Planning as Model Checking for Extended Goals in Non-deterministic Domains

Marco Pistore and **Paolo Traverso**
ITC-IRST, Via Sommarive 18, 38050 Povo, Trento, Italy
{pistore,traverso}@irst.itc.it

Abstract

Recent research has addressed the problem of planning in non-deterministic domains. Classical planning has also been extended to the case of goals that can express temporal properties. However, the combination of these two aspects is not trivial. In non-deterministic domains, goals should take into account the fact that a plan may result in many possible different executions and that some requirements can be enforced on all the possible executions, while others may be enforced only on some executions. In this paper we address this problem. We define a planning algorithm that generates automatically plans for extended goals in non-deterministic domains. We also provide preliminary experimental results based on an implementation of the planning algorithm that uses symbolic model checking techniques.

1 Introduction

Most real world planning domains are intrinsically "non-deterministic". This is the case, for instance, of several robotics, control, and space application domains. Most often, applications in non-deterministic domains require planners to deal with goals that are more general than sets of final desired states. The planner needs to generate plans that satisfy conditions on their whole execution paths, i.e., on the sequences of states resulting from execution. E.g., in a robotic application, we may need to specify that a mobile robot should "move to a given room while avoiding certain areas all along the path".

When dealing with non-deterministic domains, the task of extending the notion of goal leads to a main key issue, related to the fact that the execution of a given plan may non-deterministically result in more than one sequence of states. Consider the previous example in the robotics context. On the one hand, we would like to require a plan that guarantees to reach the room and also guarantees that dangerous areas are avoided. On the other hand, in several realistic domains, no plan might satisfy this strong requirement. We might therefore accept plans that satisfy weaker requirements, e.g., we might accept that the robot has a possibility of reaching the room without being guaranteed to do so, but it is however guaranteed to avoid dangerous areas. Alternatively, we may require a plan that guarantees that the robot reaches the desired location, just trying, if possible, to avoid certain areas, e.g., areas that are too crowded.

In this paper, we define and implement a planning algorithm that generates automatically plans for extended goals in non-deterministic domains. Extended goals are CTL formulas [Emerson, 1990]. They can express temporal conditions that take into account the fact that an action may non-deterministically result in different outcomes. Hence extended goals allow us to distinguish between temporal requirements on "all the possible executions" and on "some executions" of a plan.

The plans built by the algorithm are strictly more expressive than plans that simply map states to actions to be executed, like universal plans [Schoppers, 1987], memory-less policies [Bonet and Geffner, 2000], and state-action tables [Cimatti *et al.*, 1998; Daniele *et al.*, 1999]. Beyond expressing conditional and iterative behaviors, the generated plans can execute different actions is a state, depending on the previous execution history. This expressiveness is required to deal with extended goals.

We have implemented the planning algorithm inside MBP [Cimatti *et al.*, 1998]. MBP uses symbolic techniques based on BDDs [Burch *et al.*, 1992] that provide the ability to represent compactly and explore efficiently large state spaces. In the paper we present preliminary experimental results that show that the proposed algorithm works in practice.

This paper is structured as follows. We first define non-deterministic planning domains and extended goals. We then define the structure of plans that can achieve extended goals. Finally, we describe the planning algorithm, describe its implementation, and show some experimental results.

2 Planning Domains

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

Definition 1 *A* planning domain *D is a tuple $(\mathcal{B}, Q, A, \rightarrow)$, where $\mathcal{B}$ is the finite set of (basic) propositions, $Q \subseteq 2^{\mathcal{B}}$ is the set of states, A is the finite set of actions, and $\rightarrow \subseteq Q \times A \times Q$ is the transition relation. We write $q \xrightarrow{a} q'$ for $(q, a, q') \in \rightarrow$.*

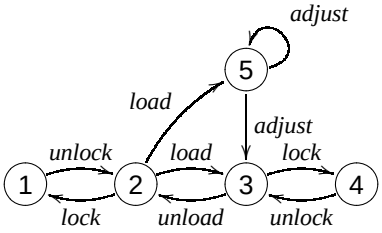

Figure 1: A simple non-deterministic domain

We require that the transition relation $\rightarrow$ is total, i.e., for every $q \in Q$ there is some $a \in A$ and $q' \in Q$ such that $q \xrightarrow{a} q'$. We denote with $\mathrm{Act}(q)$ the set of the actions that can be performed in state q: $\mathrm{Act}(q) = \{a : \exists q' . q \xrightarrow{a} q'\}$. We denote with $\mathrm{Exec}(q, a)$ the set of the states that can be reached from q performing action $a \in \mathrm{Act}(q)$: $\mathrm{Exec}(q, a) = \{q' : q \xrightarrow{a} q'\}$.

In Figure 1 we depict a simple planning domain, where an item can be loaded/unloaded to/from a container which can be locked/unlocked. Actions *load* and *adjust* are non-deterministic. *load* can either succeed, and lead to state 3 where the item is loaded correctly and the container can be locked, or it may fail, and lead to state 5, where the item needs to be adjusted to a correct position in order to lock the container. Action *adjust* may in its turn fail, and leave the item positioned incorrectly. For all states in the domain, we assume to have an action *wait* (not represented in the figure) that leaves the state unchanged. The basic propositions are loaded, locked and misplaced. loaded holds in states 3 and 4, locked in states 1 and 4, misplaced in state 5.

All the work presented in this paper is independent of the language for describing planning domains. However many of these languages (e.g., ADL-like languages as PDDL) are not able to represent non-deterministic domains and should be extended allowing for disjunctive effects of the actions. For instance, the action *load* might be described with an extension of PDDL as follows.

```
:action load
:precondition
  (and (not loaded) (not locked) (not misplaced))
:effect (OR (loaded) (misplaced))
```

3 Extended Goals

Extended goals are expressed with CTL formulas.

Definition 2 *Let $\mathcal{B}$ be the set of basic propositions of a domain D and let $b \in \mathcal{B}$. The syntax of an* (extended) goal *g for D is the following:*

$$g ::= \top \mid \bot \mid b \mid \neg b \mid g \wedge g \mid g \vee g \mid \mathrm{AX}\, g \mid \mathrm{EX}\, g \mid \mathrm{A}(g\, U\, g) \mid \mathrm{E}(g\, U\, g) \mid \mathrm{A}(g\, W\, g) \mid \mathrm{E}(g\, W\, g)$$

"X", "U", and "W" are the "next time", "(strong) until", and "weak until" temporal operators, respectively. "A" and "E" are the universal and existential path quantifiers, where a path is an infinite sequence of states. They allow us to specify requirements that take into account non-determinism. Intuitively, the formula $\mathrm{AX}\, g$ ($\mathrm{EX}\, g$) means that g holds in every (in some) immediate successor of the current state. $\mathrm{A}(g_1\, \mathrm{U}\, g_2)$ ($\mathrm{E}(g_1\, \mathrm{U}\, g_2)$) means that for every path (for some path) there exists an initial prefix of the path such that g_2 holds at the last state of the prefix and g_1 holds at all the other states along the prefix. The formula $\mathrm{A}(g_1\, \mathrm{W}\, g_2)$ ($\mathrm{E}(g_1\, \mathrm{W}\, g_2)$) is similar to $\mathrm{A}(g_1\, \mathrm{U}\, g_2)$ ($\mathrm{E}(g_1\, \mathrm{U}\, g_2)$) but allows for paths where g_1 holds in all the states and g_2 never holds. Formulas $\mathrm{AF}\, g$ and $\mathrm{EF}\, g$ (where the temporal operator "F" stands for "future" or "eventually") are abbreviations of $\mathrm{A}(\top\, \mathrm{U}\, g)$ and $\mathrm{E}(\top\, \mathrm{U}\, g)$, respectively. $\mathrm{AG}\, g$ and $\mathrm{EG}\, g$ (where "G" stands for "globally" or "always") are abbreviations of $\mathrm{A}(g\, \mathrm{W}\, \bot)$ and $\mathrm{E}(g\, \mathrm{W}\, \bot)$, respectively. A remark is in order: even if $\neg$ is allowed only in front of basic propositions, it is easy to define $\neg g$ for a generic CTL formula g, by "pushing down" the negations: for instance $\neg \mathrm{AX}\, g \equiv \mathrm{EX}\, \neg g$ and $\neg \mathrm{A}(g_1\, \mathrm{W}\, g_2) \equiv \mathrm{E}(\neg g_2\, \mathrm{U}(\neg g_1 \wedge \neg g_2))$.

Goals as CTL formulas allow us to specify different interesting requirements on plans. Let us consider first some examples of *reachability goals*. $\mathrm{AF}\, g$ ("reach g") states that a condition should be guaranteed to be reached by the plan, in spite of non-determinism. $\mathrm{EF}\, g$ ("try to reach g") states that a condition might possibly be reached, i.e., there exists at least one execution that achieves the goal. As an example, in Figure 1, the strong requirement $(\mathsf{locked} \wedge \neg \mathsf{loaded}) \rightarrow \mathrm{AF}\, (\mathsf{locked} \wedge \mathsf{loaded})$ cannot be satisfied, while the weaker requirement $(\mathsf{locked} \wedge \neg \mathsf{loaded}) \rightarrow \mathrm{EF}\, (\mathsf{locked} \wedge \mathsf{loaded})$ can be satisfied by unlocking the container, loading the item and then (if possible) locking the container. A reasonable reachability requirement that is stronger than $\mathrm{EF}\, g$ is $\mathrm{A}(\mathrm{EF}\, g\, \mathrm{W}\, g)$: it allows for those execution loops that have always a possibility of terminating, and when they do, the goal g is guaranteed to be achieved. In Figure 1, the goal $(\mathsf{locked} \wedge \neg \mathsf{loaded}) \rightarrow \mathrm{A}(\mathrm{EF}\, (\mathsf{locked} \wedge \mathsf{loaded})\, \mathrm{W} (\mathsf{locked} \wedge \mathsf{loaded}))$ can be satisfied by a plan that unlocks, loads, and, if the outcome is state 3, locks again, while if the item is misplaced (state 5) repeatedly tries to adjust the position of the item until (hopefully) state 3 is reached, and finally locks the container.

We can distinguish among different kinds of *maintainability goals*, e.g., $\mathrm{AG}\, g$ ("maintain g"), $\mathrm{AG}\, \neg g$ ("avoid g"), $\mathrm{EG}\, g$ ("try to maintain g"), and $\mathrm{EG}\, \neg g$ ("try to avoid g"). For instance, a robot should never harm people and should always avoid dangerous areas. Weaker requirements might be needed for less critical properties, like the fact that the robot should try to avoid to run out of battery.

We can *compose reachability and maintainability goals*. $\mathrm{AF}\, \mathrm{AG}\, g$ states that a plan should guarantee that all executions reach eventually a set of states where g can be maintained. For instance, an air-conditioner controller is required to reach eventually a state such that the temperature can then be maintained in a given range. Alternatively, if you consider the case in which a pump might fail to turn on when it is selected, you might require that "there exists a possibility" to reach the condition to maintain the temperature in a desired range ($\mathrm{EF}\, \mathrm{AG}\, g$). As a further example, the goal $\mathrm{AG}\, \mathrm{EF}\, g$ intuitively means "maintain the possibility of reaching g".

Reachability – preserving goals make use of the "until operators" ($\mathrm{A}(g_1\, \mathrm{U}\, g_2)$ and $\mathrm{E}(g_1\, \mathrm{U}\, g_2)$) to express reachability goals while some property must be preserved. For instance, an air-conditioner might be required to reach a desired tem-

perature while leaving at least n of its m pumps off.

As a last example in the domain of Figure 1, consider the goal "from state 2, where the container is unlocked and empty, lock the container first, and then maintain the possibility of reaching a state where the item is loaded and the container is locked (state 4)". It can be formalized as $(\neg\mathsf{locked} \wedge \neg\mathsf{loaded} \wedge \neg\mathsf{misplaced}) \rightarrow (\mathrm{AF}\,(\mathsf{locked} \wedge \mathrm{AG}\,\mathrm{EF}\,(\mathsf{locked} \wedge \mathsf{loaded})))$. In the rest of the paper, we call this example of goal "*lock-then-load goal*".

Notice that in all examples above, the ability of composing formulas with universal and existential path quantifiers is essential. Logics that do not provide this ability, like LTL [Emerson, 1990], cannot express these kinds of goals[1].

4 Plans for Extended Goals

A plan describes the actions that have to be performed in a given state of the world. In order to satisfy extended goals, actions that have to be executed may also depend on the "internal state" of the executor, which can take into account, e.g., previous execution steps. Consider again the "lock-then-load goal" for domain in Figure 1. The plan, starting from state 2, has first to lead to state 1, and then to state 4. In state 2, the first time we have to execute action *lock*, while we have to *load* the item the second time. In general, a plan can be defined in terms of an *action function* that, given a state and an *execution context* encoding the internal state of the executor, specifies the action to be executed, and in terms of a *context function* that, depending on the action outcome, specifies the next execution context.

Definition 3 *A* plan *for a domain D is a tuple $\langle C, c_0, act, ctxt\rangle$, where:*

- *C is a set of (execution) contexts,*
- *$c_0 \in C$ is the initial context,*
- *$act : Q \times C \rightharpoonup A$ is the action function,*
- *$ctxt : Q \times C \times Q \rightharpoonup C$ is the context function.*

If we are in state q and in execution context c, then $act(q, c)$ returns the action to be executed by the plan, while $ctxt(q, c, q')$ associates to each reached state q' the new execution context. Functions *act* and *ctxt* may be partial, since some state-context pairs are never reached in the execution of the plan. An example of a plan that satisfies the lock-then-load goal is shown in Figure 2. Notice that the context changes from c_0 to c_1 when the execution reaches state 1. This allows the plan to execute different actions in state 2.

In the rest of the paper we consider only plans that are *executable and complete*. We say that plan π is *executable* if, whenever $act(q, c) = a$ and $ctxt(q, c, q') = c'$, then $q \xrightarrow{a} q'$. We say that π is *complete* if, whenever $act(q, c) = a$ and $q \xrightarrow{a} q'$, then there is some context c' such that $ctxt(q, c, q') = c'$ and $act(q', c')$ is defined. Intuitively, a complete plan always specifies how to proceed for all the possible outcomes of any action in the plan.

[1] In general, CTL and LTL have incomparable expressive power (see [Emerson, 1990] for a comparison). We focus on CTL since it provides the ability of expressing goals that take into account nondeterminism.

$act(2, c_0) = lock$	$ctxt(2, c_0, 1) = c_1$
$act(1, c_1) = unlock$	$ctxt(1, c_1, 2) = c_1$
$act(2, c_1) = load$	$ctxt(2, c_1, 3) = c_1$
	$ctxt(2, c_1, 5) = c_1$
$act(5, c_1) = adjust$	$ctxt(5, c_1, 5) = c_1$
	$ctxt(5, c_1, 3) = c_1$
$act(3, c_1) = lock$	$ctxt(3, c_1, 4) = c_1$
$act(4, c_1) = wait$	$ctxt(4, c_1, 4) = c_1$

Figure 2: An example of plan

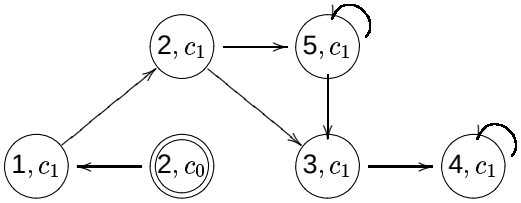

Figure 3: An example of execution structure

The execution of a plan results in a change in the current state and in the current context. It can therefore be described in terms of transitions between pairs state-context. Formally, given a domain D and a plan π, a transition of plan π in D is a tuple $(q, c) \xrightarrow{a} (q', c')$ such that $q \xrightarrow{a} q'$, $a = act(q, c)$, and $c' = ctxt(q, c, q')$. A *run* of plan π from state q_0 is an infinite sequence $(q_0, c_0) \xrightarrow{a_0} (q_1, c_1) \xrightarrow{a_1} (q_2, c_2) \xrightarrow{a_2} (q_3, c_3) \cdots$ where $(q_i, c_i) \xrightarrow{a_i} (q_{i+1}, c_{i+1})$ are transitions. Given a plan, we may have an infinite number of runs due to the nondeterminism of the domain. This is the case of the plan in Figure 2, since the execution can loop non-deterministically over the pair state-context $(5, c_1)$. We provide a finite presentation of the set of all possible runs of a plan with an *execution structure*, i.e, a Kripke Structure [Emerson, 1990] whose set of states is the set of state-context pairs, and whose transition relation corresponds to the transitions of the runs.

Definition 4 *The* execution structure *of plan π in a domain D from state q_0 is the structure $K = \langle S, R, L\rangle$, where:*

- *$S = \{(q, c) : act(q, c)$ is defined$\}$,*
- *$((q, c), (q', c')) \in R$ if $(q, c) \xrightarrow{a} (q', c')$ for some a,*
- *$L(q, c) = \{b : b \in q\}$*

As an example, the execution structure of the plan in Figure 2 is depicted in Figure 3.

We define when a goal g is true in (q, c), written $K, (q, c) \models g$ by using the standard semantics for CTL formulas over the Kripke Structure K. The complete formal definition can be found in, e.g., [Emerson, 1990]. In order to make the paper self contained, we present here some cases. Propositional formulas are treated in the usual way. $K, (q, c) \models \mathrm{AX}\, g$ iff for every path $(q, c)_0(q, c)_1(q, c)_2 \cdots$, with $(q, c) = (q, c)_0$, we have $K, (q, c)_1 \models g$. $K, (q, c) \models \mathrm{A}(g_1\, \mathrm{U}\, g_2)$ iff for every path $(q, c)_0(q, c)_1(q, c)_2 \cdots$, with $(q, c) = (q, c)_0$, there exists $i \geq 0$ such that $K, (q, c)_i \models g_2$ and, for all $0 \leq j < i$, $K, (q, c)_j \models g_1$. The definition is similar in the case of existential path quantifiers. We can now define the notion of plan that satisfies a given goal.

Definition 5 *Let D be a planning domain and g be a goal for D. Let π be a plan for D and K be the corresponding execution structure. Plan π satisfies goal g from initial state q_0, written $\pi, q_0 \models g$, if $K, (q_0, c_0) \models g$. Plan π satisfies goal g from the set of initial states Q_0 if $\pi, q_0 \models g$ for each $q_0 \in Q_0$.*

For instance, the plan in Figure 2 satisfies the lock-then-load goal from state 2.

5 Planning Algorithm

The planning algorithm searches through the domain by trying to satisfy a goal g in a state q. Goal g defines conditions on the current state and on the next states to be reached. Intuitively, if g must hold in q, then some conditions must be projected to the next states. The algorithm extracts the information on the conditions on the next states by "progressing" the goal g. For instance, if g is $\mathrm{EF}\, g'$, then either g' holds in q or $\mathrm{EF}\, g'$ must still hold in some next state, i.e., $\mathrm{EX}\,\mathrm{EF}\, g'$ must hold in q. One of the basic building blocks of the algorithm is the function *progr* that rewrites a goal by progressing it to next states. *progr* is defined by induction on the structure of goals.

- $progr(q, \top) = \top$ and $progr(q, \bot) = \bot$;
- $progr(q, b) =$ if $b \in q$ then $\top$ else $\bot$;
- $progr(q, \neg b) =$ if $b \in q$ then $\bot$ else $\top$;
- $progr(q, g_1 \wedge g_2) = progr(q, g_1) \wedge progr(q, g_2)$;
- $progr(q, g_1 \vee g_2) = progr(q, g_1) \vee progr(q, g_2)$;
- $progr(q, \mathrm{AX}\, g) = \mathrm{AX}\, g$ and $progr(q, \mathrm{EX}\, g) = \mathrm{EX}\, g$;
- $progr(q, \mathrm{A}(g\,\mathrm{U}\,g')) = (progr(q, g) \wedge \mathrm{AX}\,\mathrm{A}(g\,\mathrm{U}\,g')) \vee progr(q, g')$ and $progr(q, \mathrm{E}(g\,\mathrm{U}\,g')) = (progr(q, g) \wedge \mathrm{EX}\,\mathrm{E}(g\,\mathrm{U}\,g')) \vee progr(q, g')$;
- $progr(q, \mathrm{A}(g\,\mathrm{W}\,g')) = (progr(q, g) \wedge \mathrm{AX}\,\mathrm{A}(g\,\mathrm{W}\,g')) \vee progr(q, g')$ and $progr(q, \mathrm{E}(g\,\mathrm{W}\,g')) = (progr(q, g) \wedge \mathrm{EX}\,\mathrm{E}(g\,\mathrm{W}\,g')) \vee progr(q, g')$.

The formula $progr(q, g)$ can be written in a normal form. We write it as a disjunction of two kinds of conjuncts, those of the form $\mathrm{AX}\, f$ and those of the form $\mathrm{EX}\, h$, since we need to distinguish between formulas that must hold in all the next states and those that must hold in some of the next states:

$$progr(q, g) = \bigvee_{i \in I} \Big(\bigwedge_{f \in A_i} \mathrm{AX}\, f \wedge \bigwedge_{h \in E_i} \mathrm{EX}\, h \Big)$$

where $f \in A_i$ ($h \in E_i$) if $\mathrm{AX}\, f$ ($\mathrm{EX}\, h$) belongs to the i-th disjunct of $progr(q, g)$. We have $|I|$ different disjuncts that correspond to alternative evolutions of the domain, i.e., to alternative plans we can search for. In the following, we represent $progr(q, g)$ as a set of pairs, each pair containing the A_i and the E_i parts of a disjunct:

$$progr(q, g) = \{(A_i, E_i) \mid i \in I\}$$

with $progr(q, \top) = \{(\emptyset, \emptyset)\}$ and $progr(q, \bot) = \emptyset$.

Given a disjunct (A, E) of $progr(q, g)$, we can define a function that assigns goals to be satisfied to the next states. We denote with $assign\text{-}progr((A, E), Q)$ the set of all the possible assignments $i : Q \to 2^{A \cup E}$ such that each universally quantified goal is assigned to all the next states (i.e., if $f \in A$ then $f \in i(q)$ for all $q \in Q$) and each existentially quantified goal is assigned to one of the next states (i.e., if $h \in E$ and $h \notin A$ then $f \in i(q)$ for one particular $q \in Q$). Consider the following example in the domain of Figure 1. Let g be $\mathrm{AF}\,\mathsf{locked} \wedge \mathrm{EX}\,\mathsf{misplaced} \wedge \mathrm{EX}\,\mathsf{loaded}$ and let the current state q be 2. We have that $\mathrm{AX}\,\mathrm{AF}\,\mathsf{locked} \wedge \mathrm{EX}\,\mathsf{misplaced} \wedge \mathrm{EX}\,\mathsf{loaded} \in progr(2, g)$. If we consider action *load*, the next states are 3 and 5. Then $\mathrm{AF}\,\mathsf{locked}$ must hold in 3 *and* in 5, while $\mathsf{misplaced}$ and loaded must hold in 3 *or* in 5. We have therefore four possible state-formulas assignments $i_1, \ldots, i_4$ to be explored (in the following we write f_1 for $\mathrm{AF}\,\mathsf{locked}$, h_1 for $\mathsf{misplaced}$, and h_2 for loaded):

$$\begin{array}{ll} i_1(3) = f_1 \wedge h_1 \wedge h_2 & i_1(5) = f_1 \\ i_2(3) = f_1 \wedge h_1 & i_2(5) = f_1 \wedge h_2 \\ i_3(3) = f_1 \wedge h_2 & i_3(5) = f_1 \wedge h_1 \\ i_4(3) = f_1 & i_4(5) = f_1 \wedge h_1 \wedge h_2 \end{array}$$

In this simple example, it is easy to see that the only assignment that may lead to a successful plan is i_3.

Given the two basic building blocks *progr* and *assign-progr*, we can now describe the planning algorithm *build-plan* that, given a goal g_0 and an initial state q_0, returns either a plan or a failure.[2] The algorithm is reported in Figure 4. It performs a depth-first forward search: starting from the initial state, it picks up an action, progresses the goal to successor states, and iterates until either the goal is satisfied or the search path leads to a failure. The algorithm uses as the "contexts" of the plan the list of the active goals that are considered at the different stages of the exploration. More precisely, a context is a list $c = [g_1, \ldots, g_n]$, where the g_i are the active goals, as computed by functions *progr* and *assig-progr*, and the order of the list represents the *age* of these goals: the goals that are active since more steps come first in the list.

The main function of the algorithm is function *build-plan-aux*$(q, c, pl, open)$, that builds the plan for context c from state q. If a plan is found, then it is returned by the function. Otherwise, $\bot$ is returned. Argument *pl* is the plan built so far by the algorithm. Initially, the argument passed to *build-plan-aux* is $pl = \langle C, c_0, act, ctxt \rangle = \langle \emptyset, g_0, \emptyset, \emptyset \rangle$. Argument *open* is the list of the pairs state-context of the currently open problems: if $(q, c) \in$ *open* then we are currently trying to build a plan for context c in state q. Whenever function *build-plan-aux* is called with a pair state-context already in *open*, then we have a loop of states in which the same sub-goal has to be enforced. In this case, function *is-good-loop*$((q, c), open)$ is called that checks whether the loop is valid or not. If the loop is good, plan *pl* is returned, otherwise function *build-plan-aux* fails.

Function *is-good-loop* computes the set *loop-goals* of the goals that are active during the whole loop: iteratively, it considers all the pairs (q', c') that appear in *open* up to the next occurrence of the current pair (q, c), and it intersects *loop-goals* with the set **setof**(c') of the goals in list c'. Then, function *is-good-loop* checks whether there is some strong until goal among the *loop-goals*. If this is a case, then the loop is bad: the semantics of CTL requires that all the strong until goals are eventually fulfilled, so these goals should not

[2] It is easy to extend the algorithm to the case of more than one initial state.

```
function build-plan(q_0, g_0) : Plan
  return build-plan-aux(q_0, [g_0], ⟨∅, g_0, ∅, ∅⟩, ∅)

function build-plan-aux(q, c, pl, open) : Plan
  if (q, c) ∈ open then
    if is-good-loop((q, c), open) then return pl
    else return ⊥
  if defined pl.act[q, c] then return pl
  foreach a ∈ Act(p) do
    foreach (A, E) ∈ progr(q, c) do
      foreach i ∈ assign-progr((A, E), Exec(q, a)) do
        pl' := pl
        pl'.C := pl'.C ∪ {c}
        pl'.act[q, c] := a
        open' := conc((q, c'), open)
        foreach q' ∈ Exec(q, a) do
          c' := order-goals(i[q'], c)
          pl'.ctxt[q, c, q'] := c'
          pl' := build-plan-aux(q', c', pl', open')
          if pl' = ⊥ then next i
        return pl'
  return ⊥

function is-good-loop((q, c), open) : boolean
  loop-goals := setof(c)
  while (q, c) ≠ head(open) do
    (q', c') := head(open)
    loop-goals := loop-goals ∩ setof(c')
    open := tail(open)
  if ∃g ∈ loop-goals : g = A(_ U _) or g = E(_ U _) then
    return false
  else
    return true
```

Figure 4: The planning algorithm.

stay active during a whole loop. In fact, this is the difference between strong and weak until goals: executions where some weak until goal is continuously active and never fulfilled are acceptable, while the strong untils should be eventually fulfilled if they become active.

If the pair (q, c) is not in *open* but it is in the plan *pl* (i.e., (q, c) is in the range of function *act* and hence condition "**defined** $pl.act[q, c]$" is true), then a plan for the pair has already been found in another branch of the search, and we return immediately with a success. If the pair state-context is neither in *open* nor in the plan, then the algorithm considers in turn all the executable actions a from state q, all the different possible progresses (A, E) returned by function *progr*, and all the possible assignments i of (A, E) to $\mathrm{Exec}(q, a)$. Function *build-plan-aux* is called recursively for each destination state in $q' \in \mathrm{Exec}(q, a)$. The new context is computed by function *order-goals*$(i[q'], c)$: this function returns a list of the goals in $i[q']$ that are ordered by their "age": namely those goals that are old (they appear in $i[q']$ and also in c) appear first, in the same order as in c, and those that are new (they appear in $i[q']$ but not in c) appear at the end of the list, in any order. Also, in the recursive call, argument *pl* is updated to take into account the fact that action a has been selected from state q in context g. Moreover, the new list of open problems is updated to **conc**$((q, c), open)$, namely the pair (q, c) is added in front of argument *open*.

Any recursive call of *build-plan-aux* updates the current plan pl'. If all these recursive calls are successful, then the final value of plan pl' is returned. If any of the recursive calls returns $\bot$, then the next combination of assign decomposition, progress component and action is tried. If all these combinations fail, then no plan is found and $\bot$ is returned.

As an example, call *build-plan*$(2, \text{lock-then-load})$ is successful and returns the plan in Figure 2 where c_0 and c_1 are goals $\mathrm{AF}(\mathsf{locked} \wedge \mathrm{AG}\,\mathrm{EF}(\mathsf{locked} \wedge \mathsf{loaded}))$ and $\mathrm{AG}\,\mathrm{EF}(\mathsf{locked} \wedge \mathsf{loaded})$, respectively.

The algorithm always terminates, and it is correct and complete: given a state q of a domain D and a goal g for D, if *build-plan*$(q, g) = \pi$ then $\pi, q \models g$, and if *build-plan*$(q, g) = \bot$ then there is no plan π such that $\pi, q \models g$.

6 Symbolic Implementation and Experimental Results

We have implemented the planning algorithm and have performed some experimental evaluations. Though very preliminary, the experiments define some basic test cases for planning for CTL goals in non-deterministic domains, show that the approach is effective in practice with cases of significant complexity, and settle the basis for future comparisons.

We have implemented the algorithm inside MBP ([Cimatti *et al.*, 1998]). MBP uses symbolic techniques based on BDDs [Burch *et al.*, 1992] to overcome the problems of the explicit-state planning algorithm due to the huge size of realistic domains, and in particular of non-deterministic domains.

In order to provide a BDD-based implementation, the explicit algorithm presented in the previous section has to be revisited, taking into account the fact that BDDs work effectively on sets of states rather than on single states. For lack of space, we can not describe the symbolic BDD-based algorithm in details: further information on this algorithm, as well as on the test cases, can be found at the MBP home page `http://sra.itc.it/tools/mbp`.

One of the very few examples of planning for extended goals in non-deterministic domains that we have found in the literature is the "robot-moving-objects" search problem, presented in [Kabanza *et al.*, 1997] to test the SIMPLAN planner for some LTL-like goals. The domain consists of a set of rooms connected by doors, of a set of objects in the rooms, and of a robot that can grasp the objects and carry them to different rooms. The non-determinism in the domain is due to the fact that (some of) the doors are defective and can close without an explicit action of the robot.

We have performed experiments with different extended goals. For lack of space we report only the results for the goal of moving the objects into given rooms and keeping them there (experiment 1 in [Kabanza *et al.*, 1997]). In our framework, this goal is of the form $\mathrm{AF}\,\mathrm{AG}\,g$. The problem is parametrized in the number of the possible objects to be moved and in the number of defective doors. The time required to build the plan is reported in Figure 5 (all tests were

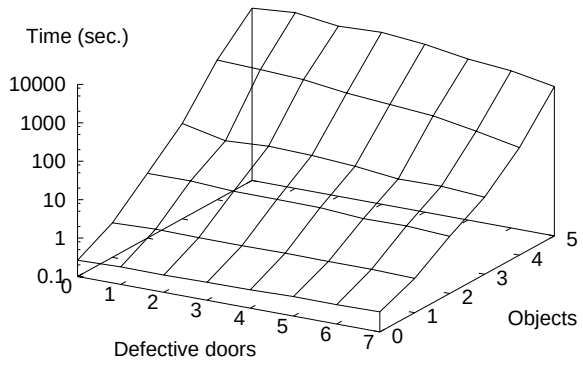

Figure 5: Experimental results

performed on a Pentium II 300 MHz with 512 Mb RAM of memory running Linux). The time scale is logarithmic and shows that the required time grows exponentially in the number of objects (this corresponds to an exponential growth in the size of the domain). Due to the usage of symbolic techniques, instead, the performance is not influenced by the non-determinism. We remark that in the case of 5 objects the domain is quite complex: it has more than 10^8 states.

The results reported in [Kabanza *et al.*, 1997] show a complementary behavior: SIMPLAN scales well with respect to the number of objects, but the explicit state search suffers significantly in the case of non-deterministic domains. We remark, however, that the efficient behavior of SIMPLAN in the case of deterministic domains depends on the enforced domain-dependent search control strategies, that are able to cut the largest part of the search graph. Our experiments show that MBP outperforms SIMPLAN, if the latter is executed without control strategies. This result is not surprising: symbolic techniques have shown to be dramatically more efficient that explicit techniques in the case of huge search space.

7 Conclusions and Related Work

In this paper we have presented an approach to automatic planning in non-deterministic domains where the goals are expressed as CTL formulas. We have implemented the algorithm by using symbolic model checking techniques, which open up the possibility to deal with large state space.

Some future objectives are the following. The symbolic implementation is still a rather naive transcription of the explicit algorithm presented in the paper: further work in needed to develop a symbolic algorithm that fully exploits the potentiality of BDDs and of the symbolic exploration of huge state spaces. Moreover, we plan to perform an extensive test with different kinds of extended goals on a set of realistic domains, to show that the approach is indeed practical. Finally, in this paper we focus on the case of full observability. An extension of the work to the case of planning for extended goals under partial observability is one of the main objectives for future research.

The problem of planning for CTL goals has never been solved before. The starting point of the work presented in this paper is the framework of "Planning via Symbolic Model Checking" (see, e.g., [Cimatti *et al.*, 1998; Daniele *et al.*, 1999; Bertoli *et al.*, 2001]). None of the previous works in this framework deals with temporally extended goals. The issue of "temporally extended goals" is certainly not new. However, most of the works in this direction restrict to deterministic domains, see for instance [de Giacomo and Vardi, 1999; Bacchus and Kabanza, 1998]. A work that considers extended goals in non-deterministic domains is described in [Kabanza *et al.*, 1997]: see Section 6 for a comparison.

Extended goals make the planning problem close to that of automatic synthesis of controllers (see, e.g., [Asarin *et al.*, 1995; Kupferman and Vardi, 1997]). However, most of the work in this area focuses on the theoretical foundations, without providing practical implementations. Moreover, it is based on rather different technical assumptions on actions and on the interaction with the environment.

References

[Asarin *et al.*, 1995] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid System II*, LNCS 999, 1995.

[Bacchus and Kabanza, 1998] F. Bacchus and F. Kabanza. Planning for Temporally Extended Goals. *Annals of Mathematics and Artificial Intelligence*, 22:5–27, 1998.

[Bertoli *et al.*, 2001] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc. of IJCAI'01*, 2001.

[Bonet and Geffner, 2000] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proc. of AIPS 2000*, 2000.

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

[Cimatti *et al.*, 1998] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Proc. of AAAI'98*, 1998.

[Daniele *et al.*, 1999] M. Daniele, P. Traverso, and M. Y. Vardi. Strong Cyclic Planning Revisited. In *Proc. of ECP'99*, 1999.

[de Giacomo and Vardi, 1999] G. de Giacomo and M.Y. Vardi. Automata-theoretic approach to planning with temporally extended goals. In *Proc. of ECP'99*, 1999.

[Emerson, 1990] 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. Elsevier, 1990.

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

[Kupferman and Vardi, 1997] O. Kupferman and M.Y. Vardi. Synthesis with incomplete information. In *Proc. of 2nd International Conference on Temporal Logic*, 1997.

[Schoppers, 1987] M. J. Schoppers. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proc. of IJCAI'87*, 1987.