Strong Cyclic Planning Revisited

Marco Daniele[1,2], Paolo Traverso[1], and Moshe Y. Vardi[3]*

[1] IRST, Istituto per la Ricerca Scientifica e Tecnologica, 38050 Povo, Trento, Italy
[2] Dipartimento di Informatica e Sistemistica, Università "La Sapienza", 00198 Roma
[3] Department of Computer Science, Rice University, Houston TX 77251, USA
daniele@irst.itc.it, leaf@irst.itc.it, vardi@cs.rice.edu

Abstract. Several realistic non-deterministic planning domains require plans that encode iterative trial-and-error strategies, e.g., "pick up a block until succeed". In such domains, a certain effect (e.g., action success) might never be guaranteed *a priori* of execution and, in principle, iterative plans might loop forever. Here, the planner should generate iterative plans whose executions always have a possibility of terminating and, when they do, they are guaranteed to achieve the goal. In this paper, we define the notion of *strong cyclic plan*, which formalizes in temporal logic the above informal requirements for iterative plans, define a planning algorithm based on model-checking techniques, and prove that the algorithm is guaranteed to return strong cyclic plans when they exist or to terminate with failure when they do not. We show how this approach can be extended to formalize plans that are guaranteed to achieve the goal and do not involve iterations (*strong plans*) and plans that have a possibility (but are not guaranteed) to achieve the goal (*weak plans*). The results presented in this paper constitute a formal account for "planning via model checking" in non-deterministic domains, which has never been provided before.

1 Introduction

Classical planning [16, 21] makes some fundamental assumptions: the planner has complete information about the initial state of the world, effects of the execution of actions are deterministic, and the solution to the planning problem can be expressed as a sequence of actions. These assumptions are unrealistic in several practical domains (e.g., robotics, scheduling, and control). The initial state of a planning problem may be partially specified and the execution of an action in the same state may have many possible effects. Moreover, plans as sequences of actions are bound to failure: non-determinism must be tackled by planning conditional behaviors, which depend on the information that can be gathered at execution time. For instance, in a realistic robotic application, the action "pick-up a block" cannot be simply described as a STRIPS-like operator [16] whose effect is that "the block is at hand" of the robot. "Pick-up a block" might result either in a success or failure, and the result cannot be known *a priori* of execution. A

* Supported in part by NSF grants CCR-9628400 and CCR-9700061.

useful plan, depending on the action outcome, should execute different actions, e.g., try to pick-up the block again if the action execution has failed.

Most often, a conditional plan is not enough: plans encoding iterative trial-and-error strategies, like "pick up a block until succeed", are the only acceptable solutions. In several realistic domains, a certain effect (e.g., action success) might never be guaranteed *a priori* of execution and, in principle, iterative plans might loop forever, under an infinite sequence of failures. The planner, however, should generate iterative plans whose executions always have a possibility of terminating and, when they do, they are guaranteed to achieve the goal.

The starting point of the work presented in this paper is the framework of *planning via model checking*, together with the related system MBP, first presented in [7] and then extended to deal with non-deterministic domains in [10, 9] (see also [18] for an introduction to Planning as Model Checking). [7] proposes the idea to use model checking techniques to do planning and proposes an algorithm for generating *weak plans*, i.e., plans that may achieve the goal but are not guaranteed to do so. [10] proposes an algorithm to generate *strong plans*, i.e., plans that are guaranteed to achieve a desired goal in spite of non-determinism. [9] extends [10] to generate *strong cyclic plans*, whose aim is to encode iterative trial-and-error strategies. However, no formal notion of strong cyclic plan is given in [9] and, as far as we know, in any other work.

In this paper we provide a framework for planning via model checking where weak, strong, and strong cyclic plans can be specified uniformly in temporal logic. In the paper, we focus on strong cyclic plans, since their formal specifications and the provision of a correct algorithm is still an open problem at the current state-of-the-art. Indeed, this paper builds on [9] making the following contributions.

- We provide a formal definition of strong cyclic plan based on the well-known Computation Tree Logic (CTL) [14]. The idea is that a strong cyclic plan is a solution such that "*for each* possible execution, *always* during the execution, there *exists* the possibility of *eventually* achieving the goal". The formalization is obtained by exploiting the universal and existential path quantifiers of CTL, as well as the "always" and "eventually" temporal connectives.
- We define a new algorithm for strong cyclic planning. It is guaranteed to generate plans that cannot get stuck in loops with no possibility to terminate. The algorithm in [9] did not satisfy this requirement. Moreover, the new algorithm improves the quality of the solutions by eliminating nonrelevant actions.
- We prove that the algorithm presented in this paper is correct and complete, i.e., it generates strong cyclic plans according to the formal definition while, if no strong cyclic solutions exist, it terminates with failure.

The results presented in this paper provide a formal account for planning via model checking that has never been given before. Indeed, after providing a clear framework for strong cyclic plans, we show how it can be easily extended to express weak and strong plans. Weak plans are such that *there exists* at least one execution that *eventually* achieves the goal, strong plans are such that *all* executions *eventually* achieve the goal.

The paper is structured as follows. We define the notion of planning problem in Section 2 and the notion of strong cyclic solutions in Section 3. The description of the planning algorithm is given in Section 4. Finally, in Section 5 we show how the framework can be extended to formalize weak and strong plans. We conclude the paper with a comparison with some related work.

2 The Planning Problem

A *(non-deterministic) planning domain* can be described in terms of *fluents*, which may assume different values in different *states*, *actions* and a *transition function* describing how (the execution of) an action leads from one state to possibly many different states.

Definition 1 (Planning Domain). *A planning domain D is a 4-tuple $\langle F, S, A, R\rangle$ where F is the finite set of fluents, $S \subseteq 2^F$ is the set of states, A is the finite set of actions, and $R : S \times A \mapsto 2^S$ is the transition function.*

Fluents belonging (not belonging) to some state s are assigned to TRUE (FALSE) in s. Our definitions deal with Boolean fluents while examples are easier to describe through fluents ranging over generic finite domains[1]. $R(s, a)$ returns all the states the execution of a from s can lead to. The action a is said to be *executable* in the state s if $R(s, a) \neq \emptyset$.

A *(non-deterministic) planning problem* is a planning domain, a set of initial states and a set of goal states.

Definition 2 (Planning Problem). *A planning problem P is a 3-tuple $\langle D, I, G\rangle$ where D is the planning domain, $I \subseteq S$ is the set of initial states and $G \subseteq S$ is the set of goal states.*

Both I and G can be represented through two Boolean functions $\mathcal{I}$ and $\mathcal{G}$ over F, which define the sets of states in which they hold. From now on, we switch between the two representations, as sets or functions, as the context requires.

Non-determinism occurs twice in the above definitions. First, we have a set of initial states, and not a single initial state. Second, the execution of an action from a state is a set of states, and not a single state.

As an explanatory example, let us consider the situation depicted in Figure 1 (left). The situation is a very small excerpt from an application we are developing for the Italian Space Agency [4]. A tray (`T`) provides two positions in which two containers ($\mathtt{C}_1$ and $\mathtt{C}_2$) for solutions may be hosted. In addition, a kettle (`K`) may host one container for boiling its solution. The kettle is provided with a switch (`S`) that can operate only if the container is well positioned on the kettle. This situation can be formalized as shown in Figure 1 (right). The set F of (non-Boolean) fluents is $\{\mathtt{C}_1, \mathtt{C}_2, \mathtt{S}\}$. $\mathtt{C}_1$ and $\mathtt{C}_2$ represent the positions of the containers, and can be `on-T` (on tray), `on-K-ok` (on kettle, steady), or `on-K-ko` (on kettle, not steady). `S` represents the status of the kettle's switch (`on` or `off`). The set of states

[1] For non-Boolean variables, we use a Boolean encoding similarly to [15].

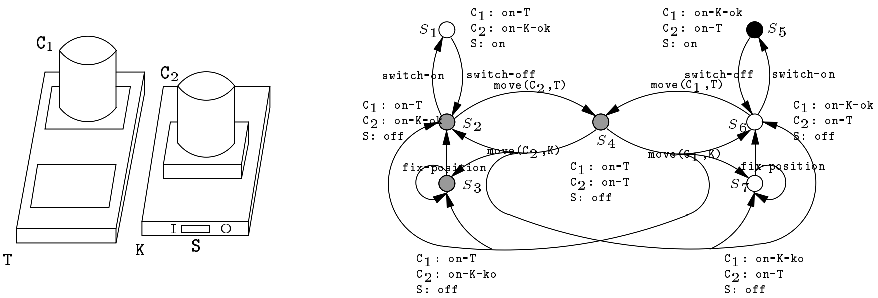

Fig. 1. An example (left) and its formalization (right).

is represented by the nodes of the graph, which define fluents' values. The set of actions is represented by the edges' labels. Actions `move(C1,T)`, `move(C2,T)`, `switch-on`, and `switch-off`, are deterministic; `move(C1,K)`, `move(C2,K)`, and `fix-position`, are not. Indeed, when moving containers from the tray to the kettle, it can happen the containers are not correctly positioned. Moreover, it can be possible the wrong container is picked up and moved upon the kettle. Thus, $R(S_4, \mathtt{move(C_1,K)}) = R(S_4, \mathtt{move(C_2,K)}) = \{S_2, S_3, S_6, S_7\}$. Still, when trying to settle a container, it is possible getting no effect. Thus, $R(S_3, \mathtt{fix\text{-}position}) = \{S_2, S_3\}$ and $R(S_7, \mathtt{fix\text{-}position}) = \{S_6, S_7\}$. The planning problem is to boil the solution contained in $\mathtt{C_1}$ starting from a situation where $\mathtt{C_1}$ is on the tray and the kettle's switch is off, that is, $\mathcal{I}$ is $\mathtt{C_1} = \mathtt{on\text{-}T} \wedge \mathtt{S} = \mathtt{off}$ (grey nodes, in Figure 1), and $\mathcal{G}$ is $\mathtt{C_1} = \mathtt{on\text{-}K\text{-}ok} \wedge \mathtt{S} = \mathtt{on}$ (black node, in Figure 1).

A remark is in order. Non-deterministic planning problems can be expressed in different specification languages. For instance, in [7, 10, 9] the $\mathcal{AR}$ language [17] is used. Alternatively, we might use any language that allows us to express non-determinism, i.e., the fact that an action has multiple outcomes or, in other words, disjunctive postconditions. STRIPS-like [16] or ADL-like [21] languages (e.g., PDDL) are not expressive enough.

3 Strong Cyclic Plans

When dealing with non-determinism, plans have to be able to represent conditional and iterative behaviors. We define plans as *state-action tables* (resembling universal plans [22]) that associate actions to states. The execution of a state-action table can result in conditional and iterative behaviors. Intuitively, a state-action table execution can be explained in terms of a reactive loop that senses the state of the world and chooses one among the corresponding actions, if any, for the execution until the goal is reached.

Definition 3 (State-Action Table). *A state-action table SA for a planning problem P is a set of pairs $\{\langle s, a\rangle : s \in S \setminus G, a \in A,$ and a is executable in $s\}$.*

The states of a state-action table may be any state, except for those in the set of goal states. Intuitively, this corresponds to the fact that when the plan achieves the goal no further action needs to be executed. Hereafter, we write

State	Action
S_1	`switch-off`
S_3	`fix-position`
S_2	`move(C₂,T)`
S_6	`move(C₁,T)`
S_4	`move(C₁,K)`
S_4	`move(C₂,K)`
S_7	`fix-position`
S_6	`switch-on`

Fig. 2. A state-action table.

STATES(SA) for denoting the set of states in the state-action table SA, i.e., STATES(SA)$= \{s : \exists a \in A.\langle s, a\rangle \in SA\}$.

Definition 4 (Total State-Action Table). *A state-action table SA for a planning problem P is* total *if, for all $\langle s, a\rangle \in SA$, $R(s, a) \subseteq$ STATES$(SA) \cup G$.*

Intuitively, in a total state-action table, each state that can be reached by executing an action either is a goal state or has a corresponding action in the state-action table. The notion of total state-action table is important in order to capture strong (cyclic) plans, i.e., plans that must be specified for all possible outcomes of actions. In Figure 2, a total state-action table related to our example is shown.

Given a notion of plan as a state-action table, the goal is to formalize strong cyclic plans in terms of temporal logic specifications on the possible executions of state-action tables. A preliminary step is to formalize the notion of execution of a state-action table.

Definition 5 (Execution). *Let SA be a state-action table for the planning problem P. An* execution *of SA starting from the state $s_0 \in$ STATES$(SA) \cup G$ is an infinite sequence $s_0 s_1 \ldots$ of states in S such that, for all $i \geq 0$, either $s_i \in G$ and $s_i = s_{i+1}$, or $s_i \notin G$ and, for some $a \in A$, $\langle s_i, a\rangle \in SA$ and $s_{i+1} \in R(s_i, a)$.*

Executions are infinite sequences of states. Depending on non-determinism, we may have many possible executions corresponding to a state-action table. Each nongoal state s_i has as successor a state s_{i+1} reachable from s_i by executing an action corresponding to s_i in the state-action table; when the sequence reaches a goal state, the execution is extended with an infinite sequence of the same goal state. Of course, nontotal state-action tables may induce also executions *dangling* at nongoal states, i.e., executions reaching a nongoal state for which no action is provided.

The total state-action tables we are interested in, i.e., strong cyclic plans, are such that, informally, all their executions either lead to the goal or loop over a set of states from which the goal could be eventually reached. With respect to the state-action table of Figure 2, an example of the former case is executing `switch-on` when at S_6, which surely leads to the goal; while an example of the latter case is executing `fix-position` in S_7 that, even if looping at S_7, may lead to S_6 and, therefore, to the goal.

In order to capture the notion of strong cyclic plan, we need a formal framework that allows us to state temporal properties of executions. We have chosen the branching time logic CTL [14], which provides universal and existential path quantifiers and temporal operators like "eventually" and "always". CTL formulas are defined starting from a finite set $\mathcal{P}$ of propositions, the Boolean connectives, the temporal connectives $\mathtt{X}$ ("next-time") and $\mathtt{U}$ ("until"), and the path quantifiers $\mathtt{E}$ ("exists") and $\mathtt{A}$ ("for all"). Given a finite set $\mathcal{P}$ of propositions, CTL formulas are inductively defined as follows:

- Each element of $\mathcal{P}$ is a formula;
- $\neg\psi$, $\psi \vee \phi$, $\mathtt{EX}\psi$, $\mathtt{AX}\psi$, $\mathtt{E}(\phi\mathtt{U}\psi)$, and $\mathtt{A}(\phi\mathtt{U}\psi)$ are formulas if ϕ and ψ are.

CTL semantics is given with respect to *Kripke structures*. A Kripke structure K is a triple $\langle W, T, L\rangle$ where W is a set of *worlds*, $T \subseteq W \times W$ is a total *transition relation*, and $L : W \mapsto 2^{\mathcal{P}}$ is a *labeling function*. A *path* π in K is a sequence $w_0 w_1 \ldots$ of worlds in W such that, for $i \geq 0$, $T(w_i, w_{i+1})$. In what follows, $K, w \models \psi$ denotes that ψ holds in the world w of K. CTL semantics is then inductively defined as follows:

- $K, w_0 \models p$ iff $p \in L(w_0)$, for $p \in \mathcal{P}$
- $K, w_0 \models \neg\psi$ iff $K, w_0 \not\models \psi$
- $K, w_0 \models \psi \vee \phi$ iff $K, w_0 \models \psi$ or $K, w_0 \models \phi$
- $K, w_0 \models \mathtt{EX}\psi$ iff there exists a path $w_0 w_1 \ldots$ such that $K, w_1 \models \psi$
- $K, w_0 \models \mathtt{AX}\psi$ iff for all paths $w_0 w_1 \ldots$ we have $K, w_1 \models \psi$
- $K, w_0 \models \mathtt{E}(\phi\mathtt{U}\psi)$ iff there exist a path $w_0 w_1 \ldots$ and $i \geq 0$ such that $K, w_i \models \psi$ and, for all $0 \leq j < i$, $K, w_j \models \phi$
- $K, w_0 \models \mathtt{A}(\phi\mathtt{U}\psi)$ iff for all paths $w_0 w_1 \ldots$ there exists $i \geq 0$ such that $K, w_i \models \psi$ and, for all $0 \leq j < i$, $K, w_j \models \phi$

We introduce the usual abbreviations $\mathtt{AF}\psi \equiv \mathtt{A}(\textsc{true}\mathtt{U}\psi)$ ($\mathtt{F}$ stands for "future" or "eventually"), $\mathtt{EF}\psi \equiv \mathtt{E}(\textsc{true}\mathtt{U}\psi)$, $\mathtt{AG}\psi \equiv \neg\mathtt{EF}\neg\psi$ ($\mathtt{G}$ stands for "globally" or "always"), and $\mathtt{EG}\psi \equiv \neg\mathtt{AF}\neg\psi$.

The executions of a total state-action table SA for the planning problem P can be encoded as paths of the Kripke structure K^P_{SA} *induced* by SA.

Definition 6 (Induced Kripke Structure). *Let SA be a total state-action table for the planning problem P. The Kripke structure K^P_{SA} induced by SA is defined as:*

- $W^P_{SA} = \textsc{States}(SA) \cup G$;
- $T^P_{SA}(s, s')$ *iff* $\langle s, a\rangle \in SA$ *and* $s' \in R(s, a)$, *or* $s = s'$ *and* $s \in G$;
- $L^P_{SA}(s) = s$.

The totality of T^P_{SA} is guaranteed by the totality of SA. Strong cyclic plans can be specified through a temporal logic formula on their executions.

Definition 7 (Strong Cyclic Plan). *A strong cyclic plan for a planning problem P is a total state-action table SA for P such that $\mathcal{I} \subseteq W^P_{SA}$ and, for all $s \in \mathcal{I}$, we have $K^P_{SA}, s \models \mathtt{AGEF}\mathcal{G}$.*

```
function StrongCyclicPlan(P)
   I := I \ G; SCP:= {⟨s, a⟩ : s ∈ S \ G, a ∈ A,  a is executable in s}; OldSCP:=⊥
   while (OldSCP≠SCP) do
      OldSCP:=SCP
      SCP:=PruneUnconnected(P, PruneOutgoing(P, SCP))
   endwhile
   if (I ⊆ States(SCP))
      then return SCP
      else  return Fail

function PruneOutgoing(P, SA)
   Outgoing := ComputeOutgoing(P, SA)
   while (Outgoing ≠ ∅) do
      SA:=SA\Outgoing
      Outgoing := ComputeOutgoing(P, SA)
   endwhile
   return SA

function PruneUnconnected(P, SA)
   ConnectedToG := ∅; OldConnectedToG := ⊥
   while ConnectedToG ≠ OldConnectedToG do
      OldConnectedToG:=ConnectedToG
      ConnectedToG:=SA ∩ OneStepBack(P, ConnectedToG)
   endwhile
   return ConnectedToG
```

Fig. 3. The algorithm.

That is, starting from the initial states, whatever actions we choose to execute and whatever their outcomes are, we always (AG) have a way of reaching the goal (EF$\mathcal{G}$). Notice that the state-action table in Figure 2 is a strong cyclic plan for the planning problem at hand.

4 The Strong Cyclic Planning Algorithm

The idea underlying our algorithm is that *sets* of states (instead of single states) are manipulated during the search. The implementation of the algorithm is based on OBDDs (Ordered Binary Decision Diagrams) [3], which allow for compact representation and efficient manipulation of sets. This opens up the possibility to deal with domains involving large state spaces, as shown by the experimental results in [9]. Our presentation is given in terms of the standard set operators (e.g., $\subseteq$, $\setminus$), hiding the fact that the actual implementation is performed through OBDD manipulation routines. In principle, however, the algorithm could be implemented through different techniques, provided that they make such set operations available. The algorithm is presented in two steps: first, algorithms computing basic strong cyclic plans are introduced (Figure 3 and 5), and then an algorithm for improving such basic solutions is given (Figure 7).

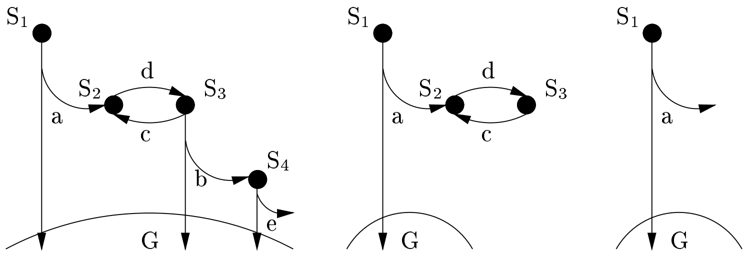

Fig. 4. Pruning the state-action table.

Given a planning problem P, StrongCyclicPlan(P) (Figure 3) generates strong cyclic plans. The algorithm starts with the largest state-action table in SCP (line 2), and repeatedly removes pairs that either spoil SCP totality or are related to states from which the goal cannot be reached (line 5). If the resulting SCP contains all the initial states (line 7), the algorithm returns it (line 8), otherwise *Fail* is returned (line 9).

Pairs spoiling SCP totality are pruned by function PruneOutgoing (lines 14–20), which iteratively removes state-action pairs that can lead to nongoal states for which no action is considered. Its core is the function ComputeOutgoing that, for a planning problem P and a state-action table SA, is defined as $\{\langle s,a\rangle \in SA : \exists s' \notin \text{States}(SA) \cup G. s' \in R(s,a)\}$. With respect to the example shown in Figure 4 (left), during the first iteration, PruneOutgoing removes $\langle S_4, e\rangle$ and, during the second one, it removes $\langle S_3, b\rangle$, giving rise to the situation shown in Figure 4 (middle).

Having removed the dangling executions results in disconnecting S_2 and S_3 from the goal, and give rise to a cycle in which executions may get stuck with no hope to terminate. This point, however, was not clear in the work presented in [9]. States from which the goal cannot be reached have to be pruned away. This task is accomplished by the function PruneUnconnected (lines 21–27) that, when given with a planning problem P and a state-action table SA, loops backwards inside the state-action table from the goal (line 25) to return the state-action pairs related to states from which the goal is reachable. Looping backward is realized through the function OneStepBack that, when given with a planning problem P and a state-action table SA, returns all the state-action pairs possibly leading to states of SA or G. Formally, $\text{OneStepBack}(P, SA) = \{\langle s,a\rangle : s \in S \setminus G, a \in A, \exists s' \in \text{States}(SA) \cup G. s' \in R(s,a)\}$. With respect to the example shown in Figure 4 (middle), PruneUnconnected removes both $\langle S_2, d\rangle$ and $\langle S_3, c\rangle$, producing the situation shown in Figure 4 (right). Having removed the above pairs re-introduces dangling executions and, therefore, requires to apply the pruning phase once more, leading to the empty set. In general, the pruning phase has to be repeated until the putative strong plan SCP is not changed either by PruneOutgoing or by PruneUnconnected (line 3).

As an alternative (see Figure 5), rather than starting with the largest state-action table, one could start with an empty state-action table in $AccSA$ (line 2)

```
1.  function StrongCyclicPlan(P)
2.    I := I \ G; SCP:= ∅; AccSA:= ∅; OldAccSA:= ⊥
3.    while (I ⊈ States(SCP) and AccSA ≠ OldAccSA) do
4.      OldAccSA:=AccSA; AccSA:=OneStepBack(P, AccSA)
5.      SCP:=AccSA; OldSCP:=⊥
6.      while (OldSCP≠SCP) do
7.        OldSCP:=SCP
8.        SCP:=PruneUnconnected(P, PruneOutgoing(P, SCP))
9.      endwhile
10.   endwhile
11.   if (I ⊆ States(SCP))
12.     then return SCP
13.     else  return Fail
```

Fig. 5. The incremental algorithm.

and incrementally extend it (line 4) until either a strong cyclic plan containing all the initial states is found, or $AccSA$ is not extendible anymore (line 3).

The strong cyclic plans returned by StrongCyclicPlan can be improved in two directions. Consider the example in Figure 6, where S_3 is the initial state. The strong cyclic plan returned by StrongCyclicPlan for such example comprises all the possible state-action pairs of the planning problem. Note, however, that the pair $\langle S_1, a\rangle$ is absolutely useless, since it is unreachable from the initial state. Furthermore, the pair $\langle S_4, d\rangle$ is useless as well, because it moves the execution away from the goal. Indeed, when reaching S_4 from S_3, one does not want to go back to S_3 through d. The algorithm for getting rid of the above is shown in Figure 7.

Function PruneUnreachable loops forward, inside the state-action table returned by the basic algorithm, collecting state-action pairs related to states that can be reached from the initial ones. Its core is the function OneStepForth (line 7) that, when given with a planning problem P and a state-action table *ReachableFromI*, returns the set of pairs related to states reachable by executing actions in *ReachableFromI*. Formally, $\text{OneStepForth}(P, \mathit{ReachableFromI}) = \{\langle s,a\rangle : s \in S, a \in A, a \text{ is executable in } s \text{ and } \exists\langle s',a'\rangle \in \mathit{ReachableFromI}, s \in R(s',a')\}$. *ReachableFromI* is initialized with the pairs related to initial states by GetInit (line 4), defined as $\text{GetInit}(P, SCP) = \{\langle s,a\rangle \in SCP : s \in I\}$. With respect to Figure 6, this first optimization phase chops out the pair $\langle S_1, a\rangle$

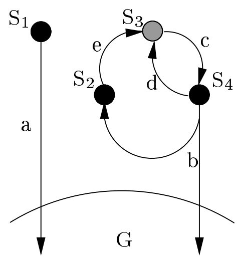

Fig. 6. Problems of the basic algorithm.

1. **function** OPTIMIZE(P, *SCP*)
2. **return** SHORTESTEXECUTIONS(P, PRUNEUNREACHABLE(P, *SCP*))

3. **function** PRUNEUNREACHABLE(P, *SCP*)
4. *ReachableFromI* := GETINIT(P, SCP); *OldReachableFromI*:= $\perp$
5. **while** (*ReachableFromI*$\neq$ *OldReachableFromI*) **do**
6. *OldReachableFromI*:=*ReachableFromI*
7. *ReachableFromI* := *ReachableFromI* $\cup$ SCP $\cap$ ONESTEPFORTH(P, *ReachableFromI*)
8. **endwhile**
9. **return** *ReachableFromI*

10. **function** SHORTESTEXECUTIONS(P, *SCP*)
11. *Shortest* := $\emptyset$; *OldShortest* := $\perp$
12. **while** (*Shortest*$\neq$ *OldShortest*)
13. *OldShortest*:=*Shortest*
14. *LastAdded* := SCP $\cap$ ONESTEPBACK(P, *Shortest*)
15. *Shortest*:=*Shortest* $\cup$ PRUNEVISITED(*LastAdded*,*Shortest*)
16. **endwhile**
17. **return** *Shortest*

Fig. 7. Optimization.

while, with respect to the state-action table of Figure 2, $\langle S_1, \texttt{switch-off}\rangle$ is removed.

Function SHORTESTEXECUTIONS chops out all the pairs $\langle s, a\rangle$ that do not start one of the shortest executions leading from s to the goal. Indeed, executions passing through s can still reach the goal through one of the shortest ones. Shortest executions are gathered in *Shortest* as a set of state-action pairs by looping backward (line 14) inside the (optimized through PRUNEUNREACHABLE) state-action table returned by the basic algorithm, and by introducing new pairs only when related to states that have not been visited yet (line 15). This latter task is performed by PRUNEVISITED, defined as PRUNEDVISITED(*LastAdded*, *Shortest*) $= \{\langle s, a\rangle \in$ *LastAdded* $: s \notin$ STATES(*Shortest*)$\}$. With respect to Figure 6, this second optimization phase chops out the pair $\langle S_4, d\rangle$ while, with respect to the state-action table of Figure 2, $\langle S_6, \texttt{move(C}_1\texttt{,T)}\rangle$ is removed.

The algorithms for generating and optimizing strong cyclic plans are guaranteed to terminate, are correct and complete (the proofs can be found in [11]):

Theorem 1. *Let P be a planning problem. Then*

1. OPTIMIZE(P, STRONGCYCLICPLAN(P)) *terminates.*
2. OPTIMIZE(P, STRONGCYCLICPLAN(P)) *returns a strong cyclic plan for P if and only if one exists.*

5 Extensions: weak and strong solutions

In this paper we focus on finding strong cyclic solutions, which has been an open problem at the current state-of-the-art for plan generation. However, strong cyclic plans are of course not the only interesting solutions. In some practical domains, it may be possible for the planner to generate strong plans, i.e., plans which are not iterative and guarantee goal achievement. In other applications, a plan may be allowed to lead to failures in very limited cases, i.e., some forms of weak solutions might be acceptable. A planner may be required to generate solutions of different "strength" according to the application domain.

Strong and weak plans have been introduced in [10]. We show here how they can be specified as temporal formulas on plan executions. This requires to generalize Definitions 5 and 6 for taking into account state-action tables that are not total. Given the state-action table SA for the planning problem P, we first define $\textsc{Closure}(SA) = \{s \notin \textsc{States}(SA) : \langle s', a' \rangle \in SA, s \in R(s', a')\} \cup G$.

Definition 8 (Execution). *Let SA be a state-action table for the planning problem P. An* execution *of SA starting from the state $s_0 \in \textsc{States}(SA) \cup \textsc{Closure}(SA)$ is an infinite sequence $s_0 s_1 \ldots$ of states in S such that, for all $i \geq 0$, either $s_i \in \textsc{Closure}(SA)$ and $s_i = s_{i+1}$, or $s_i \notin \textsc{Closure}(SA)$ and, for some $a \in A$, $\langle s_i, a \rangle \in SA$ and $s_{i+1} \in R(s_i, a)$.*

Definition 9 (Induced Kripke Structure). *Let SA be a state-action table for the planning problem P. The Kripke structure K_{SA}^P induced by SA is defined as*

- $W_{SA}^P = \textsc{States}(SA) \cup \textsc{Closure}(SA)$;
- $T_{SA}^P(s, s')$ *iff* $\langle s, a \rangle \in SA$ *and* $s' \in R(s, a)$, *or* $s = s'$ *and* $s \in \textsc{Closure}(SA)$;
- $L_{SA}^P(s) = s$.

In the case of total state-action tables, since $\textsc{Closure}(SA) = G$, these latter definitions collapse into the previous ones.

Definition 10 (Weak Plan). *A* weak plan *for a planning problem P is a state-action table SA for P such that $\mathcal{I} \subseteq W_{SA}^P$ and, for all $s \in \mathcal{I}$, we have $K_{SA}^P, s \models \mathtt{EF}\mathcal{G}$.*

Definition 11 (Strong Plan). *A* strong plan *for a planning problem P is a total state-action table SA for P such that $\mathcal{I} \subseteq W_{SA}^P$ and, for all $s \in \mathcal{I}$, we have $K_{SA}^P, s \models \mathtt{AF}\mathcal{G}$.*

6 Conclusions and Related Work

In this paper we have presented a formal account for strong cyclic planning in non-deterministic domains. We have formalized the notion of strong cyclic plans, i.e., plans encoding iterative trial-and-error strategies that always have a possibility of terminating and, when they do, are guaranteed to achieve the goal in spite of non-determinism. Strong cyclic plans are plans whose executions satisfy the CTL formula $\mathtt{AGEF}\mathcal{G}$, where $\mathcal{G}$ is a propositional formula representing the set of goal states. We have shown how this approach can also embed "strong" and "weak" plans, whose executions have to satisfy the CTL formulas $\mathtt{AF}\mathcal{G}$ and

$\mathtt{EF}\mathcal{G}$, respectively. We have defined an algorithm that is guaranteed to generate strong cyclic plans and to terminate, and have implemented it in MBP, a planner built on top of the symbolic model checker NuSMV [6]. MBP is currently used in an application for the "Italian Space Agency" (ASI) [4].

A future goal is to extend the planning task from the task of finding a plan which leads to a set of states (the goal) to the task of synthesizing a plan which satisfies some specifications in some temporal logic. This makes the planning task very close to controller synthesis (see, e.g., [1, 20]), which considers both exogenous events and non-deterministic actions. From the controller synthesis perspective, in this paper we synthesize memoryless plans. Due to its generality, however, the work in [1, 20] does not allow for concise solutions as state-action tables, and it is to be investigated how it can express and deal with strong cyclic plans. [19] proposes an approach to planning that has some similarities to the work on synthesis but abandons completeness for computational efficiency.

Most of the work in planning is focused on deterministic domains. Some works extend classical planners to "contingent" planners (see, e.g., [27]), which generate plans with conditionals, or to "conformant" planners [23, 8], which try to find strong solutions as sequences of actions. Nevertheless, neither existing contingent nor existing conformant planners are able to generate iterative plans as strong cyclic solutions. Some deductive planning frameworks (see, e.g., [24, 25]) can be used to specify desired plans in non-deterministic domains. Nevertheless, the automatic generation of plans in these deductive frameworks is still an open problem. Some works propose an approach that is similar to planning via model checking. The TLplan system [2] (see also [12] for an automata-theoretic approach) allows for control strategies expressed as Linear Time Temporal Logic (LTL) [14] and implements a forward chaining algorithm that has strong similarities with LTL standard model checking [26]. However, the planner deals only with deterministic domains. Moreover, it is not clear how it could be extended to express strong cyclic solutions (where both a universal and existential path quantifiers are required) and to generate them. [5] proposes a framework based on process algebra and mu-calculus for reasoning about nondeterministic and concurrent actions. The framework is rather expressive, but it does not deal with the problem of plan generation. In planning based on Markov Decision Processes (MDP) (see, e.g., [13]), policies (much like state-action tables) are constructed from stochastic automata, where actions induce transitions with an associated probability, and states have an associated reward. The planning task is reduced to constructing optimal policies w.r.t. rewards and probability distributions. There is no explicit notion of weak, strong, and strong cyclic solution.

References

1. E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid System II*, volume 999 of *LNCS*. Springer Verlag, 1995.
2. F. Bacchus and F. Kabanza. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence*, 1998. Submitted for pubblication.

3. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
4. A. Cesta, P. Riccucci, M. Daniele, P. Traverso, E. Giunchiglia, M. Piaggio, and M. Shaerf. Jerry: a system for the automatic generation and execution of plans for robotic devices - the case study of the Spider arm. In *Proc. of ISAIRAS-99*, 1999.
5. X.J. Chen and G. de Giacomo. Reasoning about nondeterministic and concurrent actions: A process algebra approach. *Artificial Intelligence*, 107(1):29–62, 1999.
6. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a reimplementation of SMV. Technical Report 9801-06, IRST, Trento, Italy, January 1998.
7. A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. In *ECP97*, pages 130–142, 1997.
8. A. Cimatti and M. Roveri. Conformant Planning via Model Checking. In *Proc. of ECP99*, 1999.
9. A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Proc. of AAAI98*, 1998.
10. A. Cimatti, M. Roveri, and P. Traverso. Strong Planning in Non-Deterministic Domains via Model Checking. In *Proc. of AIPS98*, 1998.
11. M. Daniele, P. Traverso, and M. Y. Vardi. Strong Cyclic Planning Revisited. Technical Report 9908-03, IRST, Trento, Italy, August 1999.
12. G. de Giacomo and M.Y. Vardi. Automata-theoretic approach to planning with temporally extended goals. In *Proc. of ECP99*, 1999.
13. T. Dean, L. Kaelbling, J. Kirman, and A. Nicholson. Planning Under Time Constraints in Stochastic Domains. *Artificial Intelligence*, 76(1-2):35–74, 1995.
14. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 16, pages 995–1072. Elsevier, 1990.
15. M. Ernst, T. Millstein, and D. Weld. Automatic SAT-compilation of planning problems. In *Proc. of IJCAI-97*, 1997.
16. R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.
17. E. Giunchiglia, G. N. Kartha, and V. Lifschitz. Representing action: Indeterminacy and ramifications. *Artificial Intelligence*, 95(2):409–438, 1997.
18. F. Giunchiglia and P. Traverso. Planning as Model Checking. In *Proc. of ECP99*, 1999.
19. R. Goldman, D. Musliner, K. Krebsbach, and M. Boddy. Dynamic Abstraction Planning. In *Proc. of AAAI97*, 1998.
20. O. Kupferman and M.Y. Vardi. Synthesis with incomplete information. In *Proc. of 2nd International Conference on Temporal Logic*, pages 91–106, 1997.
21. J. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. of KR-92*, 1992.
22. M. J. Schoppers. Universal plans for Reactive Robots in Unpredictable Environments. In *Proc. of IJCAI87*, pages 1039–1046, 1987.
23. D. Smith and D. Weld. Conformant Graphplan. In *AAAI98*, pages 889–896.
24. S. Steel. Action under Uncertainty. *J. of Logic and Computation, Special Issue on Action and Processes*, 4(5):777–795, 1994.
25. W. Stephan and S. Biundo. A New Logical Framework for Deductive Planning. In *Proc. of IJCAI93*, pages 32–38, 1993.
26. M Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 November 1994.
27. D. Weld, C. Anderson, and D. Smith. Extending Graphplan to Handle Uncertainty and Sensing Actions. In *Proc. of AAAI98*, pages 897–904, 1998.