On the Synthesis of Situation Control Rules under Exogenous Events*

Froduald Kabanza
Université de Sherbrooke, Faculté des Sciences/DMI
Sherbrooke, Québec J1K 2R1 Canada
Email:kabanza@dmi.usherb.ca

Abstract

One approach for computing plans for reactive agents is is to check goal statements over state trajectories modeling predicted behaviors of an agent. This paper describes a powerful extension of this approach to handle time, safety, and liveness goals that are specified by Metric Temporal Logic formulas. Our planning method is based on an incremental planning algorithm that generates a reactive plan by computing a sequence of partially satisfactory reactive plans converging towards a completely satisfactory one. Partial satisfaction means that an agent controlled by the plan accomplishes its goal only for some environment events. Complete satisfaction means that the agent accomplishes its goal whatever the environment event occur during the execution of the plan. As such, our planner can be stopped at any time to yield a useful plan.

Keywords: Planning, control, reactive agents, temporal goals.

Introduction

Reactive agents play an increasingly important role in many computer applications ranging from robotics and manufacturing to process control and software interfaces. One key component for such agents is a planner that generates a reactive plan for a given goal and environment. A reactive plan specifies the actions to be executed by a reactive agent in different situations that can likely be encountered in order to satisfy a given goal. This paper addresses the problem of generating reactive plans for discrete-event reactive agents. By discrete-event, it is meant that the behaviors of the agent in a given environment can be modeled by a state transition system.

A reactive plan is comparable to a strategy for an agent playing a game (like chess) against an environment. Such an agent chooses the move to make at every instant according to a game strategy (i.e., a reactive plan) that takes into account the moves played by the environment. However, for most of the applications we are interested in, the agent and the environment do not politely take turns as do players in a game. In contrast, the time instants at which the environment actions occur are generally unpredictable. The goal might also be more complex than simply reaching a winning state.

One approach for generating reactive plans is to project forwards actions, enumerating state sequences, pruning those falsifying the goal and enabling those satisfying the goal (Drummond and Bresina 1990). This paper discusses an extension of this approach to handle more complex goal types, deal with exogenous events, and reason about infinite behaviors. Our approach builds on a previous controller synthesis framework that uses Metric Temporal Logic (MTL) goals (Barbeau *et al.* 1995). We propose a simpler planning algorithm based on a model of uncontrollable actions by nondeterminism. Exogenous, uncontrollable actions are specified by using ADL operators; the planner translates them into nondeterministic transitions to generate state sequences. This yields a simpler mechanism for implementing control-directed backtracking. We also explain how to deal with liveness goals.

Our work is also related to the synthesis of classic plans with temporally extended goals (Bacchus and Kabanza 1996). In classic plans, there is no exogenous actions and no infinite behaviors. As such, classic plans are seen as a particular case of reactive plans.

Forward-chaining planning is natural for the manipulation of complex temporal goals using modal temporal logics, because such logics have models that are sequences of states. The main limitation of this approach is the state explosion problem. Search control mechanisms are necessary in order to implement an efficient forward-chaining planner. Some authors

*This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds pour la formation de chercheurs et l'aide à la recherche (FCAR).

use state transition probabilities to enumerate only the states that can most likely occur during the execution (Drummond and Bresina 1990). Partial-order search techniques have also been proposed to limit the state explosion due to interleaving independent actions (Godefroid and Kabanza 1991). The experiments reported herein relies on the use of search control formulas to prune irrelevant state sequences (Bacchus and Kabanza 1995).

The remainder of this paper is organized as follows. The next section Section discusses the action representation. Section presents the logic used to specify goal statements. Section describes the plan representation. Section describes the planner algorithm. Finally, we conclude with an the evaluation of the planner on a simulated problem.

Specifying Actions

The reactive plan synthesis problem is understood as a particular case of multi-agent planning problem, in which there is one *reactive agent* executing *controllable actions* to react to events generated by other agents in the environment executing *uncontrollable actions*. The problem is to generate a reactive plan for the reactive agent to achieve a given goal whatever action is executed by an environment agent.

Primitive actions for all the agents are specified by using ADL (Action Description Language) operators (Pednault 1989). An operator describes the precondition, effects, duration, and controllability status for a schema of actions. Examples of ADL operators are shown in Figure 1 for a reactive system consisting of a scheduler, processes and resources. The scheduler must allocate resources to processes according to a given goal. Each time the scheduler has deallocated the resource, it enters a busy state in which the only possible action is to wait. The scheduler represents a reactive agent for which one wants to generate a reactive plan, while the processes compose its environment.

Variables are terms starting with a "?". An action is obtained from an operator by replacing free occurrences of variables by constants. The *delete list* of an action consists of literals q such that $p \Rightarrow q$ is in the delete list and p holds; the *add list* is obtained likewise. As usual, an action is enabled in a state if its precondition holds in the state. In this case, the application of the action to the state yields a successor state obtained by removing the delete list and adding the add list.

Two actions can be executed simultaneously if they belong to two different agents. The transitions from a state represent all possible ways of simultaneously executing actions that are enabled in this state; it is assumed that all the actions enabled in a state have the same time duration. The states that result from the simultaneous execution of a set of actions are leafs of the tree obtained by interleaving the actions. As noted by Emerson ((Emerson 1990), page 1017), by picking a sufficiently fine level of granularity for the primitive actions, any behavior produced by true concurrency (i.e., true simultaneity of actions) can be simulated by interleaving.

To consider an example, let us assume that there are two processes (p_1 and p_2), and one resource (r). In state $\{\}$, the following sets of simultaneous actions are possible:

$$\{wait(s), wait(p_1), wait(p_2)\},$$
$$\{wait(s), request(p_1, r), wait(p_2)\},$$
$$\{wait(s), wait(p_1), request(p_2, r)\}, \text{ and}$$
$$\{wait(s), request(p_1, r), request(p_2, r)\}.$$

Uncontrollable actions are removed from transition labels to obtain nondeterministic transitions, as in Figure 2. Each transition from state $\{\}$ in this figure corresponds to one of the previous sets of actions. The correspondence can be easily established by comparing states and action effects.

As far as the planner is concerned, the only requirement is to be able to compute the successors of a state under the application of an action. Thus, henceforth we will assume that the primitive actions are specified by a state transition function *succ* returning the list of actions that are possible in each world state and their corresponding durations and successors. More specifically, $succ(w)$ returns a list $\{(a_1, t_1, l_1), \ldots, (a_n, t_n, l_n)\}$, where a_i is an action possible in w, t_i its time duration, and l_i its set of nondeterministic successors.

As explained above, *succ* is specified via ADL operators. Of course, one could specify it more explicitly by using a state transition system or nondeterministic operators. This can be, however, cumbersome and error prone when there are many interacting agents. It is more practical to specify the effects of each action taken isolately, so that the planner automatically derives the effects of simultaneous executions by using a theory of interaction; in our case, the theory of interaction models simultaneous executions by interleaving actions.

Goals

Since the behaviors of a reactive system are described by sequences of states, it seems natural to specify goals using formulas that are interpreted over state sequences. Modal temporal logics have been proven useful for specifying temporal properties of concurrent systems modeled by state sequences (Manna and Pnueli

Actions for the scheduler

allocate(s,?r,?p)
PRECOND: $requesting(?r, ?p) \wedge \neg busy(s)$
ADD: $true \Rightarrow using(?p, ?r)$
DELETE: $true \Rightarrow requesting(?p, ?r)$
DURATION: 1 CONTROLLABLE: *true*

deallocate(s,?r,?p)
PRECOND: $using(?r, ?p) \wedge \neg busy(s)$
DELETE: $true \Rightarrow using(?r, ?p)$
ADD: $true \Rightarrow busy(s)$
DURATION: 1 CONTROLLABLE: *true*

wait(S)
DELETE: $busy(s) \Rightarrow busy(s)$
DURATION: 1 CONTROLLABLE: *true*

Actions for processes

request(?p,?r)
PRECOND: $is\text{-}process(?p) \wedge is\text{-}resource(?r) \wedge \neg(requesting(?p, ?r) \wedge using(?p, ?r))$
ADD: $true \Rightarrow requesting(?p, ?r)$
DURATION: 1 CONTROLLABLE: *false*

release(?p,?r)
PRECOND: $using(?p, ?r)$
ADD: $using(?p, ?r) \Rightarrow using(?p, ?r)$
DURATION: 1 CONTROLLABLE: *false*

wait(?pro)
PRECOND: $is\text{-}process(?p)$
DURATION: 1 CONTROLLABLE: *false*

Figure 1: ADL operators for a scheduler and processes

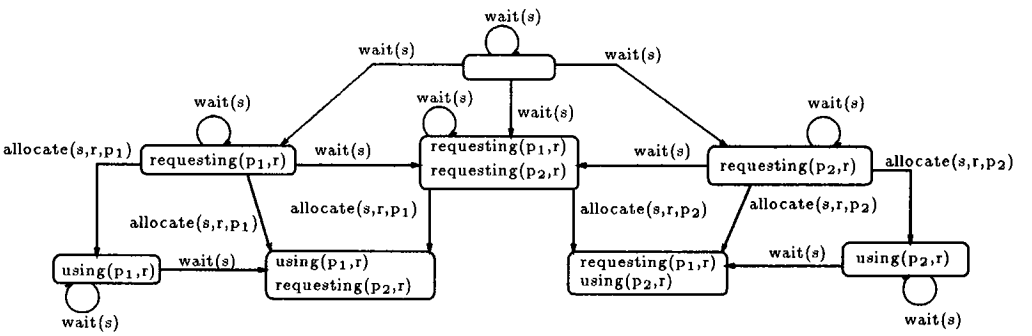


Figure 2: Some state transitions for a scheduler and two processes

1991). To fix a context, we use Metric Temporal Logic (MTL) (Koymans 1990).

Syntax

MTL formulas are constructed from an enumerable collection of propositional symbols; the Boolean connectives $\wedge$ (and) and $\neg$ (not); and the temporal modalities $\bigcirc_{\sim t}$ (next), $\Box_{\sim t}$ (always), and $U_{\sim t}$ (until), where $\sim$ denotes $\leq$ or $\geq$ and t is a real number. The formula formation rules are:

- every propositional symbol p is a formula and
- if f_1 and f_2 are formulas, then so are $\neg f_1$, $f_1 \wedge f_2$, $\bigcirc_{\sim t} f_1$, $\Box_{\sim t} f_1$ and $f_1 U_{\sim t} f_2$.

In addition to these basic rules, we use the standard abbreviations $f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2)$ (f_1 or f_2), $f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$ (f_1 implies f_2), and $\Diamond_{\sim t} f \equiv true\, U_{\sim t} f$ (eventually f).

The intuitive meaning of MTL formulas is captured by using the natural language interpretation for logical connectives and by noting that when the time constraint "$\leq t$" or "$\geq t$" is associated to a modal connective, the modal formula must hold before or after the time period t has expired. For example, $f_1 \rightarrow f_2$ is read as "f_1 implies f_2," $\bigcirc_{\leq t} f$ as "the next state is in the time interval $[0, t]$ and satisfies f," $\Box_{\leq t} f$ as "always f on the time interval $[0, t]$," $\Diamond_{\leq t} f$ as "eventually f on the time interval $[0, t]$," and $f_1 U_{\geq t} f_2$ as "f_1 until f_2 on the time interval $[t, \infty[$".

Semantics

MTL formulas are interpreted over models of the form $M = \langle \mathcal{W}, \pi, \mathcal{D} \rangle$, where

- $\mathcal{W}$ is an infinite sequence of world states $w_0 \ldots w_k \ldots$;
- π is a function that evaluates propositional symbols in a world state: $\pi(p, w) = true$ if proposition p holds in world state w;[1] and

[1] Usually, $\pi(p, w)$ is true if $p \in w$. However, propositions can also be interpreted by invoking a function specified by

- $\mathcal{D}$ is a time transition function: $\mathcal{D}(w_i, w_{i+1})$ returns a real number, which is the time duration for the transition (w_i, w_{i+1}).

As usual, we write $\langle M, w\rangle \models f$ if state w in model M satisfies formula f. When the model is understood, we simply write $w \models f$. In addition to the standard rules for Boolean connectives, we have the following rules for temporal connectives. For a state w_i in a model M, a real number $d = \mathcal{D}(w_i, w_{i+1})$ the duration of the transition (w_i, w_{i+1}), a propositional symbol p, formulas f_1 and f_2:

- $w_i \models p$ *iff* $\pi(p, w_i) = true$;
- $w_i \models \bigcirc_{\leq t} f_1$, *iff* $t - d \geq 0$ and $w_{i+1} \models f_1$;
- $w_i \models \bigcirc_{\geq t} f_1$, *iff* $t - d \leq 0$ and $w_{i+1} \models f_1$;
- $w_i \models \Box_{\leq t} f_1$, *iff* $t - d > 0$ and $w_i \models f_1$ and $w_{i+1} \models \Box_{\leq (t-d)} f_1$; or $t - d = 0$ and $w_i \models f_1$; or $t - d < 0$;
- $w_i \models \Box_{\geq t} f_1$, *iff* $t - d > 0$ and $w_{i+1} \models \Box_{\geq (t-d)} f_1$; or $t - d \leq 0$ and $w_i \models f_1$ and $w_{i+1} \models \Box_{\geq 0} f_1$;
- $w_i \models f_1 \, U_{\leq t} \, f_2$ *iff* $t-d > 0$ and ($w_i \models f_2$ or ($w_i \models f_1$ and $w_{i+1} \models f_1 \, U_{\leq (t-d)} \, f_2$)); or $t-d = 0$ and $w_i \models f_2$;
- $w_i \models f_1 \, U_{\geq t} \, f_2$, *iff* $t - d > 0$ and $w_{i+1} \models f_1 \, U_{\geq (t-d)} \, f_2$; or $t - d \leq 0$ and ($w_i \models f_2$ or ($w_i \models f_1$ and $w_{i+1} \models f_1 \, U_{\geq 0} \, f_2$));

Finally, we say that model M (or sequence $\mathcal{W}$) satisfies a formula f if $w_0 \models f$.

For example, the formula $\Diamond_{\geq 0} \Box_{\geq 0} p$ states that p must eventually be made true and then maintained true thereafter. The formula

$$\Box_{\geq 0}(\neg(using(p_1, r) \wedge using(p_2, r)) \wedge \quad (1)$$
$$(requesting(p_1, r) \rightarrow \Diamond_{\leq 4} using(p_1, r)) \wedge \quad (2)$$
$$(requesting(p_2, r) \rightarrow \Diamond_{\leq 4} using(p_2, r))) \quad (3)$$

states that p_1 and p_2 must never use resource r at the same time (subformula 1) and each process requesting resource r must obtain the right to use it within 4 time units (subformulas 2 and 3).

A goal of the form $\Box_{\geq 0}(q \rightarrow \Diamond_{\leq t} p)$ is satisfied by an agent that continuously senses the current world state, checking if q holds, to execute actions making p true within t time units. There is no single final state in which we can consider that the goal has been satisfied. Instead, we have final cycles representing infinite behaviors that satisfy the goal.

Reactive Plans

A reactive plan is represented as a set of *situation control rules* (SCR) (Drummond 1989). In the original definition, an SCR maps a world state to a set of actions that can be executed simultaneously. In our case, only one reactive agent executes controllable actions in reaction to actions executed by environment agents; there is only one action for each SCR. The original definition of SCRs was extended by adding labels and sequencing to allow the interpretation process to be biased by the recommendation from the previously executed SCR (Kabanza 1992). Other authors experimented with this extension in telescope control applications (Drummond *et al.* 1994).

An SCR is a tuple $[s, w, a, S]$, where s is a plan state, w is a world state, a is an action, and S is a set of plan states. An agent executes a reactive plan by finding an SCR $[s, w, a, S]$ such that w holds in the current situation. Then, the action a is executed and the resulting situation is determined from S by finding a plan state associated with a world state holding in the new situation. If many states hold, one with the largest set of propositional symbols is selected. If this still leaves many candidates, any of them is selected. In practice, the evaluation of the truth value of a proposition might involve sensing operations as well as internal computations.

The execution of a reactive plan never terminates, so that it produces an infinite sequence of states. Nevertheless, finite executions can be simulated by using an SCR with a *wait* action. A reactive plan is *satisfactory* if each sequence of states resulting from its execution satisfies the goal in the sense of MTL semantics, whatever nondeterministic successor is selected for each action.

Figure 3 shows a reactive plan for the scheduler, two processes (p_1 and p_2), and one resource (r). Propositions and action names are written in a Lisp-like notation. As illustrated by this example, a reactive plan can contain different SCRs with the same world state (see states 9 and 12).

Planner

The basic operations in the planning process are to check safety goals, liveness goals, and controllability, and to generate SCRs that enable only actions on satisfactory sequences.

A safety goal states that something bad must never happen on a sequence of world states (Manna and Pnueli 1991). It is characterized by the fact that, when it is violated by a sequence, the violation always occurs on a finite prefix of the sequence. Safety goals are of the form $\Box_{\leq t} g_1$, $\Box_{\geq t} g_1$, and $g_1 \, U_{\leq t} \, g_2$. Indeed, a violation of such a formula always occurs on a finite sequence. Specifically, the violation occurs when a state not satisfying g_1 is met. For the formula $g_1 \, U_{\leq t} \, g_2$, the violation also occurs when a number of transitions for

the user taking a world state as argument.

```
[STATE 0 WORLD () ACTION (wait s) SUCCESSORS (0 1 5 13)]
[STATE 1 WORLD ((requesting p2 r)) ACTION (allocate s r p2) SUCCESSORS (2 10)]
[STATE 2 WORLD ((using p2 r)) ACTION (wait s) SUCCESSORS (2 3)]
[STATE 3 WORLD ((requesting p1 r) (using p2 r)) ACTION (deallocate s r p2) SUCCESSORS (4)]
[STATE 4 WORLD ((busy s) (requesting p1 r)) ACTION (wait s) SUCCESSORS (5 12)]
[STATE 5 WORLD ((requesting p1 r)) ACTION (allocate s r p1) SUCCESSORS (6 11)]
[STATE 6 WORLD ((using p1 r)) ACTION (wait s) SUCCESSORS (6 7)]
[STATE 7 WORLD ((requesting p2 r) (using p1 r)) ACTION (deallocate s r p1) SUCCESSORS (8)]
[STATE 8 WORLD ((busy s) (requesting p2 r)) ACTION (wait s) SUCCESSORS (1 9)]
[STATE 9 WORLD ((requesting p1 r) (requesting p2 r)) ACTION (allocate s r p2) SUCCESSORS (10)]
[STATE 10 WORLD ((requesting p1 r) (using p2 r)) ACTION (wait s) SUCCESSORS (3)]
[STATE 11 WORLD ((requesting p2 r) (using p1 r)) ACTION (wait s) SUCCESSORS (7)]
[STATE 12 WORLD ((requesting p1 r) (requesting p2 r)) ACTION (allocate s r p1) SUCCESSORS (11)]
[STATE 13 WORLD ((requesting p1 r) (requesting p2 r)) ACTION (allocate s r p1) SUCCESSORS (7)]
```

Figure 3: A reactive plan for a process scheduler

which the sum of durations is greater than t has been traversed. The violation of a safety goal leads to dead ends during the enumeration of state sequences.

A liveness goal states that something good must eventually happen (Manna and Pnueli 1991). It is characterized by the fact that it can only be violated by an infinite sequence of states (i.e., a cycle of states in our case). Liveness goals are of the form $g_1\, U_{\geq t}\, g_2$. Indeed, for such a formula, there is no bound on the time when g_2 should occur after t time units. In other words, it must be checked over the infinite time interval $[t, \infty[$. The violation of a liveness goal leads to a bad cycle.

Controllability is checked by ensuring that, for each action of an SCR matching a given state, every nondeterministic successor is on a satisfactory sequence (i.e., a sequence satisfying the safety and liveness conditions involved in the goal).

Note that a goal can involve interconnected safety and liveness subgoals. They are syntactically determined by checking their main temporal connectives as indicated above. However, one must take into account the fact that the negation connective changes the temporal modalities. Indeed, $\neg(\Box_{\sim t}\, g)$ is equivalent to $\Diamond_{\sim t}\, \neg g$, while $\neg(g_1\, U_{\sim t}\, g_2)$ is equivalent to $(\Box_{\sim t}\, \neg g_2) \vee (\neg g_2\, U_{\sim t}\, \neg g_1)$. To avoid checking these equivalences, our planner automatically transforms the goal into an equivalent formula in which only propositional symbols are negated. This is done by using the above two temporal equivalences, usual De Morgan laws, and equivalences for the connectives $\wedge, \vee$, and $\neg$.

Checking Safety Goals

The procedure for checking safety goals relies on two observations on MTL interpretation rules. First, the truth value of an MTL formula on a sequence of states depends on the durations of transitions rather than on the time stamps of states. Second, the truth value of an MTL formula on a sequence $w_i w_{i+1} \ldots$ is established by evaluating a *present* requirement in w_i and postponing a *future* requirement to be checked in w_{i+1}.

Thus, an MTL formula is checked on a sequence of states by labeling each state with the goal to be validated on sequences outgoing from it. The process of computing the goal labeling a state from that of a parent is called to *progress the goal from the parent state through the successor state* (see Figure 4). The input consists of an MTL goal g, a successor world state w, a real number t, and a function π that evaluates the truth value of a proposition in a world state. Formula g labels a state w', which is a parent of w. The real number d is the duration of the action executed in w' to produce w. The output is a formula representing the future requirement of g with respect to w, that is, the goal to label w.

The goal progression algorithm is merely an implementation of MTL interpretation rules. The violation of a maintenance requirement is detected by returning *false* when evaluating a proposition that is in the scope of an *always* modal connective (case 2 in the progression algorithm). The violation of a time bound for an eventuality is also detected by returning *false* when the time bound decreases to 0 before the eventuality is satisfied (case 8 in the progression algorithm). Since states labeled *false* are dead ends, it can be shown that it is impossible to form a cycle that does not satisfy a bounded-time eventuality (Barbeau *et al.* 1995).

Checking Liveness Goals

If the goal involves unbounded-time eventualities, cycles that do not satisfy the goal can be formed because

Progress-goal(g,w,d,π)
1. **case** g
2. p (p a proposition): $\pi(p, w)$;
3. $\neg g_1$: $\neg$*Progress-goal*(g_1, w, d, π);
4. $g_1 \wedge g_2$: *Progress-goal*$(g_1, w, d, \pi) \wedge$ *Progress-goal*(g_2, w, d, π);
5. $g_1 \vee g_2$: *Progress-goal*$(g_1, w, d, \pi) \vee$ *Progress-goal*(g_2, w, d, π);
6. $\Box_{\leq t}\, g_1$: **case** $t - d > 0$: *Progress-goal*$(g_1, w, 0, \pi) \wedge \Box_{\leq (t-d)}\, g_1$
   $t - d = 0$: *Progress-goal*$(g_1, w, 0, \pi)$;
   otherwise: *true*;
7. $\Box_{\geq t}\, g_1$: **case** $t - d > 0$: $\Box_{\geq (t-d)}\, g_1$
   otherwise: *Progress-goal*$(g_1, w, 0, \pi) \wedge \Box_{\geq 0}\, g_1$;
8. $g_1\, U_{\leq t}\, g_2$: **case** $t - d > 0$: *Progress-goal*$(g_2, w, 0, \pi) \vee$
   (*Progress-goal*$(g_1, w, 0, \pi) \wedge g_1\, U_{\leq (t-d)}\, g_2$)
   $t - d = 0$ *Progress-goal*$(g_2, w, 0, \pi)$;
   otherwise: *false*;
9. $g_1\, U_{\geq t}\, g_2$: **case** $t - t > 0$: $g_1\, U_{\geq (t-d)}\, g_2$
   otherwise: *Progress-goal*$(g_2, w, 0, \pi) \vee$
   (*Progress-goal*$(g_1, w, 0, \pi) \wedge g_1\, U_{\geq 0}\, g_2$)

Figure 4: Goal progression algorithm

when the progression of a formula of the form $g_1\, U_{\geq t}\, g_2$ has decreased t to 0, the progression keeps the formula invariant provided that g_1 is not violated. Hence, the state in which g_2 must be achieved can be postponed forever within a cycle, without causing a state labeled *false*.

This problem is solved by adding a mechanism for propagating unbounded-time eventualities to check that they are eventually satisfied. However, due to space limitations, the eventuality propagation process is not detailed in this paper. In fact, it basically simulates the eventuality automaton construction in a standard decision procedure for Linear Temporal Logic (LTL) (Wolper 1989). The only difference is that here we do that incrementally, simultaneously with the goal progression. From this perspective, formula progression procedure simulates the construction of a local automaton for the LTL decision procedure (of course, we do that by taking into account time constraints, which are absent in LTL).

The expansion of a state s is graphically illustrated by Figure 5. Figure 5(a) shows the successors of $s.world$ as given in the input specification for the planner. Figure 5(b) shows the expansion of s computed by the planner. The decomposition of goals into disjunctive normal form introduces an additional level of nondeterminism. Each successor of s under the application of an action a_i is labeled by a goal g_{ijk} and a set of eventualities E_{ijk}, where j corresponds to the jth nondeterministic successor of the action as given in the world transition system and k corresponds to the kth disjunct of the decomposition of the progression of $s.goal$ through the ith world state.

The set of transitions labeled with the same action from a state is called a *link*. The set of states corresponding to the same world state in a link, but labeled with a different goal and set of eventualities, is called a *split*.

Checking Controllability

The basic idea behind our planner is better explained by considering Figure 5 again. Roughly, the planner input consists of a goal and a state transition function specifying world-state successors as in Figure 5(a). The planner must find SCRs for the initial state and for each state that can be reached thereafter. Thus, SCRs are not required for all possible states in the domain, but rather only the states that can be reached due to nondeterministic effects have to be considered. When a state contains many different satisfactory SCRs, only one of them need be produced. Ideally, it should be the optimal one.

For instance, if an SCR specifies that the agent must execute action a_i in w, then there must exist an SCR for each nondeterministic successor of a_i. No SCR is required for successors of actions other than a_i that are possible in the current state. All nondeterministic successors of a_i must be covered because the environment decides which of them occurs. The agent has no control over this choice, but it can sense to observe which of them occurs. In return, whatever state w_{ij} results in the execution of a_i, the agent has the freedom to select the goal to satisfy among the g_{ijk}.

From another perspective, the action selection represents an or-joint for the planner in the sense that only one action needs to be proven satisfactory in each state

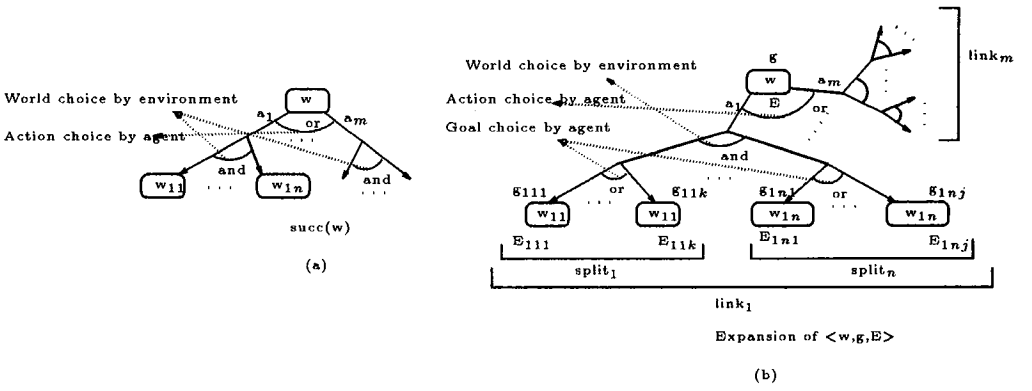


Figure 5: Expansion of a state s

likely to be encountered. In contrast, nondeterministic successors for an action represent an and-joint in the sense that the planner must find a successful action for each of them. Finally, goal-decomposition represents an or-joint because only one disjunct needs to be satisfied for each state.

Planner Algorithm

Our planner explores the and-or state space by expanding each state using the goal progression and eventuality propagation procedures to generate a set of SCRs enabling only actions that are on satisfactory sequences.

The planner input consists of an initial world state, a function *succ* that yields the transitions from each world state, a goal formula, and a function π that evaluates propositional symbols in world states. The planner computes a set of initial states from the world state and the goal (there can be many initial states due to goal decomposition); then the planner repeatedly calls the search process on each of these initial states until finding a satisfactory set of SCRs.

The set of initial states is determined as follows: the goal formula is put into normalized-negation form, by pushing the negation connective inwards next to propositional symbols (this form is preserved by the goal progression algorithm); then, the obtained formula is progressed into the initial world state using the time duration 0 to check that no safety condition is initially violated; finally, the resulting formula is put into disjunctive normal form and an initial state is created for each disjunct.

The search process explores the and-or state as illustrated by Figure 5 starting from a given initial state. A depth-first strategy is used with a stack to detect *on the fly* satisfactory cycles: a cycle is formed when the state on the top of stack is equal to another state in the stack; the cycle is satisfactory if some state between them is labeled with an empty set of eventualities.

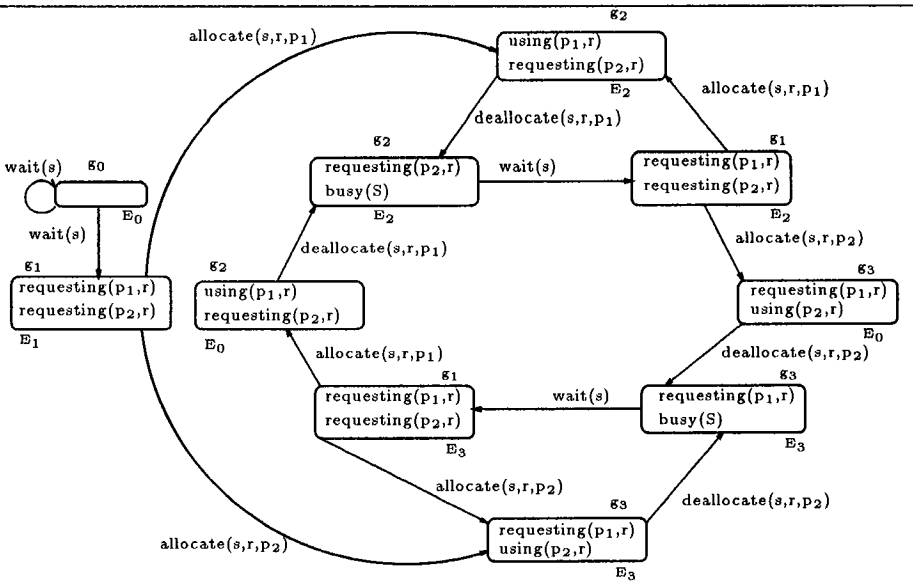
An SCR is generated from each expanded link as follows. Given the current link under expansion, an integer is associated to the state origin of the link (i.e., the state being expanded), such that all expanded states are associated with different integers. This integer represents a plan state, labeled with the world state of the extended state, which is origin of the link; this plan state is the antecedent of the SCR, whose consequent contains the action labeling the current link; the successors are the plan states corresponding the the first states in the slits of the link. However, these states might be later changed when search discovers (after backtracking) that they are not on satisfactory sequences (see below).

The expansion of a split fails when none of its states can be completed by a satisfactory cycle. In this case, the link containing the split also fails, so that the search process removes the corresponding SCR, backtracks to consider a sibling link, and generates a new SCR corresponding to this link. The search process terminates successfully after a backtracking phase to the initial state, with a satisfactory link.

It can be shown that the sequence of partially satisfactory plans computed by the search process converges towards a completely satisfactory one, whenever it exists.[2] By partially satisfactory, it is meant that the SCRs in the plan do not match all the states that can likely occur during the execution or might map wrong actions from some states. Complete satisfaction is understood in the sense given in Section . The convergence is not monotonic since SCRs can be removed and added until a solution is obtained.

Figure 6 shows a partial description of a graph ob-

[2] A detailed description of the planner algorithm and proofs of correctness and completeness will be given in the extended version of this paper.



Goals:

$$g_0 = \Box_{\geq 0}(\neg(using(p_1,r) \wedge using(p_2,r)) \wedge (requesting(p_1,r) \rightarrow \Diamond_{\geq 0}\, using(p_1,r)) \wedge (requesting(p_2,r) \rightarrow \Diamond_{\geq 0}\, using(p_2,r)))$$

$$g_1 = g_0 \wedge \Diamond_{\geq 0}\, using(p_1,r) \wedge \Diamond_{\geq 0}\, using(p_2,r) \quad g_2 = g_0 \wedge \Diamond_{\geq 0}\, using(p_2,r) \quad g_3 = g_0 \wedge \Diamond_{\geq 0}\, using(p_1,r)$$

Eventuality sets:

$$E_0 = \{\} \quad E_1 = \{\Diamond_{>0}\, using(p_1,r), \Diamond_{>0}\, using(p_2,r)\} \quad E_2 = \{\Diamond_{>0}\, using(p_2,r)\} \quad E_3 = \{\Diamond_{>0}\, using(p_1,r)\}$$

Figure 6: An extension of a state transition system with liveness goals

tained with the goal g_0 for the process scheduling example, assuming two processes (p_1 and p_2) and one resource (r). The initial state is labeled with the goal g_0 (the disjunctive normal form for g_0 is g_0) and the set of unbounded-time eventualities that are conjuncts of g_0. The goals labeling the other states are obtained as follows: for each transition (w_i, w_{i+1}), the goal g_{i+1} labeling w_{i+1} is obtained from g_i labeling w_i by the equation $g_{i+1} = \textit{Progress-goal}(g_i, w_{i+1}, 1, \pi)$. The eventuality set of a state s' that is a successor of s is obtained by *Propagate-eventualities*(s, s'). It can be checked that any cycle containing a state labeled with an empty set of eventualities (i.e., E_0) is satisfactory. The plan of Figure 3 was obtained from the graph partially represented by this figure.

Conclusion

Robustness and reliability of reactive agents depend, in part, on their ability to reason about their environments to plan. This paper presented a planning method for reactive agents that handles complex safety and liveness goals with time constraints. A plan generated by our planner is, by construction, proven to satisfy the goal whatever action the environment takes among those specified.

A protype planner was implemented in Common Lisp and experimented in a robot domain consisting of connected rooms, objects in the rooms, and a robot that moves objects to indicated rooms (see Figure 7). The objects are labeled from a to e; the robot (labeled r) is holding object a. Rooms are indicated by the letter r followed by a number. There is also a corridor; doors between rooms are indicated by shaded regions. The primitive actions for the robot are to grasp an object in the room, release an object being grasped, open a door, close a door, and move from a room to an adjacent one when the connecting door is opened.

Three other agents operate concurrently with the robot to form a multi-agent scenario. These are a *kid* process that moves between rooms to close doors arbitrarily; a *producer* that generates objects in given rooms (producer rooms) at any time; and a *consumer* that removes from the domain every object released in given rooms (consumer rooms). The number of objects that can be generated by the consumer is finite, but an object that has been consumed can be reproduced, so that we have infinite behaviors. The concurrent executions cause nondeterminism about the outcome of ac-

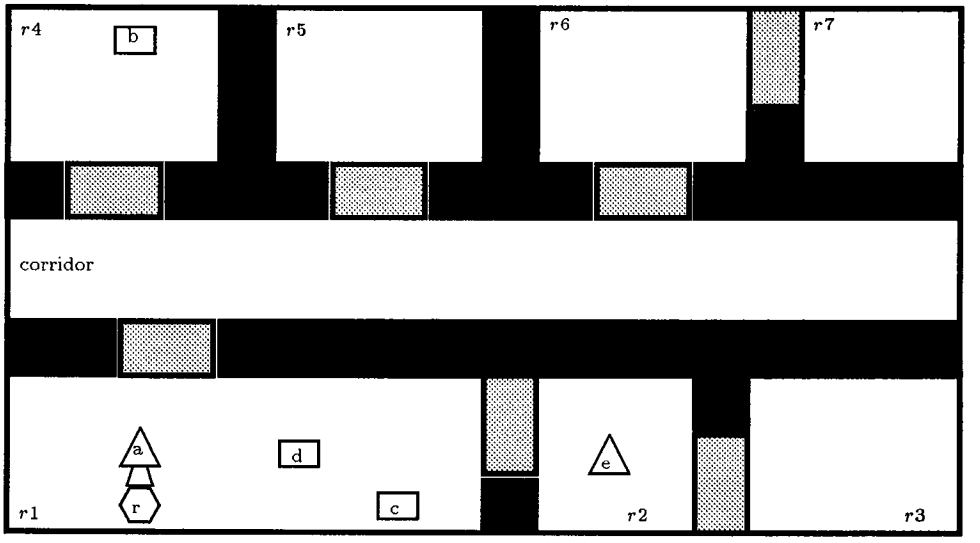


Figure 7: A simulated factory delivery domain

tions. However, our current implementation does not support probabilities yet; it assumes that all nondeterministic transitions have an equal likelihood.

We experimented goals of delivering produced objects to the consumer, using conjunctions of subformulas of the form $in(obj, proom) \rightarrow \Diamond_{\geq 0}(in(obj, croom) \wedge \neg holding(robot, obj))$, where *obj* is a producer object, *proom* is a producer room, and *croom* is a consumer room. We define the size of a given problem as the sum of different rooms involved in the goal and the number of doors affected by the kid-process. Using the search control formula given in the appendix and a blind depth-first search (without any heuristic), we obtained the results in Figure 8: The time is in cpu seconds on a Sun SPACRstation LX. The reported result for each size is an average of 10 iterations such that, at each iteration, the *producer rooms*, *consumer rooms*, and *kid-doors* are chosen randomly.

Search control formulas are useful not only for pruning irrelevant sequences from the search space, but also for pruning inefficient sequences. This strategy is applicable for inefficient behaviors that are easily identifiable. For example, the fact that opening and closing a door is not an efficient behavior can be easily captured by a search control formula. This strategy does not allow us, however, to generate plans that are constructively proven optimal. It is the user that must make sure that he has specified sufficient formulas to prune nonoptimal sequences. One future research direction will be to investigate mechanisms allowing computation of plans that are constructively proven optimal. Another future research topic deals with the use of abstractions to group together many states that have common properties.

References

F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. In *Proc. of 3rd European Workshop on Planning (EWSP)*, Assisi, Italy, September 1995.

F. Bacchus and F. Kabanza. Planning for temporally extended goals. To appear in Proc. of 13th National Conference on Artificial Intelligence (AAAI 96), Portland, Oregon, 1996.

F. Bacchus. TLPLAN user's manual. University of Waterloo, ON, Canada. Anonymous ftp: ftp://logos.uwaterloo.ca/pub/bacchus/tlplan/tlplan-manual.ps, 1995.

M. Barbeau, F. Kabanza, and R. St-Denis. Synthesizing plant controllers using real-time goals. In *Proc. of 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 791–798, 1995.

M. Drummond and J. Bresina. Anytime synthetic projection: Maximizing probability of goal satisfaction. In *Proc. of 8th National Conference on Artificial Intelligence*, pages 138–144, 1990.

M. Drummond, K. Swanson, and J. Bresina. Scheduling and execution for automatic telescopes. In M. Zwebeen and M.S. Fox, editors, *Intelligent Scheduling*, pages 341–369. Morgan Kaufmann Publishers, 1994.

M. Drummond. Situated control rules. In *Proc. of the first international conference on Principles of Knowledge Representation and Reasoning*, pages 103–113. Morgan Kofmann, 1989.

E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Sci-*

Size	1	2	3	4	5	6	7
Time	1.0	16.78	32.33	132.76	103.94	222.75	500.0
Generated States	101.3	753.0	1061.09	1946.7	1244.8	2244.8	4000.0

Figure 8: Experimental results

ence, volume B, pages 995–1072. MIT Press/Elsevier, 1990.

P. Godefroid and F. Kabanza. An efficient reactive planner for synthesizing reactive plans. In *Proc. of 9th National Conference on Artificial Intelligence*, pages 640–645, 1991.

F. Kabanza. *Reactive Planning of Immediate Actions.* PhD thesis, Departement d'Informatique, Universite de Liège, Belgium, October 1992.

R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time Systems*, 2(4):255–299, 1990.

Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.

E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the Situation Calculus. In *First International Conference on Principles of Knowledge Representation and Reasoning (KR' 89)*, pages 324–332, 1989.

P. Wolper. On the relation of programs and computations to models of temporal logic. In *Proc. Temporal Logic in Specification*, pages 75–123. Lecture Notes in Computer Science, Vol. 398, 1989.

Search Control Formula Used in the Experiment

By convention, variable names start with "?" and comment lines start with ";". Formulas are written in Lisp notation. The form *if-then-else* has the usual intuitive meaning, which is defined as an abbreviation of an *implies* construct. The procedure that propagates search-control formula also accepts first-order quantification over predicate variables. More specifically, we use a form of bounded quantification, which is borrowed from the TLPLAN system (Bacchus 1995): the construct $\forall[x : f_1(x)]f_2(x)$ (i.e., `(forall (x) (f_1 x) (f_2 x))` in Lisp notation) is equivalent to $\forall x(f_1(x) \rightarrow f_2(x))$.

We do not except that the reader will completely understand the formula below at the first glance; this might require familiarity with the formula notation used by in the implementation, which is the same as in the TLPLAN system (see (Bacchus 1995)). The formula is provided here only to give an idea of how a realistic search control formula looks like.

```
(always
  (forall (?room) (in robot ?room)
    (and
      ;; grasp only relevant objects only
(forall (?object) (in ?object ?room)
  (implies
    (and
      (not (holding robot ?object))
      (not (goal (holding robot ?object)))
      (not (exists (?room2)
             (goal (in ?object ?room2))
             (not (= ?room2 ?room)))))
      (next (not (holding robot ?object)))))
;; keep held object until in its room and
;; don't come back until having released it
(forall (?object ?room2)
  (goal (in ?object ?room2))
  (implies
    (holding robot ?object)
    (and
      (if-then-else (= ?room ?room2)
        (next (not (holding robot ?object)))
        (next (holding robot ?object)))
      (next (implies
              (not (in robot ?room))
              (until
                (not (in robot ?room))
                (in ?object ?room2)))))))
(forall (?door ?room2) (door/room ?room)
  (and
    ;; keep non kid-doors opened.
    (implies
      (and (opened ?door)
           (or (not (exists (?door2)
                      (kid-doors)
                      (= ?door ?door2)))
               (not (exists (?t)
                      (clock kid 0 ?t)))))
      (next (opened ?door)))
    ;;open doors only when there exist
    ;;objects to move.
    (implies
      (and
        (not (opened ?door))
        (not (goal (opened ?door)))
        (not (exists (?object ?room)
               (in ?object ?room)
               (exists (?room2)
                 (goal (in ?object ?room2))
                 (not (= ?room2 ?room))))))
        (next (not (opened ?door)))))))))
```