Planning as Model Checking

Fausto Giunchiglia[1,2] and Paolo Traverso[1]

[1] IRST, Istituto per la Ricerca Scientifica e Tecnologica, 38050 Povo, Trento, Italy
[2] Dipartimento di Informatica e Studi Aziendali, Universita' di Trento, Italy
fausto@irst.itc.it, leaf@irst.itc.it

Abstract. The goal of this paper is to provide an introduction, with various elements of novelty, to the Planning as Model Checking paradigm.

1 Introduction

The key idea underlying the Planning as Model Checking paradigm is that planning problems should be solved model-theoretically. Planning domains are formalized as semantic models. Properties of planning domains are formalized as temporal formulas. Planning is done by verifying whether temporal formulas are true in a semantic model.

The most important features of the proposed approach are:

- The approach is well-founded. Planning problems are given a clear and intuitive (semantic) formalization.
- The approach is general. The same framework can be used to tackle most research problems in planning, e.g., planning in deterministic and in nondeterministic domains, conditional and iterative planning, reactive planning.
- The approach is practical. It is possible to devise efficient algorithms that generate plans automatically and that can deal with large size problems.

The goal of this paper is to provide an introduction, with various elements of novelty, to the Planning as Model Checking paradigm. The core of the paper are Sections 2, 3, 4, and 5. Section 2 gives a brief introduction to the model checking problem. Section 3 shows how planning problems can be stated as model checking problems. Section 4 shows how our formalization can be used to tackle various planning problems. Section 5 shows how the approach can be extended to nondeterministic domains. Section 6 shows how Planning as Model Checking can be implemented. Our implementation relies heavily on existing work in the context of finite-state program verification (see [9] for an overview), and in particular on the work described in [8, 2, 20]. Section 7 discusses the related work.

2 Model Checking

The Model Checking problem is the problem of determining whether a formula is true in a model. Model checking is based on the following fundamental ideas:

1. A domain of interest (e.g., a computer program, a reactive system) is described by a semantic model.
2. A desired property of the domain (e.g., a specification of a program, a safety requirement for a reactive system) is described by a logical formula.
3. The fact that a domain satisfies a desired property (e.g., the fact that a program meets its specifications, that a reactive system never ends up in a dangerous state) is determined by checking whether the formula is true in the model, i.e., by model checking.

We formalize domains as Kripke Structures. We restrict ourselves to the case of finite domains, i.e., domains which can be described by Kripke Structures with a finite number of states. A Kripke Structure K is a 4-tuple $\langle W, W_0, T, L\rangle$, where

1. W is a finite set of *states*.
2. $W_0 \subseteq W$ is a set of initial states.
3. $T \subseteq W \times W$ is a binary relation on W, the *transition relation*, which gives the possible transitions between states. We require T to be total, i.e., for each state $w \in W$ there exists a state $w' \in W$ such that $(w, w') \in T$.
4. $L : W \mapsto 2^{\mathcal{P}}$ is a *labeling function*, where $\mathcal{P}$ is a set of atomic propositions. L assigns to each state the set of atomic propositions true in that state.

A Kripke Structure encodes the possible evolutions of the domain (or behaviours) as *paths*, i.e., infinite sequences $w_0 w_1 w_2 \ldots$ of states in W such that, for each i, $(w_i, w_{i+1}) \in T$. We require that paths start from an initial state $w_0 \in W_0$. By requiring that T is total, we impose that all paths are infinite.

As a simple example of Kripke Structure, consider Figure 1. It depicts a simple domain, where an item can be loaded/unloaded to/from a container which can be locked/unlocked. The corresponding Kripke Structure is the following:

1. $W = \{1, 2, 3, 4\}$
2. $W_0 = \{2\}$
3. $T = \{(1,2), (2,1), (2,2), (2,3), (3,2), (3,4), (4,3)\}$
4. $L(1) = \{Locked\}$, $L(2) = \emptyset$, $L(3) = \{Loaded\}$, $L(4) = \{Loaded, Locked\}$.

We formalize temporal properties of domains in Computation Tree Logic (CTL) [12]. Given a finite set $\mathcal{P}$ of atomic propositions, CTL formulas are inductively defined as follows:

1. Every atomic proposition $p \in \mathcal{P}$ is a CTL formula;
2. If p and q are CTL formulas, then so are
   (a) $\neg p$, $p \vee q$,

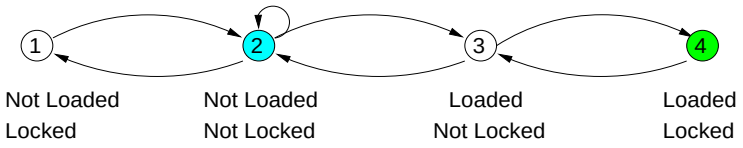

Fig. 1. An example of Kripke Structure

(b) $\mathbf{AX}p$, $\mathbf{EX}p$,
(c) $\mathbf{A}(p\mathbf{U}q)$, and $\mathbf{E}(p\mathbf{U}q)$)

$\mathbf{X}$ is the "next time" temporal operator; the formula $\mathbf{AX}p$ ($\mathbf{EX}p$) intuitively means that p holds in every (in some) immediate successor of the current state. $\mathbf{U}$ is the "until" temporal operator; the formula $\mathbf{A}(p\mathbf{U}q)$ ($\mathbf{E}(p\mathbf{U}q)$) intuitively means that for every path (for some path) there exists an initial prefix of the path such that q holds at the last state of the prefix and p holds at all the other states along the prefix. Formulas $\mathbf{AF}p$ and $\mathbf{EF}p$ (where the temporal operator $\mathbf{F}$ stands for "future" or "eventually") are abbreviations of $\mathbf{E}(\top\mathbf{U}p)$ and $\mathbf{A}(\top\mathbf{U}p)$ (where $\top$ stands for truth), respectively. $\mathbf{EG}p$ and $\mathbf{AG}p$ (where $\mathbf{G}$ stands for "globally" or "always") are abbreviations of $\neg\mathbf{AF}\neg p$ and $\neg\mathbf{EF}\neg p$, respectively. In the example in Figure 1, $\mathbf{EF}$ *Loaded* holds in state 2, since *Loaded* holds eventually in a state of the path $2, 3, 4, 3, 4\ldots$. Instead, $\mathbf{AF}$ *Loaded* does not hold in state 2, since *Loaded* does not hold in any state of the path $2, 2, \ldots$.

CTL semantics is given in terms of Kripke Structures. We write $K, w \models p$ to mean that p holds in the state w of K. Let p be a CTL formula. $K, w \models p$ is defined inductively as follows:

- $K, w \models p$ iff $p \in L(w)$, $p \in \mathcal{P}$
- $K, w \models \neg p$ iff $K, w \not\models p$
- $K, w \models p \vee q$ iff $K, w \models p$ or $K, w \models q$
- $K, w \models \mathbf{AX}p$ iff for all paths $\pi = w_0 w_1 w_2 \ldots$, with $w = w_0$, we have $K, w_1 \models p$
- $K, w \models \mathbf{EX}p$ iff there exists a path $\pi = w_0 w_1 w_2 \ldots$, with $w = w_0$, such that $K, w_1 \models p$
- $K, w \models \mathbf{A}(p\mathbf{U}q)$ iff for all paths $\pi = w_0 w_1 w_2 \ldots$, with $w = w_0$, there exists $i \geq 0$ such that $K, w_i \models q$ and, for all $0 \leq j < i$, $K, w_j \models p$
- $K, w \models \mathbf{E}(p\mathbf{U}q)$ iff there exists a path $\pi = w_0 w_1 w_2 \ldots$, with $w = w_0$, and $i \geq 0$ such that $K, w_i \models q$ and, for all $0 \leq j < i$, $K, w_j \models p$

We say that p is true in K ($K \models p$) if $K, w \models p$ for each $w \in W_0$. The Model Checking Problem for a CTL formula p and a Kripke Structure K is the problem of determining whether p is true in K.

Algorithms for model checking exploit the structure of CTL formulas. For instance, an atomic formula p is model checked by verifying that $p \in L(s)$ for all $s \in W_0$. As another example, model checking $\mathbf{AX}p$ ($\mathbf{EX}p$) is performed by model checking p in all states (in some state) s' such that $(s, s') \in T$, for each

```
1.  function MCheckEF(p,K)
2.    CurrentStates:= ∅;
3.    NextStates := States(p,K);
4.    while NextStates ≠ CurrentStates do
5.      if (W0 ⊆ NextStates)
6.        then return True;
7.      CurrentStates := NextStates;
8.      NextStates := NextStates ∪ OneStepMCheck(NextStates,K);
9.    endwhile
10.     return False;
```

Fig. 2. Model Checking $\mathbf{EF}p$.

$s \in W_0$. Finally, $\mathbf{A}(p\mathbf{U}q)$ or $\mathbf{E}(p\mathbf{U}q)$, can be model checked by exploiting the fact that

$$\begin{aligned} p\mathbf{U}q = {} & q \vee \\ & (p \wedge \mathbf{X}q) \vee \\ & (p \wedge \mathbf{X}p \wedge \mathbf{XX}q) \vee \\ & \ldots \end{aligned}$$

As a simple explanatory example, we show in Figure 2 a possible algorithm for model checking the CTL formula $\mathbf{EF}p$, with $p \in \mathcal{P}$. Given a Kripke Structure $K = \langle W, W_0, T, L\rangle$, and a propositional formula p, the algorithm starts by computing the set of states where p holds (line 3). We have in fact that:

$$\textsc{States}(p, K) = \{s \in W : \ p \in L(s)\} \tag{1}$$

Then, MCheckEF explores the state space of K. It repeatedly accumulates in *NextStates* the states returned by OneStepMCheck (line 8). Given a set of states $States \subseteq W$, OneStepMCheck returns the set of states which have at least one immediate successor state in *States*:

$$\textsc{OneStepMCheck}(States, K) = \{s \in W : \ \exists s'.\ (s' \in States \ \wedge \ T(s, s'))\} \tag{2}$$

Notice that $\mathbf{EF}p$ always holds in each state in *NextStates*. The loop terminates successfully if *NextStates* contains all the initial states (termination condition at line 5). MCheckEF returns *False* if *NextStates* does not contain all the initial states and there are no new states to explore, i.e., *NextStates* = *CurrentStates*. Termination is guaranteed by the monotonicity of the operator accumulating states and by the fact that the set of states is finite. When applied to the Kripke Structure in Figure 1 and to the formula $\mathbf{EF}(Loaded \wedge Locked)$, MCheckEF starts with state 4 in *NextStates*, after the first step *NextStates* is {3,4}, after the second step *NextStates* is {2,3,4}, and then the algorithm stops returning *True*.

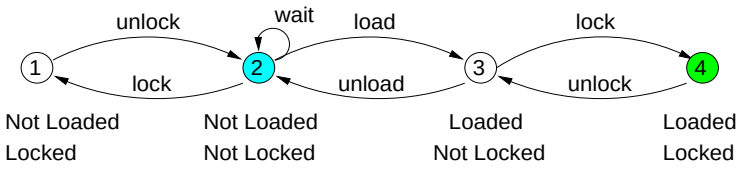

Fig. 3. An example of Planning Domain

3 Planning as Model Checking

The underlying idea of the Planning as Model Checking paradigm is to generate plans by determining whether formulas are true in a model. The fundamental ingredients are the following:

1. A planning domain is described by a semantic model, which defines the states of the domain, the available actions, and the state transitions caused by the execution of actions.
2. A planning problem is the problem of finding plans of actions given planning domain, initial and goal states.
3. Plan generation is done by exploring the state space of the semantic model. At each step, plans are generated by checking the truth of some suitable formulas in the model.

A *planning domain* D is a 4-tuple $\langle F, S, A, R\rangle$ where

1. F is a finite set of *fluents*,
2. $S \subseteq 2^F$ is a finite set of *states*,
3. A is a finite set of *actions*,
4. $R : S \times A \mapsto S$ is a *transition function*. The action $a \in A$ is said to be *executable* in $s \in S$ if $R(s, a) \neq \emptyset$.

In this section, we restrict ourselves to *deterministic actions*, which, given a state, lead to a single state. Notice that R is a function. Notice also that planning domains are general enough to express actions with secondary effects [21], which can be described in ADL-like languages (e.g., PDDL) but not in pure STRIPS-like languages.

In Figure 3 we depict a simple planning domain. It is obtained from the example in Figure 1 by labeling transitions with actions. The corresponding planning domain is the following.

1. $F = \{Loaded, Locked\}$
2. $S = \{ \{\neg\ Loaded,\ Locked\}, \{\neg\ Loaded, \neg\ Locked\},$
   $\{\ Loaded, \neg\ Locked\}, \{\ Loaded,\ Locked\}\ \}$
3. $A = \{\ lock,\ unlock,\ load,\ unload,\ wait\}$

```
1.  function PLAN(P)
2.    CurrentStates := ∅;
3.    NextStates := G;
4.    Plan := ∅;
5.    while (NextStates ≠ CurrentStates) do
6.        if I ⊆ NextStates
7.            then return Plan;
8.        OneStepPlan := ONESTEPPLAN(NextStates, D);
9.        Plan := Plan ∪ PRUNESTATES(OneStepPlan, NextStates);
10.       CurrentStates := NextStates;
11.       NextStates := NextStates ∪ PROJECTACTIONS(OneStepPlan);
12.   return Fail;
```

Fig. 4. A "Planning as Model Checking" Algorithm.

4. $R = \{$ $(\{\neg\ Loaded,\ Locked\}, unlock, \{\neg\ Loaded,\ \neg\ Locked\})$
$(\{\neg\ Loaded,\ \neg\ Locked\},\ lock, \{\neg\ Loaded,\ Locked\})$
$(\{\neg\ Loaded,\ \neg\ Locked\}, wait, \{\neg\ Loaded,\ \neg\ Locked\})$
$(\{\neg\ Loaded,\ \neg\ Locked\}, load, \{Loaded,\ \neg\ Locked\})$
$(\{Loaded,\ \neg\ Locked\}, unload, \{\neg\ Loaded,\ \neg\ Locked\})$
$(\{Loaded,\ \neg\ Locked\}, lock, \{Loaded,\ Locked\})$
$(\{Loaded,\ Locked\}, unlock, \{Loaded,\ \neg\ Locked\})$ $\}$

A *planning problem* P for a Planning Domain $D = \langle F, S, A, R\rangle$ is a 3-tuple $\langle D, I, G\rangle$, where $I = \{s_0\} \subseteq S$ is the initial state, and $G \subseteq S$ is the set of goal states.

Notice that we are restricting ourselves to planning problems with a completely specified initial situation, namely problems with a single initial state. Notice also that planning problems allow for conjunctive and disjunctive goals, and, more in general, for goals that can be described by any propositional formula.

Intuitively, plans specify actions to be executed in certain states. More precisely, a *plan* π for a planning problem $P = \langle D, I, G\rangle$ with planning domain $D = \langle F, S, A, R\rangle$ is defined as

$$\pi = \{\langle s, a\rangle :\ s \in S, a \in A\} \tag{3}$$

We call $\langle s, a\rangle$ a *state-action pair*.

We say that a plan is executable if all its actions are executable, namely if we have that $\pi = \{\langle s, a\rangle :\ s \in S, a \in A, R(s, a) \neq \emptyset\}$. In the following, we consider plans which have at least one state-action pair $\langle s, a\rangle$ with $s \in I$. A simple algorithm for plan generation is presented in Figure 4. PLAN searches backwards from G to I. At each step (see line 8), given a set of states *States*, ONESTEPPLAN computes the set of state-action pairs $\langle s, a\rangle$ such that the action a leads from s to a state in *States*:

$$\begin{aligned}&\textsc{OneStepPlan}(\mathit{States}, D) = \\ &\quad \{\langle s, a\rangle : s \in S, a \in A, \exists s'.\ (s' \in \mathit{States} \wedge s' = R(s, a))\}\end{aligned} \tag{4}$$

$\textsc{PruneStates}(\mathit{OneStepPlan}, \mathit{NextStates})$ eliminates from $\mathit{OneStepPlan}$ the state-action pairs the states of which are already in $\mathit{NextStates}$, and thus have already been visited.

$$\textsc{PruneStates}(\pi, \mathit{States}) = \{\langle s, a\rangle \in \pi : s \notin \mathit{States}\} \tag{5}$$

$\textsc{ProjectActions}$, given a set of state-action pairs, returns the corresponding set of states.

$$\textsc{ProjectActions}(\pi) = \{s : \langle s, a\rangle \in \pi\} \tag{6}$$

Consider the example in Figure 3, with 2 as initial state and 4 as goal state. $\textsc{Plan}$ starts from state 4, after the first step Plan is $\{\langle 3, \mathit{lock}\rangle\}$, and after the second step Plan is $\{\langle 2, \mathit{load}\rangle, \langle 3, \mathit{lock}\rangle\}$. Notice that the pair $\{\langle 4, \mathit{unlock}\rangle\}$ is eliminated by $\textsc{PruneStates}$. Therefore the algorithm stops returning $\{\langle 2, \mathit{load}\rangle, \langle 3, \mathit{lock}\rangle\}$.

Let us now see in which sense the planning problem is a model checking problem:

1. The planning domain is a semantic model.
2. The planning problem is specified through a set of goal states that corresponds to a formula representing a desired property of the domain.
3. Plan generation is done by checking whether suitable formulas are true in a semantic model.

More precisely, let $K = \langle W, W_0, T, L\rangle$ and $P = \langle D, I, G\rangle$ with $D = \langle F, S, A, R\rangle$ be a Kripke Structure and a planning problem, respectively. W, W_0 and T correspond to S, I and R, respectively. The set of atomic propositions $\mathcal{P}$ of the labeling function L corresponds to the set of fluents F. We have the following differences:

1. The arcs defined by R are labeled by actions.
2. R is not required to be total. Indeed, in planning domains we may have states where no actions are executable.
3. R is a function. Indeed, we are in deterministic domains. We extend to nondeterminism in Section 5.
4. I is a singleton. We extend to partially specified initial situations in Section 5.

Therefore, a planning problem corresponding to a Kripke Structure
$K = \langle W, W_0, T, L\rangle$ is $P = \langle D, I, G\rangle$,where

1. $D = \langle F, S, A, R\rangle$ with
   (a) $F = \mathcal{P}$,
   (b) $S = W$,
   (c) $A = \{u\}$,

(d) $R = \{(s, u, s') : (s, s') \in T\}$;
2. $I = W_0$.

It is now worthwhile to compare the algorithms MCheckEF and Plan. The basic routine of MCheckEF, OneStepMCheck (see (2)), can be defined in terms of the basic routine of Plan, OneStepPlan (see (4)), on the planning domain D_k corresponding to the Kripke Structure K:

$$\begin{aligned} &\textsc{OneStepMCheck}(\mathit{States}, K) = \\ &\qquad \textsc{ProjectActions}(\textsc{OneStepPlan}(\mathit{States}, D_k)) \end{aligned} \tag{7}$$

Notice that MCheckEF and Plan are very similar. The main difference is that MCheckEF returns either *True* or *False*, while Plan returns either a plan or a failure. Let p be a propositional CTL formula such that $K, w \models p$ for all $w \in G$ and $K, w \not\models p$ for all $w \notin G$. Then, $K \models \mathbf{EF}p$ iff there exists a plan satisfying the planning problem P corresponding to K. Therefore, we have reduced planning to the model checking of the formula $\mathbf{EF}p$. This corresponds to an underlying assumption of classical planning: the requirement for a plan is merely on the final states resulting from its execution. However, previous papers on planning (see, e.g., [26, 24, 25]) have stated that the planning problem should be generalized to the problem of finding plans that satisfy some specifications on their execution paths, rather than on the final states of their execution. The Planning as Model Checking paradigm can be a good approach for extending the classical planning problem.

4 Situated Plans

The plans we construct are actually "situated plans" (see, e.g., [15, 14]), namely plans that, at run time, are executed by a reactive loop that repeatedly senses the state, selects an appropriate action, executes it, and iterates, e.g., until the goal is reached. Indeed, a plan $\pi = \{\langle s, a\rangle : s \in S, a \in A\}$ can be viewed as the *iterative plan*

$$\begin{aligned} &\textbf{while } s \in \{s : \langle s, a\rangle \in \pi\} \textbf{ do} \\ &\quad a \text{ such that } \langle s, a\rangle \in \pi \end{aligned} \tag{8}$$

Plans as defined in Section 3 are rather general: they do not depend on the goal of the planning problem, and no condition is imposed on the fact that a plan should attempt to achieve a goal. A condition that a plan should satisfy is that, if a goal is achieved, then the plan stops execution. We call these plans, *goal preserving plans.*

A *goal preserving plan* is a set of state-action pairs
$\pi = \{\langle s, a\rangle : s \in S, a \in A, s \notin G\}$.

The goal preserving condition can be generalized. A plan, rather than "not acting in a goal state", can still act without abandoning the set of goal states. Consider

for instance the task of a robot for surveillance systems: it may be required, after reaching a given location, e.g., an area of a building, to move inside that area without leaving it.

A *dynamic goal preserving plan* is a set of state-action pairs $\pi = \{\langle s, a\rangle : s \in S, a \in A, s \in G \supset R(s, a) \in G\}$.

The goal preserving condition can be even weaker. Consider a surveillance robot which is required to repeatedly charge batteries in a given location in order to explore a building (the robot should eventually charge its batteries infinitely often). Let the goal states represent the charge battery location. This can be specified by requiring the plan to "pass through" the set of goal states infinitely often.

A *fair goal preserving plan* is a set of state-action pairs π such that for each $\langle s_0, a_0\rangle \in \pi$ with $s_0 \in G$, there exists $\{\langle s_1, a_1\rangle, \ldots \langle s_n, a_n\rangle\} \subseteq \pi$ such that $s_{i+1} = R(s_i, a_i)$ for each $0 \leq i \leq n-1$, and $s_n \in G$

A condition that a plan should satisfy is that it should achieve the goal. We call such plans, *goal achieving plans*. Intuitively, for each state-action pair in a goal achieving plan, there should be a path leading from the state of the state-action pair to a goal state. More precisely, this requirement can be described as follows: "all the state-action pairs $\langle s, a\rangle$ contained in a plan π should be such that, either a leads from s to the goal ($R(s, a) \in G$), or a leads from s to a state s' such that π contains $\langle s', a'\rangle$ and a' leads from s' to the goal ($R(s', a') \in G$), and so on. This informal requirement can be formalized as follows:

A *goal achieving plan* is defined inductively as follows.

1. $\pi = \{\langle s, a\rangle : s \in S, a \in A, R(s, a) \in G\}$ is a goal achieving plan
2. If π' is a goal achieving plan, then $\pi = \pi' \cup \{\langle s, a\rangle\}$ such that $R(s, a) \in \{s' : \langle s', a'\rangle \in \pi'\}$ is a goal achieving plan

We can now define situated plans.

A *situated plan* is a goal preserving and goal achieving plan.

Situated plans can be "robust" to unexpected action outcomes, namely, when executed, they can achieve the goal in spite of the fact that some actions may have outcomes that are not modeled. Consider the plan $\{\langle 2, \mathit{load}\rangle, \langle 3, \mathit{lock}\rangle\}$ in the example in Figure 3. The plan is robust to the fact that the execution of *load* may lead from state 2 to state 2 (leave the item unloaded). The plan is not robust to the fact that the execution of *load* may lead from state 2 to state 1 (leave the item unloaded and accidentally lock the container).

Universal plans [23] are particular cases of situated plans. The intuitive idea is that a universal plan maps sets of states to actions to be executed for each possible situation arising at execution time.

A *universal plan* is a situated plan π such that the states in π are all the possible states of the domain, i.e., for each $s \in S$ there exists $\langle s, a \rangle \in \pi$.

The definition of universal plan can be generalized to include situations which are not modeled by states of the planning domain, as far as the situations can be described by the language of the planner. A plan of this kind is a situated plan π such that for each $s \in 2^F$, there exists $\langle s, a \rangle \in \pi$.

As a very specific case, we can define plans which resemble classical plans, namely plans which consist of sequences of actions that lead from the initial state to a goal state.

A *quasi-classical plan* is a plan π such that
1. $\pi = \{\langle s_1, a_1 \rangle, \ldots, \langle s_n, a_n \rangle\}$
2. $s_1 = s_0 \in I$,
3. $R(s_i, a_i) = s_{i+1}$ for each $1 \leq i \leq n$,
4. $s_{n+1} \in G$

At planning time, a quasi-classical plan is similar to a classical plan. However, a quasi-classical plan is executed like a classical plan just under the main hypothesis of classical planning that the model of the world is perfect. Consider again the quasi-classical plan $\{\langle 2, \mathit{load} \rangle, \langle 3, \mathit{lock} \rangle\}$ in the example in Figure 3. At planning time, the plan is similar to the classical plan *load*, *lock*. Suppose that, the execution of *load* leads twice from state 2 to state 2 (leaves the item unloaded). The execution of the classical plan *load*, *lock* does not achieve the goal (state 4). The execution of the quasi-classical plan $\{\langle 2, \mathit{load} \rangle, \langle 3, \mathit{lock} \rangle\}$ results in the execution of the action *load* three times followed by the execution of the action *lock*, and achieves the goal. In this case, the execution of the quasi-classical plan is equivalent to the execution of the classical plan *load*, *load*, *load*, *lock*.

The different behaviour of a classical plan and a quasi-classical plan at execution time is a consequence of the fact that a quasi-classical plan does not impose *a priori* before execution any partial/total order on the execution of actions. We see plans as *sets*, and as such, the order of the execution of actions is determined at execution time, depending on the actual situation of the world.

We have now enough terminology to state and discuss several interesting properties of the algorithm PLAN in Figure 4.

- Let us first consider *correctness*. PLAN returns situated plans. Actions are guaranteed to be executable since state-action pairs are constructed from the planning domain. The termination condition at line 6 guarantees that the initial state is in the returned plan. The returned plan is goal preserving since states-action pairs whose states are in G are pruned away by PRUNESTATES. Finally, the returned plan is a goal achieving plan, since ONESTEPPLAN, at each step, when applied to the set of states *NextStates*, finds situated plans for the planning problems with initial states in *NextStates* and goal G.
- There may be different notions of *completeness*. A first notion is "if there exists a situated plan, PLAN finds it". The algorithm is complete in this sense, since it explores the state space exhaustively. A second notion is: "if there is

no situated plan, PLAN terminates with failure". PLAN always terminates. It is guaranteed to terminate even in the case there is no solution. This follows from the fact that *NextStates* is monotonically increasing at each step, and the number of states S is finite. Termination in the case a solution does not exists is a property which is not guaranteed by several state of the art planners, e.g., SATplan [19], BlackBox [18] and UCPOP [21]. A third notion of completeness is: "the planning algorithm returns *all* the possible situated plans". PLAN is *not* complete in this sense, since it stops after that the initial state is reached. The algorithm can be easily modified to satisfy this notion of completeness by continuing state exploration after the initial state is reached. This can be done by eliminating the termination condition at lines 6 and 7, and by modifying the return statement at line 12 as follows:

```
12.    if I ⊆ NextStates
13.        then return Plan;
14.        else return Fail;
```

- Consider now *optimality*. We can define optimality from the initial state, or more in general, from all the states in the plan. Intutively, a plan is optimal if for each state in the plan, each sequence of states, obtained by traversing the planning domain according to the plan, is of minimal length. PLAN is optimal in this general sense. This follows from the fact that the search is breadth-first.
- Finally consider the combination of *optimality* and the third notion of *completeness*: "the planning algorithm returns *all* the possible optimal situated plans". PLAN satisfies this property. Indeed, at each iteration, ONESTEPPLAN accumulates all the state-action pairs which lead to the subgoal. As a consequence of this property, given a set of state-action pairs returned by PLAN, the planner can select one of the many possible plans for execution, and can switch freely from one plan to another during execution.

5 Non-determinism

Several realistic applications need non-deterministic models of the world. Actions are modeled as having possible multiple outcomes, and the initial situation is partially specified. Non-deterministic planning domains and non-deterministic planning problems are obtained from definitions given in Section 3 by extending them with transition relations and sets of initial states.

A *planning domain* D is a 4-tuple $\langle F, S, A, R\rangle$ where

1. F is a finite set of *fluents*,
2. $S \subseteq 2^F$ is a finite set of *states*,
3. A is a finite set of *actions*,
4. $R \subseteq S \times A \times S$ is a *transition relation*.

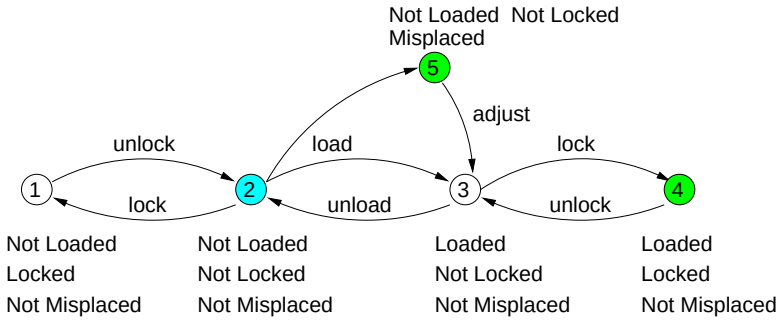

Fig. 5. An example of Non-deterministic Planning Domain

In Figure 5 we depict a simple non-deterministic planning domain. The action *load* can have two effects: either the item is correctly loaded, or it is misplaced in a wrong position. An action *adjust* can reposition the item.

A *planning problem* P for a planning domain $D = \langle F, S, A, R\rangle$ is a 3-tuple $\langle D, I, G\rangle$, where $I \subseteq S$ is the set of initial states, and $G \subseteq S$ is the set of goal states.

Along the lines described in Section 3, we can provide different specifications of plans as sets of state-action pairs for the non-deterministic case. However, in non-deterministic planning problems, we need to distinguish different kinds of solutions. Consider, for instance, the plan $\{\langle 2, \mathit{load}\rangle, \langle 3, \mathit{lock}\rangle\}$ in the planning domain in Figure 5. In general, we cannot say whether this plan does or does not achieve the goal state 4. It depends on the outcome of the non-deterministic action *load*. The plan *may* achieve the goal, and this is the case if *load* leads to state 3, or may not achieve the goal, in the case the outcome is state 5. Consider now the plan $\{\langle 2, \mathit{load}\rangle, \langle 3, \mathit{lock}\rangle, \langle 5, \mathit{adjust}\rangle\}$. It does achieve the goal, independently of the outcome of *load*. This plan specifies a conditional behaviour, of the kind "load; if load fails, then adjust; lock". We distinguish therefore between plans which may achieve the goal and those which are guaranteed to do so.

> A *weak (strong) plan* is a set of state-action pairs which may (is guaranteed to) achieve the goal.

The distinction between weak and strong plans was first introduced in [7]. The algorithm PLAN (Figure 4), if applied to non-deterministic planning problems, finds weak plans. A Planning as Model Checking algorithm for weak plans (searching the state space forward rather than backward) was first presented in [4]. Planning as Model Checking for strong plans was first solved in [7], where the notion of plan as set of state-action pairs was first introduced. The algorithm PLAN is essentially obtained from the algorithm for strong planning proposed in [7], simply replacing its basic routine STRONGPREIMAGE with ONESTEPPLAN.

$$\begin{aligned}&\textsc{StrongPreImage}(\mathit{States}, D) = \\ &\quad \{\langle s, a\rangle : s \in S, a \in A, \forall s'.\ (s' \in \mathit{States} \wedge R(s, a, s'))\}\end{aligned} \tag{9}$$

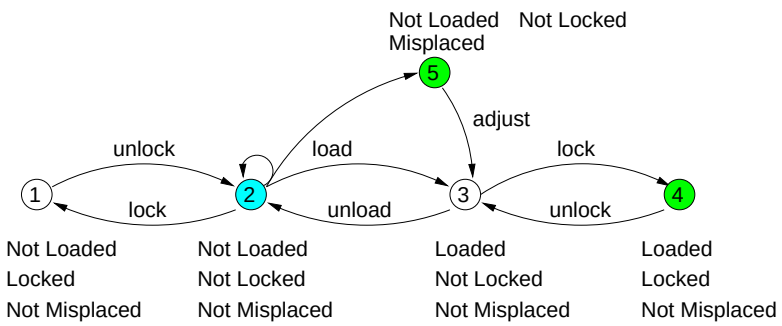

Fig. 6. An example of a domain for strong cyclic planning

The only difference between OneStepPlan (see (4)) and StrongPreImage (see (9)) is that the existential quantifier "$\exists s' \ldots$" (looking for weak plans) is replaced with the universal quantifier "$\forall s' \ldots$" (looking for strong plans)[1].

In [5], the authors exploit the Planning as Model Checking paradigm in order to generate strong plans as conformant plans, i.e., sequences of actions, rather than sets of state-action pairs.

Consider now the example in Figure 6. The action *load* has three possible outcomes: either it loads the item correctly, or it misplaces it, or it leaves the item unloaded (e.g., the robot simply fails to pick-up the item). In such a domain, there is no strong plan. The plan $\{\langle 2, load\rangle, \langle 3, lock\rangle, \langle 5, adjust\rangle\}$ may in principle loop forever, leaving the situation in state 2. This plan specifies an iterative behaviour, of the kind "load until the item is either loaded or misplaced; if it is misplaced, then adjust; lock". Plans encoding iterative trial-and-error strategies, like "pick up a block until succeed", are the only acceptable solutions in several realistic domains, where a certain effect (e.g., action success) might never be guaranteed *a priori* before execution. The planner should generate iterative plans that, in spite of the fact that they may loop forever, have good properties.

> A *strong cyclic plan* is a plan whose executions always have a possibility of terminating and, when they do, they are guaranteed to achieve the goal.

The plan $\{\langle 2, load\rangle, \langle 3, lock\rangle, \langle 5, adjust\rangle\}$ is a strong cyclic plan for the goal state 4. Indeed, execution may loop forever in state 2, but it has always a possibility of terminating (*load* may non-deterministically lead to states 3 and 5), and, if it does, the plan achieves the goal both from state 3 and from state 5. The plan $\{\langle 2, load\rangle, \langle 3, lock\rangle\}$ is not a strong cyclic solution.

The problem of planning for strong cyclic solutions was first tackled in [6], where an efficient algorithm was proposed. The algorithm looks for strong plans,

[1] In the case of non-deterministic actions $s' = R(s, a)$ in (4) should be replaced with $R(s, a, s')$.

and, if it does not find one, iterates backward by applying OneStepPlan, and then by computing the strongly connected components of the set of states produced by OneStepPlan. A formal account for strong cyclic planning was first given in [10] where strong cyclic solutions are formalized as CTL specifications on the possible executions of plans. 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". More precisely, strong cyclic plans are plans whose executions satisfy the CTL formula $\mathbf{AGEF}\mathcal{G}$, where $\mathcal{G}$ is a propositional formula representing the set of goal states. Strong and weak plans are plans whose executions have to satisfy the CTL formulas $\mathbf{AF}\mathcal{G}$ and $\mathbf{EF}\mathcal{G}$, respectively.

6 Planning via Symbolic Model Checking

We now consider the problem of implementing Planning as Model Checking. The problem is that realistic planning domains result most often in large state spaces. With this problem in mind we have defined planning domains and problems which are strictly related to Kripke Structures and CTL specifications. This has allowed us to exploit all the work done within the Computer Science community in the area of *symbolic model checking* [2, 20], based on Ordered Binary Decision Diagrams (OBDD's) [1][2]. As a practical consequence, we have implemented the Model Based Planner (MBP), a planner built on top of NuSMV [3], a state of the art OBDD based symbolic model checker. MBP can deal efficiently with rather large size planning problems. For instance, it manages to find strong cyclic plans in non-deterministic domains with more than 10^7 states in a few minutes [6].

In the rest of this section, in order to keep the paper self contained, we review the idea of OBDD-based symbolic model checking. We show how it can be applied to the Planning as Model Checking approach. As a matter of presentation, we keep distinct the description of symbolic model checking (Section 6.1) from the description of its OBDD-based implementation (Section 6.2).

6.1 Symbolic Representation

The fundamental ideas of *Planning via Symbolic Model Checking* are the following:

1. The Planning Problem is represented symbolically: the sets of states and the transitions of the semantic model are represented symbolically by logical formulas.
2. Plans are represented symbolically as formulas.
3. Planning is performed by searching through *sets* of states, rather than single states, by evaluating the assignments verifying (falsifying) the corresponding logical formulas.

[2] Various alternatives of the notions used in this paper have been provided. For instance, models can be formalized as ω-automata [27] and another common temporal logic is Linear Time Temporal Logic (LTL) [12]

Let $D = \langle F, S, A, R \rangle$ be a Planning Domain. Its symbolic representation is a boolean formula. We construct the boolean formula corresponding to D as follows. We associate to each fluent in F a boolean variable. Let $\underline{x} = x_1, \ldots, x_n$ be a vector of n distinct boolean variables, where each x_i corresponds to a distinct fluent in F (let n be the cardinality of F). S and each subset Q of S can be represented by a boolean formula in the variables $\underline{x}$, that we write as $S(\underline{x})$ and $Q(\underline{x})$, respectively.

Consider the example in Figure 3. $\underline{x}$ is *Loaded*, *Locked*. Since $S = 2^F$, $S(\underline{x}) = \top$. State 1 ($\{\neg$ *Loaded*, *Locked*$\}$) is represented by $\neg$*Loaded* $\wedge$ *Locked* and $Q = \{3, 4\} = (\{$*Loaded*, $\neg$*Locked*$\}, \{$*Loaded*, *Locked*$\}\})$ by the formula $Q(\underline{x}) =$ *Loaded*.

We associate to each action in A a boolean variable. Let $\underline{a} = a_1, \ldots, a_m$ be a vector of m distinct boolean variables (also distinct from each x_i), where each a_i corresponds to a distinct action in A (let m be the cardinality of A). For simplicity, in the following, we restrict ourselves to the case where formulas in the m boolean variables $\underline{a}$ can be redefined as formulas in the variable *Act*, which can be assigned m distinct values $a_1, \ldots a_m$. A boolean encoding of a formula in *Act* into a boolean formula in $a_1, \ldots, a_m$ can be easily defined: e.g., $Act = a_1 \leftrightarrow a_1 \wedge \neg a_2 \wedge \ldots \neg a_m$. In order to keep the notation simple, from now on, when we write a_1 in a boolean formula we mean $a_1 \wedge \neg a_2 \wedge \ldots \neg a_m$, and similarly for the other a_i's.

The transition function R is represented symbolically by a formula $R(\underline{x}, \underline{a}, \underline{x}')$, where $\underline{x}' = x'_1, \ldots, x'_n$ is a vector of n distinct boolean variables (also distinct from each x_i and a_i). Intuitively, $\underline{x}'$ is used to represent the value of fluents after the execution of an action. For instance, let R be the transition from state 1 to state 2 caused by *unlock* in Figure 3. Then

$$R = (\{\neg \textit{Loaded}, \textit{Locked}\}, \textit{unlock}, \{\neg \textit{Loaded}, \neg \textit{Locked}\})$$

and

$$R(\underline{x}, \underline{a}, \underline{x}') = (\neg \textit{Loaded} \wedge \textit{Locked} \wedge \textit{unlock}) \supset \neg \textit{Loaded}' \wedge \neg \textit{Locked}'$$

The symbolic representation of the Planning Domain in Figure 3 is the following.

1. $\underline{x} = \{\textit{Loaded}, \textit{Locked}\}$
2. $\underline{a} = \{\textit{lock}, \textit{unlock}, \textit{load}, \textit{unload}, \textit{wait}\}$
3. $\underline{x}' = \{\textit{Loaded}', \textit{Locked}'\}$
4. $R(\underline{x}, \underline{a}, \underline{x}) = ((\neg \textit{Loaded} \wedge \textit{Locked} \wedge \textit{unlock}) \supset \neg \textit{Loaded}' \wedge \neg \textit{Locked}') \wedge$
   $((\neg \textit{Loaded} \wedge \neg \textit{Locked} \wedge \textit{lock}) \supset \neg \textit{Loaded}' \wedge \textit{Locked}') \wedge$
   $((\neg \textit{Loaded} \wedge \neg \textit{Locked} \wedge \textit{wait}) \supset \neg \textit{Loaded}' \wedge \neg \textit{Locked}') \wedge$
   $((\neg \textit{Loaded} \wedge \neg \textit{Locked} \wedge \textit{load}) \supset \textit{Loaded}' \wedge \neg \textit{Locked}') \wedge$
   $((\textit{Loaded} \wedge \neg \textit{Locked} \wedge \textit{unload}) \supset \neg \textit{Loaded}' \wedge \neg \textit{Locked}') \wedge$
   $((\textit{Loaded} \wedge \neg \textit{Locked} \wedge \textit{lock}) \supset \textit{Loaded}' \wedge \textit{Locked}') \wedge$
   $((\textit{Loaded} \wedge \textit{Locked} \wedge \textit{unlock}) \supset \textit{Loaded}' \wedge \neg \textit{Locked}')$

Intuitively, symbolic representations of sets of states and transitions can be very compact: the number of variables in a formula does not depend in general

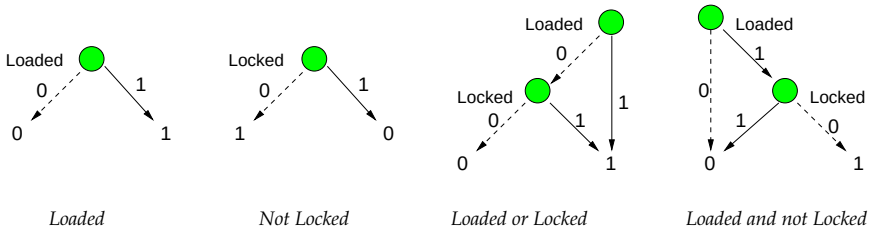

Fig. 7. OBDD's for *Loaded*, ¬ *Locked*, *Loaded* ∨ *Locked*, and *Loaded* ∧ ¬ *Locked*

on the number of states or transitions the formula represents. Given a Planning Domain with e.g., 10^6 states in S, where the fluent *Loaded* is true in a subset Q of e.g., 5×10^5 states, S is represented by the formula $\top$, Q by *Loaded*, and the empty set by $\bot$.

A symbolic representation of a Planning Problem $P = \langle D, I, G \rangle$ is obtained from the symbolic representation of the Planning Domain D, and from the boolean formulas $I(\underline{x})$ and $G(\underline{x})$. A symbolic plan for a symbolic planning domain D is any formula $\phi(\underline{x}, \underline{a})$. For instance, the symbolic plan for the situated plan {(2,*load*),(3,*lock*)} is $(\neg Loaded \wedge \neg Locked \supset load) \wedge (Loaded \wedge \neg Locked \supset lock)$.

Consider now the algorithm PLAN in Figure 4. Notice that it explores sets of states, rather than single states. The formula representing ONESTEPPLAN, can be written as a Quantified Boolean Formula (QBF).

$$\exists \underline{x'}(States(\underline{x'}) \wedge R(\underline{x}, \underline{a}, \underline{x'}))) : \tag{10}$$

An equivalent propositional boolean formula can be easily obtained. For instance, $\exists y.\phi(x, y)$ can be rewritten to $\phi(x, \top) \vee \phi(x, \bot)$.

6.2 OBDD-based implementation

Planning via Symbolic Model Checking can still be implemented in different ways. A technique which has been successfully applied in the area of formal verification is that known as *Symbolic Model Checking* [2, 20], which makes use of Ordered Binary Decision Diagrams (OBDD's) [1]. OBDD's are a compact representation of the assignments satisfying (and falsifying) a given boolean formula. Binary Decision Diagrams (BDD's) are rooted, directed, binary, acyclic graphs with one or two terminal nodes labeled 1 or 0 (for truth and falsity, respectively). A node in a BDD corresponds to a variable in the corresponding boolean formula. The two out-coming arcs from a BDD node represent the assignments of the corresponding variable to true (the arc is labeled with 1) and false (labeled with 0). OBDD's are BDD's with a fixed linear ordering on the propositional variables, which results in a corresponding order of the BDD nodes. Given a variable ordering, OBDD's are a canonical representation for boolean formulas. Figure 7 depicts the OBDD's for the formulas *Loaded*, ¬ *Locked*, *Loaded* ∨ *Locked*, and

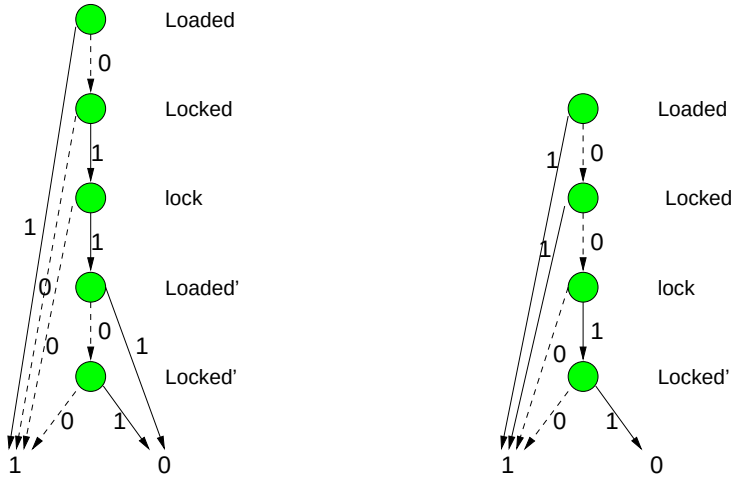

Fig. 8. OBDD's for the formulas (11) and (12)

Loaded $\wedge \neg$ *Locked*, where the variable ordering is *Loaded* $\prec$ *Locked*. Figure 8 gives the OBDD's encoding the formulas

$$(\neg Loaded \wedge Locked \wedge lock) \supset \neg Loaded' \wedge \neg Locked' \tag{11}$$

$$(\neg Loaded \wedge \neg Locked \wedge lock) \supset \neg Locked' \tag{12}$$

where the variable ordering is $Loaded \prec Locked \prec lock \prec Loaded' \prec Locked'$.

Operations on two sets S_1 and S_2, e.g., the union $S_1 \cup S_2$ and the intersection $S_1 \cap S_2$, can be viewed as composing the corresponding formulas with corresponding connectives, e.g., $S_1(\underline{x}) \vee S_2(\underline{x})$ and $S_1(\underline{x}) \wedge S_2(\underline{x})$, and as a consequence, as operations on the corresponding OBDD's. For instance, in Figure 7, the OBDD for the formula *Loaded* $\wedge \neg$*Locked* (the rightmost one in the figure) is obtained from the the OBDD's for the formulas *Loaded* and $\neg$*Locked* (the two leftmost ones) by simply replacing the terminal node labeled "1" of the OBDD for *Loaded* with the OBDD for $\neg$*Locked*.

OBDD's retain the advantage of a symbolic representation: their size (the number of nodes) does not necessarily depend on the actual number of assignments (each representing, e.g., a state or a state transition). Furthermore, OBDD's provide an efficient implementation of the operations needed for manipulating sets of states and transitions, e.g., union, projection and intersection.

7 Related Work

The idea of a model-theoretic approach to planning is along the lines proposed in [17], which argues in favor of a model-theoretic approach to knowledge representation and reasoning in Artificial Intelligence.

Planning as Model Checking is a major conceptual shift w.r.t. most of the research done in planning so far, like STRIPS-like planning (see, e.g., [13, 21]) and deductive planning (see, e.g., [24, 25]). The framework is much more expressive than STRIPS-like planning, and is closer to the expressiveness of deductive planning frameworks.

The Planning as Model Checking approach has provided the possibility to tackle and solve planning problems which have never been solved so far, like strong planning [7] and strong cyclic planning [6, 10] in non-deterministic domains. The experimental results reported in [7, 6, 5] show that Planning as Model Checking can be implemented efficiently.

Planning as propositional satisfiability [19, 18] is conceptually similar to Planning as Model Checking (even if technically different), since it is based on the idea that planning should be done by checking semantically the truth of a formula. The framework of planning as propositional satisfiability has been limited so far to deterministic classical problems.

The work in [22] exploits the idea of Planning as Model Checking presented in [4, 7, 6] to build an OBDD-based planner. [11] proposes a framework similar to ours, which is based on an automata-theoretic approach to planning for LTL specifications. In [11] there is no notion of situated plans.

8 Conclusions and Acknowledgements

The goal of this paper has been to provide an introduction, with various elements of novelty, to the Planning as Model Checking paradigm. Various papers on this topic have been published – see the references. The three papers which introduce the two key intuitions are: [4] that first introduces the idea of seeing planning as a semantic problem, and [7, 6] that first introduce the idea of seeing a plan as a set of state-action pairs. The first idea was envisaged by the first author of this paper as a follow on of the work, described in [16], where he first introduced and analyzed the graph of the states (the Kripke Structure) of a planning problem. The second idea was envisaged by the authors of [7, 6] as an effective and practical way to represent conditional and iterative plans. The ideas reported in Sections 3 and 4 had never been written before. Many people in our group have given substantial contributions to the development of the approach, the most noticeable are: Alessandro Cimatti, Marco Roveri, Enrico Giunchiglia and Marco Daniele.

References

1. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
2. 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, June 1992.

3. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.
4. A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. In S. Steel and R. Alami, editors, *Proceeding of the Fourth European Conference on Planning*, number 1348 in Lecture Notes in Artificial Intelligence, pages 130–142, Toulouse, France, September 1997. Springer-Verlag. Also ITC-IRST Technical Report 9705-02, ITC-IRST Trento, Italy.
5. A. Cimatti and M. Roveri. Conformant Planning via Model Checking. In Susanne Biundo, editor, *Proceeding of the Fifth European Conference on Planning*, Lecture Notes in Artificial Intelligence, Durham, United Kingdom, September 1999. Springer-Verlag.
6. A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Proceeding of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, Madison, Wisconsin, 1998. AAAI-Press. Also IRST-Technical Report 9801-10, Trento, Italy.
7. A. Cimatti, M. Roveri, and P. Traverso. Strong Planning in Non-Deterministic Domains via Model Checking. In *Proceeding of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, Carnegie Mellon University, Pittsburgh, USA, June 1998. AAAI-Press.
8. E. Clarke, O. Grumberg, and D. Long. Model Checking. In *Proceedings of the International Summer School on Deductive Program Design*, Marktoberdorf, Germany, 1994.
9. E. M. Clarke and O. Grumberg. Research in automatic verification and finite-state concurrent systems. *Annual Review of Computer Science*, 2(1):269–289, 1987.
10. M. Daniele, P. Traverso, and M. Y. Vardi. Strong Cyclic Planning Revisited. In Susanne Biundo, editor, *Proceeding of the Fifth European Conference on Planning*, Lecture Notes in Artificial Intelligence, Durham, United Kingdom, September 1999. Springer-Verlag.
11. G. de Giacomo and M.Y. Vardi. Automata-theoretic approach to planning with temporally extended goals. In Susanne Biundo, editor, *Proceeding of the Fifth European Conference on Planning*, Lecture Notes in Artificial Intelligence, Durham, United Kingdom, September 1999. Springer-Verlag.
12. 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.
13. 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.
14. M. Georgeff. An embedded reasoning and planning system. In J. Tenenberg, J. Weber, and J. Allen, editors, *Proc. from the Rochester Planning Workshop: from Formal Systems to Practical Systems*, pages 105–128, Rochester, 1989.
15. M. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proc. of the 6th National Conference on Artificial Intelligence*, pages 677–682, Seattle, WA, USA, 1987.
16. F. Giunchiglia. Abstrips abstraction – Where do we stand? Technical Report 9607-10, ITC-IRST, Trento, Italy, July 1996. To appear 1999 in the Artificial Intelligence Review.

17. J. Y. Halpern and M. Y. Vardi. Model Checking vs. Theorem Proving: A Manifesto. In J. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representations and Reasoning: Proceedings fo the Second International Conference*, pages 325–334, 1991.
18. H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the AIPS-98 Workshop on Planning as Combinatorial Search*, 1998.
19. Henry Kautz and Bart Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proc. AAAI-96*, pages 1194–1201, 1996.
20. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.
21. J. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. of KR-92*, 1992.
22. R. Jensen and M. Veloso. Obdd-based universal planning: Specifying and solving planning problems for synchronized agents in non-deterministic domains. Technical report, CMU, Carnegie Mellon University, USA, 1999.
23. M. J. Schoppers. Universal plans for Reactive Robots in Unpredictable Environments. In *Proc. of the 10th International Joint Conference on Artificial Intelligence*, pages 1039–1046, 1987.
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. P. Traverso and L. Spalazzi. A Logic for Acting, Sensing and Planning. In *Proc. of the 14th International Joint Conference on Artificial Intelligence*, 1995. Also IRST-Technical Report 9501-03, IRST, Trento, Italy.
27. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of LICS86*, pages 332–344, 1986.