning Group
DISA, University of Trento
Via Inama 5, 38100 Trento, Italy
fausto@irst.itc.it

M. Roveri
IRST, Povo, 38100 Trento, Italy
Mechanized Reasoning Group
DSI, University of Milano
Via Comelico 39, 20135 Milano, Italy
roveri@irst.itc.it

Abstract

Planning via Model Checking is a novel approach to planning. It is based on the reformulation of a planning problem to the exploration of a finite state automaton. Automata exploration can be efficiently performed with Model Checking, a formal verification technique widely applied in design of industrial systems. For very large domains, however, a state explosion can arise. Abstractions can be used to tackle the problem. In this paper we show how Abstraction techniques developed in classical planning can be lifted to the Planning via Model Checking paradigm.

Introduction

Planning via Model Checking is a novel approach to planning (Cimatti *et al.* 1997). It is based on the reformulation of a planning problem to the exploration of a finite state automaton. The advantages of this approach are many-fold. First, the domain can be described with an expressive language, allowing for the expression of ramifications, constraints and nondeterministic actions. Second, very efficient model checking techniques (Burch *et al.* 1992; Clarke, Grumberg, & Long 1994), developed in the context of formal verification, are used to compactly encode and exhaustively explore the automaton. Based on this paradigm, more expressive properties can be analyzed, such as the synthesis of plans which are guaranteed to achieve a goal in non-deterministic domains for every possible behavior of the environment (Cimatti, Roveri, & Traverso 1998).

For very large domains, however, a state explosion can arise. That is, the exhaustive exploration of the automaton turns out to be unacceptably expensive. Abstraction is a well known divide-and-conquer technique, used in different forms to tackle large search problems (see (Giunchiglia & Walsh 1992) for a survey). Goal of this document is to describe how abstraction techniques devised in classical planning (see, e.g., (Sacerdoti 1974; Knoblock 1994; Bacchus & Yang 1994; Yang, Tenenberg, & Woods 1996; Bundy *et al.* 1996)) can be combined with the Planning via Model Checking approach. Our main goal is to scale up to very large problems, where computation space turns out to be the problem, rather than trying to gain efficiency and thus reducing time.

This paper is structured as follows. Section overviews Planning. Section briefly summarizes and discusses the Planning via Model Checking paradigm. Section reviews the application of abstractions in planning, focusing on Abstrips abstractions, and shows how it is possible to lift abstraction to the planning via model checking paradigm. Section describes possible extensions and future work on how these ideas can be extended. Section draws some conclusions.

Planning

In planning there are two basic notions: state and operator. In the following we suppose that states are described by sets of ground atomic formulas written in an appropriate first order language. Furthermore, with an abuse of terminology, we confuse a state with the set of atomic formulas describing it.[1] We say that a state s satisfies an atomic formula p or, equivalently, that p holds in s, meaning $p \in s$.

Operators can be thought of as partial functions which take (are applied to) a state, called the source state, and return (achieve) a state, called the target state. We also say that (the application of) an operator o achieves a formula p (with p atomic) meaning that o achieves a state s, with $p \in s$. In the following, we suppose that an operator o is a tuple of the form $\langle P, A, D\rangle$, where P, A and D are sets of ground atomic formulas, called respectively (o's) preconditions, addlist and deletelist. P establish what must hold in the source state for o to be applicable, A what must hold in the target state, D what must not hold in the target state.

Usually, in the planning literature, operators are defined allowing for the use of free variables in the preconditions, addlist and deletelist – see for instance (Smith

*Parts of this paper are quoted from (Cimatti *et al.* 1997) and (Giunchiglia 1996)

[1] The notion of state is very subtle; for instance there are definitions of state and description of a state such that the same description describes many possible states. Some of these subtleties are actually exploited, for instance in order to achieve efficiency, still maintaining correctness; see (Lifschitz 1986) for a discussion.

& Peot 1992).[2] We call the operators defined in this way operator schemas. Each operator schema defines the class of operators obtainable by instantiating its free variables in all the possible ways.

A (planning search) problem can be articulated as the problem of achieving a goal state starting from an initial state through a sequence of operator applications[3]. Solving a problem means constructing a "plan", i.e. a sequence of operators of the form $o_1, o_2, \ldots, o_n$ such that, if s_1 is the initial state, for all i, o_i is applicable in s_i and leads to s_{i+1}, and s_{n+1} achieves the goal.

A (planning search) space is the set of all the possible operator applications and states inside which one looks for a plan. We say that a plan is applied to a state, and that it achieves a state or a precondition, generalizing in the obvious way what was written above for operators. (Lifschitz 1986; Knoblock 1991; Bäckström & Jonsson 1995) provide three examples of how (some of) these notions can be formally defined.

Planning via Model Checking

The Planning via Model Checking approach (Cimatti *et al.* 1997) is based on the reformulation of the planning problem to the exploration of a finite state automaton.[4] We use OBDDs (Ordered Binary Decision Diagrams) (Bryant 1992) and symbolic model checking techniques (Burch *et al.* 1992; Clarke, Grumberg, & Long 1994) to compactly encode the automaton and perform the exploration.

The planning domain is described in $\mathcal{AR}$ language (Giunchiglia, Kartha, & Lifschitz 1997), an expressive high level action language which allows for the representation of ramifications, constraints and nondeterministic actions. For the sake of simplicity, in this paper we only consider STRIPS domains, which can be trivially represented in $\mathcal{AR}$.

The first step is to define the finite state automaton $\mathcal{A}$ corresponding to a domain description $\mathcal{D}$. We call *State* the set of states of $\mathcal{A}$. Intuitively, a state can be thought of as (a compact representation of) a truth assignment to the ground atomic formulas of $\mathcal{D}$. Each transition of the automaton represents the application of an operator in a given state. The transitions are described by the relation $Res(s, o, s')$, where o is an operator and s and s' are states. $Res(s, o, s')$ holds iff o is applicable in s, and s' is the result of the application of o in s.

Let o be an operator of the form $\langle P, A, D \rangle$. For a STRIPS description, *Res* can be constructed as follows:[5]

```
function planner(Goal)
Acc[0] := New[0] := Init;
i := 0;
while (New[i] != ∅)
 Reached_goals := (New[i] ∩ Goal);
 if (Reached_Goals != ∅)
  then return choose-plan(i);
 i := i + 1;
 Acc[i] := Acc[i-1] ∪
            { s' : Res(s, A, s'),
                  s ∈ New[i-1], A ∈ Actions };
 New[i] := Acc[i]  \  Acc[i-1];
return failure;
```

```
function choose-plan(N)
V[N] := choose-element(Reached_Goals);
for (j := N - 1,...,0) do
 Back[j]  := { (s,A) : Res(s, A, V[j+1]),
                       s ∈ New[j+1] };
 (V[j],A[j+1]) := choose-element(Back[j]);
return A[1,...,N];
```

Figure 1: The planning algorithm

- a state s has an outgoing arc labeled by o iff o is applicable in s, i.e. s satisfies the preconditions of o.
- for each arc from s to s' labeled by o, (i.e. $Res(s, o, s')$ holds), s' is the state which satisfies each formula in A, falsifies each formula in D, and agrees with s on the other formulae.

Some remarks. Being STRIPS operators deterministic, it is clear that for each s there is *at most* one arc labeled by o. The construction of the automaton also explicitly specifies what *does not change*, i.e. the automaton already embodies a solution to the frame problem.

A planning problem over a given planning domain is defined by a description of a (unique) initial state *Init*, and a set of goal states *Goal*. (In the following, by *Init* and *Goal* we will mean both the description and the actual sets of states of the corresponding automaton.) In Planning via Model Checking, finding a solution to a planning problem amounts to finding a path in the automaton representing the domain which connects the *Init* state with a state in *Goal*.

Intuitively, the planning algorithm for the exploration of the automaton performs a breadth-first search, starting from *Init* and applying operators, generating successor states until either the goal is reached or a fix point is found. The basic idea is that during search, *sets of states* are manipulated and combined, rather than single states.

The planning procedure is reported in Figure 1. We use standard set operators (e.g. $\subseteq, \setminus$). `A[1,...,N]` is an array of actions (i.e. operators). At each step of

[2] Preconditions, addlist and deletelist are sometimes defined as sets of (ground) literals or even (in some work in the theory of actions) as arbitrary first order formulas.

[3] For the goals in this paper, it is sufficient to consider sequential plans. Furthermore we suppose that there is only one initial state.

[4] Our approach is restricted to the finite case. This hypothesis, though quite strong, is acceptable for many practical cases.

[5] The formal details of the construction for the general case can be found in (Cimatti *et al.* 1997).

the loop, the set of states accessible after up to `i` operator applications is generated and stored in `Acc[i]`. This is done by starting from previously unexplored (i.e. new) states generated at previous step, contained in `New[i-1]`. Notice that all possible operators are considered. Indeed, the set `Acc[i]` is computed through *Res*`(s, A, s')`, where `A` ranges over the set of all possible actions `Actions`. This computation, also known as relational product, is performed by special purpose routines. `New[i]` is computed as the set-difference of the accessible states `Acc[i]` with the accessible states at the previous step, i.e. `Acc[i-1]`. If `New[i]` is empty, then the entire space of accessible states has been covered without finding a goal, and `planner` returns `failure`. Otherwise, `planner` checks whether a state in `New[i]` is also a goal state (`New[i]` $\cap$ *Goal*) and stores the result in `Reached_goals`. If this is the case, `planner` calls `choose-plan` with argument `i`, i.e. the length of the plan. Intuitively, `choose-plan` builds a plan by traversing backwards the list of sets of states built by `planner`, and choosing one (of possibly many) action sequence leading to a goal state. The first step is to (arbitrarily) select one accessible goal state, namely `V[N]`. Then, at each `j`-th step, the set of pairs state-action `(s,A)` which can take to `V[j+1]` is generated and stored in `Back[j]`. One pair is arbitrarily chosen, and the action is stored in `A[j+1]`. The action reconstruction is necessary as the "forward" analysis performed by `planner` does not keep track of the actions which have been executed.

MBP (Model Based Planner) (Cimatti *et al.* 1997) is a planner implemented on the model checker NuSMV (Cimatti *et al.* 1998). MBP codifies the automaton using a symbolical representation based on Ordered Binary Decision Diagrams (OBDDs) (Bryant 1992). MBP implements (variations of) the above algorithm and a procedure for temporal projection. Furthermore, MBP has been extended with the ability to generate and execute complex plans containing the usual programming logic constructs, including conditionals, iterations and non-deterministic choice (Cimatti, Roveri, & Traverso 1998). The use of model checking techniques is of crucial importance for the compact representation of conditional plans.

Abstraction in Planning via Model Checking

Planning via Model Checking can be very effective. In many cases MBP performs comparably with state of the art planners, although currently it is based on a blind search. (Optimizations of different kinds are under study.) The key problem is the so called *state explosion problem*, i.e. the amount of resources (memory, time) needed to complete the exploration of the automaton when large domains are codified.

In this section we discuss how abstraction techniques can be introduced in planning via model checking. We focus on Abstrips abstractions, although many of these considerations can be extended to other forms of abstraction. Abstrips planning can be intuitively described as the sequence of the following four steps:

1. Start from a given planning problem, called the "ground" problem. (From now on, we call "ground" the operators, goal, ... defining the ground problem);
2. Generate a new problem, called the "abstract" problem, obtained by deleting (not considering) a certain set of preconditions.[6] (From now on, we call "abstract" the operators, goal, ... defining the abstract problem);
3. Find an abstract plan by solving the abstract problem;
4. Find a ground plan by refining the abstract plan.

Refining an abstract plan means achieving all the preconditions not considered in the abstract problem by reasoning in the ground space. Refinement is needed as, in general, an abstract solution is not a ground solution.

In most cases the abstract problem is further abstracted to obtain a new abstract problem, and so on for some number n of abstract problems (and spaces). When this happens the i-th abstract space is the ground space of the $(i+1)$-th abstract space, with $i = 1, n-1$. Thus, step 2 above is iterated n times, step 3 is performed only in the n-th abstract space, and then refinement (step 4) is iterated n times from the n-th abstract space backward to the ground space. In this case we say that we have a hierarchy of abstraction spaces, with bottom the ground space and top the abstraction space generated last. We also say that a space (also called level) is lower or higher than another with the obvious meaning.

A detailed discussion of how Abstrips abstractions affect the efficiency of search is out of the scope of this paper, and can be found in (Giunchiglia 1996).

Let us consider the application of an Abstrips abstraction in the context of Planning via Model Checking. We assume given the automaton representing the ground planning domain $\mathcal{A}_G$.

Generating the abstract space The first problem to be addressed is to identify which preconditions have to be abstracted. Several heuristics have been investigated (see, e.g., (Knoblock 1994; Bacchus & Yang 1994; Bundy *et al.* 1996)). The application of the selected abstraction results in an automaton $\mathcal{A}_A$ representing the abstract space. The basic advantage is that $\mathcal{A}_A$ has a much smaller number of states than $\mathcal{A}_G$. Different states in $\mathcal{A}_G$ can indeed be abstracted into the same state of $\mathcal{A}_A$.

Solving the problem in the abstract space Solving the abstract problem amounts to searching for a path connecting *Init* with *Goal* in the abstract automaton. This can be done with the algorithm presented in

[6] This is how Abstrips abstractions are used in, *e.g.*, (Sacerdoti 1973), which is slightly different from how they are used in, *e.g.*, (Knoblock 1991).

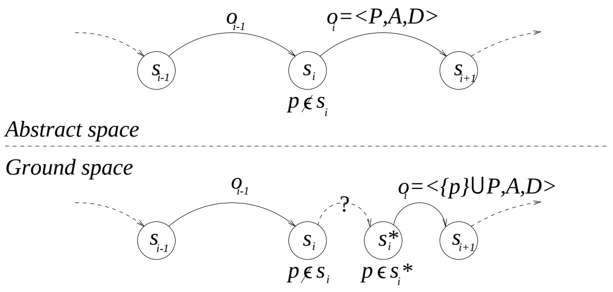

Figure 2: Unabstracting a path.

previous section. Given the reduced number of states and the corresponding augmented connectivity of $\mathcal{A}_A$, a solution found will be shorter (or equal) to solutions in the ground space. The abstract plan is a sequence of operator applications of the following form:

$$s_1, o_1, s_2, o_2, \ldots, s_{n-1}, o_{n-1}, s_n$$

where s_1 is the initial state and s_n is a goal state.

Refining the abstract solution Adding back the preconditions previously deleted means going back to the ground space. The new fact is that now, as extra information, we have the abstract plan. Let us consider the addition of a single precondition to a single abstract operator application (things generalize straightforwardly).

Let $\langle s_i, o_i, s_{i+1}\rangle$ be the operator application to be refined, where $o_i = \langle P, A, D\rangle$ is the abstract operator. Let p be the added precondition, and let $o_i = \langle\{p\}\cup P, A, D\rangle$ be a corresponding ground operator.

There are two possibilities. The first is that (a state corresponding to) s_i satisfies p, i.e. the operator application can be imported "as is" in the ground space. In other words, if $Res_G(s_i, o_i, s_{i+1})$ holds, we are done.

The second is that s_i does not satisfy p. In this situation various strategies are possible. The most common strategy is to keep the abstract plan and try to fix it by building a sub-plan which achieves p (see for instance the descriptions of this strategy in (Sacerdoti 1973; Giunchiglia & Villafiorita 1996)). The idea depicted in Figure 2 is to find an intermediate state s_i^*, with $p \in s_i^*$, as the source of the application of o_i, look for a path, which may not exist, between s_i and s_i^*, and leave everything else unmodified.

This is done with the following two steps:

- Select s_i^*. In general, there may be more than one s_i^*. A possible strategy which can be applied with the planning via model checking approach is to compute the set of all eligible states, as follows

$$S^* \doteq \{s^* | Res(s^*, o_i, s_{i+1})\}$$

This operation can be performed by means of efficient model checking techniques.

- Try to "bridge the gap" between s_i and the set S^* of the corresponding s_i^*. This is done by calling the planner on the new (and hopefully simpler) ground problem, where s_i is the initial state and S^* is the set of goal states.

Notice the difference between this approach and the classical approach to refinement. The latter is usually performed by means of meta-variables standing for terms in the ground space (Giunchiglia & Villafiorita 1996). The process may require the instantiation of such meta-variables, and the solution of the planning problem thus obtained. However, it may be the case that such problem can not be solved because of a wrong instantiation. This may require backtracking, finding a different instance, and trying to solve a new problem. Even if meta-variables are not used, and a disjunction of ground atoms is used to describe the set of possible goal, with a syntactic approach we have to face a proof by cases.

In our approach we exploit the ability to deal with sets of states in two ways. First, by considering one-shot all the possible s^*, we eliminate the possibility of backtracking due to a wrong instantiation. Second, having a larger set of goal states makes it easier to find a solution: the breadth first search algorithm allows to check several alternative goals without substantial computational overhead.

Clearly, this process has to be carried out for each operator application in the abstract plan. However, the overall result is that the search for a (possibly long) plan in the ground space has been reduced to the refinement of a set of operator applications, i.e. a set of (hopefully easier) planning problems in the ground space.

This appears to be particularly promising given the qualitative behavior of Planning via Model Checking. It is extremely efficient to explore exhaustively the state space within a small distance from the starting point. However, longer plans tend to generate a state explosion problem, due to the fact that the set of states reachable with a large number of actions may be extremely high, and therefore its representation can be hard to manipulate.

Discussion and Future Work

Some observations are in order. The basic difficulty with abstraction is the fact that the abstract plan may not be refinable to a correct ground solution, and this implies backtracking and generation of new solutions in the abstract space. In order to detect earlier possible un-refinable plans, different strategies can be envisaged to select what "gap" of the plan is to be refined first. Also, one might put a bound to the length of plans found in the ground space in order to "fill the gaps" from s_i to s_i^*, or put a computational limit on the sizes of OBDDs representing the sets of states.

Furthermore, as discussed in (Giunchiglia 1996), it is important to find plans which can be refined. The planning algorithm guarantees to find a plan of minimal length in the abstract space, which in general may not be an optimal choice. However, it is possible to save the status of the search in the abstract space, and implement an iterative deepening strategy by restarting the computation of longer abstract plans.

Other forms of abstraction are to be investigated. For instance, in (Knoblock 1994) not only the domain is abstracted, but also the goal. This implies that, in general, a predicate/fluents might no longer be relevant to the planning problems. Several techniques have been developed in model checking (e.g. cone of influence reduction) for eliminating parts of the domain which are irrelevant to deciding a goal. These techniques are in general very effective, as they allow to reduce the number of state variables in the automaton, which may be a critical cause of state explosion.

Conclusions

Planning via Model Checking is a novel approach to planning. It is based on the reformulation of a planning problem to the exploration of a finite state automaton. Automata exploration can be efficiently performed with Model Checking, a formal verification technique, currently applied to industrial design.

For very large domains, however, a state explosion can arise. Abstractions can be used to tackle the problem. In this paper we have shown how Abstraction techniques developed in classical planning can be lifted to the Planning via Model Checking paradigm.

In the future, we plan to implement the discussed mechanisms in MBP, and perform a thorough experimental analysis.

References

Bacchus, F., and Yang, Q. 1994. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence* 71:43–100.

Bäckström, C., and Jonsson, P. 1995. Planning with Abstraction Hierarchies can be Exponentially Less Efficient. In *Proc. of the 14th International Joint Conference on Artificial Intelligence*, 1599–1604.

Bryant, R. E. 1992. Symbolic Boolean manipolation with ordered binary-decision diagrams. *ACM Computing Surveys* 24(3):293–318.

Bundy, A.; Giunchiglia, F.; Sebastiani, R.; and Walsh, T. 1996. Calculating Criticalities. *Artificial Intelligence* 88(1-2).

Burch, J. R.; Clarke, E. M.; McMillan, K. L.; Dill, D. L.; and Hwang, L. J. 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation* 98(2):142–170.

Cimatti, A.; Giunchiglia, E.; Giunchiglia, F.; and Traverso, P. 1997. Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. In *Proceeding of the Fourth European Conference on Planning*, Lecture Notes in Artificial Intelligence. Toulouse, France: Springer-Verlag. Also ITC-IRST Technical Report 9705-02, ITC-IRST Trento, Italy.

Cimatti, A.; Clarke, E.; Giunchiglia, F.; and Roveri, M. 1998. NuSMV: a reimplementation of SMV. Technical Report 9801-06, IRST, Trento, Italy.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Strong Planning in Non-Deterministic Domains via Model Checking. In *Proceeding of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98).* Carnegie Mellon University, Pittsburgh, USA: AAAI-Press. Extended version available as ITC-IRST Technical Report 9801-07, ITC-IRST Trento, Italy.

Clarke, E.; Grumberg, O.; and Long, D. 1994. Model Checking. In *Proceedings of the International Summer School on Deductive Program Design.*

Giunchiglia, F., and Villafiorita, A. 1996. ABSFOL: a Proof Checker with Abstraction. In McRobbie, M., and Slaney, J., eds., *Proceedings of the 13th Conference on Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, 136–140. New Brunswick, New Jersey, USA: Springer. Also IRST-Technical Report 9602-20, IRST, Italy and DIST-Technical Report 96-0036, DIST, University of Genova, Italy.

Giunchiglia, F., and Walsh, T. 1992. A Theory of Abstraction. *Artificial Intelligence* 57(2-3):323–390. Also IRST-Technical Report 9001-14, IRST, Trento, Italy.

Giunchiglia, E.; Kartha, G. N.; and Lifschitz, V. 1997. Representing action: Indeterminacy and ramifications. *Artificial Intelligence* 95(2):409–438.

Giunchiglia, F. 1996. Abstrips abstraction – Where do we stand? Technical Report 9607-10, ITC-IRST, Trento, Italy.

Knoblock, C. 1991. *Automatically Generating Abstractions for Problem Solving.* Ph.D. Dissertation, School of Computer Science.

Knoblock, C. A. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* 68:243–302.

Lifschitz, V. 1986. On the semantics of strips. In Georgeff, M., and Lansky, A., eds., *Reasoning about Actions and Plans, Proceedings of the 1986 Workshop*, 1–9. Morgan Kaufmann Publ. Inc.

Sacerdoti, E. 1973. Planning in a Hierarchy of Abstraction Spaces. In *Proceedings of the 3rd International Joint conference on Artificial Intelligence.*

Sacerdoti, E. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115–135.

Smith, D. E., and Peot, M. A. 1992. A Critical Look at Knoblock's Hierarchy Mechanism. In *Proc. 1st International conference Artificial Intelligence planning systems (AIPS-92)*, 307–308.

Yang, Q.; Tenenberg, J. D.; and Woods, S. 1996. On the Implementation and Evaluation of `AbTweak`. *Computational Intelligence* 12.