Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains

Alessandro Cimatti, Marco Roveri, Paolo Traverso
IRST, Povo, 38050 Trento, Italy
{cimatti,roveri,leaf}@irst.itc.it

Abstract

Most real world environments are non-deterministic. Automatic plan formation in non-deterministic domains is, however, still an open problem. In this paper we present a practical algorithm for the automatic generation of solutions to planning problems in non-deterministic domains. Our approach has the following main features. First, the planner generates Universal Plans. Second, it generates plans which are guaranteed to achieve the goal in spite of non-determinism, if such plans exist. Otherwise, the planner generates plans which encode iterative trial-and-error strategies (e.g. try to pick up a block until succeed), which are guaranteed to achieve the goal under the assumption that if there is a non-deterministic possibility for the iteration to terminate, this will not be ignored forever. Third, the implementation of the planner is based on symbolic model checking techniques which have been designed to explore efficiently large state spaces. The implementation exploits the compactness of OBDDs (Ordered Binary Decision Diagrams) to express in a practical way universal plans of extremely large size.*

Introduction

Several planning domains (e.g. robotics, navigation and control) are non-deterministic. The external environment can be highly dynamic, incompletely known and unpredictable, and the execution of an action in the same state may have - non-deterministically - possibly many different effects. For instance, a robot may fail to pick up an object, or while a mobile robot is navigating, doors might be closed/opened, e.g. by external agents. Moreover, the initial state of a planning problem may be partially specified. For this reason, the fundamental assumption underlying classical planning (see e.g. (Fikes & Nilsson 1971; Penberthy & Weld 1992)) to consider only deterministic domains appears to be too restrictive in many practical cases. There are several reactive planners (see e.g. (Georgeff & Lansky 1986)) which relax this assumption, and deal with non-determinism at execution time, i.e. are able to test the different possible outcomes of the executed action and are able to execute iterative plans implementing trial-and-error strategies.

*Copyright 1998, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

However, the ability of generating automatically such plans is still an open problem. The automatic generation of plans in non-deterministic domains presents indeed some peculiar difficulties, related to the fact that the execution of a given plan depends on the non-determinism of the domain and may result, in general, in more than one state. In several applications, a planner should be able, whenever possible, to generate "safe" plans, i.e. plans which are *guaranteed* to achieve the goal despite of non-determinism, for all possible outcomes of execution due to non-determinism. We call such plans *strong solutions*, to stress the difference with *weak solutions*, i.e. plans which might achieve the goal but are not guaranteed to do so. A key problem to plan generation in non-deterministic domains is that an iterative plan (e.g. "pick up block A until succeed") might in principle loop forever, under an infinite sequence of failures. However, this could be an acceptable solution, considering that the probability of success increases with the number of iterations. More than acceptable, in certain domains a trial-and-error strategy might be the *only* acceptable solution, because a certain effect might never be guaranteed.

In this paper we address the problem of the automatic generation of plans in non-deterministic domains. We present a planning algorithm which generates iterative and conditional plans which repeatedly sense the world, select an appropriate action, execute it, and iterate until the goal is reached. These plans are similar to universal plans (Schoppers 1987) in the sense that they map a set of states to an action to be executed. This allows the planner to decide at execution time what to do next in order to achieve the goal, depending on the actual outcomes of non-deterministic actions and the actual state of the environment. To tackle the known problem of the large amount of space required for universal plans, we encode them as OBDDs (Ordered Binary Decision Diagrams) (Bryant 1986). OBDDs allow for a very concise representation of universal plans, since the dimension, i.e. the number of nodes, of an OBDD does not necessarily depend on the actual number of states and corresponding actions. The planning algorithm generates universal plans by incrementally building the OBDDs representing universal plans using symbolic model checking techniques (see, e.g., (Clarke, Grumberg, & Long 1994; Burch *et al.* 1992)).

The planning algorithm returns strong solutions, i.e. plans which are guaranteed to achieve the goal in spite of non-determinism, if such plans exist. Otherwise, the planning algorithm generates iterative trial-and-error strategies which are guaranteed to achieve the goal under the assumption that, if there is a possibility for the iteration to terminate, this will not be ignored forever. We call such solutions, *strong cyclic solutions.* Intuitively, they are "strong" since they still assure that, under the given assumption, there are no executions of the plan which do not achieve the goal. This extends significantly the applicability of the planning algorithm to the domains where trial and error strategies are acceptable or the only possible solutions. If no strong cyclic solution exists, a failure is returned. The planning procedure always terminates. Moreover, the returned solutions are optimal in the following sense: strong solutions have the "worst" execution of minimal length among the possible non-deterministic plan executions.

A remark is in order. In some practical domains, a reasonable plan may not be strong, e.g. lead to failures in very limited cases, i.e. some forms of weak solutions might be acceptable. In this paper we focus on strong and strong cyclic solutions since they are open problems at the current state of the art for plan generation. In general, the "right" kind of solution to be searched for may be in a continuum from weak to strong solutions and highly depend on the integration of planning with execution and control. Nevertheless, planning for strong solutions and strong cyclic solutions allows us to explore new building blocks for planning (and also for integrating planning with execution) which can be combined in different ways.

This paper builds on the framework of "planning via model checking", first presented in (Cimatti *et al.* 1997), which proposes a planning procedure limited to weak solutions as sequences of basic actions. The planning procedure presented in (Cimatti, Roveri, & Traverso 1998) extends the work in (Cimatti *et al.* 1997), but it cannot deal with trial-and-error strategies.

This paper is structured as follows. We first review the language that we use for describing non-deterministic domains. We define formally the universal plans generated by the planner and discuss their representation in terms of OBDDs. Then, we describe the planning procedure which generates strong and strong cyclic plans and discuss its properties. Finally, we describe some experimental results, discuss related works and draw some conclusions.

Non-Deterministic Domains

We describe non-deterministic domains through the $\mathcal{AR}$ language (Giunchiglia, Kartha, & Lifschitz 1997). There are three main reasons for this choice. First, $\mathcal{AR}$ is one of the most powerful formalisms solving the "frame problem" and the "ramification problem". Second, it provides a natural and simple way to describe non-deterministic actions and environmental changes. Third, its semantics is given in terms of finite state automata. This allows us to apply symbolic model checking to generate plans by searching for a solution through the automata. Rather than reviewing the formal syntax and semantics of $\mathcal{AR}$ (the reader is referred to (Giunchiglia, Kartha, & Lifschitz 1997) for a detailed formal account), we describe the language through the simple example in figure 1. The planning problem is to move a pack from the train station to the airport. Sentence (1) states that the pack is at the traffic light after we drive the train from the train station, i.e. *at-light* holds after we execute the action *drive-train* in a state in which *at-station* holds. *at-light* and *at-station* (as well as *at-airport*) are propositional inertial fluents, i.e. fluents which obey to the law of inertia. Propositional fluents are atomic formulas. In sentence (2), *light* is a non-propositional fluent whose value is *green* or *red*, *light* $=$ *green* is an atomic formula and *at-light* $\wedge$ *light* $=$ *green* is a formula. A formula is a propositional combination of atomic formulae. *light* is a non-inertial fluent, i.e. its value may non-deterministically change independently of action executions and thus not obey to the law of inertia. This allows us to describe non-deterministic environmental changes. Sentence (3) states that *drive-train* can be executed only in states where the formula following **prec** holds (A **prec** P is an abbreviation for A **causes** $\bot$ **if** $\neg P$, where $\bot$ stands for falsity). *wait-at-light* can be executed only at the traffic light (sentence (4)). For the law of inertia, it does not affect the position of the pack. Only non-inertial fluents (*light*), may change their value when it is executed. Sentences (5) and (6) are ramification constraints: they state that the pack is in exactly one of the three possible locations. Sentences (7) and (8) define the initial states and the goal of the planning problem, respectively.

$$\textit{drive-train}\ \textbf{causes}\ \textit{at-light}\ \textbf{if}\ \textit{at-station} \quad (1)$$
$$\textit{drive-train}\ \textbf{causes}\ \textit{at-airport}\ \textbf{if}\ \textit{at-light} \wedge \textit{light} = \textit{green} \quad (2)$$
$$\textit{drive-train}\ \textbf{prec}\ \textit{at-station} \vee (\textit{at-light} \wedge \textit{light} = \textit{green}) \quad (3)$$
$$\textit{wait-at-light}\ \textbf{prec}\ \textit{at-light} \quad (4)$$
$$\textbf{always}\ \neg(\textit{at-station} \wedge \textit{at-light} \wedge \textit{at-airport}) \quad (5)$$
$$\textbf{always}\ \textit{at-station} \leftrightarrow (\neg(\textit{at-light} \leftrightarrow \neg\textit{at-airport})) \quad (6)$$
$$\textbf{initially}\ \textit{at-station} \quad (7)$$
$$\textbf{goal}\ \textit{at-airport} \quad (8)$$

Figure 1: An example of $\mathcal{AR}$ description

This example shows how $\mathcal{AR}$ allows for describing non-deterministic actions (*drive-truck*) and non-deterministic environmental changes (non-inertial fluent *light*). It is also possible to express conditional effects, i.e. the fact that an action may have different effects depending on the state in which it is executed (e.g., see action *drive-train*). All of this makes the language

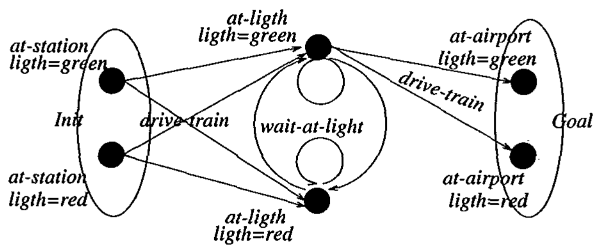


Figure 2: The automaton of the example of figure 1

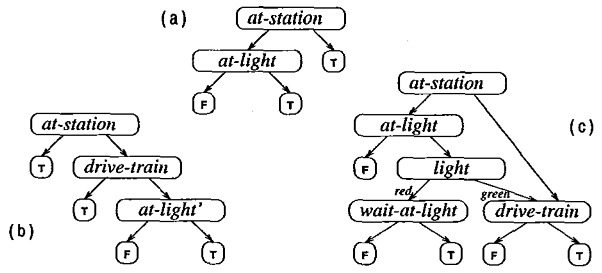


Figure 3: Some examples of OBDDs

far more expressive than STRIPS-like (Fikes & Nilsson 1971) or ADL-like (Penberthy & Weld 1992) languages.

An $\mathcal{AR}$ domain description is given semantics in terms of automata. The states of the automaton are the valuations to fluents, a *valuation* being a function that associates to each fluent one of its values. The transitions from state to state in the automaton represent execution of actions. The resulting automaton takes into account the law of inertia. It is thus possible to give semantics to a planning problem description in terms of a triple, $<Res, Init, Goal>$. *Res* defines the transitions from states to states, i.e. $Res(s, \alpha, s')$ holds for all actions α and states s, s' such that α leads from s to s' in the automaton. *Init* and *Goal* are the sets of initial states and goal states, respectively. Figure 2 shows the automaton corresponding to the example in figure 1.

In the rest of this paper we call $\mathcal{D}$ a given planning problem description, $\mathcal{F}$ the finite set of its fluents, and $\mathcal{A}$ the finite set of its actions, and we assume a given (non-deterministic) planning problem $<Res, Init, Goal>$, where *Init* is non-empty.

We represent automata symbolically by means of OBDDs (Ordered Binary Decision Diagrams) (Bryant 1986). OBDDs are a compact representation of the assignment satisfying (and falsifying) a given propositional formula[1]. Graph (a) in figure 3 is the OBDD for the formula *at-station* $\vee$ *at-light*. Each box is a node. The reading (from top to bottom) is "If *at-station* is true (right arc) then the formula is true, else (left arc) if *at-light* is true the formula is true, else it is false". The above OBDD represents the set of states of the automaton where either *at-station* or *at-light* hold. These are the four leftmost states in figure 2. Notice that this includes all the possible combinations of values of *light*.

The transitions of the automaton are also represented by OBDDs. The idea is to have variables to store the action labeling the transition, and the value of fluents *after* the execution of the action. OBDD (b) in figure 3 represents the four transitions corresponding to the execution of action *drive-train* in *at-station*. The informal reading is that if *at-station* holds, and the action *drive-train* is executed, then the primed variable *at-light'*, representing the value of the fluent *at-light* after the execution, must have value true. The variables *light* and *light'* do not occur in the OBDD, as all the combinations of assignments are possible.

[1]For non-propositional variables, we use a boolean encoding similarly to (Ernst, Millstein, & Weld 1997).

A basic advantage of OBDDs is that their size (i.e. the number of nodes) does not necessarily depend on the actual number of assignments (each representing, e.g., a state or a transition). Furthermore, OBDDs provide for an efficient implementation of the operations needed for manipulating sets of states and transitions, e.g. union, projection and intersection.

Universal Plans

Our goal is to find strong solutions for non-deterministic planning problems, i.e. establish course of actions which are guaranteed to achieve a given goal regardless of non-determinism. However, searching for strong solutions in the space of classical plans, i.e. sequences of basic actions, is bound to failure. Even for a simple problem as the example in figure 1, there is no sequence of actions which is a strong solution to the goal. Non-determinism must be tackled by planning conditional behaviors, which depend on the (sensing) information which can be gathered at execution time. For instance, we would expect to decide what to do when we get at the traffic light according to the status of the traffic light. Furthermore, it must be possible to represent trial-and-error strategies, which are important when an action can not guarantee to achieve an effect. To this extent we use universal plans (Schoppers 1987), which associate to each possibly reachable state an action, and implement a reactive loop, where the status of the world is sensed, the corresponding action is chosen, and the process is iterated until actions are no longer available. Intuitively, a universal plan can be seen as a set of pairs state-action that we call state-action table (*SA*).

In order to make this notion precise, we extend the (classical) notion of plan (i.e. sequence of actions) to include non-deterministic choice, conditional branching, and iterative plans.

Definition 0.1 ((Extended) Plan) *Let Φ be the set of formulae constructed from fluents in $\mathcal{F}$. The set of (Extended) Plans $\mathcal{P}$ for $\mathcal{D}$ is the least set such that*

if $\alpha \in \mathcal{A}$, then $\alpha \in \mathcal{P}$;
if $\alpha, \beta \in \mathcal{P}$, then $\alpha; \beta \in \mathcal{P}$;
if $\alpha, \beta \in \mathcal{P}$, then $\alpha \cup \beta \in \mathcal{P}$;

if $\alpha, \beta \in \mathcal{P}$ *and* $p \in \Phi$, *then* **if** p **then** α **else** $\beta \in \mathcal{P}$;
if $\alpha \in \mathcal{P}$ *and* $p \in \Phi$, *then* **if** p **then** $\alpha \in \mathcal{P}$;
if $\alpha \in \mathcal{P}$ *and* $p \in \Phi$, *then* **while** p **do** $\alpha \in \mathcal{P}$.

Extended plans are a subset of program terms in Dynamic Logic (DL) (Harel 1984) modeling constructs for SDL and non-deterministic choice. (See (Cimatti, Roveri, & Traverso 1998) for a semantic account of extended plans.) In the following we call $Exec[\alpha](S)$ the set of states resulting from the execution of action α in any state in the set S.

$$Exec[\alpha](S) = \{s' | s \in S, Res(s, \alpha, s')\}$$

In this framework, the universal plan corresponding to a given state-action table *SA* can be thought of as an extended plan of the form

$$UP_{SA} \doteq \textbf{while } Domain_{SA} \textbf{ do } IfThen_{SA}$$

where $Domain_{SA} \doteq \bigvee_{\exists\alpha.<\alpha,s>\in SA} s$ and

$IfThen_{SA} \doteq \bigcup_{\langle s,\alpha\rangle \in SA}$ **if** s **then** α.

Intuitively, $Domain_{SA}$ is a formula expressing the states in the state-action table *SA*, and $IfThen_{SA}$ is the plan which, given a state s, executes the corresponding action α such that $\langle s, \alpha\rangle \in SA$. The universal plan for the example in figure 2 is the following.

```
while (at-light ∨ at-station) do
 if (at-light ∧ light = green) then drive-train   ∪
 if  (at-light ∧ light = red)  then wait-at-light ∪
 if        (at-station)        then drive-train   ∪
```

Universal plans can be efficiently executed in practice. However, reasoning about universal plans may require the manipulation of data structures of extremely large size. In order to tackle this problem, we represent universal plans symbolically using OBDDs. OBDD (c) in figure 3 represents the universal plan of the example. A universal plan is encoded as (an OBDD representing) a propositional formula in fluent variables (representing the state of affairs) and action variables (representing the actions to be executed). The models of this formula represent the associations of actions to states. This approach allows for a very concise representation of state-action maps. Notice for instance that, in OBDD (c) in figure 3, the right lowest subgraph has multiple pointers, thus resulting in sharing.

The Planning Algorithm

In this section we describe the planning procedure StrongCyclicPlan, which looks for a universal plan solving the non-deterministic planning problem given in input. The underlying intuition is that *sets* of states (instead of single states) are manipulated during search. The implementation of the algorithm is based on manipulation of OBDDs, which allow for compact representation and efficient manipulation of sets of states

```
1. procedure StrongCyclicPlan(Init, Goal)
2.   OldAcc := ∅; Acc := Goal; SA := ∅;
3.   while (Acc ≠ OldAcc)
4.     if Init ⊆ Acc
5.       then return SA;
6.     PreImage := StrongPreImage(Acc);
7.     PrunedPreImage := PruneStates(PreImage, Acc);
8.     if PrunedPreImage ≠ ∅
9.       then SA := SA ∪ PrunedPreImage;
10.        OldAcc := Acc;
11.        Acc := Acc ∪ ProjAct(PrunedPreImage);
12.      else StrongCycles;
13. return Fail;
```

Figure 4: The Strong Cyclic Planning Algorithm

and state-action tables. In the following we will present the routines by using standard set operators (e.g. $\subseteq, \setminus$), hiding the fact that the actual implementation is performed in terms of OBDD manipulation routines.

StrongCyclicPlan, shown in figure 4, is given the set of states satisfying the goal *Goal*, and the set of initial states *Init*. It returns a state-action table *SA* representing a universal plan solving the problem, if a solution exists, otherwise it returns *Fail*. In order to construct *SA*, the algorithm loops backwards, from the goal towards the initial states. At each step, an extension to *SA* is computed, until the set of states *Acc*, for which a solution has been found, includes the set of initial states *Init* (step 4). If no extension to *SA* was possible, i.e. *Acc* is the same as at previous cycle (*OldAcc*), then there is no solution to the problem, the loop is exited (step 3), and *Fail* is returned (step 13).

At each step, two attempts are made to extend *SA*. The first, based on the StrongPreImage function, finds the state-action pairs which necessarily lead to previously visited states, regardless of non determinism (steps 6-11). If the first attempt fails, the StrongCycles procedure looks for an extension to *SA* implementing a cyclic trial-and-error strategy.

The basic step StrongPreImage is defined as

$$\text{StrongPreImage}(S) = \{<s,\alpha> : s \in State, \alpha \in \mathcal{A}, \emptyset \neq Exec[\alpha](s) \subseteq S\} \quad (9)$$

where *State* is the set of the states of the automaton. StrongPreImage(S) returns the set of pairs state-action $<s, \alpha>$ such that action α is applicable in state s ($\emptyset \neq Exec[\alpha](s)$) and the (possibly non-deterministic) execution of α in s necessarily leads inside S ($Exec[\alpha](s) \subseteq S$). Intuitively, StrongPreImage(S) contains all the states from which S can be reached with a one-step plan regardless of non-determinism. PruneStates(*PreImage*, *Acc*) (step 7) eliminates from *PreImage* the state-action pairs whose states are already in *Acc*, and thus have already been visited.

If *PreImage* contains new states (step 8), we have a one-step plan leading necessarily to *Acc*, *PrunedPreImage* is added to *SA* (step 9) and *Acc* is

```
1. procedure StrongCycles
2.   $WpiAcc := \emptyset; OldWpiAcc := \top; CSA := \emptyset;$
3.   while ($CSA = \emptyset$ and $OldWpiAcc \neq WpiAcc$)
4.     $IWpi :=$ WPreImage(ProjAct($WpiAcc$) $\cup$ $Acc$);
5.     $CSA :=$ $Outgoing :=$ PruneStates($IWpi$, $Acc$);
6.     while ($CSA \neq \emptyset$ and $Outgoing \neq \emptyset$)
7.       $Outgoing :=$ OSA($CSA$, ProjAct($CSA$) $\cup$ $Acc$);
8.       $CSA := CSA \setminus Outgoing$;
9.     if ($CSA \neq \emptyset$)
10.      then $SA := SA \cup CSA$; $OldAcc := Acc$;
11.           $Acc := Acc \cup$ ProjAct($CSA$);
12.      else $OldWpiAcc := WpiAcc$;
13.           $WpiAcc := WpiAcc \cup IWpi$;
```

Figure 5: The Strong Cycles Algorithm

updated with the new states in *PreImage* (step 11). ProjAct, given a set of state-action pairs, returns the corresponding set of states.

If *PreImage* does not contain new states, then the StrongCycles procedure is called (step 12). StrongCycles (figure 5) operates on the variables *SA*, *Acc* and *OldAcc* of StrongCyclicPlan. The basic step of StrongCycles is the WPreImage function (which stands for "weak-pre-image"), defined as

$$\text{WPreImage}(S) = \{<s,\alpha> : s \in State,\ \alpha \in \mathcal{A},\ Exec[\alpha](s) \cap S \neq \emptyset\} \quad (10)$$

WPreImage(S) returns the set of pairs state-action $<s,\alpha>$ such that there exists at least one execution of α in s which leads inside S. At each cycle, StrongCycles applies the WPreImage routine to the set of states visited so far. At the j-th iteration, *IWpi* contains potential cycles which can lead to *Acc* in j steps. The inner loop (steps 6-8) detects cycles necessarily leading to *Acc*, and stores them in *CSA* (Connected *SA*). The basic step of the iteration is the OSA (Outgoing *SA*) routine which, given a set of state-action pairs *CSA* and a set of states S, computes the state-action pairs which may lead out of S:

$$\text{OSA}(CSA, S) = \{<s,\alpha> \in CSA \ .\ Exec[\alpha](s) \not\subseteq S\} \quad (11)$$

The state-action pairs which may lead out of *Acc* and *CSA* are stored in *Outgoing* (step 7) and eliminated from *CSA* (step 8). The process loops until all the *Outgoing* state-action pairs have been eliminated, or *CSA* is empty. In the former case, the state-action pairs stored in *CSA* are added to *SA* and StrongCycles returns control to StrongCyclicPlan. In the latter case, StrongCycles performs the $j+1$ iteration, i.e. looks for cycles which can lead to *Acc* in $j+1$ steps. This is done by accumulating in *WpiAcc* the states visited so far. This process is iterated till either a cycle is found (the inner loop exits with a non empty *CSA*) or a fix point is reached, i.e. there are no new state-action pairs to be visited, i.e. *WpiAcc* is the same as at previous cycle (*OldWpiAcc*).

StrongCyclicPlan is guaranteed to terminate. The construction of the sets *Acc* and *WpiAcc* is monotonic and has a least fix point (see (Clarke, Grumberg, & Long 1994) for a similar proof). Moreover, if StrongCyclicPlan returns a state-action table *SA*, then the corresponding universal plan UP_{SA} is a strong cyclic solution. If StrongCyclicPlan returns a solution without invoking StrongCycles then the corresponding universal plan UP_{SA} is a (non-cyclic) strong solution. If StrongCyclicPlan returns *Fail* then there exists no strong cyclic solution. Finally, StrongCyclicPlan avoids cycles whenever possible.

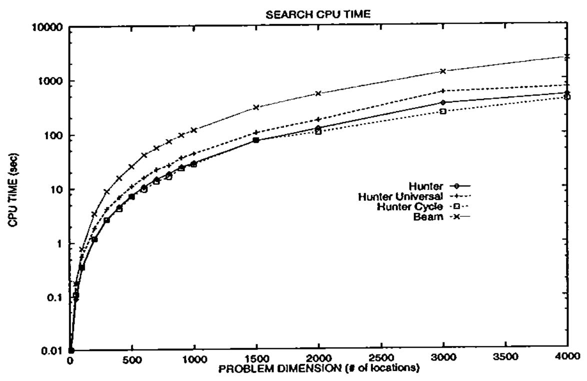
Experimental Results

The algorithm described in previous section is the core of MBP, a planner built on top of NUSMV, an OBDD-based symbolic model checker (Cimatti *et al.* 1998). MBP allows for generating the automaton starting from the action description language, finding strong (cyclic) solutions to planning problems, and directly executing universal plans encoded as OBDDs. At each execution step, the planner encodes in terms of OBDDs the information acquired from the environment, retrieves the action corresponding to the current state by accessing the state-action table of the universal plan, and activates its execution.

We performed an experimental analysis of the automatic generation of universal plans. Though very preliminary, the experiments define some basic test cases for non-deterministic planning, show that MBP is able to deal in practice with cases of significant complexity, and settle the basis for future comparisons.

One of the very few examples of a non-deterministic domain that we have found in the literature is the "hunter-prey" moving target search problem, presented in (Koenig & Simmons 1995) as an explanatory case study, where a hunter must reach the same position of a moving prey. We have parameterized the problem in the number of possible locations of the hunter and the prey (in (Koenig & Simmons 1995) 6 locations are used), and we have developed an automatic problem generator. We have run the planner on problems of different size, from 6 to 4000 locations. The results are depicted in figure 6 (all tests were performed on a 200MHz Pentium PRO with 64Mb RAM.) The search time needed for the 4000 locations case is less than 8 minutes. The size of the OBDD representing the state-action table is about 10000 nodes.

We have also experimented with the "universal" version of the problem, where the initial situation is completely unspecified. This version of the problem is much harder, as it requires to traverse the whole state space. However, we obtain search times of the same order of magnitude. Even more interestingly, the number of nodes of the OBDD representing the state action table is about 8000 nodes, *lower* than in the original problem. This contrasts with the fact that the state-action table covers a higher number of states (16 millions states against 4 millions in the original case). This is a clear



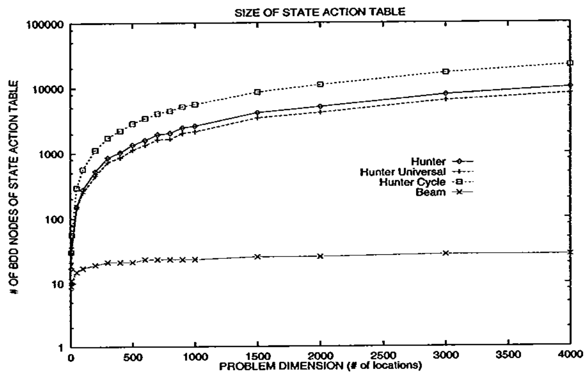
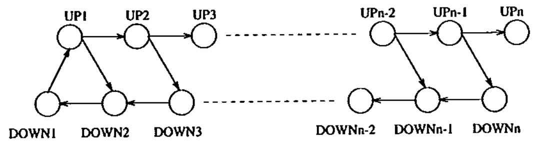


Figure 6: Search times (sec.) and size of the SA (number of nodes) for the different tests

example of the advantage of OBDDs: the size of an OBDD in terms of nodes is not, in general, related to the information "extensionally" encoded in it.

The above problems allow for strong solutions without cycles. We have then modified the problem by requiring that the hunter must "grab" the prey once they are in the same location. The prey may nondeterministically escape from a grab attempt, and this results in cycles. Again, we have experimented with different configurations of this domain, from 6 to 4000. As shown in figure 6, the search times are again of the same order of magnitude, while the size of the state action tables grows to about 22000 nodes.

Finally, in order to stress the strong cycle algorithm, we have experimented with a further example where the distance of the cycles in the solution is proportional to the dimension of the configuration. The problem consists of an agent whose purpose is to walk to the end of a beam. At each position on the beam, the walk action moves non-deterministically from position UPi-1 to UPi or to position DOWNi. If the agent falls down, it has to go back to the starting position (DOWN1) and retry from the beginning.



The results show that the regularity of the domain contains the computation load, and greatly reduces the size of the OBDDs of the resulting state-action table.

Conclusion and related work

In this paper we have presented an approach to automatic plan generation in non-deterministic domains. In order to tackle non-determinism, we construct universal plans which encode conditional iterative course of actions. We present a planning algorithm which automatically generates universal plans which are guaranteed to achieve a goal regardless of the non-determinism and are able to encode iterative trial-and-error strategies. The planning procedure always terminates, even in the case no solution exists. In case the solution does not involve cycles, it is optimal, in the sense that the worst-case length of the plan (in terms of number of actions) is minimal with respect to other possible solution plans. In principle, the algorithm is general, and its implementation could make use of different techniques. We have chosen OBDDs, which allow for an efficient encoding and construction of universal plans. We have tested the planning algorithm on some examples and the performance are promising.

Our work relies heavily on the work on symbolic model checking: we search the state space by manipulating sets of states and re-use the basic building blocks implemented in state of the art model checkers, e.g. those for computing pre-images and fix-point and for manipulating OBDDs. Nevertheless, (symbolic) model checking has been designed for verification, i.e. checking whether an automaton satisfies a given property. Its application to a synthesis problem like plan generation is a major conceptual shift which has required novel technical solutions. This has required the design of a novel algorithm which constructs and accumulates state-action pairs which are guaranteed to lead to a set of states for any action outcome.

The work described in this paper is limited in the following respects. First, compared to the previous research in planning, we restrict to a finite number of states. This hypothesis, though quite strong, allows us to treat many significant planning problems. Second, the algorithm presented in this paper can be significantly optimized. For instance, when looking for cycles which can lead to a set of states in n steps, it does not make use of the previous computations searching for loops which can lead to the set of states in $n-1$ steps. Moreover, it does not fully exploit several existing model checking optimization techniques, e.g. variable reordering, cone of influence reduction, frontier simplification, partitioning (Clarke, Grumberg, & Long 1994; Burch *et al.* 1992). Third, the current search strategy is committed to one particular notion of optimality, i.e.

it looks first for strong solutions. This might not be the most natural strategy for all the application domains. Finally, we do not have the complete assurance that the approach can scale up to very complex problems. Even if the preliminary results are promising, and even if symbolic model checking has been shown to do well in complex industrial verification tasks, we still need to experiment with extensive tests and with real-world large-scale applications.

Only few previous works address the automatic generation of (universal) plans in domains which are not deterministic. In (Kabanza, Barbeau, & St-Denis 1997), a planning algorithm searches the non-deterministic automaton representing the domain in order to solve complex goals specified as temporal logic formulae. The approach is based on explicit state search, and heuristics are required to contain the state explosion problem. Koenig and Simmons (Koenig & Simmons 1995) describe an extension of real-time search (called min-max LRTA*) to deal with non-deterministic domains. As any real-time search algorithm, min-max LRTA* does not plan ahead (it selects actions at-run time depending on a heuristic function) and, in some domains, it is not guaranteed to find a solution.

The approach to planning based on Markov Decision Processes (MDP) (see, e.g. (Dean *et al.* 1995; Cassandra, Kaelbling, & Littman 1994)) shares some similarities with our approach: it deals with actions with possible multiple outcomes, it generates a policy which encodes a universal plan, and it uses a representation based on automata. There are several differences, both technical and in focus. The MDP approach deals with stochastic domains, while we deal only with non-deterministic domains. We do not deal with probabilistic information and rewards, which can provide a greater expressivity for describing planning problems. On the other hand, the fact that we are limited to non-deterministic domains allows us to generate automatically the automata starting from a high level description in $\mathcal{AR}$. $\mathcal{AR}$ provides a natural way of describing ramifications and non-deterministic environmental changes (through the use of non-inertial fluents) and releases the user from the task of describing the effects of the law of inertia. Problem descriptions in $\mathcal{AR}$ are very concise and simple, compared with the corresponding automata (see for instance the simple example in figures 1 and 2). (Dean *et al.* 1995) describes an algorithm which can be used for anytime planning, by finding the solution for a restricted state space and then increasing iteratively the state space which is considered. Our algorithm searches for a solution backward from the goal to the initial states. It would be interesting to apply the same kind of strategy used in (Dean *et al.* 1995) within our approach, for instance by searching first for a weak solution, and then extending it step-by-step with state-action pairs returned by the STRONGPREIMAGE.

Acknowledgments

We thank Marco Daniele for finding a bug in an early version of the planning algorithm.

References

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* C-35(8):677–691.

Burch, J.; Clarke, E.; McMillan, K.; Dill, D.; and Hwang, L. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2):142–170.

Cassandra, A.; Kaelbling, L.; and Littman, M. 1994. Acting optimally in partially observable stochastic domains. In *Proc. of AAAI-94*. AAAI-Press.

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

Cimatti, A.; Giunchiglia, E.; Giunchiglia, F.; and Traverso, P. 1997. Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. In *Proc. of ECP-97.*

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Strong Planning in Non-Deterministic Domains via Model Checking. In *Proc. of AIPS-98.* AAAI-Press.

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

Dean, T.; Kaelbling, L.; Kirman, J.; and Nicholson, A. 1995. Planning Under Time Constraints in Stochastic Domains. *Artificial Intelligence* 76(1-2):35–74.

Ernst, M.; Millstein, T.; and Weld, D. 1997. Automatic SAT-compilation of planning problems. In *Proc. IJCAI-97.*

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2(3-4):189–208.

Georgeff, M., and Lansky, A. L. 1986. Procedural knowledge. *Proc. of IEEE* 74(10):1383–1398.

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

Harel, D. 1984. Dynamic Logic. In Gabbay, D., and Guenthner, F., eds., *Handbook of Philosophical Logic,* volume II. D. Reidel Publishing Company. 497–604.

Kabanza, F.; Barbeau, M.; and St-Denis, R. 1997. Planning control rules for reactive agents. *Artificial Intelligence* 95(1):67–113.

Koenig, S., and Simmons, R. 1995. Real-Time Search in Non-Deterministic Domains. In *Proc. of IJCAI-95,* 1660–1667.

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. of KR-92.*

Schoppers, M. J. 1987. Universal plans for Reactive Robots in Unpredictable Environments. In *Proc. of IJCAI-87,* 1039–1046.