Policy Generation for Continuous-time Stochastic Domains with Concurrency

Håkan L. S. Younes and **Reid G. Simmons**
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{lorens, reids}@cs.cmu.edu

Abstract

We adopt the framework of Younes, Musliner, & Simmons for planning with concurrency in continuous-time stochastic domains. Our contribution is a set of concrete techniques for policy generation, failure analysis, and repair. These techniques have been implemented in TEMPASTIC, a novel temporal probabilistic planner, and we demonstrate the performance of the planner on two variations of a transportation domain with concurrent actions and exogenous events. TEMPASTIC makes use of a deterministic temporal planner to generate initial policies. Policies are represented using decision trees, and we use incremental decision tree induction to efficiently incorporate changes suggested by the failure analysis.

Introduction

Most existing methods for planning under uncertainty are impractical for domains with concurrent actions and events (Bresina *et al.* 2002). While discrete-time Markov decision processes (MDPs) can be used to handle concurrent actions (Guestrin, Koller, & Parr 2002), this approach is restricted to *synchronous* execution of sets of actions. Continuous-time MDPs (Howard 1960) can be used to model *asynchronous* systems, but are restricted to events and actions with exponential delay distributions. In many occasions, the exponential distribution is inadequate to accurately model the stochastic behavior of a system or system component. Component life time, for example, is often best modeled using a Weibull distribution because it describes increasing and decreasing failure rates (Nelson 1985). The semi-Markov decision process (SMDP) (Howard 1971) permits non-exponential distributions, but this model is not closed under concurrent composition. This means that a system consisting of two concurrent SMDPs with finite state spaces cannot in general be modeled by a finite (or even countable) state-space SMDP.

Younes, Musliner, & Simmons (2003) have proposed a framework for planning with concurrency in continuous-time stochastic domains. This framework allows path based plan objectives expressed using the continuous stochastic logic (CSL) (Aziz *et al.* 2000; Baier *et al.* 2003). The framework is based on the Generate, Test and Debug (GTD) paradigm (Simmons 1988), and relies on statistical techniques (discrete event simulation and acceptance sampling) for CSL model checking developed by (Younes & Simmons 2002b) to test if a policy satisfies specified plan objectives.

We adopt this framework and contribute a set of concrete techniques for representing and generating initial policies using an existing deterministic temporal planner, robust sample path analysis techniques for extracting failure scenarios, and efficient policy repair techniques. These techniques have been implemented in TEMPASTIC, a novel temporal stochastic planner accepting domain descriptions in an extension of PDDL described by Younes (2003) for expressing general stochastic discrete event systems.

The domain model that we use allows for actions and exogenous events with random delay governed by general continuous probability distributions. We enforce the restriction that only one action can be enabled at any point in time, but we can still model concurrent processes by having a start action for each process with a short delay, and using an exogenous event with extended delay to signal process completion. Although the only continuous resource we consider in this paper is time, we briefly discuss at the end how the techniques presented here could be extended to work with more general continuous resources.

Framework

We adopt the framework of Younes, Musliner, & Simmons (2003) for planning with concurrency in continuous-time stochastic domains, based on the Generate, Test and Debug (GTD) paradigm proposed by Simmons (1988). The domain model is a continuous-time stochastic discrete event system, and policies are generated to satisfy properties specified in a temporal logic. The approach resembles that of Drummond & Bresina (1990) for probabilistic planning in discrete-time domains.

Algorithm 1 shows the generic hill-climbing procedure, FIND-POLICY, proposed by Younes, Musliner, & Simmons (2003) for probabilistic planning based on the GTD paradigm. The input to the procedure is a model $\mathcal{M}$ of a stochastic discrete event system, an initial state s_0, and a goal condition ϕ. The result is a policy π such that the stochastic process $\mathcal{M}[\pi]$ (i.e. $\mathcal{M}$ controlled by π) satisfies ϕ when execution starts in s_0. The procedure GENERATE-INITIAL-POLICY returns a seed policy for the policy search algorithm. Later in this paper, we will describe in detail how to implement this procedure using a deterministic temporal planner. TEST-POLICY returns true iff the current policy satisfies the goal condition, and this procedure can be implemented using discrete event simulation and statistical hypothesis testing as described by Younes, Musliner, & Simmons (2003). The simulation traces generated during policy verification can, as described in later sections, be used by DEBUG-POLICY to find reasons for failure and return a repaired policy. The repaired policy is compared to the currently best policy by BETTER-POLICY, which returns the better of the two policies. This procedure can also be implemented using statistical techniques, reusing the samples generated during verification as suggested by Younes, Musliner, & Simmons (2003).

Algorithm 1 Generic planning algorithm for probabilistic planning based on the GTD paradigm.

```
FIND-POLICY(M, s0, φ)
  π0 ⇐ GENERATE-INITIAL-POLICY(M, s0, φ)
  if TEST-POLICY(M, s0, φ, π0) then
    return π0
  else
    π ⇐ π0
    loop    ▷ return π on break
      repeat
        π′ ⇐ DEBUG-POLICY(M, s0, φ, π)
        if TEST-POLICY(M, s0, φ, π′) then
          return π′
        else
          π′ ⇐ BETTER-POLICY(π, π′)
      until π′ ≠ π
      π ⇐ π′
```

The proposed algorithm is sound if TEST-POLICY never accepts a policy that does not satisfy the goal condition. Since we rely on statistical techniques in our implementation of this procedure, our planner can only give probabilistic guarantees regarding soundness. The same holds for completeness because TEST-POLICY can reject a good policy with some probability, but we sacrifice completeness in any case by using local search.

Model of Uncertainty

Although FIND-POLICY does not rely on any specific model of stochastic discrete event systems, we here describe the model of uncertainty used in our implementation of the algorithm. A model $\mathcal{M}$ consists of a set S of states and a set E of events. Each event $e \in E$ has an enabling condition ϕ_e determining the set of states in which e is enabled. An event can trigger when it is enabled, causing an instantaneous state transition in the system, but there is uncertainty in the time from when an event becomes enabled until it triggers. The time that an event e has to remain enabled before it triggers is governed by a probability distribution $F(t; e)$. Multiple events can be enabled simultaneously, representing concurrent processes. We can associate a real-valued clock $c(e)$ with each event e that is currently enabled, representing the time until e is scheduled to trigger. When an event becomes enabled, having previously been disabled, the value of $c(e)$ is sampled from the distribution $F(t; e)$. Events race to trigger in a state s, and the event with the smallest clock value causes a transition to a state s' determined by a probability distribution over successor states $p(s'; s, e)$ defined for each event. Let e^* denote the triggering event in s. The clock values for events that remain enabled in s' but did not trigger in s are decremented by $c(e^*)$. New clock values are sampled for events that become enabled in s', including e^* if it remains enabled after triggering. The probability of multiple events triggering simultaneously is zero if all delay distributions are continuous.

The model we have described is known in queuing theory as a *generalized semi-Markov process* (GSMP), first introduced by Matthes (1962). A GSMP can intuitively be viewed as the composition of concurrent semi-Markov processes, and captures the essential dynamical structure of a stochastic discrete event system (Glynn 1989). Note that a GSMP model associates a local clock with each event, which differentiates it from time-dependent MDPs (Boyan & Littman 2001) where there is a single global clock.

For the purpose of planning, we identify a set $E_{\mathrm{a}} \subset E$ of actions (controllable events) that can be disabled at will. The remaining events, $E_{\mathrm{e}} = E \setminus E_{\mathrm{a}}$, are *exogenous events* beyond the control of the decision maker. An exogenous event $e \in E_{\mathrm{e}}$ is always enabled in a state s if ϕ_e is satisfied in s. For an action to be enabled, it must be selected by the current policy to be enabled in addition to having its enabling condition satisfied. We assume that E_{a} always contains a null-action a_ϵ representing idleness.

We work with a factored representation of the state space and adopt the syntactic extension of PDDL proposed by Younes (2003) for specifying actions and events with random delay. This allows us to compactly represent complex planning domains. Figure 1 shows part of the definition of a transportation domain that will be used to illustrate the techniques introduced in this paper. The declarations of one action schema ("check-in") and one event schema ("fill-plane") are shown. All events instantiated from the "fill-plane" event schema have an exponentially distributed delay with rate 0.01, while actions instantiated from the "check-in" action schema have a deterministic delay of 1 time unit.

Goal Formalism

We adopt the *continuous stochastic logic* (CSL) (Aziz *et al.* 2000; Baier *et al.* 2003) as a formalism for expressing probabilistic *temporally extended* goals in continuous-time domains. The syntax of CSL is defined as

$$\phi ::= \top \big| a \big| \phi \wedge \phi \big| \neg\phi \big| \mathcal{P}_{\bowtie p}\left(\phi\, \mathcal{U}^{\leq t}\, \phi\right) \big| \mathcal{P}_{\bowtie p}\left(\phi\, \mathcal{U}\, \phi\right),$$

where a is an atomic proposition, $p \in [0, 1]$, $t \in \mathbb{R}_{\geq 0}$, and $\bowtie \in \{\leq, \geq\}$.

Regular logic operators have their usual semantics. A probabilistic formula $\mathcal{P}_{\bowtie p}(\rho)$ holds in a state s if and only if the set of paths starting in s and satisfying the path formula ρ is p' and $p' \bowtie p$. A path of a stochastic process is a sequence of states and holding times:

$$\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots$$

```
(define (domain transportation)
  ...
  (:delayed-action check-in
          :parameters (?pers - person ?plane - airplane ?loc - airport)
          :delay 1
          :condition (and (at ?pers ?loc) (at ?plane ?loc)
                          (or (not (full ?plane)) (has-reservation ?pers ?plane)))
          :effect (and (not (at ?pers ?loc)) (in ?pers ?plane) (not (has-reservation ?pers ?plane))))
  (:delayed-event fill-plane
          :parameters (?plane - airplane ?loc - airport)
          :delay (exponential 0.01)
          :condition (and (not (full ?plane)) (at ?plane ?loc))
          :effect (full ?plane))
  ...)
```

Figure 1: Part of a domain description with exogenous events and continuous delay distributions.

A path formula $\phi_1 \; \mathcal{U}^{\leq t} \; \phi_2$ ("time-bounded until") holds over a path σ if and only if ϕ_2 holds in some state s_i such that $\sum_{j=0}^{i-1} t_j \leq t$ and ϕ_1 holds in all states s_j for $j < i$. The formula $\phi_1 \; \mathcal{U} \; \phi_2$ ("until") holds over a path σ if and only if $\phi_1 \; \mathcal{U}^{\leq t} \; \phi_2$ holds over σ for some $t \in \mathbb{R}_{\geq 0}$.

A wide variety of goals can be expressed in CSL. Table 1 shows examples of achievement goals, goals with safety constraints on execution paths, and maintenance/prevention goals. In this paper we focus on goal conditions of the form $\mathcal{P}_{\bowtie p} \left(\phi_1 \, \mathcal{U}^{\leq t} \, \phi_2 \right)$, where both ϕ_1 and ϕ_2 are regular propositional formulae.

Initial Policy Generation

Given a planning problem $\langle \mathcal{M}, s_0, \phi \rangle$, we want to find a stationary policy $\pi : S \rightarrow E_{\mathrm{a}}$ such that ϕ holds in s_0 for the stochastic process $\mathcal{M}[\pi]$. Algorithm 1 outlines a procedure for finding such a policy by means of local search. The efficiency of the procedure will depend on the quality of the initial policy returned by GENERATE-INITIAL-POLICY. A quick solution would be to simply return the null-policy mapping every state to the idle action a_ϵ, but this ignores the goal condition of the planning problem. If we can make a more informed choice for an initial policy, it is likely to have fewer bugs than the null-policy, thus requiring fewer repairs.

We propose an implementation of GENERATE-INITIAL-POLICY that relaxes the original planning problem by ignoring uncertainty, and solves the resulting deterministic planning problem using an existing temporal planner. Our implementation uses a slightly modified version of VHPOP (Younes & Simmons 2003), a heuristic partial order causal link (POCL) planner with support for PDDL2.1 (Fox & Long 2003) durative actions.

Conversion to Deterministic Planning Problem

We relax a continuous-time probabilistic planning problem by treating all events of a model equally, ignoring the fact that some events are not controllable. In other words, all events are considered to be actions that the deterministic planner can choose to include in a plan. We can eliminate probabilistic effects by splitting events with probabilistic effects into multiple events with deterministic effects. Each new event would have the same enabling condition as the original event and an effect representing a separate outcome of the original event's probabilistic effect. Furthermore, instead of a probability distribution over possible event durations, we associate an interval with each event representing the possible durations for the event. This interval is simply the support of the probability distribution for the event delay. The deterministic temporal planner is allowed to select any duration within the given interval for an event that is part of a plan. We can now represent each event as a PDDL2.1 durative action with an interval constraint on the duration, with the enabling condition of the event as an invariant ("over all") condition of the durative action that must hold over the duration of the action, and with the effects associated with the end of the durative action. Figure 2 shows a stochastic event with delay distribution $U(0, 10)$ and a probabilistic effect with two outcomes, and it also shows the two durative actions with deterministic effects that would be used to represent the stochastic event. The purpose of the transformation is to make every possible outcome of a stochastic event available to the deterministic planner.

```
(:delayed-event crash
        :delay (uniform 0 10)
        :condition (up)
        :effect (probabilistic 0.4 (down) 0.6 (broken)))

(:durative-action crash1
        :duration (and (>= ?duration 0)
                       (<= ?duration 10))
        :condition (over all (up))
        :effect (at end (down)))
(:durative-action crash2
        :duration (and (>= ?duration 0)
                       (<= ?duration 10))
        :condition (over all (up))
        :effect (at end (broken)))
```

Figure 2: A stochastic event (top) and two durative deterministic actions (bottom) representing the stochastic event.

A CSL goal condition of the form $\mathcal{P}_{\geq p} \left(\phi_1 \, \mathcal{U}^{\leq t} \, \phi_2 \right)$ is converted into a goal for the deterministic planning problem as follows. We make ϕ_2 a goal condition that must be-

Goal description	Formula
reach office with probability at least 0.9	$\mathcal{P}_{\geq 0.9}\left(\top\,\mathcal{U}\ \mathit{office}\right)$
reach office within 17 time units with probability at least 0.9	$\mathcal{P}_{\geq 0.9}\left(\top\,\mathcal{U}^{\leq 17}\ \mathit{office}\right)$
reach office within 17 time units with probability at least 0.9 while not spilling coffee	$\mathcal{P}_{\geq 0.9}\left(\neg\mathit{coffee\text{-}spilled}\,\mathcal{U}^{\leq 17}\ \mathit{office}\right)$
reach office within 17 time units with probability at least 0.9 while recharging at least every 5 time units with probability at least 0.5	$\mathcal{P}_{\geq 0.9}\left(\mathcal{P}_{\geq 0.5}\left(\top\,\mathcal{U}^{\leq 5}\ \mathit{recharging}\right)\ \mathcal{U}^{\leq 17}\ \mathit{office}\right)$
remain stable for at least 8.2 time units with probability at least 0.7	$\mathcal{P}_{\leq 1-0.7}\left(\top\,\mathcal{U}^{\leq 8.2}\ \neg\mathit{stable}\right)$

Table 1: Examples of goals expressible in CSL.

come true no later than t time units after the start of the plan, while ϕ_1 becomes an invariant condition that must hold until ϕ_2 is satisfied. We can represent this goal in the temporal POCL framework as a durative action with no effects, with an invariant condition ϕ_1 that must hold over the duration of the action, and a condition ϕ_2 associated with the end of the action. We add the temporal constraints that the start of the goal action must be scheduled at time 0 and that the end of the action must be scheduled no later than at time t. VHPOP records all such temporal constraints in a *simple temporal network* (Dechter, Meiri, & Pearl 1991) allowing for efficient temporal inference during planning. A plan now represents an execution of actions and exogenous events satisfying the path formula $\phi_1\,\mathcal{U}^{\leq t}\ \phi_2$, possibly ignoring the adverse effects of some exogenous events which is left to the debugging phase to discover.

For CSL goals of the form $\mathcal{P}_{\leq p}\left(\phi_1\,\mathcal{U}^{\leq t}\ \phi_2\right)$, we instead want to find plans representing executions not satisfying the path formula $\phi_1\,\mathcal{U}^{\leq t}\ \phi_2$. We then use $\neg\phi_2$ as an invariant condition that must hold in the interval $[0, t]$. Note that it is not necessary to achieve ϕ_1 in order for $\phi_1\,\mathcal{U}^{\leq t}\ \phi_2$ to be false, so we do not include ϕ_1 in the deterministic planning problem. This means that an empty plan will satisfy the goal condition, unless ϕ_2 holds in the initial state in which case the problem lacks solution, and we return the null-policy as an initial policy for these goals.

There are a few additional constraints inherited from the model that we enforce in the modified version of VHPOP. The first is that we do not allow concurrent actions. This is due to the restriction on policies being mappings from states to single actions. The restriction is not severe, however, since an "action" with extended delay can be modeled as a controllable event with short delay to start the action and an exogenous event to end the action, allowing for additional actions to be executed before the temporally extended action completes. The second constraint is that separate instances of the same exogenous event cannot overlap in time. For example, if one instance of the "fill-plane" event is made enabled at time t_1 and is scheduled to trigger at time t_2, then no other instance of "fill-plane" can be scheduled to be enabled or trigger in the interval $[t_1, t_2]$. This constraint follows from the GSMP domain model. Both constraints are of the same nature and is represented in the planner as a new flaw type, associated with two events e_1 and e_2, that can be resolved in two ways analogous to promotion and demotion for regular POCL threat resolution: either the end of e_1 must come before the start of e_2, or the start of e_1 must come after the end of e_2.

A final adjustment to "close the gap" between events is made to an otherwise complete plan before it is returned. It ensures that events are scheduled to become enabled at the triggering of some other event, and not at an arbitrary point in time. This restriction also follows from the GSMP domain model.

From Plan to Policy

Given a plan, we now want to generate a policy. We represent a policy using a *decision tree* (cf. Boutilier, Dearden, & Goldszmidt 1995), and we generate a policy from a plan by converting the plan to a set of training examples $\langle s_i, e_i\rangle$, $s_i \in S$ and $e_i \in E_{\mathrm{a}}$, and then generating a decision tree from these training examples. The training examples are obtained by serializing the plan returned by VHPOP and executing the sequence of events starting in the initial state.

A plan returned by VHPOP is a set of triples $\langle t_i, e_i, d_i\rangle$, where e_i is an event, t_i is the time that e_i is scheduled to become enabled, and d_i is the delay of e_i (i.e. e_i is scheduled to trigger at time $t_i + d_i$). We serialize a plan by sorting the events in ascending order based on their trigger time, breaking ties nondeterministically. The first event to trigger, call it e_0, is applied to the initial state s_0, resulting in a state s_1. If $e_0 \in E_{\mathrm{a}}$, then this gives rise to a training example $\langle s_0, e_0\rangle$. Otherwise, the first event gives rise to the training example $\langle s_0, a_\epsilon\rangle$, signifying that we are waiting for something beyond our control to happen in state s_0. We continue to generate training examples in this fashion until there are no unprocessed events left in the plan. Given a set of training examples for the initial plan, we use regular decision tree induction (see, e.g., Quinlan 1986) to generate an initial policy.

To illustrate the process of generating an initial policy, consider the planning problem described by Younes, Musliner, & Simmons (2003), which is a continuous-time variation of a problem developed by Blythe (1994). In this problem, the goal is to have a person transport a package from CMU in Pittsburgh to Honeywell in Minneapolis with probability at least 0.9 in at most 300 time units without losing it on the way. In CSL, this goal can be expresses as $\mathcal{P}_{\geq 0.9}\left(\neg\mathit{lost}_{\mathrm{pkg}}\,\mathcal{U}^{\leq 300}\ \mathit{at}_{\mathrm{me,honeywell}} \wedge \mathit{carrying}_{\mathrm{me,pkg}}\right)$.

Figure 3(a) shows the plan generated by the deterministic temporal planner. The plan schedules two events to become enabled at time zero, one being the action to enter a taxi at CMU, and the other being the exogenous event causing the plane to depart from Pittsburgh to Minneapolis. Actions are identified by an entry in the second column of the table in Figure 3(a). The "enter-taxi" action is scheduled to trigger first, resulting in a training example mapping the initial state to this action. The following state is mapped to the first "depart-taxi" action, while the state following the triggering of that action is mapped to the idle action. This is because the next event ("arrive-taxi") is not an action. Eight additional training examples can be extracted from the plan, and the decision tree representation of the policy learned from the eleven training examples is shown in Figure 3(b). This policy, for example, maps all states satisfying $at_{\text{pgh-taxi,cmu}} \wedge at_{\text{me,cmu}}$ to the action labeled a_1 (the first "enter-taxi" action in the plan), while states where $at_{\text{pgh-taxi,cmu}}$, $at_{\text{plane,mpls-airport}}$, and $at_{\text{me,pgh-airport}}$ are all false and $in_{\text{me,plane}}$ is true are mapped to the idle action a_ϵ.

Additional training examples can be obtained from plans with multiple events scheduled to trigger at the same time by considering different trigger orderings of the simultaneous events. If two events e_1 and e_2 are both scheduled to trigger at time t, we would get one set of training example by applying e_1 before e_2, and a second set by applying e_2 before e_1.

Policy Debugging and Repair

During verification of a policy π for a planning problem $\langle \mathcal{M}, s_0, \phi \rangle$, we generate a set of sample execution paths starting in s_0 for the stochastic process $\mathcal{M}[\pi]$. If the policy π does not satisfy the goal condition, then these sample paths can help us understand the "bugs" of π and provide us with valuable information on how to repair the policy.

We next present the techniques for sample path analysis we use in our implementation of the DEBUG-POLICY procedure. The result of the analysis is a set of ranked failure scenarios. A failure scenario can be fed to the deterministic temporal planner, which will try to generate a plan taking the failure scenario into account. The resulting plan, if one exists, can be used to repair the current policy.

Sample Path Analysis

Policy verification generates a set of sample paths $\boldsymbol{\sigma} = \{\sigma_1, \ldots, \sigma_n\}$, with each sample path being of the form

$$\sigma_i = s_0 \xrightarrow{t_{i0},e_{i0}} s_{i1} \xrightarrow{t_{i1},e_{i1}} \ldots \xrightarrow{t_{i,k_i-1},e_{i,k_i-1}} s_{ik_i}.$$

We start the sample path analysis by computing a value, relative to a goal formula $\mathcal{P}_{\geq p}\left(\phi_1 \,\mathcal{U}^{\leq t}\, \phi_2\right)$, for each state occurring in some sample path. This is done by constructing a stationary Markov process representing the sample paths. The state space for this Markov process is the set of states occurring in some sample path. The transition probabilities $p(s'; s)$ are defined as the number of times s' is immediately followed by s in the sample paths divided by the total number of occurrences of s. We assign the value $+1$ to states satisfying ϕ_2 and the value -1 to states satisfying $\neg(\phi_1 \vee \phi_2)$. We represent the exceeding of the time bound t along a sample path with a special event e_τ leading to a state s_τ that also is assign the value -1 (for a goal formula $\mathcal{P}_{\leq p}(\rho)$, all the $+1$ and -1 values are interchanged). The value of the remaining states is computed using the recurrence

$$V(s) = \gamma \sum_{s' \in S} p(s'; s) V(s'),$$

where $\gamma < 1$ is a discount factor. The value of a state signifies the closeness to a success or failure state, ignoring timing information and only counting the number of transitions. A large positive value indicates closeness to success, while a large negative value indicates closeness to failure. The discount factor permits us to control the influence a success or failure state s has on the value of states at some distance from s.

The next step is to assign a value to each event occurring in some sample path. Each triple $s \xrightarrow{e} s'$ is given the value $V(s') - V(s)$ and the value $V(e)$ of an event e is the sum of the values of all triples that e is part of. We also compute the mean μ_e and standard deviation σ_e over triples involving e. The event with the largest negative value can be thought of as the "bug" contributing the most to failure, and we want to plan to avoid this event or to prevent it from having negative effects.

Finally, we construct a failure scenario for each event e by combining the information from all failure paths σ_i (paths ending in a state with value -1) containing a triple $s \xrightarrow{e} s'$ such that $V(s') - V(s) < \mu_e + \sigma_e$. The reason for the cutoff is to not include information from failure paths where an event contributes to failure significantly less than on average so that the aggregate information is representative for the "bug" being considered. A failure scenario is a sequence of events paired with time points and is constructed from a set of paths by associating each event occurring in all paths with the average trigger time for the event. Table 2 shows how two example failure paths are combined into a single failure scenario. Event e_1 occurs twice in both failure paths and therefore also occurs twice in the failure scenario, while event e_4 only appears in the first path and is thus excluded from the scenario.

Failure Path 1	**Failure Path 2**	**Failure Scenario**
e_1 @ 1.2	e_1 @ 1.6	e_1 @ 1.4
e_2 @ 3.0	e_2 @ 3.2	e_2 @ 3.1
e_1 @ 4.5	e_3 @ 4.4	e_1 @ 4.5
e_3 @ 4.8	e_1 @ 4.5	e_3 @ 4.6
e_4 @ 6.8	e_5 @ 6.4	e_5 @ 6.7
e_5 @ 7.0	-	-

Table 2: Example of failure scenario construction from two failure paths.

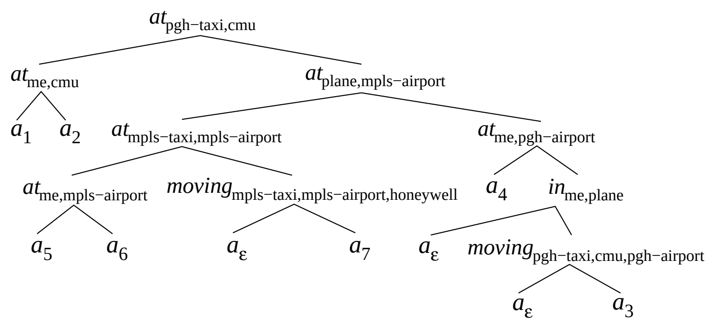
Planning for Failure Scenarios

We select the failure scenario for the event with the lowest value and try to generate a plan for the selected scenario that achieves the goal. If this fails, we try planning for the next worst failure scenario, and continue in this manner until we find a promising repair or run out of failure scenarios.

We plan for a failure scenario by incorporating the events and timing information of the scenario into the planning problem that we pass to the temporal deterministic planner.

t_i:$e_i[d_i]$	act.
0:(enter-taxi me pgh-taxi cmu)[1]	a_1
0:(depart-plane plane pgh-airport mpls-airport)[60]	
1:(depart-taxi me pgh-taxi cmu pgh-airport)[1]	a_2
2:(arrive-taxi pgh-taxi cmu pgh-airport)[20]	
22:(leave-taxi me pgh-taxi pgh-airport)[1]	a_3
23:(check-in me plane pgh-airport)[1]	a_4
60:(arrive-plane plane pgh-airport mpls-airport)[90]	
150:(enter-taxi me mpls-taxi mpls-airport)[1]	a_5
151:(depart-taxi me mpls-taxi mpls-airport honeywell)[1]	a_6
152:(arrive-taxi mpls-taxi mpls-airport honeywell)[20]	
172:(leave-taxi me mpls-taxi honeywell)[1]	a_7

(a) Plan for simplified deterministic planning problem.



(b) Policy generated from plan in (a).

Figure 3: (a) Initial plan and (b) policy for transportation problem. Leafs in the decision tree are labeled by actions, with labels taken from the table in (a). To find the action selected by the policy for a state s, start at the root of the decision tree. Traverse the tree until a leaf node is reached by following the left branch of a decision node if s satisfies the test at the node and following the right branch otherwise.

Given a failure scenario $e_1@t_1, \ldots, e_k@t_k, \ldots, e_n@t_n$ associated with the event e_k, we generate a sequence of states $s_0, \ldots, s_n$, where s_0 is the initial state of the original planning problem and s_i for $i > 0$ is the state obtained by applying e_i to state s_{i-1}. We can plan to avoid the bad event e_k by generating a planning problem with initial state s_i for $i < k$. By choosing i closer to k, we can potentially avoid planning for situations that the current policy already handles well. Our implementation iterates over the possible starts states from $i = k - 1$ to $i = 0$. If a solution is found for some i, then we do not have to attempt further initial states. For each planning problem that we generate, we limit the number of search nodes explored by VHPOP. In case the search limit is reached, we attempt an earlier initial state, or try to plan for the next worst failure scenario if we already are at $i = 0$.

Given an initial state s_i, we incorporate the events following s_i in the failure scenario into the planning problem in the form of a set of *event dependency trees* T_i and a set of *untriggered* events U_i. Each node in an event dependency tree stores an event and a trigger time for the event relative the parent node (or relative the initial state for root nodes). The children of a node for an event e represent events that depend on the triggering of e to become enabled. The set U_i represents events that are enabled in all states s_j but differ from all events e_j for $j \geq i$, and these events should not be allowed to trigger between time 0 and t_n in the deterministic planning problem. An event dependency tree node can be associated with a set of untriggered events as well, these being events that should not be allowed to trigger between the triggering of the event associated with the node and time t_n.

We define the sets T_i and U_i for state s_i recursively. The base case is $T_n = \emptyset$, with U_n containing all events enabled in s_n. For $i < n$, let $\delta = t_i - t_{i-1}$ (or simply t_i for $i = 1$) and construct a tree τ_i consisting of a single node with event e_i and trigger time δ. For each tree $\tau \in T_{i+1}$:

- if the event at the root of τ is an action, then add τ to T_i.
- if the event at the root of τ is enabled in s_i, then add δ to the trigger time of the root node and add the resulting tree to T_i.
- if the event at the root of τ is disabled in s_i, then add τ to the children of τ_i.

e_i @ t_i	**Label**
(enter-taxi me pgh-taxi cmu) @ 0.909091	a_1
(depart-taxi me pgh-taxi cmu pgh-airport) @ 1.81818	a_2
(fill-plane plane pgh-airport) @ 13.284	e_3
(arrive-taxi pgh-taxi cmu pgh-airport) @ 30.0722	e_4
(leave-taxi me pgh-taxi pgh-airport) @ 30.9813	a_5
(lose-package me pkg pgh-airport) @ 44.0285	e_6

Figure 4: Failure scenario for the policy in Figure 3(b) associated with the “fill-plane” event.

Let U be the set of events $e \in U_{i+1}$ not enabled in s_i. Then $U_i = U_{i+1} \setminus U \cup \{e_{i+1}\}$ and the root node of τ_i is associated with the set U. Finally, add τ_i to T_i.

Figure 4 shows an actual failure scenario for the policy in Figure 3(b). For this scenario, in the state right before the “fill-plane” event, there are three event trees: one with $e_4@28.254$ as the sole node, one with $e_3@11.4658$ as the sole node, and a final tree with a_5 at the root and $e_6@13.0472$ as a child node.

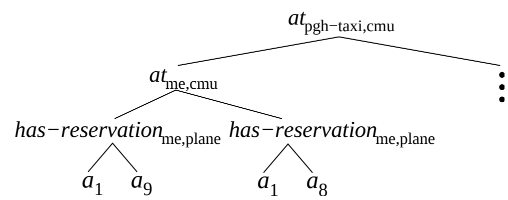
We incorporate the event trees in T_i with exogenous events at the root into the deterministic planning problem by forcing all the events in these trees to be part of the plan. Events at root nodes are scheduled to become enabled at time 0 and to trigger at the time stored at the node, and events at non-root nodes are scheduled to become enabled at the time the parent event triggers and scheduled to trigger t time units after the parent event triggers (t being the time stored at the node). The planner is allowed to disable the effects of a forced event by disabling its enabling condition. This can easily be handled in a POCL framework by treating the enabling condition as an effect condition that can be disabled by means of *confrontation* (see, e.g., Weld 1994). The set U_i, and sets of untriggered events associated with event dependency tree nodes, impose further scheduling constraints that restrict the possibilities for the deterministic planner, forcing it to produce a plan consistent with the timing information contained in the failure scenario. For an example of an untriggered event, consider the failure scenario in Figure 4. There is a "move-taxi" event for the Pittsburgh taxi that becomes enabled immediately after it arrives at the Pittsburgh airport, but the "move-taxi" event is treated as an untriggered event since it does not appear in the failure scenario. This means that the deterministic planner is not permitted to schedule a "move-taxi" event for the Pittsburgh taxi until after any events in the plan that are part of the failure scenario.

Once a plan is found for a failure scenario, we extract a set of training examples from the plan as described earlier in this paper. We update the current policy by incorporating the additional training examples into the decision tree using incremental decision tree induction (Utgoff, Berkman, & Clouse 1997). This requires that we store the old training examples in the leaf nodes of the decision tree, and some additional information in the decision nodes, but we avoid having to generate the entire decision tree from scratch. We adapt the algorithm of Utgoff, Berkman, & Clouse to our particular situation by always giving precedence to new training examples over old ones in case of inconsistencies, and by only restructuring the decision tree after incorporating all new training examples.

Figure 5(a) shows a plan for the failure scenario in Figure 4, with the state after the "enter-taxi" action as the initial state for the planning problem. The policy after incorporating the training examples generated from the plan is shown in Figure 5(b). The entire right subtree for the repaired policy is the same as for the initial policy, so it does not have to be regenerated.

t_i:$e_i[d_i]$	act.
0:(leave-taxi me pgh-taxi cmu)[1]	a_8
0:(depart-plane plane pgh-airport mpls-airport)[60]	
0:(fill-plane plane pgh-airport)[12.3749]	
1:(make-reservation me plane cmu)[1]	a_9
2:(enter-taxi me pgh-taxi cmu)[1]	a_1
3:(depart-taxi me pgh-taxi cmu pgh-airport)[1]	a_2
4:(arrive-taxi pgh-taxi cmu pgh-airport)[20]	
24:(leave-taxi me pgh-taxi pgh-airport)[1]	a_3
25:(check-in me plane pgh-airport)[1]	a_4
60:(arrive-plane plane pgh-airport mpls-airport)[90]	
150:(enter-taxi me mpls-taxi mpls-airport)[1]	a_5
151:(depart-taxi me mpls-taxi mpls-airport honeywell)[1]	a_6
152:(arrive-taxi mpls-taxi mpls-airport honeywell)[20]	
172:(leave-taxi me mpls-taxi honeywell)[1]	a_7

(a) Plan for failure scenario.



(b) Repaired policy.

Figure 5: (a) Plan for failure scenario in Figure 4 using the second state as initial state, and (b) the policy after incorporating the training examples from the plan in (a). The right subtree of the root node is identical to that of the initial policy in Figure 3(b), and is only indicated by three dots.

Results

The results in this section were generated on a PC with a 650 MHz Pentium III processor running Linux. A search limit of 10,000 explored nodes was set for the deterministic planner. We used the additive heuristic described by Younes & Simmons (2002a) with VHPOP, a variation for POCL planning of the additive heuristic for state space planning first proposed by Bonet, Loerincs, & Geffner (1997).

First we consider the transportation problem described so far in this paper. There are several things that can go wrong with the initial policy in Figure 3(b): the plane can become full or leave before we get to the Pittsburgh airport to check in, the Minneapolis taxi can be serving other customers when we arrive at the Minneapolis airport, and the package can get lost if we stand with it at an airport for too long. The top part of Table 3 shows the worst three "bugs" for the initial policy as determined by the sample path analysis. The numbers in the table are averages over five runs with different random seeds, and we used the parameters $\alpha = \beta = 0.01$ (error probability) and $\delta = 0.005$ (half-width of indifference region) with the verification algorithm (see (Younes & Simmons 2002b) for details on the meaning of these parameters). The influence of these parameters on the complexity of the verification algorithm are discussed by Younes *et al.* (2004). By a wide margin, the worst bug is that the plane becomes full before we have a chance to check in. Losing the package at Minneapolis airport comes in second place. Note that the package is more often lost at Pittsburgh airport than at Minneapolis airport, but this bug is not ranked as high because it only happens when the plane already has been filled.

The "fill-plane" bug is repaired by making a reservation before leaving CMU, resulting in the policy shown in Figure 5(b). The top three bugs for this policy are shown in the bottom part of Table 3. Now, losing the package at Minneapolis airport appears to be the only severe bug left. Note that losing the package at Pittsburgh airport no longer ranks in the top three because the repair for the "fill-plane" bug took care of this bug as well. The package is lost at Minneapolis airport because the taxi is not there when we arrive, and the repair found by the planner is to store the package in a safety box until the taxi returns. The policy resulting from this repair satisfies the goal condition so we are done.

Table 4 shows running times for the different parts of the planning algorithm on two variations of the transportation problem. The first problem uses the original transportation

	Event	Rank	Value	$\mu_e + \sigma_e$	Paths
first policy	(fill-plane plane pgh-airport)	1.0	-24.1	-0.36	41.8
	(lose-package me pkg mpls-airport)	2.0	-14.7	-0.76	15.0
	(lose-package me pkg pgh-airport)	3.2	-6.8	-0.15	36.4
second policy	(lose-package me pkg mpls-airport)	1.0	-94.3	-0.70	101.6
	(arrive-plane plane pgh-airport mpls-airport)	2.4	-19.9	0.04	99.4
	(move-taxi mpls-taxi mpls-airport)	2.6	-18.2	0.06	107.4

Table 3: Top ranking "bugs" for the first two policies of the transportation problem. All numbers are averages over five runs.

domain, while the second problem replaces the possibility of storing a package with an action for reserving a taxi and uses the probability threshold 0.85 instead of 0.9. The verification time is inversely proportional to the logarithm of the error bounds α and β (cf. Younes *et al.* 2004). We can see that the sample path analysis takes very little time. The time for the first repair is about the same for both problems, which is not surprising as exactly the same repair applies in both situations. The second repair takes longer time for the second problem because we have to go further back in the failure scenario in order to find a state where we can apply the taxi reservation action so that it has desired effects. We observe that the sample path analysis finds the same major bugs despite random variation in the sample paths across runs and varying error bounds.

Discussion

We have presented concrete techniques for policy generation, debugging, and repair that can be used in the framework of Younes, Musliner, & Simmons (2003) for planning in continuous-time domains with concurrency. We represent policies using decision trees, which are generated from training examples extracted from a serial plan produced by a deterministic temporal planner. Our debugging technique utilizes the samples generated during policy verification, and reliably identifies the two major bugs in our transportation example. The sample path analysis results in a set of failure scenarios that help guide the replanning effort required to repair a policy. These failure scenarios could also be useful in helping humans understand negative behavior of continuous-time stochastic systems, and can be thought of as corresponding to "counter-examples" in non-probabilistic model checking.

The policies we generate are stationary, but we could easily extend our techniques to generate non-stationary policies by adding a time stamp to each training example extracted from a plan and allowing numeric tests in the decision tree. If the deterministic planner we use supports planning with continuous-valued resources other than time, then we could also lift the restriction on only allowing boolean state variables in the domain descriptions.

We are currently trying to avoid some replanning by not always planning from the initial state when planning for a failure scenario. We could potentially save more effort by using a different goal than the original goal, for example by considering some cross section of the previous partial order plan and plan for a goal that is the conjunction of link conditions crossing the cut. Alternatively, we could reuse the most recent plan and use transformational plan operators to repair the plan.

Extensions of the planning framework to decision theoretic planning are also under consideration (Ha & Musliner 2002). The techniques presented in this paper may be useful in this setting as well. Instead of assigning the values -1 and $+1$ to terminal states during sample path analysis, we could assign values according to a user defined value function, with the failure scenarios then indicating execution paths that bring down the overall value for the policy being analyzed.

Acknowledgments. This paper is based upon work supported by DARPA and ARO under contract no. DAAD19–01–1–0485, and a grant from the Royal Swedish Academy of Engineering Sciences (IVA). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsors.

References

Aziz, A.; Sanwal, K.; Singhal, V.; and Brayton, R. 2000. Model-checking continuous-time Markov chains. *ACM Transactions on Computational Logic* 1(1):162–170.

Baier, C.; Haverkort, B. R.; Hermanns, H.; and Katoen, J.-P. 2003. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering* 29(6):524–541.

Blythe, J. 1994. Planning with external events. In *Proc. Tenth Conference on Uncertainty in Artificial Intelligence*, 94–101. Morgan Kaufmann Publishers.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *Proc. Fourteenth National Conference on Artificial Intelligence*, 714–719. AAAI Press.

Boutilier, C.; Dearden, R.; and Goldszmidt, M. 1995. Exploiting structure in policy construction. In *Proc. Fourteenth International Joint Conference on Artificial Intelligence*, 1104–1111. Morgan Kaufmann Publishers.

Boyan, J. A., and Littman, M. L. 2001. Exact solutions to time-dependent MDPs. In *Advances in Neural Information Processing Systems 13: Proc. 2000 Conference*. Cambridge, MA: The MIT Press. 1026–1032.

Bresina, J.; Dearden, R.; Meuleau, N.; Ramakrishnan, S.; Smith, D. E.; and Washington, R. 2002. Planning under

		first policy			*second policy*			*third policy*
		Verification	**Analysis**	**Repair**	**Verification**	**Analysis**	**Repair**	**Verification**
problem 1	$\alpha = \beta = 10^{-1}$	0.044	0.008	0.642	0.232	0.012	0.014	0.176
	$\alpha = \beta = 10^{-2}$	0.084	0.014	0.640	0.470	0.012	0.018	0.344
	$\alpha = \beta = 10^{-4}$	0.160	0.004	0.646	0.974	0.020	0.022	0.698
problem 2	$\alpha = \beta = 10^{-1}$	0.072	0.006	0.666	2.372	0.036	2.468	0.606
	$\alpha = \beta = 10^{-2}$	0.140	0.006	0.670	5.490	0.074	2.496	1.318
	$\alpha = \beta = 10^{-4}$	0.272	0.010	0.682	10.036	0.128	2.568	2.494

Table 4: Running times, in seconds, for different stages of the planning algorithm for the original transportation problem (problem 1) and the modified transportation problem (problem 2) with varying error bounds (α and β). All numbers are averages over five runs.

continuous time and resource uncertainty: A challenge for AI. In *Proc. Eighteenth Conference on Uncertainty in Artificial Intelligence*, 77–84. Morgan Kaufmann Publishers.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49(1–3):61–95.

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

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20. Forthcoming.

Glynn, P. W. 1989. A GSMP formalism for discrete event systems. *Proceedings of the IEEE* 77(1):14–23.

Guestrin, C.; Koller, D.; and Parr, R. 2002. Multiagent planning with factored MDPs. In *Advances in Neural Information Processing Systems 14: Proc. 2001 Conference*. Cambridge, MA: The MIT Press. 1523–1530.

Ha, V., and Musliner, D. J. 2002. Toward decision-theoretic CIRCA with application to real-time computer security control. In *Papers from the AAAI Workshop on Real-Time Decision Support and Diagnosis Systems*, 89–90. AAAI Press. Technical Report WS-02-15.

Howard, R. A. 1960. *Dynamic Programming and Markov Processes*. New York, NY: John Wiley & Sons.

Howard, R. A. 1971. *Dynamic Probabilistic Systems*, volume II. New York, NY: John Wiley & Sons.

Matthes, K. 1962. Zur Theorie der Bedienungsprozesse. In *Trans. Third Prague Conference on Information Theory, Statistical Decision Functions, Random Processes*, 513–528. Publishing House of the Czechoslovak Academy of Sciences.

Nelson, W. 1985. Weibull analysis of reliability data with few or no failures. *Journal of Quality Technology* 17(3):140–146.

Quinlan, J. R. 1986. Induction of decision trees. *Machine Learning* 1(1):81–106.

Simmons, R. G. 1988. A theory of debugging plans and interpretations. In *Proc. Seventh National Conference on Artificial Intelligence*, 94–99. AAAI Press.

Utgoff, P. E.; Berkman, N. C.; and Clouse, J. A. 1997. Decision tree induction based on efficient tree restructuring. *Machine Learning* 29(1):5–44.

Weld, D. S. 1994. An introduction to least commitment planning. *AI Magazine* 15(4):27–61.

Younes, H. L. S., and Simmons, R. G. 2002a. On the role of ground actions in refinement planning. In *Proc. Sixth International Conference on Artificial Intelligence Planning and Scheduling Systems*, 54–61. AAAI Press.

Younes, H. L. S., and Simmons, R. G. 2002b. Probabilistic verification of discrete event systems using acceptance sampling. In *Proc. 14th International Conference on Computer Aided Verification*, volume 2404 of *LNCS*, 223–235. Springer.

Younes, H. L. S., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research* 20:405–430.

Younes, H. L. S.; Kwiatkowska, M.; Norman, G.; and Parker, D. 2004. Numerical vs. statistical probabilistic model checking: An empirical study. In *Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. Forthcoming.

Younes, H. L. S.; Musliner, D. J.; and Simmons, R. G. 2003. A framework for planning in continuous-time stochastic domains. In *Proc. Thirteenth International Conference on Automated Planning and Scheduling*, 195–204. AAAI Press.

Younes, H. L. S. 2003. Extending PDDL to model stochastic decision processes. In *Proc. ICAPS-03 Workshop on PDDL*, 95–103.