SimPlan. A Simulation Based Planner for Reactive Agents

Froduald Kabanza
Dept. de math-info
Universite de Sherbrooke
Sherbrooke, Qubec, J1K 2R1
Canada
kabanza@dmi.usherb.ca

Abstract

Planning is the activity of generating plans, that is, finding combinations of actions to be executed in the future in order to achieve a desired behavior. SimPlan is a general-purpose planning system geared towards reactive systems. It can deal with cyclic executions and uncontrollable environment's events. One key contribution of SimPlan to the planning problem is the use of *lookahead behaviors* that convey strategic planning knowledge in cyclic plans. Another key contribution is the possibility of expressing the quality of desired plans as a function of the probability of failing, the cost of executing individual actions, and the desire to achieve a goal behavior. These two contributions are made possible by a uniform integration of Markov decision planning and model checking.

1 What is SimPlan ?

Programming a system to do something is to certain extent a decision-making problem, that is, the problem of deciding the instructions to be executed at different points. These decisions can be made at different levels of abstraction, with the lowest level corresponding to primitive units of computations and higher levels corresponding to combinations of primitive units of computations. In the programming theory, a primitive unit of computation is called an *instruction* and combinations of instructions are called *programs*. In the planning theory, a primitive unit of computation is called an *action* and combinations of actions are called *plans*. Planning is the activity of generating plans, that is, finding combinations of actions to be executed in the future in order to achieve a desired behavior.

SimPlan is a software for planning rules that specify actions to be executed by a system at different points in time, depending on the current situation, to accomplish a given task. SimPlan is based on the simulation of system's behaviors, and this is why we call it a simulation-based planner, or SimPlan for short.

Given a specification of actions executable by a system and a goal that must be achieved by the system, SimPlan simulates these actions, and derives a plan from a comparison of simulation traces with the goal. The comparison is done by verifying that the trace satisfies the goal, that is, by checking that the trace is a model of the goal. Hence, SimPlan uses a model-checking technique. The main advantage with this approach is its simplicity. The simulation process is easily implemented by a forward-chaining search engine that looks ahead in the space of possible executions to determine points where specific actions must be enabled or disabled. Since the search engine explores the actual state space of the system (i.e., a state in the search space corresponds to a system state), it makes it easy to specify goals and to verify that traces satisfy goals. The major drawback with this approach is the state explosion, but a part of it can be mastered by using *lookahead behaviors*.

SimPlan is a general-purpose planning tool in the sense that it can be applied to different domains. For instance, it is currently being applied in a mobile robot domain. Figure 1 shows a snapshot of two mobile robots during an experiment consisting of finding objects and delivering them to designated locations. To achieve such goals, several control processes are involved at different levels of abstraction, to avoid obstacle, to move along selected corridors, to move towards a particular goal point, to grasp objects or release them. SimPlan can compute plans containing rules that will coordinate these basic control processes to prevent bad things from happening (e.g., preventing a robot from engaging in stairways or staying for a certain period in the same region) while making certain good things happen (e.g., making a robot eventually reaching a desired location).

Figure 1: Mobile robots

The remainder of this paper is structured as follows. In the following section, we introduce the notion of behaviors and other related terminology. This is followed by the definitions of languages used to specify behaviors. We then dis-

cuss the notion of plan quality and how it relates to goals. This leads us to the planning method. Finally, we conclude with a discussion on related work.

2 States, Behaviors and Plans

The execution of one action may change the value one or more control variables. For instance, a move action change the robot's *position* and *translational velocity* control variables, while a turn action affects the robot's *angular velocity* control variable. A system *state* is the specification of values for all control variables at a given instant time. Some control variables may be related to an interface between the system and its environment. For example, a robot state may include the current robot's position, its speed, positions of immediate obstacles and requests from users. A state is changed by the execution of an action by the system or its environment.

An *execution sequence* is a sequence of states the system goes through when interacting with its environment. Not all computations are made to terminate. We often have open-loop reactive programs whose role is to respond to incoming events. Mobile robots, for example, involve obstacle avoidance processes that continually monitor objects in front of the robot to avoid them. Without loss of generality, we assume that an execution sequence is terminated by a cycle. Cycles account for open-loop programs and implement non-terminating behaviors. Truly terminating behaviors will be represented by equivalent execution sequences obtained by adding a cycle on terminal states using a "wait action".

A *primitive behavior* is the set of all possible execution sequences the system may ever be engaged in for all imaginable tasks. A *goal behavior* is a set of execution sequences that the system is allowed to go through, depending on the task you wish the system to accomplish. A goal behavior is always a subset of the primitive behavior. For example if we want a robot to deliver a package to some place, then we will specify that request as a goal. In this case, the goal behavior is the set of all execution sequences that contain a state in which the package's position coincides with the desired place. A *lookahead behavior* is a set of execution sequences that are relevant to a goal. This is behavior is superset of a goal behavior. It is not intended to be a behavior of the system itself, but a behavior for SimPlan. Since SimPlan computes a plan by simulating the primitive behavior and verifying the goal over traces of simulated execution sequences, to make this efficient, it helps to provide guidelines along which the simulation should be conducted. A lookahead behavior specifies those guidelines. When SimPlan is a given a lookahead behavior, it only simulates execution sequences that are consistent with it. For example, consider again a mobile robot that needs a plan for delivering an object to a particular location. One useful lookahead behavior states that:

> *when the goal is to bring some object o from a room x to a room y, then it is irrelevant to simulate behaviors in which the robot would be grasping any other object than o.*

When searching for plan for moving objects from one place to another, SimPlan won't simulate execution sequences that are inconsistent with the above lookahead behavior.

A *plan* is a mapping from states to actions that implements a set of decisions made in different states about the action to execute. Since the execution of a single action changes a state, the execution of a plan causes execution sequences. A *plan behavior* is the set of execution sequences a system may go through when controlled by the rules in the plan.

3 Languages for Primitive Behaviors

The system for which you want to generate plans may interact with other external processes. In this case, we say that the other processes are part of the environment. The system may influence the environment's actions, but in general it does not control them. For instance, a robot may influence our own moves – we tend to avoid it – but it does not control them. Thus, when specifying the primitive behavior, we must take into account not only the actions of the system but also its environment's actions.

Primitive behaviors are represented by a *Markov decision process*. This is a *stochastic automaton*, whose actions are associated with rewards that expressed their desirability degree within plans.[1] A *stochastic automaton* in turn is a graph of states, with nondeterministic state transitions representing actions of the system and associated with corresponding state transition probabilities. Environment's actions are not explicitly represented, but are implicitly conveyed by nondeterministic transitions.

Figure 2 shows an example of an nondeterministic graph representing a system that has four possible states. In each state the system executes exactly one action among those enabled (i.e., having corresponding outgoing transitions) to reach another state. For example, if the system is currently in state s_0, it may execute either action w_1 or action a_4 . Action w_1 leaves the state unchanged (it does not affect any control variable). Action a_4 moves the system either to state s_1 or to state s_2. When simulating the system behavior, it is impossible to determine which of these outcomes will materialize. But during a real execution, we assume that the system has appropriate sensors and reckoning tools to determine its current state, and hence the state resulting from the execution of its own actions. Nondeterministic action outcomes are mutually exclusive, but they must represent all possibilities. By adding state transition probabilities, we obtain a stochastic automaton. For instance, we may state that action a_4 has probability 0.5 of moving to state s_1 and probability 0.5 of moving to s_2. Since action outcomes represent all possibilities, the sum of transition probabilities corresponding to an action in a state must be equal to 1. By adding action rewards to the stochastic automaton we obtain a Markov decision process.

We assume that the system executes exactly one action in every state. Since "waiting" is also considered as an action, we can still model the possibility of doing nothing by putting "wait" action on appropriate transitions. In states where there are several possible actions (this is the case for all states in Figure 2), the system will have to chose exactly one. Of course the choice must depend on the goal the system has to achieve. This is why the system needs a plan to make good choices.

The specification of a Markov decision process for primitive behaviors is done recursively by defining an *initial state* and a *state transition function*. The state transition function takes a state as input and returns the set of transitions possible from that state. More specifically, given a state, such a transition function returns a list of the form

(action-name duration reward successors)

[1] In Markov decision processes, rewards are usually associated to states [7, 5] and express the state desirability. In SimPlan, we found it more practical to associate them to actions, but this is a mere syntactic sugar.

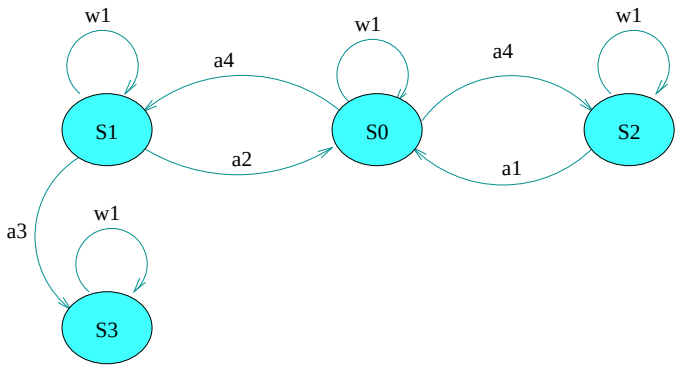

Figure 2: A nondeterministic transition system

where *successors* is in turn a list of pairs of the form *(state probability)*. Such a tuple means that in the current state, the action has the specified duration and reward and will lead to one of the specified successor with the corresponding probability.

4 Language for Goal and Lookahead Behaviors

In SimPlan, goals and lookahead behaviors are specified in a particular kind of modal temporal logic, called Metric Temporal Logic (MTL). A statement in the MTL language is called a formula. An MTL formula expresses a property of state changes for a system over time, that is, a temporal property. Hence, the meaning of MTL formulas is given in terms of sequences of states. One expresses a particular goal or lookahead behavior by writing an MTL formula. That way, the goal or lookahead behavior is the set of sequences of states satisfying the formula.

The basic elements of MTL language are predicates and connectives. We have two types of predicates : *state predicate* and *achieve meta-predicate*. A *state predicate* is an atomic property describing a fact true about the system state. An *achieve meta-predicate* is an atomic property describing a fact true of a goal.[2] With state predicates, we can make statements about the desirable system behavior (goals). With achieve meta-predicates, we can in addition make statements how goals should be satisfied (i.e., lookahead behaviors). Connectives are used to build more complex statements from predicates.

State Predicates

Syntactically, a state predicate is a list of symbols, where the first symbol is the *name* of the predicate and the other symbols are optional; when present, they represent objects about which the property named by the predicate holds. The symbols after the proposition name are also called its *arguments*. Arguments can be constants or variables. *Variables* are symbols starting with an "?", while *constants* are symbols starting with any alphabetic letter. Constants denote well defined objects related to the system state, while variables denote indefinite objects. A predicate containing only constant arguments is also called a *proposition*. There are two built-in state predicates: **true** (true in every states) and **false** (not true in any state). Other predicates are defined by the user by providing their state-based interpretation function.

[2]The achieve-meta predicate is an extension to the standard MTL logic [1]. It is a generalization of the GOALP predicate in TLPLAN [3].

Achieve Meta-Predicates

An *Achieve meta-predicate* is specified as a list of two elements. The first element must be the symbol `ach` and the second element must be a state predicate. Given a state predicate `p`, then `(ach p)` means that the system must eventually make `p` true. This is a statement about things that the system must do, that is, a statement about the current goal. Thus, achieve meta-predicates are only used in the specification of *lookahead behaviors* to specify strategies about how SimPlan should behave when searching for a plan that achieve a given goal.

First-order logic connectives

The traditional connectives for first-order logic are used to write complex formulas by combining predicates: `not` (negation), `and` (conjunction), `or` (disjunction), `implies` (implication), `forall` (universal quantifier) and `exists` (existential quantifier). Given two formulas `f1` and `f2`, the following examples illustrate the syntax used in SimPlan: `(not f1)`, `(and f1 f2)`, `(or f1 f2)`, `(implies f1 f2)`, where are formulas.

However, SimPlan uses quantification over a finite number of objects (i.e., bounded quantification). Syntactically, a quantified formula is a list of the quantification symbol, a sublist of variables, a description of their corresponding quantification domain and a formula. Typical `forall` and `exists` quantified formulas look like this: `(forall VARS BINDP f1)` and `(exists VARS BINDP f1)`. `VARS` is a list of variables, `BINDP` a predicate or meta-predicate describing the quantification domain and `f1` a subformula.

Roughly, `(forall VARS BINDP f1)` means that `f1` holds for all things in `VARS` such that `BINDP` is true. In other words, the scope of the quantification is bounded by the set of things in `VARS` that would make `BINDP` true. Similarly, `(exists VARS BINDP f)` means that `f` holds for something in `VARS` such that `BINDP` is true.

Temporal Connectives

A formula written by using the previous constructs is a first-order logic formula. It expresses properties true of a single state. For instance, the following formulas express state properties :

- `(in robot 1)`, i.e., the robot is in room `1`,
- `(and (in robot 1) (in a 2))`, i.e., the robot is in room `1` and object `a` in room `2`.
- `(forall (?x) (ach (in ?x 2)) (in ?x 1))`, i.e., all objects that must be in room `2` are currently in room `2`.

To express properties of behaviors we must add new connectives that relates to sequences of states. After all, behaviors are sets of execution sequences. Properties that hold of sequences of states are also called temporal properties, because they express properties having a temporal scope. We use the following temporal operators, with their intuitive meaning: `next`, `eventually`, `always`, and `until`.

To specify a formula with any of these temporal connective, one must also specify the corresponding temporal scope, that is, the time interval over which the property holds. Time interval are specified as time constraints in the

following way. Given an integer `n`, then we have the following time intervals:

- `(< n)`, i.e., the time interval starting from "now" and ending just before `n` time units have elapsed,
- `(<= n)`, i.e., the time interval starting from "now" and ending after `n` time units have elapsed,
- `(> n)`, i.e., the infinite time interval starting after `n` time units have elapsed, or
- `(>= n)`, i.e., the infinite time interval starting just before `n` time units have elapsed.

A temporal formula is specified as a list formed of a temporal connective, a time constraint, one or two subformulas depending on the temporal connective. Only the connective `until` takes two subformulas. The following examples illustrate the syntax:

- ```
  (always (>= 0)
    (implies
      (and (in a 1) (in robot 1))
      (next (>= 0) (holding robot a)))
  ```
  That is, each time both object a and the robot are in room 1, then in the next state the robot is holding object a.
- ```
   (always (>= 0)
    (forall (?x) (in a ?x)
      (implies
        (and (in robot ?a)
             (ach (holding robot a)))
        (until (in robot ?x)
               (holding robot a)))))
  ```
  That is, each time the robot and object a are in the same room and the robot has the goal of grasping the object, then the robot must stay there until grasping the object. This formula may be used to express a lookahead behavior.

5 Language for Specifying Plans

A *plan* is a list of *decision rules*. A decision rule in turn consists of an *id*, a *state*, and a *decision*, where

- *id* is a natural number that uniquely identifies the decision rule;
- *state* is a system state;
- *decision* is a list whose first element is an action's name and the remaining elements are decision rule's *id*s;

A decision-rule specifies an action to be executed in a state and the *ids* of decision rules matching the successor states that are expected from the execution of the action.

The role of a plan is to control a system in such a way to satisfy a given goal. The basic procedure for interpreting a plan is an open loop that alternates between *fetch-a-decision-rule* and *fire-the-corresponding-action* operations (Figure 3). At step 3.(a), some sensing may have to be involved to determine the current state. The fact that the procedure has no predefined terminating condition is consistent our assumption that reactive systems are generally made to react to external events with no predefined ending point. Of course, we may have a goal requiring a reactive system to make certain things and stop. For example, we may ask a robot reach a given location and that's it. In such a case, SimPlan produces a plan that ends in an idle state. The corresponding decision-rule makes the system cycle over the same state, performing a "wait" action.

```
Procedure Basic-Plan-Interpret()
1. get the current state of the system;
2. let the set of relevant decision rules be the set of
   rules matching the current system state
3. repeat until interrupted
  (a) get-a-decision-rule matching the current state from
      the set of relevant decision rules;
  (b) execute the action associated to the decision rule;
  (c) set the set of relevant decision rules to be the
      set of rules associated with the action;
```

Figure 3: Basic procedure for interpreting a plan

6 Plan Quality

The execution of plan controls the system by forcing it to chose particular actions at different execution points. But how can we evaluate the quality of the resulting behavior ? Does it satisfy the goal ? Is it optimal or at least efficient enough in terms of resources engaged by the execution of individual actions ? If the plan is approximate – for instance some decision rules specify the wrong actions or wrong successor states –, how can we measure the plan quality ?

For instance, assume that a robot's primitive behavior is specified in terms of *turn-left*, *turn-right* or *go-straight* actions. Given the goal of reaching a particular point and sufficient map knowledge, the robot must decide what action to execute at every point in order to reach the desired destination. At a given point, assume that a *turn-right* would lead the robot to its destination with 1 probability of success (because landmarks are very precise along that path), and with traveled distance of 100 meters. Still at the same point, assume that a *turn-left* also may lead the robot to its destination, but with 0.2 probability of erring in a wrong direction (the landmarks along that path are not precise enough), however with only a traveled distance of 20 meters. Should the robot turn left or turn right ?

Such a decision depends on how we prefer quick responses from the robot (traveling as fast as possible) over goal failures (the 0.2 probability of erring away of the destination). This preference is in part conveyed by reward associated to actions. The other part, as seen later, is conveyed by rewards associated to some specially marked states that characterize good execution sequences. Rewards convey knowledge about the plan quality. On the other hand, the requirement of reaching the goal location is a temporal constraint and will be conveyed by the temporal logic formula `(eventually (in robot x))`, where x is the desired location. That way, assuming move actions are rewarded an amount disproportionate to the traveled distance, say $1/d$ for a distance d, a *turn-left* leads to an expect reward of 0.04 (that is, $0.8 \times 1/20$), while a *turn-right* leads to an expected reward of 0.02 (that is, $1/50$). In this case, a plan generated by SimPlan will advice the robot to turn left, because this has a higher expected reward.

Definition 1 *The* value of a plan *in a state s is the expected amount of rewards that would be obtained by traversing endlessly the plan structure, starting from that state. More formally, let's note*

- *$V(P, s)$ the value of a plan P in state s; $P(s)$ the planned action matching s as given by the decision rules in P;*
- *$succ(s)$ the set of successors of s for the planned action $P(s)$;*
- *$GR(s, a)$ the* goal-based reward *of action a in state s,*

- $Pr(s, a, s')$ *the probability of reaching state* s' *by executing action* a *in state* s *(as given in the recursive state transition function definition), and*
- df *a real number greater or equal than* 0*, but strictly smaller than* 1 *(i.e.,* discounting factor *used to weigh the contribution of future rewards to the reward of the current state)*

The value $V(P, s)$ *of a plan* P *in a state* s *is recursively defined as follows:*

$$V(P, s) = R(s) + df \times \sum_{s' \in succ(s)} (Pr(s, P(s), a) \times R(s'))$$

The goal-based reward $GR(s, a)$ is a function of three things: a goal associated to state s, an *eventuality set* also associated to state s, and the reward for action a. The manner in which the goal and *eventuality sets* contribute to the reward of an action is explained later when pre-requisite concepts will have been introduced.

Let's note N the size of plan P, that is, its number of decision rules. The previous recursive definition of $V(P, s)$ defines N linear equations of N unknowns (the $V(P, s)$). Hence, one obtain $V(P, s)$ for all states s by simply solving these linear equations.

Definition 2 *A plan is* optimal *if there is no other plan that has a higher value in any state.*

7 Planning Method

SimPlan's planning method is a combination of simulation execution sequences specified by the primitive behavior and verification of the goal and lookahead behaviors over simulation traces, to derive satisfactory plans.

The verification of temporal logic formulas over state transition systems is done using algorithms described in [8]. The verification of safety requirements is done using the *formula progression algorithm*, while the verification of liveness requirements is done using the *eventuality progression algorithm*. We have slightly modified the definition of the formula progression algorithm to handle appropriately *achieve meta-predicates* and their related *goal worlds*.

Given a recursive state transition function for Markov decision process, SimPlan applies those algorithms to define a recursive definition of another Markov decision process that is obtained by composing the input transition function with a *formula progression function* and an *eventuality progression function*. We call the new function an *extended recursive state transition function*. The formula and eventuality progression algorithms are characterized by the following properties.

Property 1 *When a state has a goal or lookahead formula that is the proposition* `(false)`*, the sequences containing that state do not satisfy the goal (or lookahead) formula of the initial state.*

This property is the basis for checking safety requirements conveyed by goal formulas. It is also the foundation for controlling the simulation process by using lookahead behaviors. Since a lookahead formula expresses behaviors that are relevant for a given a goal, execution sequences that violate it are not expanded. In other words, every state that has the lookahead formula `(false)` is a dead end. States that have the goal `(false)` are also dead ends, but for a different reason: their corresponding sequence do not satisfy the goal.

Function $GR(s, a)$

- `if` s `has an empty eventuality set then return 1;`
- `if` s `has the goal (false), then return -1;`
- `otherwise, return 0.`

Figure 4: Default reward function

Function $GR(s, a)$

- `if` s `has an empty eventuality set then return` *`a value for good states`*`;`
- `if` s `has the goal (false), then return` *`a value for bad states`*`;`
- `otherwise, return the` *`reward for action a`* `as given by the recursive state transition function definition.`

Figure 5: A cost-sensitive reward function

Property 2 *The goal formula holds on every infinite state sequence unwound from an execution sequence whose cycle contains an empty eventuality set. On the other hand, for every infinite execution sequence that can be generated by a recursive transition function, there is a corresponding extended cyclic execution sequence whose cycle contains an empty eventuality set.*

These properties follow, respectively, from Theorem 11 and Theorem 12 in [8], but the proofs require additional arguments to account for the extensions made for managing achieve meta-predicates.

Property 2 tells us that an extended execution sequence is satisfactory if it contains a state with an empty eventuality set on its cycle. In other words, an infinite sequence generated by the extended state transition function will be satisfactory if a state with an empty eventuality set infinitely often on that sequence. On the other hand, a key characteristic of optimal plans is that they tend to drive the system most often through states with the highest rewards. Together, these two observation entail that if we reward execution sequences satisfying the goal according to Property 2, penalize execution sequences that violate the goal according to Property 1, and leave the other states with no rewards, we obtain a plan that tend to achieve the given temporal logic goal. This is the default rewarding schema in SimPlan. It is formally described by the function $GR(s, a)$ that returns the reward of action a in state s (see Figure 4)

SimPlan can also accept user-specified reward functions. This makes it possible to specify for example, cost-sensitive rewarding duration that make compromise between the satisfaction of temporal goal and the cost of executing actions. Such a scheme is given in Figure 5. In this function, the values for good or bad states are set depending on how the user want to trade the logical goal satisfaction over action reward optimization. In fact, the previous default rewarding schema is a special case in which the reward of actions is replaced by 0.

To sum up, the overall planning algorithm for SimPlan is sketched as follows:

- `The input consists of`
  - `an initial state,`
  - `a recursive state transition function (or a higher-level specification from which it can be obtained, such as` STRIPS `or` ADL `actions),`
  - `a goal formula, and`
  - `a lookahead formula.`

- The output is a plan.
- The algorithm sketch is
  1. Define an extended recursive state transition function from the input.
  2. Then, iteratively apply the extended state transition function to the initial state and descendants obtained thereof, except descendants having the goal (false) or the lookahead formula (false).
  3. This state expansion produces a Markov decision process whose state space coverage grows with the number of iterations.
  4. At different iteration steps, apply the *policy iteration algorithm* defined in [7, 5] to current Markov decision process, using a goal-rewarding function. This produces a plan that is optimal with respect to the current Markov decision process.
  5. Stop when the plan at the current iteration is satisfactory.

At step 1, no single state at all is generated. The definition of the extended recursive state transition function is simply a composition of functions. The state pruning effect of lookahead behaviors is made operational by not expanding states with lookahead formula `(false)`.

8 Implementation

SimPlan is written in Common Lisp. The site for downloading the system and other related documents is `http://www.dmi.usherb.ca/~kabanza/simplan`. The document SimPlan*'s User Manual* provides instructions on how to download SimPlan, install it and use it.[3] Below is a trace for a very simple artificial domain matching Figure 2.

```
SimPlan 1.0
Init state: s0
Goal: (eventually (p 3))
Mode: iemdp
LAS : (true)     (i.e., lookahead formula)
Reactive Plan:
[ID 0 state s0 ACTION (a4 1 2)]
[ID 1 state s2 ACTION (a1 0)]
[ID 2 state s1 ACTION (a3 3)]
[ID 3 state s3 ACTION (w1 4)]
[ID 4 state s3 ACTION ((w 1) 4)]
```

The goal is to reach state s_3, starting in s_0. This plan was obtained using the default reward function (which ignores individual action rewards), assuming 0.5 probability for each of the outcomes for action $a4$. It is interesting to note that there exists no correct plan for achieving this goal with probability 1 of success, because action a_4 has always the possibility of moving to state s_2. SimPlan yields the best plan for this goal, although partially correct – partially because the system has a non zero probability of cycling over $s_0 s_2$ endlessly.

9 Related Work

The combination of Markov decision planning and model checking has already been explored in [2]. They advocated a past temporal logic, but we use a future temporal logic. In a future temporal logic, statements refer to future executions of the system, while in a past temporal logic they refer to past executions. Past temporal statements are not appropriate for describing cyclic behaviors since, by definition, the past is a finite history and cannot involve cyclic execution sequences. On the other hand, since lookahead behaviors are aimed at avoiding unnecessary future state expansions, they are better expressed by future temporal logic formulas.

The reduction of the planning problem to a model-checking problem has also been discussed in [4]. This reduction is exploited to take advantage of efficient encoding techniques that were developed in the area of verification. However, the generation of plans that make a tradeoff between the logical notion of goal satisfaction, failure probability and cost of executing actions is not the objective pursued in [4] and hence this issue was not addressed. The management of strategic planning knowledge is not discussed either.

A similar observation can be made for [6]. This is an approach for reducing the planning problem to an automata-theoretic model checking approach. More specifically, the reduction is described assuming Büchi automata over infinite computations. However, this reduction does not take into account probabilistic knowledge, strategic planning knowledge, nor knowledge about the cost of executing actions.

[3] Only the source code is currently available, so it takes a Lisp interpreter to use it.

10 Acknowledgment

This research was supported in part by the Canadian *Natural Science and Engineering Research Council* (NSERC) and the Québec *Fond pour la formation des chercheurs et l'aide à la recherche* (FCAR).

References

[1] R. Alur and T. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1):35–77, 1993.

[2] F. Bacchus, C. Boutilier, and A. Grove. Rewarding behaviors. In *Proc. of 13th National Conference on Artificial Intelligence (AAAI 96)*, pages 1160–1167. AAAI Press/MIT Press, 1996.

[3] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[4] A. Cimatti and M. Roveri. Conformant planning via model checking. In *Proc. of European Conference on Planning (ECP 99)*. Lecture Notes in Artificial Intelligence. Springer Verlag., 1999.

[5] T. Dean, L. P. Kaelbling, J. Kerman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1–2):35–74, 1995.

[6] G. De Giacomo and M.Y. Vardi. Automata-theoretic approach to planning for temporally extended goals. In *Proc. of European Conference on Planning (ECP 99)*. Lecture Notes in Artificial Intelligence. Springer Verlag., 1999.

[7] R. A. Howard. *Dynamic Programming and Markov Processes*. The M.I.T. Press, 1960.

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