Synthesizing Plant Controllers Using Real-time Goals*

M. Barbeau, F. Kabanza, and R. St-Denis
Département de mathématiques et d'informatique
Université de Sherbrooke
Sherbrooke, Québec J1K 2R1 Canada
Email: {barbeau, kabanza, stdenis}@dmi.usherb.ca

Abstract

This paper introduces a novel planning method for reactive agents. Our planning method handles, in a single framework, issues from AI, control theory, and concurrency that have so far been considered apart. These issues are mostly controllability, safety, bounded liveness, and real time. Our approach is founded on the supervisory control theory and on Metric Temporal Logic (MTL). The highlights of our method consist of a new technique for incrementally checking MTL goal formulas over sequences of states generated by actions and a new method for backtracking during search by taking into account uncontrollable actions.

1 Introduction

A *reactive plan* is a program that specifies the actions to be executed by an agent that continuously reacts to the occurrence of discrete events from its environment. The automatic synthesis of reactive plans is a key component in the design of autonomous systems. It is of interest in AI planning [Georgeff and Lansky, 1987; Drummond and Bresina, 1990; Dean *et al.*, 1993], control theory [Ramadge and Wonham, 1989; Makungu *et al.*, 1994], and concurrency theory [Emerson, 1990]. In these areas, the terms *control rules*, *supervisor*, and *reactive programs* are respectively used to denote reactive plans. However, the paradigms used to generate such plans vary widely. In AI planning, the focus is on representing and reasoning about actions in an efficient and natural way. This has led researchers in this field to concentrate on heuristic search techniques. Unfortunately, little research in AI planning has taken into account the concepts of controllable actions and liveness goals.

In the area of control theory, the emphasis has been on reasoning about the issues of controllability and observability of actions [Ramadge and Wonham, 1989]. A key characteristic of models of actions in this field is that they distinguish between controllable and uncontrollable actions. This facilitates reasoning about controllability issues.

In concurrency, modal temporal logic has been recognized as a useful tool for dealing with liveness properties. For instance, real-time modal temporal logic can express bounded-time response properties of the form "each occurrence of event e leads to satisfaction of condition c within 5 time units" [Alur and Henzinger, 1993]. One interesting development in this area is *model checking* [Emerson, 1990]. In this approach, verifying a program is viewed as evaluating the truth of a temporal formula on a temporal structure representing the program.

In this paper, we describe a method for generating reactive plans. Our method borrows concepts and concerns from all the three areas outlined above. From AI planning, we take the concern of reasoning about actions and using heuristic search to generate plans. From control theory, we borrow the concept of controllable action. As in concurrency theory, we are also concerned with the representation and reasoning about real time, safety, and liveness goals.

To be more specific, our planner takes as input a goal, a description of the primitive actions of a reactive agent, which are *controllable*, and those of its environment, which are *uncontrollable*. The planner returns a reactive plan for achieving the goal. Note that both the agent and its environment are referred to as the *process* or *plant* in control theory. We generate plans essentially by searching among sequences of states that represent executions of primitive actions, checking whether they satisfy the goal or not. Reactive plans are more or less immediately extracted from satisfactory sequences. We describe goals using Metric Temporal Logic (MTL) formulas [Alur and Henzinger, 1993].

Recent research has investigated connections between AI planning, control, and concurrency. This includes work on representation of AI control programs [Nilsson, 1994], a recent book by Dean and Wellman on planning and control [Dean and Wellman, 1991], work on the integration of planning and concurrency ideas [Gode-

*This work was supported by the Canadian Government through its NSERC programs.

froid and Kabanza, 1991; Kabanza, 1990], and work on the synthesis of controllers in temporal logic frameworks [Thistle and Wonham, 1986; Fusaoka *et al.*, 1983]. We view our contribution to this research in three respects. First, our planner integrates concepts and concerns from the above three fields deeper than any existing comparable framework. Hence, besides its more general capabilities, our planner contributes to a better understanding of issues in the boundaries of these fields.

Second, we introduce a method for checking MTL goals over state sequences *on the fly* (i.e., incrementally). In contrast to other techniques for checking similar real-time formulas (e.g., [Alur and Henzinger, 1993; Alur *et al.*, 1993]), our technique does not require an explicit storage of the state-transition graph over which formulas are checked. Rather, our planner generates and checks state sequences *on the fly*. This allows us to generate plans using a standard forward-chaining search. It also lets our planner generate plan control rules incrementally. In this way, our planner can be considered as having *anytime* capability. According to Dean and Boddy [Dean and Body, 1988], an *anytime* algorithm is one that can be stopped at different stages of its processing and yield an approximate, but useful, result.

Third, we introduce a search technique, called *control-directed backtracking*. This technique uses selective backtracking to states from which the planner can expand controllable paths. We use an example to demonstrate that this approach can allow the planner to prune the search space significantly. This search technique is reminiscent of, but quite different from, the mechanisms used in *dependency-directed backtracking* [Stallman and Sussman, 1978] and partial-order search using Mazurkiewicz's *traces* [Godefroid and Kabanza, 1991].

Although our planning approach applies to standard MTL and more general logics, due to space limitations, we will describe it only for a restricted class of MTL formulas called *bounded-time MTL formulas (BMTL)*. Even with this restriction, our planner handles problems that are beyond the scope of comparable AI planning or control-theoretic frameworks. For example, as was mentioned above, our planner treats various types of bounded-time goals.

The remainder of this paper is organized in three parts, beginning with the description of our model of actions and plans. This is followed by a brief overview of MTL and a description of our planning method.

2 Actions and Plans

We adopt the following model of plans from Ramadge and Wonham[1] [Ramadge and Wonham, 1989]. A process is modeled as a spontaneous generator of sequences of actions. The set of actions $\mathcal{A}$ of the process are partitioned into two disjoint subsets $\mathcal{A}_u$ and $\mathcal{A}_c$. The actions in $\mathcal{A}_u$ are uncontrollable, while those in $\mathcal{A}_c$ are controllable. Hence, the evolution of the process can be controlled by prohibiting the occurrence of some controllable actions at certain points. A *control input* is a subset γ of $\mathcal{A}$ satisfying the condition $\mathcal{A}_u \subseteq \gamma$. If $a \in \gamma$, then a is permitted to occur. Let L denote the set of sequences of actions that can be generated by the process and $\Gamma \subseteq 2^{\mathcal{A}}$ represent the set of all control inputs. A *plan* is a map $f : L \to \Gamma$ yielding, for each sequence $w \in L$ of generated actions, the control input $f(w)$ to be applied to the process at that point. Language L_f denotes the sequences of actions generated by the process under the supervision of f. Let ϵ be the empty string, $a \in \mathcal{A}$, and $w \in \mathcal{A}^*$. The language L_f is formally defined as follows:

(1) $\epsilon \in L_f$ and
(2) $wa \in L_f$ iff $w \in L_f, a \in f(w)$, and $wa \in L$.

In our framework, the *process* consists of a reactive agent and its environment, both performing primitive actions (see Figure 1). The actions performed by the agent are controllable, whereas those performed by the environment are uncontrollable. We model the interaction between the agent and its environment by interleaving their actions.

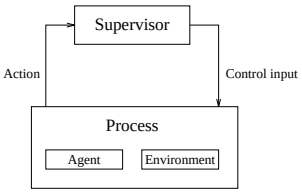


Figure 1: A model of control

Actions There are various models of actions in the fields of AI, concurrency, and control theory. Representations of actions in planning are mostly based on STRIPS-like models [Fikes and Nilsson, 1972]. Models of actions in concurrency and control theory are often based on state-transition machines such as automata or Petri nets. Nevertheless, all these models are more or less theoretically equivalent. For example, a state-transition model can be derived from a STRIPS-like formalism by generating the states reachable from one or several initial states. Hence, without loss of generality, we assume a STRIPS-like formalism in this paper.

The STRIPS model of actions includes two basic concepts, namely, world state and primitive action. A *world state* is defined as a set of propositions. Propositions model facts about states of the agent and its environment. Every primitive action is described by a *precondition* (a conjunction of propositions enabling the action), a *delete-list* (a list of propositions retracted by the execution of the action), and an *add-list* (a list of propositions asserted by the execution of the action). Note that, in

[1] As was mentioned in the introduction, we use the term "plan" as a synonym for supervisor.

general, variables are used to describe schemata of instances of actions.

For the sake of clarity, we use the following functions to manipulate actions. Let S denote a world state and a an action. The transition function $\delta(a, S)$ returns the successor world state of S after execution of action a. The function $Controllable(a)$ returns *true* if a is controllable, otherwise it returns *false*. The function $\delta(a, S)$ is defined only when a is enabled in S. When it is defined, the result of $\delta(a, S)$ is obtained by removing from S the propositions in the delete-list of a and then adding the propositions in the add-list of a.

Operational representation of a plan For convenience, we adopt the following automaton representation of a plan f. A plan is represented by a pair $\langle R, \varphi \rangle$, where $R = \langle Y, \mathcal{A}, \xi, y_0 \rangle$ is a finite-state automaton[2] and $\varphi : Y \rightarrow \Gamma$ is an output map yielding, for each state of R, the control input to be applied to the process at that point. A pair $\langle R, \varphi \rangle$ represents a plan f if for every $w \in L_f$, $\varphi(\xi(w, y_0)) = f(w)$. Intuitively, the value of f on the sequence of actions w is obtained by the application of w to R, causing R to move from its initial state y_0 to the state $y = \xi(w, y_0)$, and then calculating the control input $\varphi(y)$.

3 Metric Temporal Logic

In this section, we summarize the MTL formulas that we use to describe goals [Alur and Henzinger, 1993].

Syntax MTL formulas are constructed from an enumerable collection of propositional symbols, the boolean connectives $\wedge$ (and) and $\neg$ (not), and the temporal modalities $U_{\sim x}$ (until), $\Box_{\sim x}$ (always), where $\sim$ denotes $=$, $\leq$, or $\geq$, and x is an integer. The formula formation rules are (1) every propositional symbol p is a formula, and (2) if f_1 and f_2 are formulas, then so are $\neg f_1$, $f_1 \wedge f_2$, $\Box_{\sim x} f_1$, and $f_2 \, U_{\sim x} \, f_1$.[3] The following abbreviations are standard: $\Diamond_{\sim x} f \equiv true \, U_{\sim x} \, f$ (eventually f) and $f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$ (f_1 implies f_2).

Semantics MTL formulas are interpreted over models of the form $M = \langle S, \pi, \mathcal{T} \rangle$, where S is an infinite sequence of states $s_0, s_1, \ldots$; π is a function that takes a proposition and a state as input and returns *true* if the proposition holds in the state; and $\mathcal{T}$ is a function that associates a time stamp with each state. As usual, we write $\langle M, s \rangle \models f$ if state s in model M satisfies formula f. In addition to the standard rules for the boolean connectives, we have that, for a state s_i in a model M, a propositional symbol p, and formulas f_1 and f_2:

- $\langle M, s_i \rangle \models p$ *iff* $\pi(p, s_i) = true$;
- $\langle M, s_i \rangle \models \Box_{\sim x} f_1$ *iff* $\forall j$ such that $\mathcal{T}(s_j) \sim \mathcal{T}(s_i) + x$, $\langle M, s_j \rangle \models f_1$;
- $\langle M, s_i \rangle \models f_1 \, U_{\sim x} \, f_2$ *iff* $\exists j$ such that $\mathcal{T}(s_j) \sim \mathcal{T}(s_i) + x$, $\langle M, s_j \rangle \models f_2$, and $\langle M, s_k \rangle \models f_1 \ \forall k$, $i \leq k < j$.

Finally, we say that the model M (or sequence of states $S \in M$) satisfies a formula f if $\langle M, s_0 \rangle \models f$.

Bounded-time MTL Formulas As was mentioned in the introduction, we describe our planning algorithm only for *bounded-time MTL formulas*. These formulas are defined in terms of *normalized MTL formulas*. A *normalized MTL formula* is one in which only propositional symbols are negated.[4] A *bounded-time MTL formula* is a normalized MTL formula in which every *until* modality is of the form $U_{\leq x}$.

Example 3.1 The formula $\Diamond_{\leq 10} \Box_{\geq 0} p$ (i.e., p holds within 10 time units and remains true afterwards) expresses a classical goal requirement in AI planning and stability in control theory [Fusaoka *et al.*, 1983]. In Figure 7, the formula g_1 expresses the safety (mutual exclusion) requirement that $c(i)$ and $m(i)$, $1 \leq i \leq 4$, are never true simultaneously. The formula g_2 expresses the liveness requirement that $c(2)$ and $c(4)$ become simultaneously true infinitely often, at intervals shorter than 5 time units. ■

4 Planning Method

4.1 Global view

Given a set of primitive actions, an initial state, and a goal, our planner searchs for sequences of world states, starting from the initial state, and checking *on the fly* whether or not sequences of states expanded so far satisfy the goal. Reactive plans are extracted from satisfactory sequences. Our search process differs from a classical forward-chaining exploration in three respects. First, the check of MTL goals requires that we actually search in a space of pairs where each pair consists of a world state and an MTL formula, rather than simply a space of world states. Second, transitions corresponding to uncontrollable actions are subject to special treatment. Third, when we reach a sink, we backtrack taking into account the fact that some transitions are uncontrollable. As will be illustrated, our backtracking mechanism can lead to considerable savings compared to *blind* backtracking.

To check formulas over state sequences *on the fly*, we associate an MTL formula to each world state. The formula attached to a state must be satisfied by each sequence starting from this state. The planner uses a mechanism of *progressing* MTL subformulas through

[2] In this paper, all the states of automata are accepting.

[3] Note that the standard MTL also has a *congruence modulo constraint* and a *next* modality [Alur and Henzinger, 1993].

[4] For any MTL formula, we can obtain an equivalent normalized formula by using logical equivalences to propagate the negation symbol inwards.

world states to compute the formulas to be attached to states. Intuitively, progressing an MTL formula through a world state consists in checking that the *present* requirement conveyed by the formula is satisfied in this state. If so, a formula representing the *future requirement* is returned, otherwise *false* is returned. Hence, given a current world state and an action, the formula representing the future requirement is paired with the successor world state obtained by applying the action to the current world state.

4.2 Formula Progression

The formula progression algorithm is described in Figure 2. The function *Prog* accepts as input an MTL formula f, a world state s, a real number t, and a function $\pi(p, s)$, which returns *true* only if the proposition p holds in the world state s. It returns an MTL formula representing the progression of f through s. This algorithm is characterized by the following theorem.

function $Prog(f,s,t,\pi)$
1. **case** f
2. p (p a proposition): $\pi(p,s)$;
3. $\neg f_1$: $\neg Prog(f_1, s, t, \pi)$;
4. $f_1 \wedge f_2$: $Prog(f_1, s, t, \pi) \wedge Prog(f_2, s, t, \pi)$;
5. $f_1 \vee f_2$: $Prog(f_1, s, t, \pi) \vee Prog(f_2, s, t, \pi)$;
6. $\Box_{=x} f_1$: **if** $x - t > 0$ **then** $\Box_{=(x-t)} f_1$
7. **else** $Prog(f_1, s, t, \pi)$;
8. $\Box_{\leq x} f_1$: **if** $x - t > 0$ **then** $Prog(f_1, s, t, \pi) \wedge \Box_{\leq (x-t)} f_1$
9. **else** $Prog(f_1, s, t, \pi)$;
10. $\Box_{\geq x} f_1$: **if** $x - t > 0$ **then** $\Box_{\geq (x-t)} f_1$
11. **else** $Prog(f_1, s, t, \pi) \wedge \Box_{\geq 0} f_1$;
12. $f_1 U_{=x} f_2$:
13. **if** $x - t > 0$ **then** $Prog(f_1, s, t, \pi) \wedge f_1 U_{=(x-t)} f_2$
14. **else** $Prog(f_2, s, t, \pi)$;
15. $f_1 U_{\leq x} f_2$:
16. **if** $x - t > 0$ **then** $Prog(f_2, s, t, \pi) \vee$
17. $(Prog(f_1, s, t, \pi) \wedge f_1 U_{\leq (x-t)} f_2)$
18. **else** $Prog(f_2, s, t, \pi)$;

end

Figure 2: Progression algorithm for BMTL formulas.

Theorem 4.1 *Let $M = \langle S, \pi, \mathcal{T} \rangle$ be any MTL model and let s_i be the ith state in the sequence of states S. Then, we have for any MTL formula f, $\langle M, s_i \rangle \models f$ if and only if $\langle M, s_{i+1} \rangle \models Prog(f, s_{i+1}, \mathcal{T}(s_{i+1}) - \mathcal{T}(s_i), \pi)$.*

The time difference $\mathcal{T}(s_{i+1}) - \mathcal{T}(s_i)$ corresponds to the duration of the action that leads to s_{i+1} from s_i. For the sake of simplicity, we assume that each action takes 1 time unit.

4.3 The Planning Algorithm

We can now detail the algorithm of our planner. The input consists of an initial world state x_0, a set of primitive actions $\mathcal{A}$, the transition function δ, an MTL goal g, and the function π (see Figure 3). It outputs a plan $\langle R, \varphi \rangle$. For the sake of clarity, the planning algorithm is divided into three parts: the prologue (Figure 3), search process (Figure 4), and epilogue (Figure 5).

function *Planner*(**in** x_0, δ, $\mathcal{A}$, g, π, **out** $\langle R, \varphi \rangle$)
1. $f_0 \leftarrow Prog(g, x_0, 0, \pi)$;
2. **if** $f_0 = false$ **then return the empty solution**;
3. $[x_0, f_0].pct \leftarrow \{\}$; $[x_0, f_0].pre \leftarrow \{\}$;
4. $[x_0, f_0].status \leftarrow undefined$;
5. $[x_0, f_0].controllable \leftarrow false$;
6. CLOSED $\leftarrow \{\}$; OPEN $\leftarrow \{[x_0, f_0]\}$;

Figure 3: Prologue to the planning algorithm

7. **repeat select** $[x, f]$ **in** OPEN;
8. $[x, f].controllable \leftarrow true$;
9. CLOSED $\leftarrow$ CLOSED $\cup \{[x, f]\}$; OPEN $\leftarrow$ OPEN $\setminus \{[x, f]\}$;
10. **for each** a **in** $\mathcal{A}$ **such that** $\delta(a, x)$ **is** *defined* **do**
/* Expand the state $[x, f]$ */
11. $x' \leftarrow \delta(a, x)$; $f' \leftarrow Prog(f, x', 1, \pi)$;
12. **if** $f' = false$ **or** $([x', f'] \in$ CLOSED **and**
13. $[x', f'].status = sink)$
14. **then if not** $Controllable(a)$
/* $[x, f]$ becomes *sink* */
15. $[x, f].controllable \leftarrow false$; $Sink([x, f])$;
16. **break**;
17. **else**
18. $\xi(a, [x, f]) \leftarrow [x', f']$;
19. **if** $[x', f'] \notin$ CLOSED **and** $[x', f'] \notin$ OPEN **then**
/* Initialization of a new state */
20. $[x', f'].pct \leftarrow \{\}$; $[x', f'].pre \leftarrow \{\}$;
21. $[x, f].status \leftarrow undefined$;
22. $[x', f'].controllable \leftarrow false$;
23. OPEN $\leftarrow$ OPEN $\cup \{[x', f']\}$;
24. **if** $Controllable(a)$ **then**
25. $[x', f'].pct \leftarrow [x', f'].pct \cup \{[a, [x, f]]\}$;
26. **else**
27. $[x', f'].pre \leftarrow [x', f'].pre \cup \{[x, f]\}$;
28. $[x, f].controllable \leftarrow false$;
29. **if** $[x, f].controllable = true$ **then**
/* Check for a trivial satisfactory state */
30. **if not** $Has\_Eventually([x, f])$
31. **then** $Satisfactory([x, f])$;
32. **until** OPEN $= \{\}$;
/* Check for non trivial satisfactory states */
33. $Check\_Non\_Trivial\_Satisfactory\_States()$;

Figure 4: Search process of the planning algorithm

Prologue The prologue consists in initializing a search graph that implements an intermediate automaton $M = \langle X \times F, \mathcal{A}, \xi, [x_0, f_0] \rangle$, where F is a set of bounded-time MTL formulas derived from g using the formula progression algorithm. The set of states $X \times F$ is partitioned into expanded and unexpanded states. Initially, the set of expanded states, CLOSED, is empty; the set of unexpanded states, OPEN, contains the initial state $[x_0, f_0]$, where $f_0 = Prog(g, x_0, 0, \pi)$. Note that $Prog(g, x_0, 0, \pi)$ returns a formula equivalent to g, since the time difference used is 0. The purpose is to check that the *present* requirement of g is satisfied in x_0.

Overview of the Search Process The search process repeatedly selects a state $[x, f]$ from OPEN and proceeds as follows. For each action enabled in x, it

```
    /* Compute the reactive plan (supervisor) */
34. if [x0, f0].status = satisfactory
35. then R ← ⟨CLOSED_sat, A, ξ_sat, [x0, f0]⟩;
36.    else return the empty solution;
       /* Compute the feedback function φ */
37. for each [x, f] in CLOSED_sat do
38.    for each a in A do
39.       if δ(a, x) is defined and ξ_sat(a, [x, f]) is defined
40.       then φ([x, f]) ← φ([x, f]) ∪ {a};
end
```

Figure 5: Epilogue to the planning algorithm

computes a successor $[x', f']$, where $x' = \delta(a, x)$ and $f' = Prog(f, x', 1, \pi)$.

Epilogue The epilogue derives a plan $\langle R, \varphi \rangle$ from the intermediate automaton M as follows. First, the planner removes the states that are not *satisfactory* from M to obtain the automaton R (lines 34-35). Then φ is obtained by including in $\varphi(s)$ an action a for each transition (s, a, s') of R (lines 37-40). Hence, a crucial step in our planning process is to determine *satisfactory* states.

Example 4.1 Let us consider the maze example pictured in Figure 6 (taken from [Ramadge and Wonham, 1989]). The maze has five rooms. There are a cat and a mouse moving from one room to another through doorways. Every doorway is either for the exclusive passage of the cat $(c_1, \ldots, c_7)$ or the mouse $(m_1, \ldots, m_6)$. Doorway c_7 is bidirectional and uncontrollable, while others are unidirectional and controllable (i.e., they can be closed). The goal of this process consists of two requirements. First, the cat and the mouse must never occupy the same room simultaneously. Second, after leaving their initial rooms (room 2 for the cat and room 4 for the mouse), they must return within 5 time units. These requirements are specified by formula f_0 in Figure 7 (this formula is also commented on in Example 3.1).

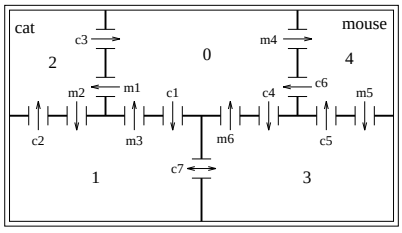


Figure 6: Maze for cat and mouse

Figure 7 details the expansion of the search graph (i.e., automaton M) for this example. Every state of the graph is composed of a world state, which is a pair consisting of the cat and mouse positions, and an MTL formula. For clarity in the figure, we only indicate formulas for some states of the graph. The initial state is $[(2, 4), f_0]$. The formulas $f_i, 1 \leq i \leq 5$, are defined in Figure 7. A proposition $c(i)$ $(m(i))$ means that the cat (mouse) is in room i. The initial formula f_0 is the result of progressing the goal $g_1 \wedge g_2$ through the world state $(2, 4)$ with a time difference of 0. States are expanded by applying actions as explained above. For instance, $[(0, 4), f_1]$ is produced by applying action c_3 to obtain the world state $(0, 4)$ and by progressing f_0 through $(0, 4)$, using a time difference of 1, to get f_1. Note that the eventually modality in f_1 is bounded by 4, which conveys the fact that one unit of time has elapsed. The expansion of $[(0, 4), f_1]$ into $[(1, 4), f_2]$ by applying the action c_1 is similar.

The expansion of $[(1, 4), f_2]$ into $[(2, 4), f_0]$ by applying c_2 deserves more comment. Following lines 15-18 of the formula progression algorithm, the progression of $\Diamond_{\leq 3}(c(2) \wedge m(4))$, which is a conjunct of f_2, through $(2, 4)$ yields $true \vee \Diamond_{\leq 2}(c(2) \wedge m(4))$, which is simplified as *true*. In this way, the progression of f_2 through $(2, 4)$ yields f_0, hence closing a cycle. Note that the infinite execution represented by this cycle satisfies f_0. In general, as explained below, each cycle generated by our search process is necessarily *satisfactory*.

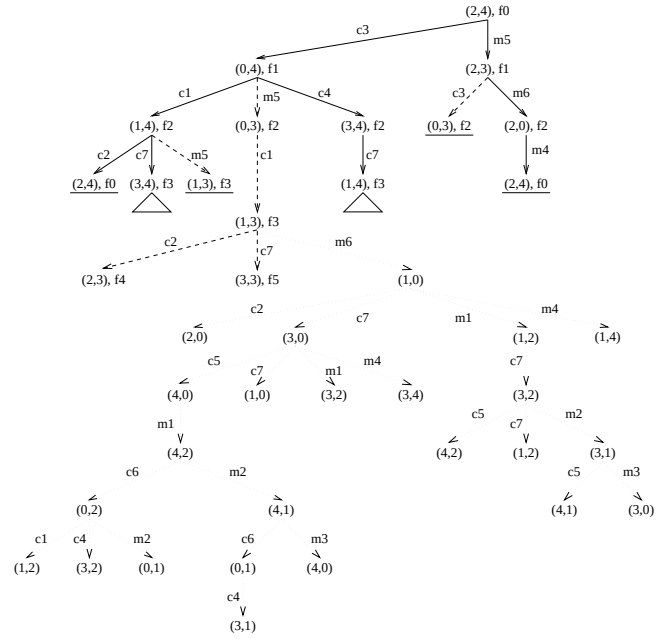
In Figure 7, solid lines represent transitions of the plan. Dashed lines represent paths that have been explored then rejected when the planner has determined that they lead to sinks. For instance, state $[(3, 3), f_5]$ is a sink because it violates the subgoal g_1 of f_0. This violation causes the progression of f_3 through $(3, 3)$ to return *false*, i.e., f_5. From such trivial sinks, the planner derives less trivial ones. For example, state $[(1, 3), f_3]$ also becomes a sink because action c_7 is uncontrollable.

Dotted lines represent parts of the search space that are not explored by our search process due to the use of the dependency-directed backtracking technique explained below. ■

Determining Satisfactory States and Sinks We extend the notion of controllable action to a notion of *controllable state*. A state is controllable if it is expanded (i.e., it is in CLOSED) and all its outgoing transitions are due to controllable actions. Each state is attached with a slot, *controllable*, indicating whether it is controllable or not. This slot is set to *false* for each newly created state. To each state is also attached a *status* that indicates whether the state is *sink*, *satisfactory*, or still *undefined*. The status of each newly created state is set to *undefined*. Then, the search process updates the status of expanded states based on the following definitions.

Definition 4.1 A *sink* is (1) a state of the form $[x, false]$ or (2) a state from which there is an outgoing uncontrollable transition to a sink or (3) a state involving an *eventually* modality and from which there is no enabled action or all enabled actions lead to sinks.

Condition 1 constitutes a basic case for determining sinks. It is implemented by the lines 14-16 of the plan-



$g_1 = \wedge_i^4 \Box_{\geq 0}(\neg c(i) \vee \neg m(i)); \quad g_2 = \Box_{\geq 0} \Diamond_{\leq 5}(c(2) \wedge m(4))$
$f_0 = g_1 \wedge g_2; \quad f_1 = f_0 \wedge \Diamond_{\leq 4}(c(2) \wedge m(4))$
$f_2 = f_0 \wedge \Diamond_{\leq 3}(c(2) \wedge m(4)); \quad f_3 = f_0 \wedge \Diamond_{\leq 2}(c(2) \wedge m(4));$
$f_4 = f_0 \wedge \Diamond_{\leq 1}(c(2) \wedge m(4)); \quad f_5 = \mathit{false}$

Figure 7: A part of the search graph for the maze problem

ner's algorithm. Condition 2 constitutes one of two recursive cases for determining sinks and is implemented in part by the function *Sink*. This function is described in Figure 8 and is discussed below. Condition 2 is also handled in part concurrently with Condition 3 by the procedure *Check_Non_Trivial_Satisfactory_States*. This is further explained below.

Definition 4.2 A state is *satisfactory* if (1) it is controllable and does not involve an eventuality modality or (2) it is in a cycle not containing a sink or (3) there exists at least one transition to a satisfactory state and every outgoing uncontrollable transition leads to a satisfactory state.

Condition 1 specifies the most trivial of two basic cases for determining satisfactory states. It is implemented by the lines 29-31. Condition 2 constitutes the other less trivial basic case that needs further explanation. Each formula progression decreases the timing constraints of bounded-time eventualities by the duration of the applied action. That is, each progression yields a formula different from the current one. Hence, the search process cannot reach an ancestor state on the current path, i.e., it cannot close a cycle, unless all the eventualities have been achieved. Theorem 4.1 implies that each path terminated by such a cycle satisfies the goal formula. Note that if we had unbounded-time eventualities, it would be possible to reach a cycle without achieving all the eventualities, since the progression of an unbounded-time eventuality does not necessarily change it. Also note that a bounded-time eventuality cannot be progressed indefinitely without being achieved, since sooner or later its timing constraint will reach 0, resulting in a sink.

```
function Sink(s);
1.  s.status ← sink;
2.  Mark_Useless(s);
3.  for each s' in s.pre do
4.      if s'.status ≠ sink then Sink(s');
end
```

Figure 8: Algorithm for deleting sinks

```
function Satisfactory(s)
1.  s.status ← satisfactory;
2.  for each s' in s.pre do
3.    if s'.status ≠ satisfactory then
4.      flag ← true;
5.      for each a in A do
6.        if ξ(a, s') is defined and
7.          (not Controllable(a)) and
8.          ξ(a, s').status ≠ satisfactory
9.        then flag ← false; break;
10.   if flag then Satisfactory(s');
11. for each [a, s'] in s.pct do
12.     if s'.status ≠ satisfactory and
13.       s'.controllable = true
14.     then Satisfactory(s');
end
```

Figure 9: Algorithm for determining satisfactory states

Condition 2 is implemented in the function *Check_Non_Trivial_Satisfactory_States*. The recursive case is handled by the procedure *Satisfactory* (Figure 9), which is called each time a basic satisfactory state has been identified. Because of space constraints, we describe *Check_Non_Trivial_Satisfactory_States* informally. Essentially, Condition 2 must label all satisfactory states of the automaton M. When all the satisfactory states have been identified, the remaining states still labeled *undefined* become sinks. In this way, the planner also takes into account Condition 3 for sinks.

Control Directed Backtracking Search The idea of this search technique is to avoid expanding descendants of nontrivial sinks. Indeed, such expansions are useless. For example, in Figure 7, when the expansion has reached the sink $[(3,3), f_5]$, through the uncontrollable transition c_7, state $[(1,3), f_3]$ also becomes a sink. It then becomes useless to expand descendants of $[(1,3), f_3]$ that cannot be reached through at least one path from the initial state.

To implement this idea, we need to attach *backpointers* to states to indicate their immediate parents: $s.pre \subseteq 2^{(X \times F)}$ contains the set of immediate parents of s for which the transition to s is caused by an uncontrollable

action; $s.pct \subseteq 2^{(\mathcal{A}_c \times (X \times F))}$ contains the set of controllable transitions incoming to s. Hence, if $[a, s'] \in s.pct$, then state s can be avoided by disabling a at s', since a is controllable.

The control-directed backtracking mechanism is then implemented by the function *Sink* (see Figure 8). Given a sink s, this function explores, backwards, all ancestors of s reachable through a path formed only of uncontrollable transitions. Each state met in this traversal is marked "*sink*" and all its descendants are marked "*useless*" provided that their predecessors are *sink* or marked "useless" (the function *Mark_Useless* runs over descendants of s, marking them "*useless*" until we meet a state already marked "useless" or for which at least one predecessor is not marked "useless"). A state marked "useless" is removed from OPEN.

Note, however, that this process might mark states for which all the incoming transitions have not yet been generated. Hence, when these currently missing transitions are generated, the "*useless*" mark are removed. The overhead incurred by these removals can be avoided by deleting states rather than marking them. Deleted states would then be re-expanded when reached again by further expansions. In this way, we replace the overhead due to updating the "useless" marks by the overhead incurred by re-expanding states. The problem here is comparable to the management of *backpointers* in the search algorithm A* [Rich and Knight, 1991] (pp. 76-79): one can either keep and update backpointers on expanded nodes, or re-expand them. Both strategies are incomparable in terms of performance.

5 Conclusion

In this paper, we have described a new planning method based on an original *control-directed backtracking* technique and a procedure for checking MTL goal formulas incrementally. Our framework generates plans of reactive systems that take into consideration safety, liveness, and real-time constraints as well as controllable actions in a uniform way. Note that the computation of the feedback function φ does not need to be done after the construction of the entire automaton R. Instead, we could detect satisfactory states at different stages of expansion and immediately update the function φ accordingly. This means that the processing done in the function *Check_Non_Trivial_Satisfactory_States* has to be interleaved with the expansion process. In this way, φ is produced incrementally. This gives *anytime* capability to our planner [Dean and Body, 1988].

As a matter of facts, the search process explained in the previous section can be seen as generating the *product* of two *timed automata* (or *timed transition graphs*): one automaton accepting all the world-state sequences, and the other accepting the legal sequences (satisfying the MTL formula). The product automaton is simply a search graph in which world-states correspond to states in the world-state automaton and MTL formulas corresponds to states in the MTL automaton. Our search process computes this product *on the fly*. Thus, rather than specifying the transitions of the world-state automaton explicitly, we use a STRIPS-like action language to specify them succintly and then recursively apply actions to generate them only when they become relevant in the search process. Similarly, rather than specifying the transitions of the MTL automaton explicitly, we progress formulas to generate them only when they become relevant in the search process. Recently, Brandin and Wonham have proposed an approach in which the transitions of the two automata are specified explicitly [Brandin and Wonham, 1994]. Our approach has at least two advantages with respect to their work. First, in our planner only world-states and MTL formulas that are relevant in the product are generated. In the Brandin and Wonham's approach, sink states and all potential time transitions must be specified explicitly in the input automata since their relevance can only be determined when constructing the product. Second, MTL specifications are declarative and are maintained as such during our search process. We think that this will facilitate debugging of controller specifications. For example, sink states rise when an MTL *always* modality is made false or the deadline of an *eventually* modality expires. These modalities provide information in a declarative form about undesired states in the process being controlled.

We are investigating different complementary strategies for coping with state explosion. Recent work in AI planning shows that forward-chaining search can be made effective using such strategies [Bacchus and Kabanza, 1995]. At the same time, work in control theory has demonstrated that a forward-chaining exploration can become effective with real-world problems using efficient implementation or modular techniques [Balemi *et al.*, 1993; Brandin, 1994]. In the same trend, we intend to investigate the application of AI planning *goal-regression* techniques ([Rich and Knight, 1991]) to generate search control knowledge that specifies the relevance of actions to a given goal. With this knowledge, forward-chaining will explore the more relevant parts of the search space before the less relevant ones. The usual *goal-regression* techniques in AI planning deal only with goals of achieving state conditions. Our approach to handle general MTL goals is being influenced by two observations. On the one hand, an MTL *eventually* modality expresses a condition to be achieved. Thus, we could parse an MTL formula to extract the sub-formulas expressing conditions to be achieved and then apply *goal-regression* techniques to determine the relevance of actions to these conditions. On the other hand, an MTL *always* modality expresses a condition to be maintained. Each uncontrollable action that could falsify such a condition is relevant since it might lead to sinks. Such relevant actions could also be determined using *goal-regression*. It thus

seems possible to interleave the *goal-regression* processes of computing relevant actions and the *forward-chaining* one of generating accessible states. The challenge will be to study the tradeoffs between these processes and the effectiveness of their combination on real-world problems. In the same line of inquiry, we are also interested in abstraction over large numbers of states during search.

References

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

[Alur *et al.*, 1993] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[Bacchus and Kabanza, 1995] F. Bacchus and F. Kabanza. Control strategies in planning. *AAAI Spring Symposium Series. Extending Theories of Action: Formal Theory and Practical Applications*, March 1995.

[Balemi *et al.*, 1993] S. Balemi, G.J. Hoffman, P. Gyugyi, H. Wong-Toi, and G.F. Frankli. Supervisory control of rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, 1993.

[Brandin and Wonham, 1994] B. A. Brandin and W. M. Wonham. Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 39(2):329–42, 1994.

[Brandin, 1994] B. A. Brandin. Supervisory control of an experimental manufacturing cell. In *Proceedings of Thirty-second Annual Allerton Conference on Communication, Control, and Computing*, pages 699–708, Urbana-Champaign, 1994.

[Dean and Body, 1988] T. Dean and M. Body. An analysis of time-dependent planning. In *Proc. of Sixth National Conference on Artificial Intelligence*, pages 49–54, 1988.

[Dean and Wellman, 1991] T. L. Dean and M. P. Wellman. *Planning and Control*. Morgan Kaufmann, 1991.

[Dean *et al.*, 1993] T. Dean, L. P. Kaelbling, J. Kerman, and A. Nicholson. Planning with deadlines in stochastic domains. In *Proc. of Eleventh National Conference on Artificial Intelligence*, pages 574–579, 1993.

[Drummond and Bresina, 1990] M. Drummond and J. Bresina. Anytime synthetic projection: Maximazing probability of goal satisfaction. In *Proc. of Eighth National Conference on Artificial Intelligence*, pages 138–144, 1990.

[Emerson, 1990] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. MIT Press/Elsevier, 1990.

[Fikes and Nilsson, 1972] R. Fikes and N. J. Nilsson. Learning and executing generalized robots. *Artificial Intelligence*, 3(4):251–288, 1972.

[Fusaoka *et al.*, 1983] A. Fusaoka, H. Seki, and K. Takahashi. A description and reasoning of plant controllers in temporal logic. In *Proceedings of the IJCAI*, pages 405–408, 1983.

[Georgeff and Lansky, 1987] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of Fifth National Conference on Artificial Intelligence*, pages 677–682, 1987.

[Godefroid and Kabanza, 1991] P. Godefroid and F. Kabanza. An efficient reactive planner for synthesizing reactive plans. In *Proc. of Ninth National Conference on Artificial Intelligence*, pages 640–645, 1991.

[Kabanza, 1990] F. Kabanza. Synthesis of reactive plans for multi-path environments. In *Proc. of Eighth National Conference on Artificial Intelligence*, pages 164–169, 1990.

[Makungu *et al.*, 1994] M. Makungu, M. Barbeau, and R. St-Denis. Synthesis of controllers with colored Petri nets. In *Proceedings of Thirty-second Annual Allerton Conference on Communication, Control, and Computing*, pages 709–718, Urbana-Champaign, 1994.

[Nilsson, 1994] N. J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[Ramadge and Wonham, 1989] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[Rich and Knight, 1991] E. Rich and K. Knight. *Artificial Intelligence*. McGraw Hill, 1991.

[Stallman and Sussman, 1978] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9, 1978.

[Thistle and Wonham, 1986] J. G. Thistle and W. M. Wonham. Control problems in a temporal logic framework. *International Journal on Control*, 44(4):943–976, 1986.