Planning Control Rules for Reactive Agents*

F. Kabanza, M. Barbeau, and R. St-Denis
{KABANZA, BARBEAU, STDENIS}@DMI.USHERB.CA
Université de Sherbrooke, Dépt. de math. et info.
Sherbrooke, Québec J1K 2R1 Canada

Abstract

A traditional approach for planning is to evaluate goal statements over state trajectories modeling predicted behaviors of an agent. This paper describes a powerful extension of this approach for handling complex goals for reactive agents. We describe goals by using a modal temporal logic that can express quite complex time, safety, and liveness constraints. Our method is based on an incremental planner algorithm that generates a reactive plan by computing a sequence of partially satisfactory reactive plans converging to 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 environment events occur during the execution of the plan. As such, our planner can be stopped at any time to yield a useful plan. An implemented prototype is used to evaluate our planner on empirical problems.

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

1. Introduction

The key characteristic of reactive agents is that they maintain an ongoing interaction with their environment to accomplish given goals. Such agents play an increasingly important role in many computer applications. A reactive agent can be a physical device (e.g., a robot) or a software process (e.g., a process scheduling system). Its executions are controlled by using a reactive program that invokes the action to be executed at each instant, depending on the concurrent situation.

The specification of a reactive program is called a reactive plan. It can be described by a mapping from situations to actions (Dean, Kaelbling, Kerman, & Nicholson, 1995) or a state transition system (Ramadge & Wonham, 1989). An important distinction between reactive plans and traditional plans in artificial intelligence is that a reactive plan has no predetermined sequence for executing its actions. Rather, the order in which the actions are executed depends on the situation sensed by the agent at the moment of execution. This makes reactive plans more practical in unpredictable environments.

A reactive plan can be compared 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

* 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).

instants at which the environment actions occur are generally unpredictable. The goal might also be more complex than simply reaching a winning state.

Our planning method is geared towards discrete-event reactive systems. Discrete-event means that the behaviors of the agent in a given environment can be modeled by a state transition system. Goals are described by using modal temporal logic formulas that can express various types of constraints including time, safety, and liveness requirements. A reactive plan is computed from an input specification describing the primitive behaviors of the agent and a goal. Specifically, the planner checks the goal incrementally over state sequences modeling the behaviors of the agent, producing a sequence of partially satisfactory plans that converges to a completely satisfactory one.

Partially satisfactory means that the goal might not be satisfied in some cases depending on the action executed by the environment. Completely satisfactory means that an agent executing the plan satisfies the goal whatever the action executed by the environment. In general, the longer the planner has been running, the better the obtained reactive plan is. This incremental capability is useful in real-time applications because the planner can be stopped prematurely to obtain a reaction for a critical situation.

The remainder of this paper is organized as follows. The next section recalls reactive plan synthesis approaches that are closely related to ours. Section 3 outlines our method and contributions. Section 4 discusses the specification of primitive behaviors for a reactive agent. Section 5 presents the logic that we use to specify goal statements. Section 6 describes the plan representation. Section 7 describes the planner algorithm. Section 8 discusses the evaluation of the planner on simulated problems. Section 9 discusses related work from a more technical point of view. Finally, we conclude in Section 10.

2. Related Work

A great deal of research in the areas of artificial intelligence (Dean et al., 1995; Barbeau, Kabanza, & St-Denis, 1995; Drummond & Bresina, 1990), control theory (Ramadge & Wonham, 1989), and program synthesis (Abadi, Lamport, & Wolper, 1989; Pnueli & Rosner, 1989) focuses on the problem of automatically generating reactive plans for discrete-event reactive systems.

Usually, the behaviors of a reactive agent are described by a state transition system. The purpose of a reactive plan is to select the agent's actions that must be executed in each situation that can likely occur during the execution in order to satisfy a given goal. The planning problem is to compute such a reactive plan, given a goal specification and a description of primitive behaviors of a reactive agent. The description of primitive behaviors takes into account environment events that may interfere with the actions executed by the agent, causing nondeterministic effects.

One planning approach is to perform a forward-chaining search, enumerating the possible state sequences starting from a given initial state and checking the goal to enable actions that are on satisfactory sequences. Of course, there must be mechanisms for controlling the explosion of the search space in order to implement a planner that is reasonably efficient. One way is to use heuristics. For example, transition probabilities can be exploited to enumerate only the states that are most likely to occur during the execution (Drummond &

Bresina, 1990). Partial-order search techniques have also been proposed to limit the state explosion due to interleaving independent actions (Godefroid & Kabanza, 1991).

Backward-chaining search is also used to generate reactive plans. Its main advantage is that it automatically avoids exploring the part of the state space that is irrelevant to the goal (Schoppers, 1987). This is, however, applicable only to goals of reaching a final state, for which relevance of actions is determined by comparing their effects to propositions in the goal or in the preconditions of actions that are already proven relevant.

Reactive plans can also be computed by applying the theory of Markov decision processes (Dean et al., 1995). By associating probabilities to transitions, a nondeterministic state transition system becomes a stochastic automaton. A goal is expressed by specifying a reward for each state, that is, a real value expressing the desirability of being in the state. The planning problem is to find a policy (i.e., a reactive plan) that maps states to actions in order to maximize the expected future rewards. It is computed by defining a system of linear equations that relates state transition probabilities to state rewards and then by applying a dynamic programming algorithm to the linear equations. Simple achievement and maintenance goals are handled by using appropriate reward functions.

Another planning approach is to use a fixpoint calculation algorithm that was originally proposed in the area of control theory (Ramadge & Wonham, 1989). A transition system describing primitive behaviors is seen as a generator for a language of all strings that denote possible sequences of events. A goal is also represented by a state transition system that defines legal sequences of events. A reactive plan (also called a controller or supervisor) is represented by a state transition system and a feedback function that disables undesired events in some states. A language is said to be controllable if it is closed under environment executions, that is, whatever uncontrollable event is concatenated to a string in the language, one obtains another string in the language. The planning problem is to determine the largest controllable language satisfying the goal. This language is characterized by the largest fixpoint of an operator over languages, which is the cornerstone for an algorithm that computes controllers.

The last approach that we discuss is generating a reactive plan with theorem proving techniques (Pnueli & Rosner, 1989). In this context, a reactive plan (also called a reactive module) is modeled as a tree of states, that is, states that may be reached by a nondeterministic execution of a reactive module. Given a temporal logic formula expressing the desired behaviors for a reactive system, one constructs a tree-automaton that accepts trees that are models of the formula. Then, a reactive module is obtained from the tree-automaton. The approach of Abadi et al. (Abadi et al., 1989) is quite similar, but the input specification is a state transition system instead of a temporal formula.

Each of the above approaches has its limitations, virtues, and applications. For instance, the fixpoint calculation algorithm is limited by the fact that it requires the entire state transition system to reside in memory. In contrast, a search-based approach facilitates the expansion of states on the fly, without having to memorize the entire state space. The theorem proving approach is limited by the fact that it does not keep separate the information about actions and states in the synthesis process, which is impractical for debugging the specification and for controlling the state explosion problem. The decision theoretic approach can cope with the state explosion problem by using transition probabil-

ities and iterative refinement techniques, yet is limited to simple goals of achievement and maintenance.

3. Our Planning Method

We model the dynamics of a reactive agent in an environment by using a nondeterministic state transition system. A goal is specified by using a modal temporal logic formula. A reactive plan is described by a set of situation control rules (Drummond, 1989). Each situation control rule specifies an action to be executed in a corresponding situation. In addition, it specifies the possible successors.

A reactive plan is computed from a nondeterministic state transition system describing the behaviors of a reactive agent and a goal, by incrementally enumerating state sequences and checking the goal to obtain actions that guide the agent along satisfactory sequences. The approach is similar in spirit to the *anytime synthetic projection algorithm* of Drummond and Bresina (Drummond & Bresina, 1990), but our planner handles much more complex goals and deals with infinite behaviors. The manner in which goals are checked over state sequences is essentially model checking.

Model checking is widely used in the verification of temporal properties (Courcoubetis, Vardi, Wolper, & Yannakakis, 1992; Wolper, 1989). It has also been applied to synchronize reactive plans (Kabanza, 1995; Rao & Georgeff, 1993) and to control search in a classical planner (Bacchus & Kabanza, 1995). To check goals over state sequences, our planner generalizes the approach of Bacchus and Kabanza by defining mechanisms for handling uncontrollable actions, liveness goals, and time constraints.

A preliminary version of our method was published in (Barbeau et al., 1995). Herein, we detail the procedure for handling liveness goals, present an incremental planning algorithm, and evaluate the planner on a number of empirical problems. We believe that our method brings at least three contributions to the reactive plan synthesis problem.

- First, we describe an extension of the search-based planning approach to deal with complex goals for reactive agents. The ability of handling complex temporal goals is desirable for autonomous agents as long as they are expected to accomplish really useful tasks.

- Second, we describe an incremental planner algorithm for such goals. In AI planning terminology, our planner can be characterized as an *anytime planner* in the sense that it can be stopped at any time to yield a useful plan.

- Finally, our planning method contributes to a deeper understanding of the relations between the research fields outlined above by integrating the concepts of controllability, safety, and liveness, and the concern of handling complex goals.

Designing a planner algorithm requires formalisms for describing primitive behaviors, specifying goals, and representing reactive plans. It also requires a technique for computing reactive plans from a specification of primitive behaviors and a goal. These issues are addressed in this paper.

4. Specifying Primitive Behaviors

Primitive behaviors are described recursively by an initial world state w_0 and a transition function *succ* between world states. The function *succ* returns a list of actions that are executable in each world state, their corresponding duration, and successors. More specifically, $succ(w)$ returns a list $((a_1, d_1, W_1), \ldots, (a_n, d_n, W_n))$, where a_i is an action that is executable in w, d_i is a strictly positive real number denoting the duration of a_i in state w, and W_i the set of nondeterministic successors resulting from the execution of a_i in w.

Intuitively, the pair $(w_0, succ)$ describes the dynamics of a reactive agent in a given environment in the sense that, by applying *succ*, we can obtain all states the agent could be in at any time when executing its own actions under the influence of actions executed by other agents in the environment, which are implicitly represented via nondeterministic transitions. When a_i is executed in w, the interference with concurrent environment actions cause uncertainty about the outcome, which can be any state among those in W_i.

Example 1 *Figure 1 shows a partial description of a nondeterministic transition system that describes the behaviors of a reactive agent (some transitions are not shown due to space limitations). In this example, the reactive agent is a scheduler s that allocates a resource r to two processes p_1 and p_2 that compose the environment for s. Each state describes a set of propositions true in the corresponding situation. For instance,* $\mathsf{requesting}(p_1, r)$ *is interpreted as "process p_1 is requesting resource r." Each action lasts one time unit. The initial state has an empty set of propositions. A process can request the use of the resource at any time, provided that it is not already requesting or using it. Each time the scheduler has deallocated the resource, it enters a busy state in which the only possible action is to wait. A process using a resource never releases it unless the scheduler explicitly deallocates the resource.*

In reality, most interesting reactive scheduling problems involve much more than simply two processes and one resource. The interactions between processes and the scheduler may also be very complex. For instance, in a distributed computing application, the scheduler many denote a process that allocates the right to execute mutually exclusive code to other processes by changing appropriately shared variables. A scheduler may also denote a process that allocates peripheral devices (e.g., printers) to other processes. In a robotics application, a scheduler may denote a robot that fetches and delivers objects to different consumer processes, which may be other robots. For the sake of clarity, we discuss our planning method by using the above simple example, but Section 8 presents more complex problems.

For most of the part, the execution of a reactive agent has no definite ending point. For instance, it may be difficult to determine when a scheduler must stop monitoring requests and allocating resources. Thus, we view the execution of a reactive agent as a never terminating one, involving endless sensing and reaction to environment events. Accordingly, the execution of reactive plan by a reactive agent produces an infinite sequence of states. Nonetheless, finite executions can be simulated by using a terminal state in which the agent continually executes a *wait* action. The *wait* action is executable in all states that can be reached by executing other actions.

In practice, *succ* can be defined via some action representation formalism that allows succinct specifications. For example, our own implementation defines *succ* by a set of ADL

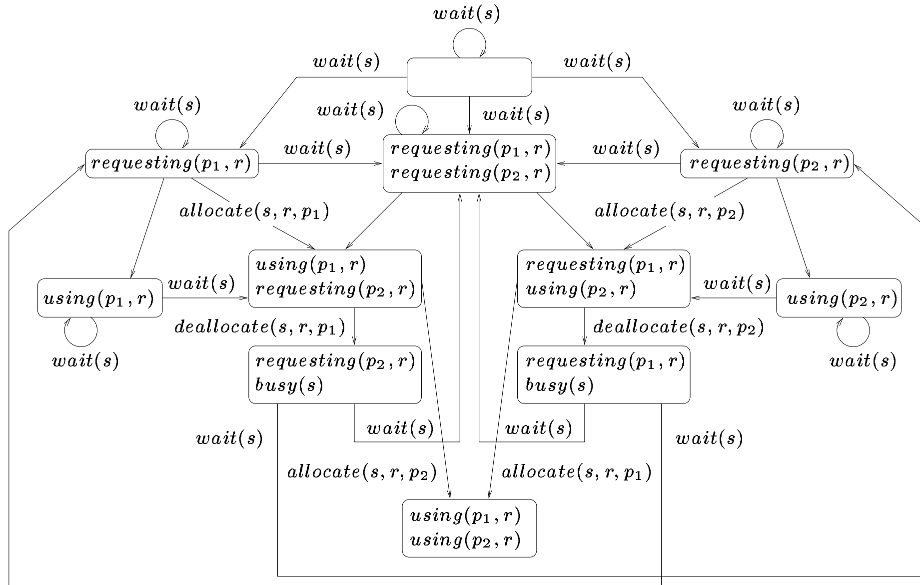


Figure 1: Partial description of a state transition system for a scheduler and two processes

operators.[1] Each operator describes the precondition, effects, duration, and controllability status for a schema of actions. Given a world state w, the planner computes $succ(w)$ by instantiating the operators to find enabled actions, which are instances of the operators. Obtained actions are then composed to take into account their possible simultaneous execution. The parallel composition of actions is achieved by interleaving them and then hiding uncontrollable actions to obtain nondeterministic transitions. The procedure for composing parallel actions is quite simple; its details are omitted due to space limitations.

Given a pair $(w_0, succ)$ that describes primitive behaviors for a reactive agent, one can specify a goal stating a particular desired behavior. Then, the role of a planner is to compute a reactive plan that selects, for each state, one action to be executed by the agent among the many possible primitive ones, in order to satisfy the given goal, whatever nondeterministic transitions are caused by environment actions. For example, in the scheduling domain, a goal would state that each time a process requests the resource, it must obtain the grant to use it within four time units. Then, a planner would automatically determine a particular strategy (i.e., reactive plan) for allocating resources that comply with this goal.

Roughly, our planning method is to view the pair $(w_0, succ)$ that describes primitive behaviors as a generator of many different reactive plans – possibly infinitely many – in the sense that, for any state w, $succ(w)$ represents different possible selections of actions. Each choice leads to a different reactive plan. From this perspective, the planning problem is essentially to determine the choices that satisfy a given goal. To solve it, one must first choose a goal language and a formal notion of goal satisfaction.

1 ADL (Action Description Language) is an extension of STRIPS operators (Pednault, 1989).

5. Specifying Goals

Modal temporal logics have been proven useful for specifying temporal properties in the verification of reactive systems (Manna & Pnueli, 1991). Formulas in such logics are interpreted over models that are infinite sequences of states. Thus, they are appropriate for specifying goals to our planner. To fix a context, we chose a particular modal temporal logic that can express time constraints. Specifically, we use Metric Temporal Logic (MTL) (Koymans, 1990). In MTL, goals of achievement are conveyed by the *eventually* and *until* modalities, whereas goals of maintenance are conveyed by the *always* and *until* modalities. Note that the *until* modality conveys both achievement and maintenance constraints.

5.1 Syntax

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

- every proposition 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 $\Diamond_{\sim t} f \equiv \mathit{true} \, U_{\sim t} \, f$ (eventually f). As usual, the language contains two atomic propositions *true* and *false*: *true* denotes valid statements ($\mathit{true} \equiv p \vee \neg p$), while *false* denotes inconsistent statements ($\mathit{false} \equiv p \wedge \neg p$).

The intuitive meaning of MTL formulas is captured by using the natural language interpretation for logical connectives and by noting that, when a time constraint "$\sim t$" is associated to a modal connective, the modal formula must hold within a time period satisfying the relation "$\sim t$". 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 closed time interval $[0, t]$ and satisfies f," $\Box_{\leq t} f$ as "always f on the closed time interval $[0, t]$," $\Diamond_{\leq t} f$ as "eventually f within t time units," or, equivalently, "eventually f on the closed time interval $[0, t]$," and $f_1 \, U_{\geq t} \, f_2$ as "f_1 until f_2 on the semi-open time interval $[t, \infty)$."

It is interesting to note that, when time is discrete, the ordering relations "$\leq t$" and "$\geq t$" can be used to define "$< t$" and "$> t$." For example, if time values are natural numbers, $(< t) \equiv (\leq t - 1)$ and $(> t) \equiv (\geq t + 1)$. It must also be noted that, although we assume real numbers for time constraints, models of formulas are discrete because they are sequences of states. In fact, the only advantage of using real numbers is that we can allow actions having durations that are real numbers. But, time values for goal constraints will be sampled only at discrete points that correspond to states.

5.2 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 $w_0 w_1 \ldots$;
- π is a binary function that evaluates propositions in a world state: $\pi(p, w)$ returns true if proposition p holds in world state w; and

- $\mathcal{D}$ is a time transition function: $\mathcal{D}(w_i, w_{i+1})$ returns a strictly positive real number, which is the time duration for the transition (w_i, w_{i+1}). In fact, $D(w_i, w_{i+1})$ denotes the duration taken by the action that causes state w_{i+1} from w_i.

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. We only give the semantic definitions for temporal connectives with the time constraints $\leq$ and $\geq$. The definitions for $<$ and $>$ are similar. For a state w_i in a model M with $d_i = \mathcal{D}(w_i, w_{i+1})$, a proposition p, formulas f_1 and f_2:

- $w_i \models p$ *iff* $\pi(p, w_i)$ returns true;
- $w_i \models \bigcirc_{\leq t} f_1$, *iff* $d_i \leq t$ and $w_{i+1} \models f_1$;
- $w_i \models \bigcirc_{\geq t} f_1$, *iff* $d_i \geq t$ and $w_{i+1} \models f_1$;
- $w_i \models \Box_{\leq t} f_1$, *iff*

  $d_i \leq t$ and $w_i \models f_1$ and $w_{i+1} \models \Box_{\leq (t-d_i)} f_1$; or
  $d_i > t$ and $w_i \models f_1$;

- $w_i \models \Box_{\geq t} f_1$, *iff*

  $d_i \leq t$ and $w_{i+1} \models \Box_{\geq (t-d_i)} f_1$; or
  $d_i > t$ and $t \neq 0$ and $w_{i+1} \models \Box_{\geq 0} f_1$; or
  $t = 0$ 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*

  $d_i \leq t$ and ($w_i \models f_2$ or ($w_i \models f_1$ and $w_{i+1} \models f_1 \, U_{\leq (t-d_i)} \, f_2$)); or
  $d_i > t$ and $w_i \models f_2$;

- $w_i \models f_1 \, U_{\geq t} \, f_2$, *iff*

  $d_i \leq t$ and $w_{i+1} \models f_1 \, U_{\geq (t-d_i)} \, f_2$; or
  $d_i > t$ and $t \neq 0$ and $w_{i+1} \models f_1 \, U_{\geq 0} \, f_2$; or
  $t = 0$ 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 $w_0 \models f$.

Example 2

1. *The formula* $\Diamond_{\geq 0} \Box_{\geq 0} \, p$ *states that* p *must eventually be made true and then maintained true thereafter.*
2. *The formula* $\Diamond_{\leq 10} \Box_{\leq 20} \, p$ *states that* p *must be made true within* 10 *time units and then maintained true for at least* 20 *time units.*

3. The formula

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

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 four 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, the execution eventually leads to cycles representing infinite behaviors that satisfy the goal.

5.3 Safety and Liveness Constraints

Any temporal goal can be seen as conveying a *safety constraint* and a *liveness constraint*. A safety constraint states that something bad must never happen during the execution, while a liveness constraint states that something good must eventually happen. Another way to understand this is that a safety constraint prohibits transitions to bad states, while a liveness constraint specifies transitions that must be eventually traversed.

Safety constraints are conveyed by MTL formulas of the form $\bigcirc_{\sim t}\, f_1$, $\Box_{\sim t}\, f_1$, for any ordering relation $\sim$, $f_1\, U_{\leq t}\, f_2$, and $f_1\, U_{<t}\, f_2$. Liveness constraints are conveyed by formulas of the form $f_1\, U_{\geq t}\, f_2$ or $f_1\, U_{>t}\, f_2$. However, a negated *always* conveys a liveness constraint, while a negated *until* conveys a safety constraint, as indicated by the following equivalences:

$$\neg(\Box_{\sim t}\, f) \;\equiv\; \Diamond_{\sim t}\, \neg f \qquad (4)$$
$$\neg(f_1\, U_{\sim t}\, f_2) \;\equiv\; (\Box_{\sim t}\, \neg f_2) \vee (\neg f_2\, U_{\sim t}(\neg f_1 \wedge \neg f_2)) \qquad (5)$$

for $\sim$ denoting any of the four ordering relations. This characterization of safety and liveness constraints is important in the description of our planning method. In fact, the difficult part of this method deals with the management of formulas that express liveness constraints.

5.4 Negative Normal Form

In general, a formula can involve different types of temporal connectives to convey both safety and liveness constraints. It is possible to check that any formula does not convey liveness constraints by first transforming it into an equivalent in *negative normal form*, that is, a form in which only propositions are negated. Then, one scans the formula to determine whether or not it contains a subformula of the form $f_1\, U_{\geq t}\, f_2$ or $f_1\, U_{>t}\, f_2$.

Any MTL formula can be transformed into an equivalent in negative normal form by propagating the negation connective inwards: negated temporal connectives are transformed by using Equation 4, Equation 5, and the equation $\neg \bigcirc_{\sim t}\, f \equiv (\bigcirc_{\sim t}\, \neg f) \vee \bigcirc_{\overline{\sim} t}\, true$, where $\overline{\sim}$ denotes the converse of the ordering relation $\sim$. Negated classical connectives ($\wedge$ and $\vee$) are transformed with standard distributive and De Morgan laws.

6. Reactive Plans

A reactive plan is represented by a set of *situation control rules* (SCR) (Drummond, 1989). In the original definition, an SCR simply maps a world state to a set of actions that can be executed simultaneously. Herein, only one reactive agent executes controllable actions in reaction to actions executed by environment agents. Hence, there is only one action for each SCR. Our representation extends the original definition of SCRs by defining plan states that are labeled by world states and a transition relation between plan states to allow the interpretation process to be biased by the recommendation from the previously executed SCR. This extension was initially proposed in (Kabanza, 1992). Other authors experimented with it in telescope control applications (Drummond, Swanson, & Bresina, 1994).

6.1 The Representation of Reactive Plans

A reactive plan is represented by a set of SCRs, where an SCR is a tuple of the form (n, w, a, N), such that:

- n is a number denoting a plan state;
- w is the world state labeling the plan state n and describing the situation in which the SCR is applicable;
- a is the action to be executed when w holds; and
- N is a set of integers denoting plan states that are nondeterministic successors of n when a is executed.

```
(STATE 0 WORLD nil ACTION (wait s) SUCCESSORS (0 1 4 10))
(STATE 1 WORLD ((requesting p2 r1)) ACTION (allocate s r1 p2) SUCCESSORS (2 9))
(STATE 2 WORLD ((using p2 r1)) ACTION (deallocate s r1 p2) SUCCESSORS (3 7))
(STATE 3 WORLD ((busy s)) ACTION (wait s) SUCCESSORS (0 1 4 10))
(STATE 4 WORLD ((requesting p1 r1)) ACTION (allocate s r1 p1) SUCCESSORS (5 8))
(STATE 5 WORLD ((using p1 r1)) ACTION (deallocate s r1 p1) SUCCESSORS (3 6))
(STATE 6 WORLD ((busy s) (requesting p2 r1)) ACTION (wait s) SUCCESSORS (1 11))
(STATE 7 WORLD ((busy s) (requesting p1 r1)) ACTION (wait s) SUCCESSORS (4 10))
(STATE 8 WORLD ((requesting p2 r1) (using p1 r1)) ACTION (deallocate s r1 p1) SUCCESSORS (6))
(STATE 9 WORLD ((requesting p1 r1) (using p2 r1)) ACTION (deallocate s r1 p2) SUCCESSORS (7))
(STATE 10 WORLD ((requesting p1 r1) (requesting p2 r1)) ACTION (allocate s r1 p1) SUCCESSORS (8))
(STATE 11 WORLD ((requesting p1 r1) (requesting p2 r1)) ACTION (allocate s r1 p2) SUCCESSORS (9))
```

Figure 2: A reactive plan for a process scheduler

Example 3 *Figure 2 shows a reactive plan for the scheduler partially represented in Figure 1 and a goal of eventually allocating resource r to each process requesting it. The formula expressing this goal is like goal 3 in Example 2, but time constraints for the eventually connectives are of the form "≥ 0." Propositions and action names are written in a Lisp-like notation.*

A plan can contain different SCRs with the same world state. In particular, this is the case with the SCR for states 10 and 11. Intuitively, the possibility of having different plan states labeled with the same world state amounts to extending the original world state space. Such an extension is necessary for many goals expressed by temporal formulas. For instance, without extending the original state space of Figure 1, it is not possible to write a reactive plan satisfying the goal of eventually allocating resource r to each requesting process. As a reactive plan is essentially a mapping from states to actions, it would allocate the resource to the same process in the world state $(requesting(p_1, r), requesting(p_2, r))$, causing the other process to never use the resource.

6.2 Executing a Reactive Plan

An agent executes a reactive plan by first fetching the SCR corresponding to the initial world state. By convention, this is the SCR with plan state 0. The corresponding world state describes the current situation before the agent executes any action. At any time, given the current SCR (n, w, a, N), the action a is executed and the SCR matching the resulting situation is determined from the successor plan states in N by getting an SCR (n', w', a', N') such that n' is in N and w' holds in the new situation. Then, a' is executed. The execution continues on endlessly by fetching an SCR matching the current situation based on the successor states given by the previously executed SCR. As finite executions are simulated by using a terminal state in which the agent performs a *wait* action endlessly, an SCR corresponding to such a state w has the form $(n, w, wait, (n))$.

6.3 Satisfactory Reactive Plans

Since nondeterministic transitions represent interference between an action executed by an agent and those performed by external processes in the environment, an agent executing a reactive plan cannot predict which nondeterministic successor will result from the current action. Hence, the sequence of states that will be generated by executing a reactive plan cannot be predicted either.

A reactive plan is deemed *completely satisfactory* if each sequence of states that may be generated by executing it satisfies the goal, whatever nondeterministic successor is selected for each action. In other words, as far as the environment executes its actions as predicted in the specified nondeterministic model of primitive behaviors, the execution of the reactive plan will satisfy the goal.

On the other hand, a reactive plan is said to be *partially satisfactory* if, by executing it, it may be possible to generate a sequence of states that does not satisfy the goal. In other words, the environment may execute an action diverting the execution to bad situations. It is often desirable to characterize the likelihood of such bad executions by associating nondeterministic transitions with probabilities. Such an extension is beyond the scope of the present paper, but the conclusion discusses some ideas about our future work in that direction.

7. Planner

Our planning approach is to view an initial world state w_0 and the function *succ* as a generator of graphs representing reactive plans. The planner searches for sequences of states satisfying the goal while taking into account nondeterministic transitions to obtain such a graph. The process for checking goals deals with sequences individually, but the overall planning process deals with graphs representing reactive plans. From a search process point of view, the violation of a safety constraint by a sequence of states always occurs on a finite prefix of the sequence. Thus, such a violation leads to dead ends during the search process of the planner. In contrast, a liveness constraint can only be violated by an infinite sequence of states. Such a violation leads to bad cycles during search. We start by sketching a planner algorithm for goals that only express safety constraints. Then, building on it, we describe a more general planner algorithm that handles any arbitrary MTL goal formula.

7.1 Planning for Safety Constraints

The process we use to check goal formulas is called *goal progression* to stress the fact that this process consists in *progressing* the goal formula forward over state sequences generated by *succ*. The main idea is to label each state with a formula that must be satisfied by each sequence starting from this state. The initial state is labeled with the input goal. Then, given any current state and its label, the label of a successor produced by *succ* is obtained by applying the algorithm described in Figure 3.

The input of the goal progression algorithm is an MTL formula f, a state w, a real number d denoting the duration of the transition from w to a successor w', and a function π that evaluates propositions in states. The output is a formula expressing the constraint that would have to be satisfied by a sequence from w' in order that the entire sequence from w satisfies f. As with the semantic definition of MTL, we give only the cases corresponding to temporal connectives with the time constraints $\leq$ and $\geq$. The cases for $<$ and $>$ are similar. It can be easily shown that the algorithm satisfies the following theorem (a proof is given in Appendix A.1 and it is based on the observation that *Progress-goal* is merely a rewriting of the MTL interpretation rules given in Section 5.2).

Theorem 1 *Let $w_0 w_1 \ldots$ denote any infinite sequence of world states, d_i the duration of the transition from w_i to w_{i+1}, and π a function that evaluates propositions in states. Then, for any state w_i and MTL formula f, $w_i \models f$* if and only if *$w_{i+1} \models$ Progress-goal(f, w_i, d_i, π).*

The phrasing of this theorem intentionally makes the satisfaction of f at w_i depend purely on the satisfaction of the progressed formula at w_{i+1}. This simply reflects the MTL interpretation rules which are also defined recursively. In order to effectively check a formula over a sequence of states, we must combine *Progress-goal* with a mechanism that systematically detects points where safety constraints of the progressed goal are violated and a mechanism that detects points where liveness constraints are violated. In this subsection, we discuss the case of safety constraints. Then, the next subsection generalizes to the case of liveness constraints.

For safety constraints, the violation of a maintenance requirement conveyed by *always* and *until* connectives is detected when evaluating propositions (case 2 in the progression algorithm). The violation of a deadline for eventually achieving a subgoal conveyed by the

$Progress\text{-}goal(f,w,d,\pi)$
1. **case** f
2. p (p a proposition): if $\pi(p,w)$ then *true* else *false*;
3. $\neg f_1$: $\neg Progress\text{-}goal(f_1, w, d, \pi)$;
4. $f_1 \wedge f_2$: $Progress\text{-}goal(f_1, w, d, \pi) \wedge Progress\text{-}goal(f_2, w, d, \pi)$;
5. $f_1 \vee f_2$: $Progress\text{-}goal(f_1, w, d, \pi) \vee Progress\text{-}goal(f_2, w, d, \pi)$;
6. $\bigcirc_{\leq t} f_1$: if $d \leq t$ then f_1 else *false*;
7. $\bigcirc_{\geq t} f_1$: if $d \geq t$ then f_1 else *false*;
8. $\Box_{\leq t} f_1$: if $d \leq t$ then $Progress\text{-}goal(f_1, w, d, \pi) \wedge \Box_{\leq (t-d)} f_1$
   else $Progress\text{-}goal(f_1, w, d, \pi)$;
9. $\Box_{\geq t} f_1$: if $d \leq t$ then $\Box_{\geq (t-d)} f_1$
   else if $t = 0$ then $Progress\text{-}goal(f_1, w, d, \pi) \wedge \Box_{\geq 0} f_1$;
   else $\Box_{\geq 0} f_1$;
10. $f_1 \, U_{\leq t} \, f_2$: if $d \leq t$ then $Progress\text{-}goal(f_2, w, d, \pi) \vee$
    $(Progress\text{-}goal(f_1, w, d, \pi) \wedge \, f_1 \, U_{\leq (t-d)} \, f_2)$
    else $Progress\text{-}goal(f_2, w, d, \pi)$;
11. $f_1 \, U_{\geq t} \, f_2$: if $d \leq t$ then $f_1 \, U_{\geq (t-d)} \, f_2$
    else if $t = 0$ then $Progress\text{-}goal(f_2, w, d, \pi) \vee$
    $(Progress\text{-}goal(f_1, w, d, \pi) \wedge f_1 \, U_{\geq 0} \, f_2)$
    else $f_1 \, U_{\geq 0} \, f_2$

Figure 3: Goal progression algorithm

until connective is detected when the time bound decreases to 0 before the eventuality is satisfied (case 10 in the progression algorithm). This can be illustrated by a simple case in which the current state is labeled with $f_1 \, U_{\leq t} \, f_2$, where f_2 contains no temporal connectives.

As long as f_2 is not satisfied, the goal is progressed by decreasing the time bound by the action duration. Should f_1 be violated or the time bound reach 0, the progression returns *false*. Since the time bound is decreased by the action duration in each state, the goal of a current state differs from that of its descendants. Thus, if we do not expand states labeled *false*, a cycle cannot be formed unless f_2 is satisfied. When f_2 is satisfied, the progression returns *true*, so that we could obtain a goal equal to that of an ancestor. This is the only way a cycle can be formed. This means that any infinite sequence unwound from a path terminated by a cycle satisfies a goal that involves only safety constraints.[2] A reactive plan is obtained from a graph composed of such paths.

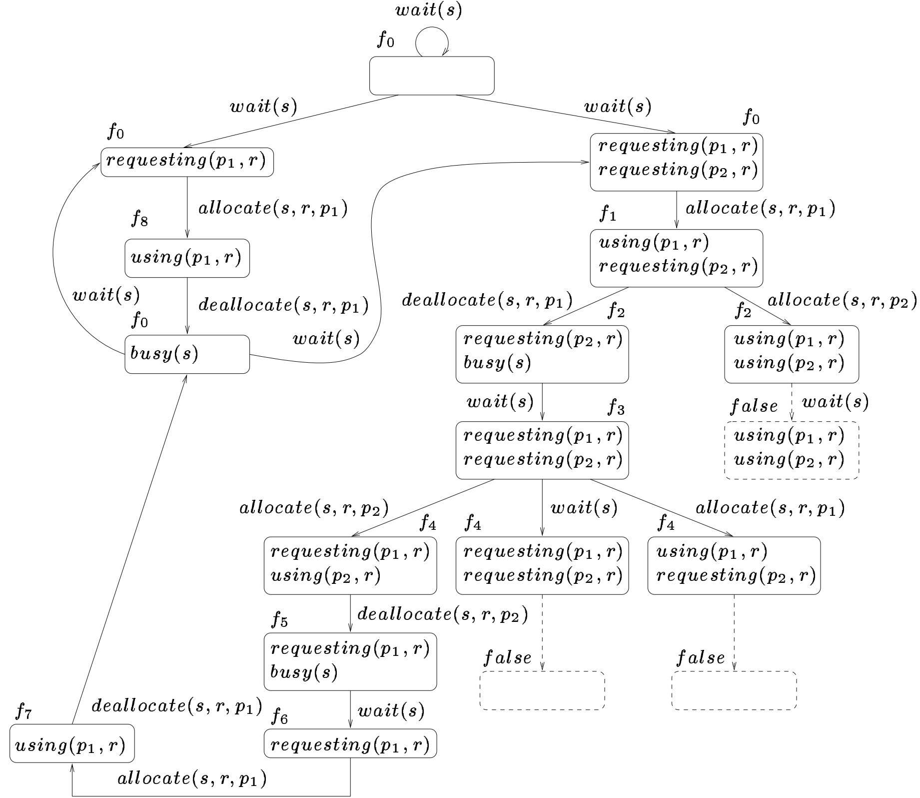
Given a goal formula f that does not involve liveness constraints, a simple planner algorithm is:

1. Generate a graph of states labeled with goal formulas by starting from a state s_0 that is w_0 labeled with f. Then, for every state s that is w labeled with $f \neq false$, for every $(a, d, W) \in succs(w)$, and for every w' in W, create a successor s' of s that consists of w' labeled with a formula f' given by the equation $f' = Progress\text{-}goal(f, w, d, \pi)$; the transition (s, s') is labeled a.

2 Later, we give proofs for more general results that involve both safety and liveness constraints (Theorems 2 and 3). The particular case of safety constraints can be easily derived from these proofs.

2. A reactive plan is easily read from the graph obtained by the previous step.

In Step 1, states labeled *false* are not expanded because, according to Theorem 1, any sequence containing such a state cannot satisfy the goal. Step 2 is more thoroughly explained later when we generalize the planner algorithm to liveness constraints. In fact, we will further see that the two steps can be carried on simultaneously by extracting the plan on the fly to obtain an incremental planner algorithm.



Goals:

$$f_0 = \Box_{\geq 0}(\ \neg(using(p_1, r) \wedge using(p_2, r)) \wedge$$
$$(requesting(p_1, r) \rightarrow \Diamond_{\leq 4}\, using(p_1, r)) \wedge$$
$$(requesting(p_2, r) \rightarrow \Diamond_{\leq 4}\, using(p_2, r)))$$

$$f_1 = f_0 \wedge \Diamond_{\leq 3}\, using(p_1, r) \wedge \Diamond_{\leq 3}\, using(p_2, r)$$

$$f_2 = f_0 \wedge \Diamond_{\leq 2}\, using(p_2, r) \qquad f_3 = f_0 \wedge \Diamond_{\leq 1}\, using(p_2, r)$$

$$f_4 = f_0 \wedge \Diamond_{\leq 3}\, using(p_1, r) \wedge \Diamond_{\leq 0}\, using(p_2, r)$$

$$f_5 = f_0 \wedge \Diamond_{\leq 2}\, using(p_1, r) \qquad f_6 = f_0 \wedge \Diamond_{\leq 1}\, using(p_1, r)$$

$$f_7 = f_0 \wedge \Diamond_{\leq 0}\, using(p_1, r) \qquad f_8 = f_0 \wedge \Diamond_{\leq 3}\, using(p_1, r)$$

Figure 4: A partial description of a graph obtained by progressing a goal

Example 4 *Figure 4 shows a partial description of the graph obtained by progressing Goal 3 in Example 2 for the function succ graphically represented by Figure 1. The initial state*

has an empty set of propositions and that all actions have a duration of one time unit. The goal is noted f_0 and labels the initial state. The goals labeling the other states are obtained as follows: for each transition (w, w'), the goal f' labeling w' is obtained from f labeling w by the equation $f' =$ Progress-goal$(f, w, 1, \pi)$. The progression of f_2 through the state $(\mathit{using}(p_1, r), \mathit{using}(p_2, r))$ yields false because the subformula $\neg(\mathit{using}(p_1, r) \wedge \mathit{using}(p_2, r))$ is violated. The progression of f_4 through $(\mathit{requesting}(p_1, r), \mathit{requesting}(p_2, r))$ also yields false because the time bound for $\diamond_{\leq 0}\, \mathit{using}(p_2, r)$ is 0, but $\mathit{using}(p_2, r)$ is not satisfied. For the same reason, the progression of f_4 through $(\mathit{using}(p_1, r), \mathit{requesting}(p_2, r))$ yields false, for any action. It can be easily checked that any infinite sequence unwound from a path terminated by a cycle satisfies f_0.

If a goal involves liveness constraints, then cycles that do not satisfy it can be formed. This is because liveness constraints are conveyed by an *unbounded-time until* (i.e., a formula of the form $f_1\, U_{\geq t}\, f_2$ or $f_1\, U_{>t}\, f_2$). With such a formula, the progression process sooner or later reaches a state at which t is decreased to 0. From that point, if f_1 is not violated and f_2 is not satisfied, the progressed goal never changes. Hence, the progression process may reach a previous state with the same goal to form a cycle, but the cycle may not contain a state satisfying f_2.[3]

Example 5 *Figure 5 shows progressions for a goal similar to that in the previous example, except that the time constraint for the eventualities is "≥ 0." Clearly, many infinite sequences unwound from paths terminated by cycles do not satisfy f_0. For example, the leftmost cycle involves two states labeled $f_2 = f_0 \wedge \diamond_{\geq 0}\, \mathit{using}(p_2, r)$, but no state along this cycle satisfies the proposition $\mathit{using}(p_2, r)$.*

7.2 Handling Liveness Constraints

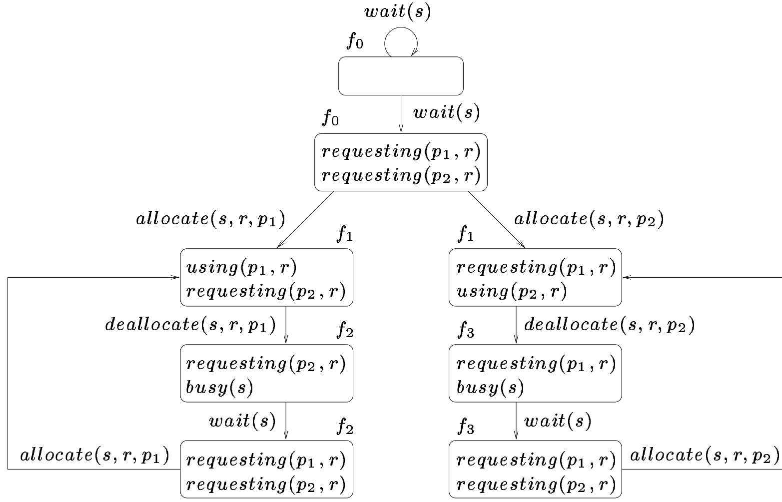
The goal progression process fails to detect the violation of liveness constraints because such constraints can only be violated by infinite sequences or cycles. This problem is tackled by keeping track of *unbounded-time until* formulas to check that they are eventually satisfied along cycles.[4] As the characterization of safety and liveness constraints with respect to temporal connectives depends on whether or not the connectives are in the scope of the negation connective, we simplify the planner algorithm by transforming the input goal into an equivalent in *negative normal form*. Thereafter, the goal progression process preserves this form because the function *Progress-goal* never introduces a negative connective symbol.

7.2.1 Decomposing Goals into Disjunctive Normal Form

In order to check liveness constraints, we need to transform every progressed goal into *disjunctive normal form*, that is, an equivalent formula of the form $g_1 \vee \cdots \vee g_n$, such that

3 Such a situation would not occur with a constraint of the form "$\leq t$" or "$< t$" because, in this case, t is decremented by the action duration for every traversed transition, as long as f_2 is not satisfied. Hence, a previously encountered formula cannot be met again on the same path unless f_2 is satisfied in-between.

4 An alternative and simpler approach for handling liveness constraints would be to approximate infinite sequences by sufficiently long finite sequences. This can be done by associating states with time stamps. In this case, since time never decreases, cycles cannot be formed. However, it may be difficult to determine a sufficient length for sequences of timed states. Moreover, the association of states with time stamps may increase the size of the state space by introducing many states that differ only in time stamps.



Goals:

$$\begin{aligned} f_0 = \Box_{\geq 0}(\ & \neg(\mathit{using}(p_1,r) \wedge \mathit{using}(p_2,r)) \wedge \\ & (\mathit{requesting}(p_1,r) \rightarrow \Diamond_{\geq 0}\,\mathit{using}(p_1,r)) \wedge \\ & (\mathit{requesting}(p_2,r) \rightarrow \Diamond_{\geq 0}\,\mathit{using}(p_2,r))) \end{aligned}$$

$$f_1 = f_0 \wedge \Diamond_{\geq 0}\,\mathit{using}(p_1,r) \wedge \Diamond_{\geq 0}\,\mathit{using}(p_2,r)$$

$$f_2 = f_0 \wedge \Diamond_{\geq 0}\,\mathit{using}(p_2,r) \quad f_3 = f_0 \wedge \Diamond_{\geq 0}\,\mathit{using}(p_1,r)$$

Figure 5: A tentative expansion of states with liveness goals

each g_i is a conjunction of the form $h_1 \wedge \cdots \wedge h_{m_i}$, where each h_j is a literal or a formula whose main connective is $\bigcirc$, $\Box$, or U. Any MTL formula in negative normal form can be transformed into an equivalent in disjunctive normal form by using standard distributive laws between the connectives $\wedge$ and $\vee$. As further explained below, this transformation allows us to label states with goals that are conjunctions of formulas so that we can easily determine the eventualities that must be progressed.

Let s be a current state during search, w the corresponding world state, and f the corresponding goal. If we followed the planner algorithm sketched in Section 7.1, given a successor w' of w, we would generate a successor s' of s whose world state is w' and goal is $f' = \mathit{Progress\text{-}goal}(f, w, d, \pi)$. This time, instead of proceeding in this way, we transform f' into an equivalent formula f'' in disjunctive normal form and then check the satisfaction of each disjunct of f'' separately. This is done by creating as many copies of w' as we have disjuncts of f'', each copy being labeled with a different disjunct. All these copies become nondeterministic successors of s.

To be more specific, we introduce the notion of an *extended state*, that is, as a world state labeled with a goal formula and a set of *unbounded-time until* formulas. By convention, we denote world states by the letter w, possibly with a subscript, and extended states by the letter s, also possibly with a subscript. Given an extended state s, its corresponding

world state is noted $s.world$, the goal $s.goal$, and the set of *unbounded-time until* formulas $s.eventualities$, because it represents formulas that must be *eventually* satisfied.

Given an initial world state w_0, a transition function *succ*, a goal formula f, and a function π for interpreting propositions, we have several initial extended states that are obtained as follows. The input formula f is transformed into an equivalent formula f' in negative normal form. Then, f' is transformed into an equivalent formula $f'' = f_1 \vee \cdots \vee f_n$ in disjunctive normal form. This gives n initial states s_i, such that $s_i.world = w_0$, $s_i.goal = f_i$, and $s_i.eventualities = \emptyset$.[5]

The algorithm for generating successors of extended states is called *Expand* (see Figure 6). Its input is an extended state s, a transition function *succ* for world states, and a function π that evaluates propositions in world states. It generates transitions from state s by applying *succ* to $s.world$ to obtain successor world states, *Progress-goal* to obtain their respective goal labels, and *Progress-eventualities* to obtain their set of eventualities labels. Step 2 handles the nondeterminism conveyed by *succ*. Step 4 handles goal decomposition by introducing an additional level of nondeterminism.

```
Expand(s, succ, π) {
1.  for each (a, d, W) in succ(s.world) {
2.    g = Progress-goal(s.goal, s.world, d, π);
3.    for each disjunct f in disjunctive-normal-form(g)
4.      for each w in W {
5.        s' := create-new-state(); s'.world := w;
6.        s'.goal := f; s'.eventualities := Progress-eventualities(s, s', d, π);
7.        generate-transition(s, a, s');}}}
```

Figure 6: Algorithm for expanding a state

Figure 7 illustrates graphically the expansion of a state s. Part (a) shows the successors of $s.world$ that would be obtained by applying $succ(s.world)$. Part (b) shows the successors of s that would be created by $Expand(s)$. Each successor of s obtained by applying an action a_i is labeled by a goal f_{ij}, where j corresponds to the jth disjunct of the disjunctive normal form of $Progress\text{-}goal(s.goal, s.world, d_i, \pi)$, given d_i the duration of the transition corresponding to a_i. The sets E_{ij} contain eventualities that are progressed from s. The other annotations in the figure explain the interpretation for each level of nondeterminism. They are discussed further later when defining the planner algorithm.

7.2.2 Progressing Eventualities

By construction, for any extended state s, $s.goal$ is a conjunction of the form $f_1 \wedge \cdots \wedge f_n$. A subformula is required to hold in a state s only if it is a conjunct f_i of $s.goal$. Eventualities are progressed only if they are conjuncts of goals labeling states. For example, let us consider a state s with $s.goal = \Box_{\geq 0}(p \rightarrow \Diamond_{\geq 0}\, q)$, where p and q are propositions. Although $\Diamond_{\geq 0}\, q$ is a subformula of $s.goal$, it would not be progressed in a set of eventualities because

5 We use the symbol $\emptyset$ to represent the empty set.

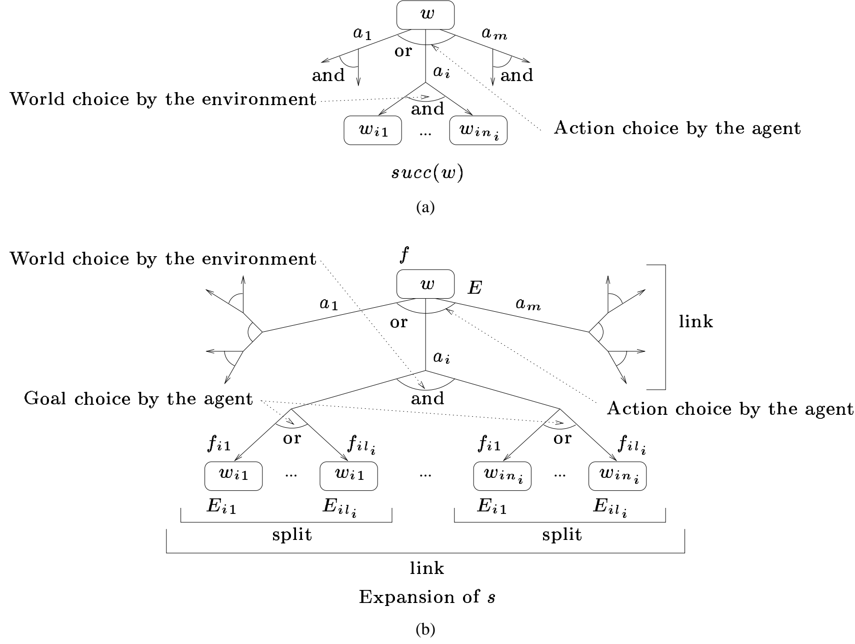


Figure 7: Expansion of an extended state s $(s.world = w, s.goal = f, s.eventualities = E)$

it is not a conjunct of $s.goal$. This is coherent with the fact that $s.goal$ only requires that $\diamond_{\geq 0}\, q$ must hold only on sequences rooted from states satisfying p, which is not the case for s. If there is a descendant s' of s that satisfies p, then, since $\Box_{\geq 0}(p \rightarrow \diamond_{\geq 0}\, q)$ is progressed through all descendants of s, any state s'' that is a successor of s' must be such that $s''.goal = \diamond_{\geq 0}\, q \wedge \Box_{\geq 0}(p \rightarrow \diamond_{\geq 0}\, q)$. The conjunct $\diamond_{\geq 0}\, q$ reflects the fact that, this time q must eventually hold on sequences rooted from s''. Hence, $\diamond_{\geq 0}\, q$ will be progressed in the set of eventualities.

Progress-eventualities(s, s', d, π)

1. if $s.eventualities$ is empty, then return the set of formulas f of the form $f'\, U_{\geq 0}\, f''$ or $f'\, U_{>0}\, f''$ that are conjuncts of $s.goal$ (i.e., if we note $s.goal = f_1 \wedge \cdots \wedge f_m$ then f must be one of the f_i), such that *Locally-entailed*(*Progress-goal*$(f'', s.world, d, \pi), s'$) returns false;
2. otherwise, return the set of formulas obtained from $s.eventualities$ by removing formulas of the form $f'\, U_{\geq 0}\, f''$ or $f_1\, U_{>0}\, f_2$ such that *Locally-entailed*(*Progress-goal*$(f'', s.world, d, \pi), s'$) returns true;

Figure 8: Eventuality progression algorithm

This is the image. In reality, we do not need to check new eventualities at every step. As described in Figure 8, we progress a set of eventualities from a state s to a successor s' by removing formulas $f'\, U_{\geq 0}\, f''$ and $f'\, U_{>0}\, f''$ such that *Progress-goal*$(f'', s.world, d, \pi)$ is *locally entailed* by s' (Step 2). Only when we reach a state with an empty set of eventualities, do we compute a new set of eventualities (Step 1). The formula returned by

Progress-goal$(f'', s.world, d, \pi)$ expresses the requirement that would have to be satisfied by the sequence from s' in order to have the entire sequence from s satisfy f''.

The *local entailment* of a formula f by an extended state s is a restricted form of the logical entailment of f by $s.goal$. A formula f is locally entailed by an extended state s if the interpretation of f yields true, assuming that a literal holds if it is a conjunct of $s.goal$, and a modal formula g holds if there is a modal formula g' that is a conjunct of $s.goal$ and differs with g at most by the time constraint of their main temporal connective; the time constraints must be such that g' implies g (see Figure 9).

For a *next* or *until* connective, the implication relation between time constraints is noted by *interval-implies-next-until*. For an *always* connective, it is noted by *interval-implies-always*. The definition of these relations follows trivially from the semantic definition of MTL formulas, by observing that time constraints actually denote time interval over which the corresponding modalities must hold. Thus, *interval-implies-next-until*$(\leq t', \leq t)$ returns true if $t' \leq t$ and *interval-implies-next-until*$(\leq t', < t)$ returns true if $t' < t$, while *interval-implies-always*$(\leq t', \leq t)$ returns true if $t' \geq t$ and *interval-implies-always*$(< t', \leq t)$ returns true if $t' > t$. The other cases are similar.

When *Progress-goal*$(f'', s.world, d, \pi)$ is *locally entailed* by s', this means that any sequence from s going through s' that violates f'' also violates $s'.goal$. As the progression of $s'.goal$ would lead to a state having the goal *false* whenever f'' is violated, we do not have to progress $f' \, U_{\geq 0} \, f''$ or $f' \, U_{>0} \, f''$ from $s'.eventualities$ to $s.eventualities$. Our notion of local entailment is clearly weaker than logical entailment: when *Locally-entailed*(f, s) returns true, this means that f is a logical consequence of $s.goal$, but the converse is not necessarily true. As will be proven later, this poses no problem with the completeness of our planner algorithm because progressed eventualities are derived from progressed goals and the goal progression process keeps intact subformulas that are relevant to the evaluation of local entailment.

Locally-entailed(f, s)
1. **case** f
2. p (p a proposition): $p = true$ or p is a conjunct of $s.goal$;
3. $\neg f_1$: $\neg$*Locally-entailed*(f_1, s);
4. $f_1 \wedge f_2$: *Locally-entailed*$(f_1, s) \wedge$ *Locally-entailed*(f_2, s);
5. $f_1 \vee f_2$: *Locally-entailed*$(f_1, s) \vee$ *Locally-entailed*(f_2, s);
6. $\bigcirc_{\sim t} f_1$: $\bigcirc_{\approx t'} f_1$ is a conjunct of $s.goal$
   for some $\approx t'$ such that *interval-implies-next-until*$(\approx t', \sim t)$ returns true;
7. $\Box_{\sim t} f_1$: $\Box_{\approx t'} f_1$ is a conjunct of $s.goal$
   for some $\approx t'$ such that *interval-implies-always*$(\approx t', \sim t)$ returns true;
8. $f_1 \, U_{\sim t} \, f_2$: $f_1 \, U_{\approx t'} \, f_2$ is a conjunct of $s.goal$
   for some $\approx t'$ such that *interval-implies-next-until*$(\approx t', \sim t)$ returns true.

Figure 9: Local entailment algorithm

Example 6 *Given the formulas* $f_1 = \neg p_5$, $f_2 = (p_1 \wedge (\Box_{\geq 0} p_2) \wedge p_3 \, U_{\leq 5} \, p_4)$, *and* $f = f_1 \, U_{\geq 0} \, f_2$, *let us consider a state* s *such that* $s.eventualities = \{f\}$, *and* $s.goal = p_1 \wedge \Box_{\geq 0} p_2 \wedge p_3 \, U_{\leq 2} \, p_4$.

Formula f_2 *is locally entailed by* s *because all its conjuncts are locally entailed by* s. *Specifically,* p_1 *is a conjunct of* $s.goal$; $\Box_{\geq 0} p_2$ *is a conjunct of* $s.goal$; *and* $p_3 \, U_{\leq 2} \, p_4$ *is a conjunct of* $s.goal$, *which entails the local entailment of* $p_3 \, U_{\leq 5} \, p_4$. *Hence, for any successor* s' *of* s, $s'.eventualities$ *must be* $s.eventualities$ *minus* $\{f\}$, *that is, the empty set. This does not mean that* f *is effectively satisfied by all sequences rooted from* s'. *Actually, we cannot be sure of that since, when* s' *is created during search, it is not expanded yet, that is, we do not know yet what the sequences from* s' *are. Rather, this means that since* $s.goal$ *is a conjunction of formulas including* p_1, $\Box_{\geq 0} p_2$, *and* $p_3 \, U_{\leq 2} \, p_4$, *then if* f *were violated by a sequence from* s, *the* goal progression process *eventually causes a state to be labeled* false.

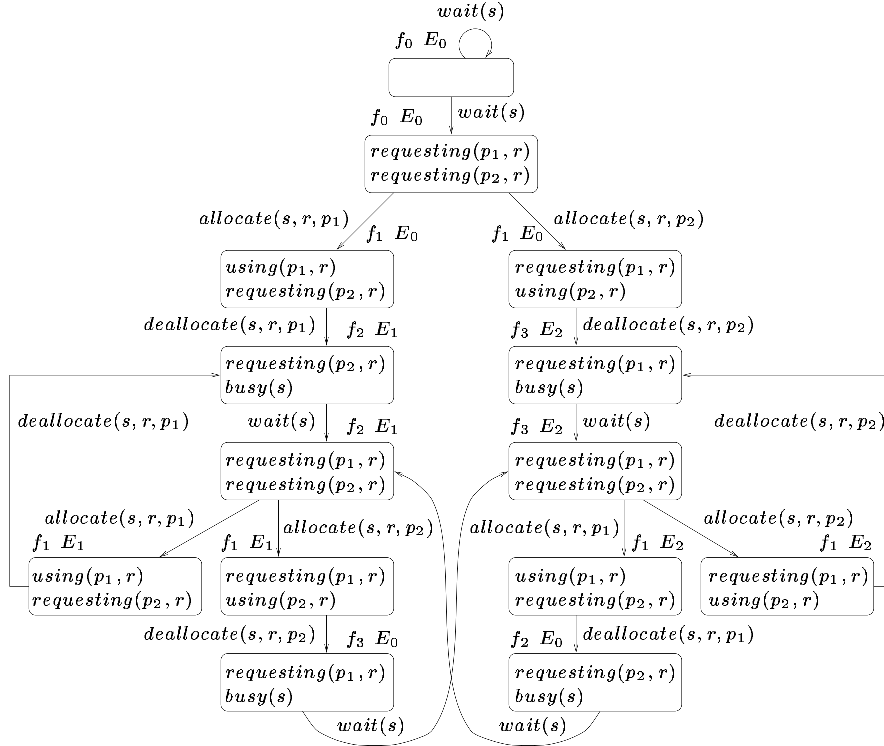
From the definition of *Progress-goal* and *Expand* one can see that, for any state s on a path obtained by applying *Expand*, the conjuncts forming $s.goal$ include eventualities that have occurred in ancestor states and that have not been satisfied meanwhile. As will be discussed in Section 7.3, from this observation, it can be shown that when a cycle contains an empty set of eventualities but no state with the goal *false*, then the infinite sequence obtained by unwinding a path terminated by the cycle satisfies the goal labeling the root of this path.

Example 7 *Figure 10 shows a partial description of the graph produced by* Expand, *for the transition function succ represented by Figure 1 and goal* f. *Again, we assume that all actions last one time unit and that the initial state has an empty set of propositions. As the main connective of* f_0 *is* $\Box_{\geq 0}$, *there is only one disjunct, hence only one initial state. Moreover, for this particular example, the disjunctive normal form of any goal progressed thereafter is the goal itself, for any state. Hence, there is no goal decomposition. The difference between this example and the previous attempt in Figure 5 is that the new graph is equipped with a mechanism for determining satisfactory cycles.*

Example 8 *We previously explained the generation of the graph in Figure 4 without taking into account eventualities and without decomposing goals. Actually,* Expand *would generate the same graph because there is no goal decomposition for this example. However, the subformula in the scope of the* always *connective would be in negative normal form. In addition, each state would be labeled with an empty set of eventualities.*

Example 9 *Figure 11 shows an example with a goal formula causing nondeterministic decompositions, for an artificial domain.*

Nondeterministic goal decompositions occur only for formulas that convey nondeterminism about the different strategies to satisfy them. This is the case with Example 9. Another more familiar example, is the goal of reaching a state satisfying p and then maintaining true thereafter. Such a goal is expressed by the formula $\Diamond_{\geq 0} \Box_{\geq 0} p$. The progression of this formula through a state that does not satisfy p yields the same formula. But, in a state satisfying p, it yields $\Box_{\geq 0} p \vee \Diamond_{\geq 0} \Box_{\geq 0} p$. This would cause two different states:



<u>Goals:</u>

$$
\begin{aligned}
f = \Box_{\geq 0}(\ & \neg(using(p_1, r) \wedge using(p_2, r)) \wedge \\
& (requesting(p_1, r) \rightarrow \Diamond_{\geq 0}\, using(p_1, r)) \wedge \\
& (requesting(p_2, r) \rightarrow \Diamond_{\geq 0}\, using(p_2, r)))
\end{aligned}
$$

$f_0 =$ disjunctive normal form of f

$f_1 = f_0 \wedge \Diamond_{\geq 0}\, using(p_1, r) \wedge \Diamond_{\geq 0}\, using(p_2, r) \quad f_2 = f_0 \wedge \Diamond_{\geq 0}\, using(p_2, r) \quad f_3 = f_0 \wedge \Diamond_{\geq 0}\, using(p_1, r)$

<u>Eventuality sets:</u>

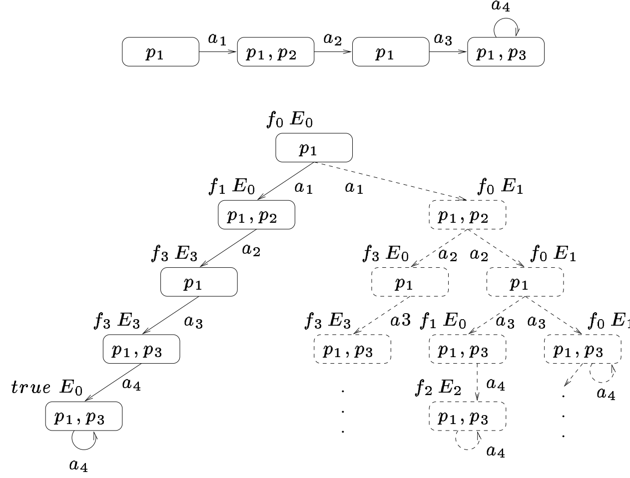
$$E_0 = \emptyset \qquad E_1 = \{\Diamond_{\geq 0}\, using(p_2, r)\} \qquad E_2 = \{\Diamond_{\geq 0}\, using(p_1, r)\}$$

Figure 10: An expansion of states with liveness goals

one labeled by $\Box_{\geq 0}\, p$, the other by $\Diamond_{\geq 0} \Box_{\geq 0}\, p$. Intuitively, the disjunct $\Box_{\geq 0}\, p$ accounts for the possibility that henceforth p cannot be falsified by the environment (therefore we must maintain p forever), while the other disjunct accounts for the possibility that p might be later falsified by the environment (hence, we must keep on progressing the eventuality to re-establish p after it has been falsified).

7.3 Properties of Sequences Generated by *Expand*

For every sequence of extended states $s_0 s_1 \ldots$ produced by *Expand*, we have a corresponding sequence of world states $w_0 w_1 \ldots$, where $w_i = s_i.world$, for $i \geq 0$. We extend the interpretation of MTL formulas to sequences of extended states produced by *Expand* as follows:



<u>Goals:</u>

$$f_0 = p_1\, U_{\geq 0}(\Diamond_{\geq 0}\, p_2 \wedge \Diamond_{\geq 0}\, p_3)$$
$$f_1 = \Diamond_{\geq 0}\, p_2 \wedge \Diamond_{\geq 0}\, p_3$$
$$f_2 = \Diamond_{\geq 0}\, p_2 \qquad f_3 = \Diamond_{\geq 0}\, p_3$$
$$E_0 = \emptyset \qquad E_1 = \{f_0\}$$
$$E_2 = \{\Diamond_{\geq 0}\, p_2\} \qquad E_3 = \{\Diamond_{\geq 0}\, p_3\}$$

Figure 11: A path of world states and a corresponding graph of extended states

we say that a sequence of extended states $s_0 s_1 \ldots$ satisfies a formula f (noted $s_0 \models f$) if the corresponding sequence of world states $w_0 w_1 \ldots$ satisfies f (i.e., $w_0 \models f$). We have the following theorem (a proof is given in Appendix A.2).

Theorem 2 *For any path terminated by a cycle $s_0 s_1 \ldots s_j \ldots s_j$ and produced by* Expand*, if for all $i \geq 0$, $s_i.goal \neq$ false, and there exists $k \geq j$ such that $s_k.eventualities = \emptyset$, then for any state s_i on the infinite sequence obtained from the path by unwinding the cycle, we have $s_i \models s_i.goal$.*

The premises given by Theorem 2 are sufficient for satisfying the goal formula, but not necessary. For example, in Figure 11, any infinite sequence starting from p_1 in the graph of extended states satisfies f_0, but the paths drawn in dashed lines are terminated by cycles that do not contain a state with an empty set of eventualities. In other words, these paths satisfy the conclusion of Theorem 2, but not its premises. Nonetheless, as stated by the following theorem, the graph generated by *Expand* must contain at least one path satisfying both the conclusion and the premises of Theorem 2. In Figure 11, this is the path drawn in solid lines. A proof of this theorem is given in Appendix A.3.

Theorem 3 *For any path terminated by a cycle $w_0 w_1 \ldots w_l \ldots w_l$, and produced by succ, the infinite sequence obtained by unwinding the cycle satisfies an MTL formula f* if and only if *the graph produced by* Expand *contains a path terminated by a cycle $s_0 s_1 \ldots s_j \ldots s_j$ such*

that (a) for all $i \geq 0$, $s_i.goal \neq$ false; (b) there exists $k \geq j$ such that $s_k.eventualities = \emptyset$; and (c) $s_0.world = w_0$ and, for any $s_{i'}$ and w_i, if $s_{i'}.world = w_i$, then $s_{i'+1}.world = w_{i+1}$.

7.4 Extracting a Reactive Plan

Formalizing a process for extracting a reactive plan from such paths requires some definitions.

Definition 1 (Splits and Links) *A* split *for a given action and world state is the set of transitions from an extended state, corresponding to the same action and leading to extended states composed of this world state but with different goals. A* link *is the set of all splits for a given action.*

Definition 2 (Bad State, Safe Action) *An extended state is* bad *if it is labeled* false *or has no enabled action that is* safe. *An action is* safe *in a given state if its corresponding link has at least one good (non-bad) state for each split.*

Definition 3 (Realization) *A* realization *is a finite subgraph of the graph generated by* Expand *that satisfies the following conditions: 1) no state is bad, 2) each simple cycle (i.e., a cycle not containing any two equal states) contains at least one state labeled with an empty set of eventualities, 3) each state is left by only one link, and 4) each split in the link contains a state with a successor in the subgraph.*

A split represents *or* nondeterminism related to the satisfaction of goals, while a link represents *and* nondeterminism related to the uncontrollability of environment choices (see Figure 7). The notion of bad state captures safety and controllability: a state is *bad* if it is labeled *false* (i.e., a safety constraint is violated) or for each action enabled from this state, there is a nondeterministic path leading to a state labeled *false* (i.e., it is impossible to avoid reaching a bad state). The notion of realization takes into account safety (Condition 1), liveness (Condition 2), and controllability (Conditions 3 and 4).

A completely satisfactory reactive plan is obtained from a realization by simply replacing extended states with integers that denote plan states. Specifically, an integer is associated with each extended state; this integer represents the plan state corresponding to the extended state. The world state labeling the plan state is that of the extended state. Then, an SCR is constructed to map each plan state to the action labeling the link leaving the corresponding extended state in the realization. The successors are those of the link in the realization. We have the following theorem (a proof is given in Appendix A.4).

Theorem 4 *Given an initial world state w_0, a transition function succ for world states, a function π that evaluates propositions, and a formula f, there exists a reactive plan satisfying f* if and only if *the graph generated by* Expand *contains a realization.*

Example 10 *The plan in Figure 2 is derived from a realization partially represented by Figure 10 after some trivial simplifications automatically made by the planner to reduce the size of* SCR*s. The plan before simplification contains* 28 *states.*

7.5 Detailed Planner Algorithm

A reactive plan is computed by searching for a realization in the graph generated by *Expand*. The input of the planner algorithm is an initial world state w_0, a transition function *succ* for world states, a goal formula f, and a function π that evaluates propositions in world states (see Figure 12). The operations of the planner are first to put the goal formula into negative normal form. Then, the obtained formula is decomposed into disjunctive normal form. Finally, the search process is called for each initial state s_0 created by considering a disjunct. The planner stops once a reactive plan is found without examining the graphs starting from the remaining initial states. The search process computes a reactive plan by putting its SCRs into some global variable that is not explicitly shown in algorithms, so that Step 8 of the planner algorithm can check that a completely satisfactory reactive plan has been completed in order to stop searching.

Planner(w_0, *succ*, f, π) {
1. let f' the negative normal form of f;
2. let f'' the disjunctive normal form of f';
3. let *Goals* the list formed of disjuncts of f'';
4. for each formula g in *Goals* {
5. create an initial extended state s_0;
6. $s_0.world := w_0$; $s_0.goal := g$; $s_0.eventualities := \emptyset$;
7. *Search*(s_0, *succ*, π);
8. if a reactive plan is found, then exit}}

Figure 12: Planner algorithm

A naive approach for defining a search algorithm is to generate a graph representing the entire space of extended states by applying the function *Expand*, and then to search for a realization in the obtained graph. We describe a more efficient approach that computes a reactive plan on the fly, without exploring the entire state space. This reduces the size of the examined state space on average, while allowing computation of finite reactive plans from infinite graphs. The main idea is to detect satisfactory cycles on the fly while generating SCRs incrementally. Roughly, for each current state w, the search process guesses at a tuple (a, d, W) of $succ(w)$ that must be used as SCR for w. Then, it expands states of W to check that they are on satisfactory sequences. Should this prove not to be the case, the search process backtracks to examine alternative tuples of $succ(w)$, that is, alternative SCRs for w.

Again, a look at Figure 7 would be helpful in understanding how the search process proceeds. It should be seen as performing an and-or depth-first exploration of the state space generated by *Expand*, trying to find a realization. For each state, the search process tries to find a link such that, for every split in the link, there is a path terminated by a satisfactory cycle. A link represents an *and-branch* in the sense that all its splits must be successful. In contrast, a split represents an *or-branch* in the sense that only one of its states must be succesful. A choice between different links from a state also represents an *or-branch*, since only one link must be found for a given state.

When a state contains many different satisfactory SCRs, only one of them needs be produced. Ideally, it should be the optimal one, but our search process presently selects any of them. For instance, in Figure 7(b), if an SCR specifies that the agent must execute action a_i in w ($1 \leq i \leq m$), 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 as a result of the action. The agent has no control over this choice, but it can sense to observe which of them occurs. In return, whatever state w_{ij} ($1 \leq j \leq n_i$) results in the execution of a_i, the agent has the freedom to select the goal to satisfy among the f_{ik} ($1 \leq k \leq l_i$).

The search algorithm is described in Figure 13. Its input consists of an initial state, a successor function *succ*, and an evaluation function π. A stack is used to record the path currently being examined. Each state s on the stack is associated with three pointers: a pointer to the current link and corresponding to the SCR for s, a pointer to the current split in this link, and a pointer to the current successor s' of s in this split.

```
Search(s, succ, π) {
1.  initialize a stack containing s;
2.  while not(empty(stack)) do
3.     if member(top(stack),rest(stack)) then
4.        if satisfactory(stack) then Backtrack(FALSE);
5.        else Backtrack(TRUE);
6.     else if top(stack).goal=false then Backtrack(TRUE)
7.          else { Expand(top(stack),succ,π);
8.                 move the current link pointer of top(stack) to
                        the first link outgoing from top(stack);
9.                 move the current split pointer of top(stack) to
                        the first split in the current link for top(stack);
10.                move the current successor pointer of top(stack) to
                        the first state of the current split of top(stack);
11.                generate an SCR for the current link of top(stack);
12.                push the current successor of top(stack) on the stack;}}
```

Figure 13: An incremental search algorithm

A cycle is completed when the state on the top of the stack is equal to another in the rest of the stack (line 3). If the cycle contains a state labeled with an empty set of eventualities, this means that it is satisfactory (tested by the function *satisfactory*, line 4). Whenever a satisfactory cycle is completed, the planner backtracks to *pop* the stack; the current split is not considered further. If the completed cycle is unsatisfactory (line 5) or if a bad state is encountered (line 6), the planner backtracks and considers the next state in the current split. If all the states of the current split have been exhausted, the planner considers the first split in the next link. Finally, if the state on the top of the stack neither completes a cycle nor is bad, it is then expanded (lines 7 to 12); that is, outgoing links and splits are calculated and considered.

The function *Backtrack* is described in Figure 14. Its argument is a flag (BAD) indicating whether backtracking is due to a *bad state* (TRUE), a nonsatisfactory cycle (TRUE), or a satisfactory cycle (FALSE). The variable *stack* initialized in the function *Search* is global to *Backtrack*. The function *Backtrack* removes states from the stack (by the while loop) until reaching a state with splits or links to inspect. If no such state is found, the while loop terminates with an empty stack.

```
Backtracking(BAD)
1.  while not(empty(stack)) do {
2.     pop the stack;
3.     if BAD then
4.        if the current split for top(stack) is not exhausted
5.        then {move the current successor pointer of top(stack) to
                       the next state in the current split of top(stack);
6.               push the current successor of top(stack) on the stack; exit;}
7.        else { /* the current link is unsafe */
8.               remove an SCR matching top(stack);
9.               if the links outgoing from top(stack) are not exhausted
10.              then { move the current link pointer of top(stack) to
                            the next link of top(stack);
11.                     move the current split pointer of top(stack) to
                                 the first split in the current link of top(stack);
12.                     move the current successor pointer of top(stack) to
                                 the first state in the current split of top(stack);
13.                     generate an SCR for the current link of top(stack);
14.                     push the current successor of top(stack) on the stack; exit;}
15.    else if the current splits are not exhausted
16.        then move the current split pointer of top(stack) to
                  the next split in the current link of top(stack);
17.             move the current successor pointer of top(stack) to
                       the first state in the current split of top(stack);
18.             push the current successor of top(stack) on the stack; exit;}
```

Figure 14: Backtracking algorithm

More specifically, the stack is *poped* (line 2) and then the flag BAD is tested (line 3). If BAD is TRUE, then backtracking was invoked because the state just removed from the stack either completes an unsatisfactory cycle or is labeled *false*. If the current split is not exhausted, its next state is pushed onto the stack (lines 5–6). If the states in the current split are exhausted, the current link becomes unsafe because it contains a split for which no state is on a satisfactory path; thus, this link must not be considered further. As a consequence, the control rule that was previously generated for the state on the top of the stack is removed in order to avoid accessibility to the link (line 8). If the links outgoing from the state on the top of the stack are not exhausted, the next one is examined by changing the pointers appropriately, putting the first state in its first split on the stack, and generating an SCR for the new link (lines 10–14). If all links have been considered, the

state becomes *bad*: in the algorithm *Backtrack*, this means that we do nothing (i.e., there is no else part for the **if** in line 9), so that current state will be *poped* at the next iteration of the while *loop*.

On the other hand, if the flag is not BAD (line 15), backtracking was invoked because the state just removed from the stack completes a satisfactory cycle. If the splits in the current link are not exhausted, the current split pointer is updated to point to the next split, and the first state in the new current split is pushed onto the stack (lines 16-18). Otherwise, if all the splits have been inspected, the current link is satisfactory and does not need further consideration: in the algorithm *Backtrack*, this means that we do nothing (i.e., there is no else part for the **if** in line 15), so that the current state will be *poped* at the next iteration of the while *loop*.

The planner succeeds whenever the stack becomes empty following a backtracking phase during which the flag BAD was false. In this case, the set of SCRs is not empty and one is applicable to the initial state. The planner fails whenever the stack becomes empty following a backtracking phase during which the flag BAD was true. In this case, no SCR is applicable to the initial state. However, the set of SCRs might not be empty; it might contain satisfactory SCRs matching states that are accessible from the initial state but only along paths with nondeterministic transitions to bad states. In other words, the SCRs correspond to a realization that is not safely reachable from the initial state.

In fact, the search process above is an enumeration of possible finite reactive plans. Thus, if the state space is finite and a finite reactive plan exists, it would be found sooner or later. If the state space is infinite, but contains a finite reactive plan, then such a plan can still be found by using an *iterative deepening* strategy (Rich & Knight, 1991), which consists in applying the search algorithm several times, by fixing the depth at each step and increasing it at the subsequent steps until a solution is found. It can be proved that the sequence of partially satisfactory plans computed by the search process converges to a completely satisfactory one, whenever one exists. Of course, the convergence is not monotonic since SCRs can be removed and added until a solution is obtained. While theoretically true, in practice, the search process may be limited by the available CPU and memory resources in problems with large state spaces. This follows from Theorem 4 and the observation that the above search algorithm is simply a depth-first search for a realization in the state space generated by *Expand*, combined with an extraction of SCRs on the fly. However, the proof is ommitted due to space limitations.

7.6 Complexity

An extended state consists of a world state, a goal, and a set of eventualities. Hence, the size for extended state space is $|W| \times |F| \times |E|$, where $|W|$ is the number of possible world states, $|F|$ is the number of different possible subgoals, and $|E|$ is the number of different unbounded-time eventualities. By abstracting over the action durations, the number of different subgoals that can be produced for a goal f using goal progression is $2^{|closure(f)|}$, where $closure(f)$ is the set of subformulas of f. It can be easily checked that $|closure(f)| \leq 2 \times N$ (with N the number of Boolean and temporal connectives).

In order to take action durations into account, let T be the maximum of the different constants that occur in a time constraint associated with a temporal connective, d the

minimum of the different action durations, and C the maximum of 1 and T/d. It can be shown that there can be at most C different time arguments for the formula progression algorithm. Hence, $|closure(f)| \leq 2 \times N \times C$. The number of different conjuncts that can be formed is $O(2^{N \times C})$. Since $|E| \leq |F|$, we have that the worst-case space complexity for the planner is $O(2^{N \times C})$ for a fixed number of world states. The worst-case time complexity is double exponential since the planner searches for cycles in an exponential state space.

This complexity analysis concerns, however, the worst case. In fact, it has been proven that the time complexity for verifying many interesting temporal formulas over concurrent systems is polynomial and sometimes linear (Emerson, Sadler, & Srinivasan, 1989). This suggests that the average complexity of our planner algorithm is much better than the worst case, as confirmed by the experiments discussed in the next section. This can also be justified by a number of observations about the planner algorithm. In fact, many goal combinations are mutually inconsistent, so that they are never generated by the planner, or are inconsistent with some world states, so that their progressions yield *false*, which causes a pruning of the state space. This is illustrated by the states labeled *false* in Figure 4. Also, many goal types never cause the goal combinatorial explosion assumed by the worst case analysis. This is the case, for example, with the goals in Figures 4 and 10: each goal decomposition yields only one disjunct. Another explanation for the search efficiency in practice is that the planner explores an and-or state space. Thus, it can find a solution plan without exploring many of the or-branches, which can be anticipated if the planner is properly guided by heuristics or search control strategies. Furthermore, the planner is incremental, so that it can be stopped with an approximate plan obtained from a small part of the state space.

8. Evaluation

We implemented the incremental planner algorithm in Common Lisp, which was used to validate the previous examples. For instance, the plan of Figure 2 was obtained after 0.21 seconds and an expansion of 34 states. More complex problems were experimented in the traditional robot domain. All the experiments reported on in this paper were run on a SUN SPARCstation LX in a Lucid Common Lisp environment.

8.1 Robot Domain

The robot domain consists of connected rooms, objects in the rooms, and a robot that moves objects to indicated rooms (see Figure 15). The objects are labeled from a to e; the robot is labeled r. In the figure, the robot 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 same 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. There also exogenous actions performed by three processes that are not explicitly shown in the figure.

The first process represents a *kid* that moves between rooms to close doors randomly. Only doors specified as *kid-doors* are affected by this process. The second process is a *producer* that generates objects in given rooms at any time. Objects that can be generated

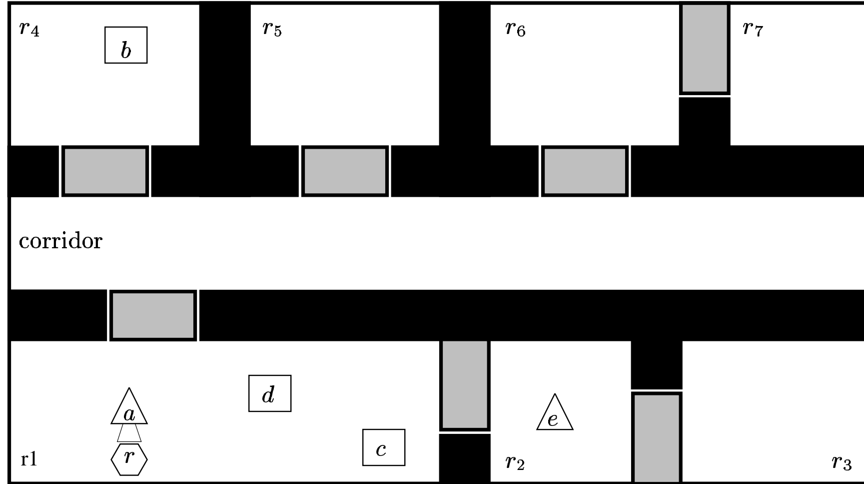


Figure 15: A simulated factory delivery domain

by the producer are specified as *producer objects*; the rooms in which they are produced are specified as *producer rooms*. The number of different producible objects is finite. An object cannot be produced when there is already an object with the same label in the domain. The last process is a *consumer* that removes every object released in a *consumer room*. An object removed by the consumer can later be regenerated by the producer. This causes infinite behaviors that are represented by cycles in the state transition system.

In each current situation, any process can either wait or perform an action that is possible, which causes nondeterminism in the state transition system. The robot's primitive actions and the process exogenous actions are specified by ADL operators using an action package borrowed from the TLPLAN system (Bacchus & Kabanza, 1995). As mentioned in Section 4, parallel composition is used to compute the state transitions *on the fly*.

8.2 Search Control Strategies

The state explosion is controlled by using search control formulas, which are temporal logic formulas that do not express legal behaviors *per se*. Rather, they express properties satisfied by sequences that involve relevant actions and that are efficient. The planner progresses them as if they were safety formulas to prune sequences that violate them. However, they are not involved in the state equality test: the planner progresses them separately. In fact, we use the TLPLAN formula progressor by Bacchus and Kabanza to handle quantified search control formulas (Bacchus & Kabanza, 1995). This requires a slight adaptation of the first-order temporal formulas progressor to express relevance of sequences with respect to MTL goals.

The following search control formulas are used in the experiments reported herein:

Formula 1 states that the planner must only expand actions that make the robot grasp relevant objects, keep the object being held by the robot until it is in the required room, keep doors opened whenever possible, and open doors only if relevant.

Formula 2 is Formula 1 conjoined with a subformula stating that the planner must avoid expanding move actions leading to a previously occupied room without having grasped or released an object.

Formula 3 is Formula 2 conjoined with a subformula stating that the planner must expand an action of moving into a room each time it has just expanded an action of opening a door connected to the room.

A detailed specification of these formulas is given in Appendix B.

Formula 1 prunes state sequences that are irrelevant to the goal, such as when the robot tries to move an object not involved in the goal. Formula 2 also prunes move actions that are relevant to the goal but executed at a bad time. Specifically, these are actions that move the robot from a room to one it has occupied previously, without having accomplished any other action than moving. This produces a more efficient plan than Formula 1. In addition, Formula 3 prunes actions that open a door without immediately moving the robot into the corresponding room. This produces reactive plans that are more efficient than those obtained with Formula 2.

```
(STATE 0 world ((in a r1) (in b r1) (in c r1) (in d r4) (in e r4) (in robot co))
      ACTION ((open robot d6c)) SUCCESSORS (1))
(STATE 1 world ((opened d6c) (in a r1) (in b r1) (in c r1) (in d r4) (in e r4) (in robot co))
      ACTION ((move robot co r6)) SUCCESSORS (2 6))
(STATE 2 world ((in a r1) (in b r1) (in c r1) (in d r4) (in e r4) (in robot r6))
      ACTION ((open robot d67)) SUCCESSORS (3))
(STATE 3 world ((opened d67) (in a r1) (in b r1) (in c r1) (in d r4) (in e r4) (in robot r6))
      ACTION ((move robot r6 r7)) SUCCESSORS (4))
(STATE 4 world ((opened d67) (in a r1) (in b r1) (in c r1) (in d r4) (in e r4) (in robot r7))
      ACTION ((wait robot)) SUCCESSORS (5))
                ...
(STATE 12 world ((opened d67) (opened d6c) (in a r1) (in b r1) (in c r1) (in d r4) (in e r4) (in robot r7))
      ACTION ((wait robot)) SUCCESSORS (9 10))
```

Figure 16: A reactive plan for reaching room 7

Figure 16 shows a partial description of a reactive plan computed by our planner by using Formula 3 for the goal of reaching room 7. The corridor is denoted by *cor* and the robot by *robot*. Objects and rooms are identified with the same symbols as in Figure 15. The door between room ri and the corridor is denoted by dic, while the door between room ri and room rj is denoted by dij. The *kid-process* randomly closes door *d6c* whenever it is open. This plan was computed in 3.49 seconds. In fact, a more efficient plan can be obtained by further constraining Formula 3 to prevent the *wait* actions in states 3 and 4.

8.3 Experiment 1

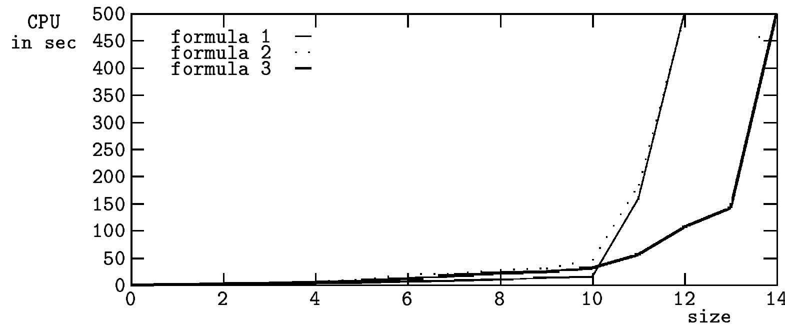
This experiment consists in moving objects into given rooms. The goal is of the form

$$\Diamond_{\geq 0} \Box_{\geq 0} (in(obj_1, room_1) \wedge \ldots \wedge in(obj_n, room_n)).$$

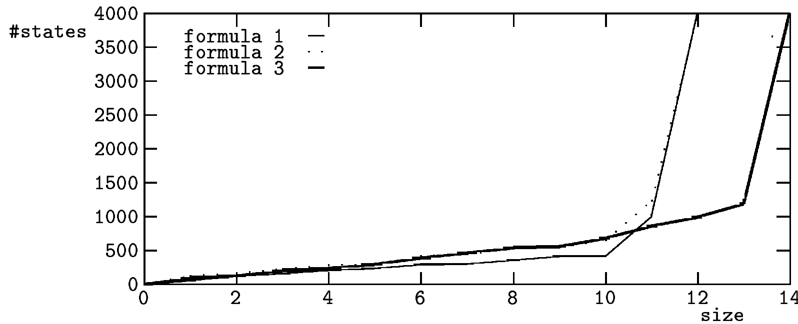
This is a goal of reaching a state satisfying a given condition and then maintain true thereafter, like in classical planners. However, contrarily to classical planners, here we have exogenous actions performed by the kid who closes doors. Thus, a reactive plan must not only ensure that a goal state can be reached, but also that it can be maintained.

The size of a problem is defined as follows. For a size $n \leq 10$, n objects are moved; in this case, we have no *kid-door*. For a size $n \geq 10$, there are 10 objects to move and $(n-10)$ *kid-doors*; that is, the kid can close up to $(n-10)$ doors at any time.

For a fixed size, the time taken by the planner depends on the initial rooms for objects, the distance to the rooms the objects must be moved to, and the room connections. To obtain representative experiments for each size, we performed 10 iterations by randomly choosing the initial rooms, goal rooms for objects, and *kid-doors*. The performance recorded for each size is then the average of the 10 iterations. Figure 17 shows the performances with respect to the CPU time in seconds (a) and the number of expanded states (b).



(a)



(b)

Figure 17: Performances for move-objects goals

The time and space performances do not depend significantly on the number of objects in the domain *per se* because the control formulas prune irrelevant sequences. Rather, they depend on the number of objects that are being moved and the number of doors that can be opened by the kid. The performance curves increase drastically from size 10 since doors can be nondeterministically closed by the kid, causing an exponential explosion in the state space.

Since Formula 3 may prune sequences that are satisfactory but not efficient, it might converge to a plan solution slower than Formula 2 or Formula 1. The same holds true for Formula 2 with respect to Formula 1. In other words, there is a price to pay for efficiency. We have tested similar goals with bounded-time constraints, yielding similar observed performance. Trial and error is involved in finding the correct time bound specification for which the goal is achievable. Otherwise, planning fails due to time constraint violation.

8.4 Experiment 2

The second experiment consists in reactively delivering produced objects to the consumer. The goal is a conjunction of formulas 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.

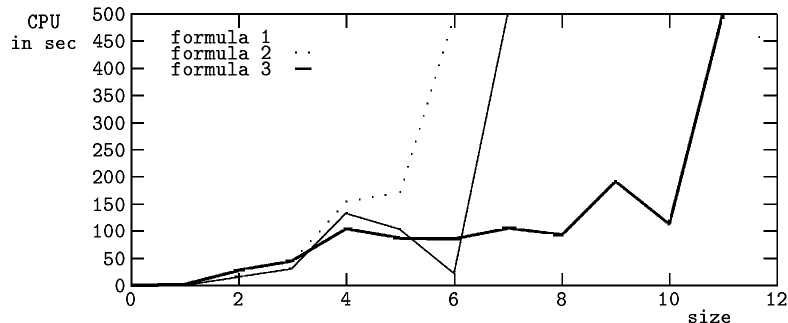
The size of a problem is the sum of different rooms involved in the goal and the number of *kid-doors*. Again, the result for each fixed size is an average over ten random iterations. Objects, rooms, and *kid-doors* are selected randomly for each iteration; *kid-doors* start being involved from size 4. The observed performances are shown in Figure 18. Formula 3 is much more efficient than the two other formulas because the state explosion starts earlier from size 4 when *kid-doors* are introduced. Since Formula 3 prunes more states than the two other formulas, it can explore larger search spaces, even though it sometimes prunes sequences that are satisfactory but not efficient.

8.5 Simulation

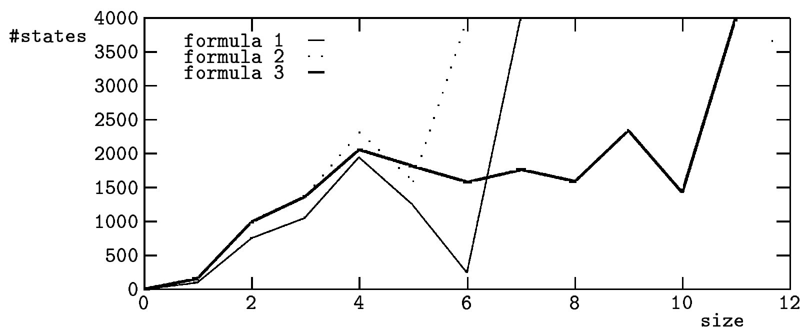
The quality of produced plans was evaluated by carrying out simulations with a robot simulator that graphically shows the effects of actions performed by the robot, kid, producer, and consumer. For instance, one can see on the screen the robot grasping an object or moving. One can also see a flashing object that is newly generated by the producer. The simulator screen looks similar to Figure 15. The simulator is also implemented in Common Lisp, using the Lucid Common Lisp packages for processes and graphics.

To measure the benefit of planning, we first tried an experiment in which the robot was controlled by using simply a hill-climbing heuristic but without a plan. For each situation, the heuristic selects the action that looks most promising with respect to the goal. For example, the doors to be opened and the room to move to are chosen depending on their Euclidean distance to the goal room. Only 5% of the goals given to the robot were achieved by using this heuristic without the planner. These goals are quite simple (at most, a size of 3) and are achieved only after many tries.

Then, the planner was run online while the robot is executing. The speed at which the robot executes actions is controlled by introducing some *sleep* instructions, to ensure that the planner processes at a higher speed. This is to prevent the planner from frequently producing obsolete SCRs. This time, the robot invokes the heuristic rule only when it is in a situation without a planned SCR. The robot was able to achieve 80% of the goals presented to it, with a size ranging from 1 to 14. However, before the planner converges to a completely satisfactory plan, the robot might backtrack in some situations because the intermediate plans that are generated by the planner are not completely satisfactory.



(a)



(b)

Figure 18: Performances for reactive produce-consume goals

9. Further Notes on Related Work

The idea of progressing temporal formulas through a sequence of states was originally introduced by Bacchus and Kabanza (Bacchus & Kabanza, 1995). Bacchus and Kabanza originally applied this idea to progress search control formulas. Recently, they extended their approach to handle MTL goals for classical plans that are sequences of actions (Bacchus & Kabanza, 1996). The problem of synthesizing classical plans is actually a special case of the problem of generating reactive plans. The problem is simpler and more efficient for three reasons. First, only finite state sequences are involved. Hence, the eventuality progression process is irrelevant because no liveness goals are involved. Second, the state space solely consists of world states; that is, the goal and eventuality labels are not involved in the test for equality between two states. Finally, there are no nondeterministic transitions.

The concept of goal progression is reminiscent of the decision procedure for linear temporal logic using the *tableau method* (Wolper, 1989). This decision procedure proceeds by constructing a Büchi automaton accepting sequences of states satisfying a temporal formula. As defined in (Thomas, 1990), a Büchi automaton is a generalization of a finite state automaton to accept infinite words. The method for constructing a Büchi automaton that accepts sequences satisfying a formula involves three phases: 1) construction of a *local automaton* accepting sequences that satisfy safety properties; 2) construction of an

eventuality automaton accepting sequences satisfying liveness properties; 3) combination of the two automata. Originally developed solely for formulas without time constraints, the technique was later generalized to formulas with time constraints (Alur & Henzinger, 1993).

Actually, a state labeled with an empty set of eventualities represents an accepting state in the sense of Büchi automata. Like the algorithm for constructing a local automaton, the goal progression algorithm is based on the property that a temporal formula is decomposable into a present and a future part. The difference is that goal progression not only constructs transitions that relate to present and future parts, but also compares these parts to the transitions of a reactive agent. Intuitively, this amounts to a composition *on the fly* of the local automaton and the state transition system for a reactive agent. On the other hand, our eventuality progression algorithm is reminiscent of the procedure for constructing an eventuality automaton, which also basically keeps track of unbounded time eventualities that must be satisfied. However, our progression of eventualities is done with respect to the goal progressions and to the transitions that are possible for a reactive agent. Again, this intuitively amounts to a composition *on the fly* of the local automaton, the eventuality automaton, and the transition system for the reactive agent.

Another major difference between our approach and the construction of Büchi automata from temporal logic specifications is that nondeterminism in such automata means *or-branches*, while nondeterminism in a graph generated by our function *Expand* means *and-branches*. Thus, our notion of nondeterminism is much more like in stochastic automata, except that we do not have transition probabilities. As such, a graph generated by *Expand* is closer to a Büchi tree automaton that accepts infinite trees of states, as defined in (Thomas, 1990). In fact, a realization is essentially a representation of many infinite trees that would be accepted by a tree automaton: each tree is obtained by unwinding cycles in the realization. Even when the graph generated by *Expand* is finite, it may include infinitely many realizations because one can repeat cycles one or many times. But our planner algorithm only searches for one realization that represents a group of paths terminated with simple cycles satisfying the conditions of Definition 3.

From a tree automaton point of view, our planning approach is related to approaches in (Pnueli & Rosner, 1989; Abadi et al., 1989) for synthesizing a reactive module that satisfies a given temporal property. A reactive module is essentially the same as a reactive plan. However, in these approaches, it is computed by constructively proving that there exists a tree automaton satisfying the desired temporal property. Although a graph generated by *Expand* could be understood as an acceptor of trees unwound from realizations, our approach rather trees such a graph as a generator of realizations. Hence, we search for a realization rather than trying to obtain it from a trace of a proof of the validity of the specification. The advantage is that we can more easily control the state explosion by adapting familiar techniques.

10. Conclusion

Robustness and reliability of reactive agents depend, in part, on their ability to reason about their environments to predict their executions and plan. In this paper, we 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. Otherwise, goal satisfaction is not guaranteed, but things cannot be as bad as if the agent had no plan whatsoever.

Our planner can be adapted to other linear modal temporal logics used in the verification of real-time systems (Alur & Henzinger, 1993). As with MTL, the semantics of these logics can be formulated in terms of evaluating the truth values of a present and future constraint. Hence, it is possible to define a corresponding goal progression procedure to apply them in our planning framework.

The representation of discrete-event systems by state sequences is limited by the state explosion problem. As explained above, one approach for controlling the state explosion problem is to use search control formulas, although the number of expanded states may become huge as the problem size increases. A possible future direction for coping with this problem is to enumerate the state space by using abstractions to group together many states that have common properties.

We have seen that 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 easily be captured by a search control formula. This strategy does not allow us, however, to generate plans that are constructively proven optimal. The user must make sure that he has specified sufficient formulas to prune nonoptimal sequences. There exist different possibilities for investigating mechanisms allowing computation of plans that are constructively proven optimal.

One possibility is to apply traditional artificial intelligence search methods, such as A^* search (Rich & Knight, 1991). Under certain conditions of *admissibility*, A^* generates plans that are constructively proven optimal with respect to costs associated with actions. The main idea is to use a heuristic function, assigning a cost to each transition in the search graph depending on how it looks close to a given goal state. Then, the planner selects the states to expand during the search in the order of their promise. The questions that must be addressed to apply these ideas to our approach are how to handle nondeterministic transitions and how to deal with cycles. In particular, A^* search is based on a notion of a final goal state, while our planner relies on final satisfactory cycles. In the same line of search strategy, a hill-climbing heuristic is already supported by our current implementation, although it was not involved in the tests we have given. In these tests, only the robot uses the hill-climbing to react heuristically.

Another possible extension is to apply decision-theoretic techniques. Dean et al. developed a planner that takes as input a state transition system, transition probabilities, and a reward function that assigns reward values to states (Dean et al., 1995). They apply dynamic programming techniques to generate a plan that maximizes the expected future rewards in each state. Given transition probabilities and reward functions, the extended transition system computed by our planner could be used as input to this decision-theoretic planner. The benefit is that our extended transition system is constructively proven to satisfy complex temporal goals that could not otherwise be expressed as reward functions. An interesting problem would be to examine the overheads introduced by the progression of goals and the progression of eventualities.

Bacchus and Kabanza initiated a research program in this direction by defining a modal temporal logic that can express a notion of *utility* for a sequence of states (Bacchus & Kabanza, 1994). The idea is that, instead of having a binary measure of satisfaction for MTL formulas (true or false), there is a utility value that expresses how close the degree of satisfaction is with respect to a given optimal value.

References

Abadi, M., Lamport, L., & Wolper, P. (1989). Realizable and unrealizable specifications of reactive systems. In *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, pp. 1–17. Lecture Notes in Computer Science, Vol. 372.

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

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

Bacchus, F., & Kabanza, F. (1994). Applying decision theory to reactive planning. *AAAI Decision-Theoretic Planning Spring Symposium*, 1–5.

Bacchus, F., & Kabanza, F. (1995). Using temporal logic to control search in a forward chaining planner. In *Proc. of 3rd European Workshop on Planning (EWSP)*, pp. 157–169.

Bacchus, F., & Kabanza, F. (1996). Planning for temporally extended goals. In *Proc. of 13th National Conference on Artificial Intelligence (AAAI 96)*, pp. 1215–1222.

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

Courcoubetis, C., Vardi, M. Y., Wolper, P., & Yannakakis, M. (1992). Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, *1*, 275–288.

Dean, T., Kaelbling, L. P., Kerman, J., & Nicholson, A. (1995). Planning under time constraints in stochastic domains. *Artificial Intelligence*, *76*, 35–74.

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

Drummond, M., & Bresina, J. (1990). Anytime synthetic projection: Maximizing probability of goal satisfaction. In *Proc. of 8th National Conference on Artificial Intelligence (AAAI 90)*, pp. 138–144.

Drummond, M., Swanson, K., & Bresina, J. (1994). Scheduling and execution for automatic telescopes. In Zwebeen, M., & Fox, M. (Eds.), *Intelligent Scheduling*, pp. 341–369. Morgan Kaufmann Publishers.

Emerson, E. A., Sadler, T., & Srinivasan, J. (1989). Efficient Temporal Reasonning. In *16th Annual ACM Symposium on Principles of Programming Languages*, pp. 166–178.

Godefroid, P., & Kabanza, F. (1991). An efficient reactive planner for synthesizing reactive plans. In *Proc. of 9th National Conference on Artificial Intelligence (AAAI 91)*, pp. 640–645.

Kabanza, F. (1992). *Reactive Planning of Immediate Actions.* Ph.D. thesis, Département d'informatique, Universite de Liège, Belgium.

Kabanza, F. (1995). Synchronizing multiagent plans using temporal logic specifications. In *Proc. of First International Conference on Multi-Agent Systems (ICMAS)*, pp. 217–225.

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

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

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

Pnueli, A., & Rosner, R. (1989). On the synthesis of an asynchronous reactive module. In *ICALP*, pp. 652–671. Lecture Notes in Computer Science, Vol 372.

Ramadge, P., & Wonham, W. (1989). The control of discrete event systems. *Proceedings of the IEEE*, *77*(1), 81–98.

Rao, A., & Georgeff, M. (1993). A model-theoretic approach to the verification of situated reasoning systems. In *Proc. of 13th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 318–324.

Rich, E., & Knight, K. (1991). *Artificial Intelligence*. McGraw Hill.

Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable environments. In *Proc. of 10th Internation Conference on Artificial Intelligence (IJCAI)*, pp. 1039–1046.

Thomas, W. (1990). Automata on infinite objects. In van Leeuwen, J. (Ed.), *Handbook of Theoretical Computer Science*, Vol. B, pp. 133–192. MIT Press/Elsevier.

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

Appendix A. Proofs

A.1 Theorem 1

Let $w_0 w_1 \ldots$ denote any infinite sequence of world states, d_i the duration of the transition from w_i to w_{i+1}, and π a function that evaluates propositions in states. Then, for any state w_i and MTL formula f, $w_i \models f$ if and only if *$w_{i+1} \models$ Progress-goal(f, w_i, d_i, π).*

Proof:

For any state w_i, if $w_i \models f$, then $w_{i+1} \models$ Progress-goal(f, w_i, d_i, π).

The proof for this direction is by induction on the structure of f.[6] The basic case is when f is a proposition. Then, according to the semantic definition of MTL, $\pi(f, w_i)$ returns true. But then, *Progress-goal*(f, w_i, d_i, π) would return the formula *true*. Since any sequence satisfies *true*, we have $w_{i+1} \models$ *Progress-goal*(f, w_i, d_i, π).

6 We could also make a proof based on a double induction on the length of a sequence and the structure of a formula. Instead, we consider an arbitrary sequence and an arbitrary state s_i on it, and then show that the conditions of the theorem hold for any formula. This approach is also used for other theorems discussed in this paper.

The inductive case is when f is a combination of any other formulas with classical or temporal connectives, as indicated by MTL syntax.[7] We detail only the cases $\bigcirc_{\sim t}\, g$, for any ordering relation $\sim$, $\Box_{\leq t}\, g$, and $g\, U_{\leq t}\, h$. The other cases are similar.

If $f = \bigcirc_{\sim t}\, g$, for any ordering relation $\sim$, and $w_i \models \bigcirc_{\sim t}\, g$, then, according to the semantic definition of MTL, $d_i \sim t$ and $w_{i+1} \models g$. But, according to the definition of *Progress-goal*, $\textit{Progress-goal}(f, w_i, d_i, \pi) = g$. Thus, $w_{i+1} \models \textit{Progress-goal}(f, w_i, d_i, \pi)$.

If $f = \Box_{\leq t}\, g$, we have two possibilities: either $d_i \leq t$ or $d_i > t$. We only show the proof for the case $d_i \leq t$. If $d_i \leq t$ and $w_i \models \Box_{\leq t}\, g$, then, according to the semantic definition of MTL, $w_i \models g$ and $w_{i+1} \models \Box_{\leq (t-d_i)}\, g$. By inductive hypothesis, if $w_i \models g$, then $w_{i+1} \models \textit{Progress-goal}(g, w_i, d_i, \pi)$. Thus, $w_{i+1} \models \textit{Progress-goal}(g, w_i, d_i, \pi) \wedge \Box_{\leq (t-d_i)}\, g$. But, since $d_i \leq t$, according to the definition of *Progress-goal*, $\textit{Progress-goal}(f, w_i, d_i, \pi) = \textit{Progress-goal}(g, w_i, d_i, \pi) \wedge \Box_{\leq (t-d_i)}\, g$. Hence, $w_{i+1} \models \textit{Progress-goal}(f, w_i, d_i, \pi)$.

If $f = g\, U_{\leq t}\, h$, we have two possibilities: either $d_i \leq t$ or $d_i > t$. We only show the proof for the case $d_i \leq t$. If $d_i \leq t$ and $w_i \models g\, U_{\leq t}\, h$, then, according to the semantic definition of MTL, $w_i \models h$ or ($w_i \models g$ and $w_{i+1} \models g\, U_{\leq (t-d_i)}\, h$). By inductive hypothesis, if $w_i \models g$, then $w_{i+1} \models \textit{Progress-goal}(g, w_i, d_i, \pi)$ and if $w_i \models h$, then $w_{i+1} \models \textit{Progress-goal}(h, w_i, d_i, \pi)$. Thus, $w_{i+1} \models \textit{Progress-goal}(h, w_i, d_i, \pi)$ or ($w_{i+1} \models \textit{Progress-goal}(g, w_i, d_i, \pi)$ and $w_{i+1} \models g\, U_{\leq (t-d_i)}\, h$). Clearly, this is equivalent to $w_{i+1} \models \textit{Progress-goal}(h, w_i, d_i, \pi) \vee (\textit{Progress-goal}(g, w_i, d_i, \pi) \wedge g\, U_{\leq (t-d_i)}\, h)$. From the definition of *Progress-goal*, since $d_i \leq t$, $\textit{Progress-goal}(g\, U_{\leq t}\, h, w_i, d_i, \pi) = \textit{Progress-goal}(h, w_i, d_i, \pi) \vee (\textit{Progress-goal}(g, w_i, d_i, \pi) \wedge g\, U_{\leq (t-d_i)}\, h)$. Hence, $w_{i+1} \models \textit{Progress-goal}(g\, U_{\leq t}\, h, w_i, d_i, \pi)$.

For any state w_i*, if* $w_{i+1} \models \textit{Progress-goal}(f, w_i, d_i, \pi)$*, then* $w_i \models f$.
The proof for this direction is quite similar to the reverse direction, but deductions are made in the opposite way. Again, we use induction on the structure of f. The basic case is when f is a proposition. Then, there are two possibilities: $\pi(f, w_i)$ returns true or $\pi(f, w_i)$ returns false. Only the first case is possible because if $\pi(f, w_i)$ returned false, then, from the definition of *Progress-goal*, $\textit{Progress-goal}(f, w_i, d_i, \pi)$ would return *false*, that is, a contradiction with the hypothesis $w_{i+1} \models \textit{Progress-goal}(f, w_i, d_i, \pi)$. Hence, $\pi(f, w_i)$ must return true. By the semantic definition of MTL, this means that $w_i \models f$.

The inductive case is when f is a combination of other formulas. We only show the proof for the case $g\, U_{\leq t}\, h$. We have two possibilities: either $d_i \leq t$ or $d_i > t$. We show the proof for the case $d_i \leq t$. By hypothesis, $w_{i+1} \models \textit{Progress-goal}(g\, U_{\leq t}\, h, w_i, d_i, \pi)$. By definition, $\textit{Progress-goal}(g\, U_{\leq t}\, h, w_i, d_i, \pi) = \textit{Progress-goal}(h, w_i, d_i, \pi) \vee (\textit{Progress-goal}(g, w_i, d_i, \pi) \wedge g\, U_{\leq (t-d_i)}\, h)$. Thus, $w_{i+1} \models \textit{Progress-goal}(h, w_i, d_i, \pi)$ or ($w_{i+1} \models \textit{Progress-goal}(g, w_i, d_i, \pi)$ and $w_{i+1} \models g\, U_{\leq (t-d_i)}\, h$). By inductive hypothesis, if $w_{i+1} \models \textit{Progress-goal}(g, w_i, d_i, \pi)$, then $w_i \models g$ and if $w_{i+1} \models \textit{Progress-goal}(h, w_i, d_i, \pi)$, then $w_i \models h$. Hence, $w_i \models h$ or ($w_i \models g$ and $w_{i+1} \models g\, U_{\leq (t-d_i)}\, h$). By the semantic definition of MTL, this means that $w_i \models g\, U_{\leq t}\, h$. ■

---

7 The inductive hypothesis is: for any state w_i, if $w_i \models g$, then $w_{i+1} \models \textit{Progress-goal}(g, w_i, d_i, \pi)$, similarly for h. Given this hypothesis, we must prove that for any other formula f constructed from g and h and for any state w_i, if $w_i \models f$, then $w_{i+1} \models \textit{Progress-goal}(f, w_i, d_i, \pi)$.

A.2 Theorem 2

For any path terminated by a cycle $s_0 s_1 \dots s_j \dots s_j$ and produced by Expand, if for all $i \geq 0$, $s_i.goal \neq$ false, and there exists $k \geq j$ such that $s_k.eventualities = \emptyset$, then for any state s_i on the infinite sequence obtained from the path by unwinding the cycle, we have $s_i \models s_i.goal$.

We provide a proof of the theorem that only considers time constraints of the form "$\leq t$" and "$\geq t$". The extension to constraints of the form "$< t$" and "$> t$" is trivial. For any infinite sequence of states $s_0 s_1 \dots$, let us define the following properties.

Property 1: For all $i \geq 0, s_i.goal \neq$ *false*.

Property 2: For all $i \geq 0$, there exists $j \geq i$ such that $s_j.eventualities = \emptyset$.

For any state s_i, we note $s_i.goal = f_1^i \wedge \cdots \wedge f_{m_i}^i$ and d_i the duration of the transition from s_i to s_{i+1}. The proof of Theorem 2 is based on the following five lemmas.

Lemma 1 *If $s_i s_{i+1} \dots$ satisfies Properties 1 and 2, and $f_p^i = \bigcirc_{\sim t}\, g$ $(1 \leq p \leq m_i)$, then (a) $d_i \sim t$ and (b) $s_{i+1}.goal$ has a conjunct that implies a disjunct of the disjunctive normal form of g.*

Proof: By definition, *Progress-goal*$(\bigcirc_{\sim t}\, g, s_i.world, d_i, \pi)$ yields *false* or g. But, by Property 1, $s_{i+1}.goal \neq$ *false*. Thus, $d_i \sim t$. As $\bigcirc_{\sim t}\, g$ is a conjunct of $s_i.goal$ and since, from the definition of *Expand*, $s_{i+1}.goal$ is a disjunct of *Progress-goal*$(s_i.goal, s_i.world, d_i, \pi)$ and the only possible result for *Progress-goal*$(\bigcirc_{\sim t}\, g, s_i.world, d_i, \pi)$ is g, then every disjunct of *Progress-goal*$(s_i.goal, s_i.world, d_i, \pi)$ will contain a conjunct that implies a disjunct of the disjunctive normal form of g. ∎

Lemma 2 *If $s_i s_{i+1} \dots$ satisfies Properties 1 and 2, and $f_p^i = g\, U_{\geq t}\, h$, for some p $(1 \leq p \leq m_i)$, then there exists a state s_j $(j \geq i)$ such that (a) $(\sum_{l=i}^{j-1} d_l) \geq t$ and $s_{j+1}.goal$ has a conjunct that implies a disjunct of the disjunctive normal form of Progress-goal$(h, s_j.world, d_j, \pi)$; and (b) for all k $(i \leq k < j)$ such that $(\sum_{l=i}^{k-1} d_l) \geq t$, $s_{k+1}.goal$ has a conjunct that implies a disjunct of the disjunctive normal form of Progress-goal$(g, s_k.world, d_k, \pi)$.*

Proof: Let us note $s_i.goal = F_i \wedge g\, U_{\geq t}\, h$, where $F_i = \bigwedge_{l \neq p} f_l^i$. From the definition of *Progress-goal*, *Progress-goal*$(s_i.goal, s_i.world, d_i, \pi) =$ *Progress-goal*$(F_i, s_i.world, d_i, \pi) \wedge$ *Progress-goal*$(g\, U_{\geq t}\, h, s_i.world, d_i, \pi)$. Let us note $F_1^i \vee \cdots \vee F_{n_i}^i$ the disjunctive normal form of the result of *Progress-goal*$(F_i, s_i.world, d_i, \pi)$. We have three possibilities: (I) $d_i \leq t$, (II) $d_i > t$ and $t \neq 0$, and (III) $t = 0$.

CASE I. According to the definition of *Progress-goal*, when $d_i \leq t$, then we must have *Progress-goal*$(g\, U_{\geq t}\, h, s_i.world, d_i, \pi) = g\, U_{\geq (t-d_i)}\, h$. By taking into account the notations above, *Progress-goal*$(s_i.goal, s_i.world, d_i, \pi) = \bigvee_{l=1}^{n_i} (F_l^i \wedge g\, U_{\geq (t-d_i)}\, h)$. From the definition of *Expand*, we must have $s_{i+1}.goal = F_q^i \wedge g\, U_{\geq (t-d_i)}\, h$, for some q $(1 \leq q \leq n_i)$. We can repeat this reasoning with states s_{i+1}, s_{i+2}, and so on. Since we required that all action durations be strictly positive, it follows that there exists a state s_j $(j \geq i)$ such that $t' = t - d_i - \cdots - d_{j-1} \geq 0$, $d_j > t'$, and $s_j.goal = F_q^{j-1} \wedge g\, U_{\geq t'}\, h$, for some q $(1 \leq q \leq n_{j-1})$. Then, either $t' \neq 0$ or $t' = 0$. If $t' \neq 0$, then s_j is in the same situation as s_i in Case II;

otherwise, s_j is in the same situation as s_i in Case III. In either case, we continue the argumentation of Case II or Case III by replacing s_i by s_j, and reach the conclusion that Lemma 2 holds.

CASE II. According to the definition of *Progress-goal*, if $d_i > t$ and $t \neq 0$, then we have *Progress-goal*$(g\, U_{\geq t}\, h, s_i.world, d_i, \pi) = g\, U_{\geq 0}\, h$. But, from the definition of *Progress-goal*, *Progress-goal*$(s_i.goal, s_i.world, d_i, \pi) = \bigvee_{l=1}^{n_i}(F_l^i \wedge g\, U_{\geq 0}\, h)$. From the definition of *Expand*, $s_{i+1}.goal = F_q^i \wedge g\, U_{\geq 0}\, h$, for some q $(1 \leq q \leq n_i)$. Thus, s_{i+1} is in the same situation as s_i in Case III. Hence, we continue the argumentation of Case III by replacing s_i by s_{i+1}, and reach the conclusion that Lemma 2 holds.

CASE III. According to the definition of *Progress-goal*, if $t = 0$, then

$$\begin{aligned}&\textit{Progress-goal}(g\, U_{\geq 0}\, h, s_i.world, d_i, \pi) = \\ &\quad \textit{Progress-goal}(h, s_i.world, d_i, \pi) \vee (\textit{Progress-goal}(g, s_i.world, d_i, \pi) \wedge g\, U_{\geq 0}\, h).\end{aligned}$$

Let $\bigvee_{l=1}^{m} g_l^i$ and $\bigvee_{l=1}^{n} h_l^i$ the disjunctive normal forms of *Progress-goal*$(g, s_i.world, d_i, \pi)$ and *Progress-goal*$(h, s_i.world, d_i, \pi)$, respectively. Then, *Progress-goal*$(s_i.goal, s_i.world, d_i, \pi) = (\bigvee_{l=1}^{n_i} \bigvee_{l'=1}^{n} F_l^i \wedge h_{l'}^i) \vee (\bigvee_{l=1}^{n_i} \bigvee_{l'=1}^{m} F_l^i \wedge g_{l'}^i \wedge g\, U_{\geq 0}\, h)$. From the definition of *Expand*, we have two possibilities: (A) $s_{i+1}.goal = F_q^i \wedge h_r^i$, for some q $(1 \leq q \leq n_i)$ and r $(1 \leq r \leq n)$; or (B) $s_{i+1}.goal = F_q^i \wedge g_r^i \wedge g\, U_{\geq 0}\, h$, for some q $(1 \leq q \leq n_i)$ and r $(1 \leq r \leq m)$.

Case III.A If $s_{i+1}.goal = F_q^i \wedge h_r^i$, then since h_r^i is a disjunct of the disjunctive normal form of *Progress-goal*$(h, s_i.world, d_i, \pi)$, part (a) of Lemma 2 is satisfied. Moreover, since $j = i$, part (b) of Lemma 2 is trivially satisfied.

Case III.B If $s_{i+1}.goal = F_q^i \wedge g_r^i \wedge g\, U_{\geq 0}\, h$, then since $s_{i+1}.goal$ contains $g\, U_{\geq 0}\, h$ as a conjunct, we can resume the reasoning from the beginning of Case III, but applied to s_{i+1}. Similarly, we can repeat this for s_{i+2}, and so on. It follows that, for any state s_k $(k \geq i)$ before a state s_j satisfying Case III.A, $s_{k+1}.goal$ must be of the form $F_q^k \wedge g_r^k \wedge g\, U_{\geq 0}\, h$, for some q and r. This means that all states before any such a state s_j satisfy part (b) of Lemma 2. But note that state s_j (assuming that it exists) satisfies part (a) of Lemma 2. Thus, to complete the proof, it remains to show that a state s_j satisfying Case III.A effectively exists.

By Property 2, there exists a state s_l after s_i such that $s_l.eventualities = \emptyset$. Either there exists a state s_j between s_i and s_l such that s_j satisfies case III.A or no such state exists. If such a state s_j exists, then this trivially ends the proof. Now, if no such state s_j exists, this means that all states between s_i and s_l fall in Case III.B. Hence, $s_l.goal$ is of the form $F_q^{l-1} \wedge g_r^{l-1} \wedge g\, U_{\geq 0}\, h$, for some q and r. From the definition of *Expand* and *Progress-eventualities*, $s_{l+1}.eventualities$ must contain $g\, U_{\geq 0}\, h$. But then, by Property 2, there must exist a state $s_{l'}$ after s_l such that $s_{l'}.eventualities = \emptyset$. From the definition of *Expand* and *Progress-eventualities*, this means that there exists a state $s_{j'}$ between s_l and $s_{l'}$ such that $s_{j'+1}.eventualities$ is obtained by removing $g\, U_{\geq 0}\, h$ from $s_{j'}.eventualities$ because h is *locally entailed* by $s_{j'}$. From this, it follows that $s_{j'}.goal = F_q^{j'-1} \wedge h_r^{j'-1}$, for some q and r such that $h_r^{j'-1}$ implies a disjunct of the disjunctive normal form of *Progress-goal*$(h, s_{j'-1}.world, d_{j'-1}, \pi)$. Hence, state $s_{j'-1}$ satisfies Case II.A of the present proof, that is, part (a) of Lemma 2. ■

Lemma 3 *If $s_i s_{i+1} \ldots$ satisfies Properties 1 and 2, and $f_p^i = g\, U_{\leq t}\, h$, for some p ($1 \leq p \leq m_i$), then there exists a state s_j ($j \geq i$) such that (a) $(\sum_{l=i}^{j-1} d_l) \leq t$ and $s_{j+1}.goal$ has a conjunct that implies a disjunct of the disjunctive normal form of Progress-goal$(h, s_j.world, d_j, \pi)$; and (b) for all k ($i \leq k < j$), $s_{k+1}.goal$ has a conjunct that implies a disjunct of the disjunctive normal form of Progress-goal$(g, s_k.world, d_k, \pi)$.*

Proof: The proof is similar to that for Lemma 2 in Case III since, for t greater than the duration of the current action, the progression of $g\, U_{\leq t}\, h$ is similar to the progression of $g\, U_{\geq 0}\, h$. But, to prove part (b) of the lemma, we use the observation that sooner or later, we reach a state at which the time constraint for the *until* connective is less than the duration of the current action. From the definition of *Progress-goal*, h will be ultimately progressed through that state. ■

Lemma 4 *If $s_i s_{i+1} \ldots$ satisfies Properties 1 and 2, and $f_p^i = \Box_{\sim t}\, g$, for some p ($1 \leq p \leq m_i$), then for all $j \geq i$ such that $(\sum_{l=i}^{j-1} d_l) \sim t$, $s_{j+1}.goal$ has a conjunct that implies a disjunct of the disjunctive normal form of Progress-goal$(g, s_j.world, d_j, \pi)$.*

Proof: The proof is similar to Lemma 2 and Lemma 3. In fact, $\Box_{\sim t}\, g$ is almost equivalent to $g\, U_{\sim t}$ *false* and is progressed like an *until* formula, except that we do not have to check that *false* is eventually satisfied. ■

Lemma 5 *If $s_0 s_1 \ldots$ satisfies Properties 1 and 2, then for all $i \geq 0$ and for all p ($1 \leq p \leq m_i$), $s_i \models f_p^i$.*

Proof: We prove this by induction on the structure of f_p^i. The basic case is when f_p^i is a proposition. In this case, $\pi(f_p^i, s_i.world)$ returns true or false. From the definition of *Progress-goal*, if $\pi(f_p^i, s_i.world)$ returns false, then *Progress-goal*$(f_p^i, s_i.world, d_i, \pi)$ returns *false*. This would cause $s_{i+1}.goal =$ *false*. But this is impossible according to Property 1. Hence, the only possible result for $\pi(f_p^i, s_i.world)$ is true. By the semantic definition of MTL, then $s_i \models f_p^i$.

The inductive case is for a formula f_p^i that is the negation of a proposition, or a formula whose main connective is $\bigcirc$, $\Box$, or U (from the definition of *Expand*, a goal labeling a state is a conjunct of such formulas). We only show the proof for $g\, U_{\leq t}\, h$.

By Lemma 3, there exists a state s_j ($j \geq i$) such that (a) $(\sum_{l=i}^{j-1} d_l) \leq t$ and $s_{j+1}.goal$ has a conjunct f_q^{j+1}, for some q ($1 \leq q \leq m_{j+1}$) that implies a disjunct of the disjunctive normal form of *Progress-goal*$(h, s_j.world, d_j, \pi)$; and (b) for all k ($i \leq k < j$), $s_{k+1}.goal$ has a conjunct f_r^{k+1}, for some r ($1 \leq r \leq m_{k+1}$), that implies a disjunct of the disjunctive normal form of *Progress-goal*$(g, s_k.world, d_k, \pi)$. In case (a), by inductive hypothesis, $s_{j+1} \models f_q^{j+1}$. Since f_q^{j+1} implies a disjunct of the disjunctive normal form of *Progress-goal*$(h, s_j.world, d_j, \pi)$, then $s_{j+1} \models$ *Progress-goal*$(h, s_j.world, d_j, \pi)$. Then, by Theorem 1, $s_j \models h$. Similarly, in case (b), for any k ($i \leq k < j$), $s_k \models g$. Hence, by the semantic definition of MTL, $s_i \models g\, U_{\leq t}\, h$. ■

Now, we can prove Theorem 2.

Proof: Let $s_0 s_1 \ldots s_j \ldots s_j$ a path terminated by a cycle and produced by *Expand*. If for all $i \geq 0$, $s_i.goal \neq$ *false*, and there exists $k \geq j$ such that $s_k.eventualities = \emptyset$, then any

infinite sequence unwound from the path satisfies Properties 1 and 2. By Lemma 5, for any state s_i on that sequence and for any f_p^i that is a conjunct of $s_i.goal$ $(1 \leq p \leq m_i)$, $s_i \models f_p^i$. But, $s_i.goal = f_1^i \wedge \cdots \wedge f_{m_i}^i$. Hence, $s_i \models s_i.goal$. ■

A.3 Theorem 3

For any path terminated by a cycle $w_0 w_1 \ldots w_l \ldots w_l$, and produced by succ, the infinite sequence obtained by unwinding the cycle satisfies an MTL formula f if and only if the graph produced by Expand contains a path terminated by a cycle $s_0 s_1 \ldots s_j \ldots s_j$ such that (a) for all $i \geq 0$, $s_i.goal \neq$ false; (b) there exists $k \geq j$ such that $s_k.eventualities = \emptyset$; and (c) $s_0.world = w_0$ and, for any $s_{i'}$ and w_i, if $s_{i'}.world = w_i$, then $s_{i'+1}.world = w_{i+1}$.

Proof:
If (a), (b), and (c), then $w_0 \models f$.
This follows trivially from Theorem 2.

If $w_0 \models f$, then (a), (b), and (c).
Let us note d_i the duration of the transition from w_i to w_{i+1}. Let us build a path of pairs $(w_0, f_0)(w_1, f_1) \ldots (w_k, f_k) \ldots (w_k, f_k)$ such that $f_0 = f$ and $f_{i+1} = \textit{Progress-goal}(f_i, w_i, d_i, \pi)$ for all $i > 0$. This path is terminated by a cycle $(w_k, f_k) \ldots (w_k, f_k)$ because it is defined from a path $w_0 w_1 \ldots w_l \ldots w_l$ that is also terminated by a cycle, and *Progress-goal* can generate only finitely many different formulas.[8]

For any f_i, let us note $f_1^i \vee \cdots \vee f_{n_i}^i$ the disjunctive normal form of f_i. By Theorem 1, since $w_0 \models f_0$, then we must have $w_1 \models f_1$. Similarly, from $w_1 \models f_1$, we have $w_2 \models f_2$. And so on, we obtain $w_i \models f_i$ for any world state w_i. From this, we have that, for every world state w_i, there exists at least one f_q^i such that $w_i \models f_q^i$. Then, using Theorem 1 again, it follows that, for every f_q^i such that $w_i \models f_q^i$, there exists f_r^{i+1} such that $w_{i+1} \models f_r^{i+1}$ and f_r^{i+1} is a disjunct of the disjunctive normal form of $\textit{Progress-goal}(f_q^i, w_i, d_i, \pi)$. Thus, there exists at least one path terminated by a cycle $(w_0, f_p^0)(w_1, f_p^1) \ldots (w_k, f_q^k) \ldots (w_k, f_q^k)$ and satisfying the property that for any state w_i, (P.1) $w_i \models f_q^i$ and (P.2) f_r^{i+1} implies a disjunct of the disjunctive normal form of $\textit{Progress-goal}(f_q^i, w_i, d_i, \pi)$. (We use the symbol P to denote P.1 and P.2.)

Now, for any pair (w_0, f_p^0) such that $w_0 \models f_p^0$, let us construct a graph G of extended states by grouping all paths rooted on this state and satisfying Property P, while extending each pair (w_i, f_q^i) with a set of eventualities $e_q^{i'}$; each tuple $(w_i, f_q^i, e_q^{i'})$ denotes an extended state $s_{i'}$ with $s_{i'}.world = w_i$, $s_{i'}.goal = f_q^i$, and $s_{i'}.eventualities = e_q^{i'}$. Precisely, G is constructed as follows. The root is a state s_0 with $s_0.world = w_0$, $s_0.goal = f_p^0$, and $s_0.eventualities = \emptyset$. Then, for any state $s_{i'}$, successors are obtained as follows: for any pair (w_{i+1}, f_r^{i+1}) that is a successor of (w_i, f_q^i) on a path rooted on (w_0, f_p^0) and satisfying Property P, $s_{i'}$ has a successor $s_{i'+1}$, with $s_{i'+1}.world = w_{i+1}$, $s_{i'+1}.goal = f_r^{i+1}$, and $s_{i'+1} = \textit{Progress-eventualities}(s_i, s_{i+1}, d_i, \pi)$.

---

8 The number of propositions and connectives in f is fixed; time values are always decreased with action durations, but never below 0. Since the path has finitely many actions, there are finitely many different durations for decrementing time values. Hence, *Progress-goal* can generate finitely many subformulas with different time constraints.

From P.2 and $s_{i+1}.eventualities = Progress\text{-}eventualities(s_i, s_{i+1}, d_i, \pi)$, it follows that G is a subgraph of the graph generated by *Expand* from s_0. The graph G cannot be empty since above we concluded that there exists at least one path satisfying Property P. By Property P.1 and Theorem 1, it follows that, for every state s of G, $s.goal \neq false$. Hence, all paths of G satisfy part (a) of Theorem 3. By Property P.1 and the definition of *Progress-eventualities*, it follows that at least one path satisfies part (b) of Theorem 3.[9] From the definition of G, all paths of G satisfy part (c) of Theorem 3. Hence, G contains a path satisfying parts (a), (b), and (c) of the theorem. As G is part of the graph generated by *Expand*, there exists a path generated by *Expand* that satisfies parts (a), (b), and (c) of Theorem 3. ■

A.4 Theorem 4

Given an initial world state w_0, a transition function succ for world states, a function π that evaluates propositions, and a formula f, there exists a reactive plan satisfying f if and only if *the graph generated by Expand contains a realization.*

Proof:
If the graph generated by Expand contains a realization, then there exists a reactive plan satisfying f. This direction is trivial. It follows from the definition of a realization and the observation that a reactive plan is merely a syntactic sugar for a realization.

If there exists a reactive plan satisfying f, then the graph generated by Expand contains a realization. A reactive plan satisfying f can be seen as a finite set of paths terminated by cycles such that an infinite sequence unwound from any of these paths satisfies f.[10] By Theorem 3, for each of these paths, the graph generated by *Expand* contains a corresponding path satisfying the criteria 1 and 2 of a realization. Now all together, these paths generated by *Expand* satisfy all the criteria of a realization. Hence, there exists a subgraph generated by *Expand* that is a realization. ■

Appendix B. Search Control Formulas

Search control formulas are progressed by using the TLPLAN package which supports first-order temporal formulas (Bacchus, 1995). Formulas are given in prefix notation. The time constraint for the modal connectives is implicitly "≥ 0". Following is the specification for search control Formula 3.

```
(always
  (forall (?room) (in robot ?room)
    (and
     (forall (?object) (holding robot ?object)
       (and
```

9 This follows by contradiction, by observing that if part (b) did not hold for any path, this would mean that there is a formula $g\,U_{\geq 0}\,h$ labeling some state w_i such that h is not locally entailed with any descendant of w_i. This contradicts Property P.1.

10 By definition, a reactive plan is finite; thus, the number of different paths terminated by cycles composing a reactive plan is also finite; but, all together, these paths may generate infinitely many infinite sequences.

```
    ;; grasp only relevant objects only
    (exists (?room2) (goal (in ?object ?room2)))
    (forall (?room2) (goal (in ?object ?room2))
    (and
      ;; keep held objects until in their rooms.
      (if-then-else (= ?room ?room2)
         (next (not (holding robot ?object)))
         (next (holding robot ?object)))
       ;;don't come back until having delivered object.
       (next (implies (not (in robot ?room))
                      (until (not (in robot ?room))
                             (in ?object ?room2))))))))
  ;;if arm empty, don't come back until having grasped an object
  (implies (not (exists (?object) (holding robot ?object)))
           (next (implies (not (in robot ?room))
                          (until (not (in robot ?room))
                                 (exists (?object)
                                         (holding robot ?object))))))
   ;; move only when there exist objects to move
   (implies (not (exists (?object ?room) (in ?object ?room)
                   (exists (?room2) (goal (in ?object ?room2))
                     (not (= ?room2 ?room)))))
            (next (in robot ?room)))
   (forall (?door ?room2) (door/room ?room)
     (and
       ;; keep doors opened, expect those controllable by the kid.
       (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))))
        ;;if the robot opens a door, it must enter the connected room
        ;;immediately.
        (implies (not (opened ?door))
                 (next (implies (opened ?door)
                                (next (in robot ?room2))))))))))
```

Formula 2 is obtained from Formula 3 by removing the subformula annotated by the comment "if the robot opens a door, it must enter the connected room immediately". Formula 1 is obtained from Formula 2 by further removing the subformulas annotated by the comments "don't come back until having delivered object" and "if arm empty, don't come back until having grasped an object".

All these formulas are general for the robot domain and do not depend on a specific initial state or goal. To fully understand them, the reader may need some familiarity with the TLPLAN goal specification notations, which are explained in (Bacchus, 1995).