Fault-Tolerant Planning under Uncertainty

Luis Pineda,[†] **Yi Lu,**[†] **Shlomo Zilberstein,**[†] **Claudia V. Goldman**[‡]

[†] School of Computer Science, University of Massachusetts, Amherst, MA 01003, USA
[‡] General Motors, Advanced Technical Center, Herzliya Pituach 4673341, Israel

Abstract

A fault represents some erroneous operation of a system that could result from an action selection error or some abnormal condition. We formally define error models that characterize the likelihood of various faults and consider the problem of fault-tolerant planning, which optimizes performance given an error model. We show that factoring the possibility of errors significantly degrades the performance of stochastic planning algorithms such as LAO*, because the number of reachable states grows dramatically. We introduce an approach to plan for a bounded number of faults and analyze its theoretical properties. When combined with a continual planning paradigm, the k-fault-tolerant planning method can produce near-optimal performance, even when the number of faults exceeds the bound. Empirical results in two challenging domains confirm the effectiveness of the approach in handling different types of runtime errors.

1 Introduction

We are concerned with decision-theoretic planning in large domains that involve uncertainty about both the outcome of actions as well as erroneous operation of the system. In the driving domain, examples of uncertainty associated with actions include the uncertainty about their duration or about traffic conditions. Examples of erroneous operation include taking the wrong turn or having a flat tire. We model erroneous operation using actions that represent various types of errors or failures, referred to as *faults*. The distinction between uncertain effects of regular actions and those captured by faults is a design choice rather than an explicit property of the domain. In practice, planning for all possible faults in advance can make an easy problem intractable.

One approach to handle faults is to generate a *universal plan* [Schoppers, 1987] – one that covers any possible situation the system may encounter, even when errors are made. But universal planning is impractical when the domain is large [Ginsberg, 1989]. In fact, when modeling the problem as a Markov decision process (MDP), a policy that covers the entire state space, such as the one generated by the value iteration (VI) or policy iteration (PI) algorithms, is a universal plan. Efficient algorithms for generating compact universal plans for MDPs have been developed, most notably LAO* [Hansen and Zilberstein, 2001], LRTDP [Bonet and Geffner, 2003] and their extensions. These algorithms exploit reachability analysis to converge on optimal universal plans that include only states that are reachable during the course of plan execution. However, as we illustrate in this paper, even such compact universal plans are unrealistic in some application domains. For example, when driving from Boston to New York, we don't really need to consider being in Washington as part of the plan, although some extremely unlikely sequence of errors could lead us there. Hence, the fundamental questions addressed in this paper are: (1) *How can we create plans that are robust to errors, without making them unnecessarily large and cumbersome?* and (2) *How can we guarantee that a valid plan will always be available at runtime, even when errors occur?*

Interest in fault-tolerant planning has a long history in control theory and engineering [Dubash *et al.*, 1992], focusing mostly on mechanisms to overcome faults via redundancy and parallelism. Within AI, our work is most closely related to that of Jensen *et al.* [2004], who examine deterministic planning systems in which actions are allowed to have additional "faulty consequences". In contrast, we allow normal actions to have multiple possible *probabilistic* outcomes. A fault is associated with the execution of a faulty action, either mistakenly performing a different normal action than the one specified by the plan (e.g., missing an exit in driving) or performing a special action that represents some faulty condition (e.g., having a flat tire). The approach we present is based on planning for a bounded number of faults, an idea previously explored by Meuleau and Smith [2003] who focus on reducing the total number of branches in a plan to k to better understand, verify or communicate the plan. We don't limit branching per se, but instead reduce the overall number of states reachable by the optimal plan.

The rest of the paper is structured as follows. In Section 2, we define runtime faults, error models, and fault-tolerant plans. In Section 3, we describe algorithms for fault-tolerant planning and in Section 4, we introduce an effective method for generating plans that address a bounded number of faults. We also analyze the properties of this k-fault-tolerant planning algorithm. In Section 5, we present the continual planning paradigm that creates a new k-fault-tolerant plan online once k faults occur. We assess the performance of the planning algorithms and the continual planning paradigm in Section 6 and conclude with a summary of the contributions.

2 Fault-Tolerant Planning Problems

We represent the planning problem using a Markov decision process (MDP). We focus in this work on stochastic shortest path problems, where the goal is to minimize the expected cost of reaching some goal state. The underlying MDP, $\mathcal{M}$, is defined by a tuple of the form: $\langle S, A, T, C, s_0, G\rangle$ where: S is a finite set of states; A is a finite set of actions; $T(s'|s,a)$ is the transition probability of ending in state s', when action a is taken in state s; $C(s,a)$ is the cost of taking action a is state s; $s_0 \in S$ is a given start state; and $G \subset S$ is a set of terminal goal states.

Starting in state s_0, the objective is to reach one of the goal states while minimizing the expected cumulative cost of the actions taken. There are no dead-ends in the problems we tackle, and there is always a *proper policy* that reaches a goal state with probability 1. Thus, we do not use discounting.

A solution is a policy π, represented as a mapping from states to actions: $\pi : S \rightarrow A$. The well-known Bellman equation defines a value function over states, $V^*(s)$, from which the optimal policy π^* can be extracted:

$$V^*(s) = \min_a [C(s,a) + \sum_{s' \in S} T(s'|s,a)V^*(s')] \qquad (1)$$

There are many algorithms for solving MDPs, most notably value iteration (VI) and policy iteration (PI) [Puterman, 1994]. These dynamic programming methods guarantee convergence on the optimal policy, but cannot scale to very large domains. We therefore employ search-based methods that exploit the fact that the set of states reachable by an optimal policy can be a small subset of S, particularly when a good heuristic is given. A good example is LAO* [Hansen and Zilberstein, 1998; 2001] – an extension of the classic search algorithms A* and AO* that can find solutions with loops. This makes it possible to solve MDPs, while leveraging the vast amount of work on reachability analysis and heuristic methods in classical planning.

An MDP policy is said to be a *universal policy* if it covers every state $s \in S$. A *partial policy* covers a subset of states in S. In fact, LAO* can generate the best partial solution graph for a given MDPs, without covering states that are not reachable by the plan. We use the notation $\mathcal{S}(\pi)$ to represent the set of states covered by a partial policy π.

2.1 Execution Time Faults

We model faults using alternate actions that take place instead of the planned action and could have undesired consequences.

Definition 1 (fault w.r.t. a plan). *We say that an action a executed in state s is a fault w.r.t. plan π if $a \neq \pi(s)$.*

Definition 2 (error model). *Given an MDP, an error model ξ is defined as a probability distribution $\xi(a'|s,a)$ over actions for each state $s \in S$ and given action $a \in A$.*

In general, the faulty action $a' \in A'$, could be the same as the intended action a, another "normal" action from A that is available to the decision maker when the MDP is solved, or a new "abnormal" action that represents some faulty (and usually undesired) operation of the system. To simplify the notation, we assume that the given transition model T is defined over all normal and abnormal actions in A'.

Definition 3 (fault-free state). *A state $s \in S$ is said to be a fault-free state w.r.t. error model ξ iff $\forall a : \xi(a|s,a) = 1$.*

A given MDP and an error model induce a new transition function T' that factors the possibility of error actions:

$$T'(s'|s,a) = \sum_{a'} T(s'|s,a')\xi(a'|s,a) \qquad (2)$$

That is, T' specifies the likelihood of outcome state s' when action a is taken in state s, with or without a fault occurrence.

Definition 4 (error model with undesirable faults). *Let V^* and π^* be the optimal value function and policy for a given MDP. We say that model ξ is with undesirable faults iff*

$$\forall s, a : V^*(s) \leq C(s,a) + \sum_{s' \in S} T'(s'|s,a)V^*(s').$$

That is, when we plug in the transition model T' that factors an error model with undesirable faults into the Bellman equation, it can only increase the cost of reaching the goal (relative to the optimal cost), i.e., errors cannot be helpful.

Proposition 1. *A sufficient condition under which any error model is with undesirable faults is that the error model include no* abnormal *actions. That is $A' \setminus A = \emptyset$.*

The property holds because when no abnormal actions exist, the error model simply replaces the best action with another normal action. But no normal action can have a lower cost compared to the optimal one, since that would contradict optimality.

2.2 Fault-Tolerant Plans

Definition 5 (fault-tolerant plan). *A plan π is fault tolerant w.r.t. error model ξ if there is no state s reachable as a result of correct or faulty operation for which $\pi(s)$ is undefined. That is, $\forall s \in \mathcal{S}(\pi) : T'(s'|s,\pi(s)) > 0 \Rightarrow s' \in \mathcal{S}(\pi)$.*

Definition 6 (k-fault-tolerant plan). *A plan π is k-fault-tolerant w.r.t. error model ξ if every state s reachable from the initial state s_0 as a result of at most k execution faults according to ξ is covered by the plan π.*

3 Finding Optimal Fault-Tolerant Plans

We say that a fault-tolerant plan is optimal, when it minimizes the expected cost of reaching a goal state *given* an error model. There is no obvious resemblance between the optimal plan for a given MDP and the optimal fault-tolerant plan. For example, the optimal plan in the driving domain may involve 12 turns. If the error model indicates that missing turns is likely, the optimal fault-tolerant plan may be an entirely different path that is a bit longer, but has fewer turns.

Proposition 2. *Given an MDP and an error model, the actions that optimize the following value function form an optimal fault-tolerant plan:*

$$\bar{V}^*(s) = \min_a [C(s,a) + \sum_{s' \in S} T'(s'|s,a)\bar{V}^*(s')].$$

Proof. (Sketch) Note that $T'(s'|s,a)$ is the induced transition model for the given MDP *plus* error model as shown in Eq. (2). The value function $\bar{V}^*$ therefore satisfies the Bellman equation for the induced MDP. Hence, $\bar{V}^*$ provides the optimal solution for the fault-tolerant planning problem. $\square$

The above equation can be solved using standard MDP solvers such as VI or PI. For large domains, value iteration is slow and is likely to limit scalability. Search-based algorithms, such as LAO*, can take advantage of reachability analysis to reduce the number of states evaluated and runtime. However, with unlimited errors the number of reachable states could be very large, diminishing the benefits of LAO*. For example, in the racetrack problem (see Section 6), LAO* is orders-of-magnitude faster than VI when there is no error model, but it is only about twice as fast once the error model is taken into account. The gap is even more striking in more complex problems such as the driving domain (see Section 6). *How can one exploit the power of reachability analysis while allowing for execution-time errors?*

4 Finding k-Fault-Tolerant Plans with LAO*

The intuitive motivation for separating non-faulty operation from faulty operation is that faults are often exceptional and rare. While it's possible that a driver misses a turn, the likelihood that it happens frequently is low. This suggests that planning in advance for a bounded number of faults – and thus limiting the number of reachable states covered by the plan – could be beneficial. For search-based MDP solvers such as LAO*, this could reduce planning time dramatically in exchange for optimality with regard to the full error model. And, as we will show later, when more errors than planned for occur, we can still operate efficiently thanks to online planning during plan execution.

We can use LAO* to produce a k-fault-tolerant plan by applying it to an augmented MDP that accounts for a bounded number of faults. The state space of the augmented MDP is $(s, j) \in S \times \{0, 1, ..., k\}$ where s is a state in the original state space and j is a bounded fault count, $0 \le j \le k$. The initial state of the augmented MDP is $(s_0, 0)$, indicating that no faults occurred. The transition function of the augmented MDP is defined as follows. When $j = k$, faults can no longer occur, so the transition function is identical to the original transition function T:

$$\forall s, a, s' \;\; Pr((s', k)|(s, k), a) = T(s'|s, a) \qquad (3)$$

When $0 \le j < k$ the transition function needs to factor the possibility of a fault occurring:

$$\forall s, a, s' \;\; Pr((s', j)|(s, j), a) = \xi(a|s, a)T(s'|s, a) \qquad (4)$$

$$\forall s, a, s' \;\; Pr((s', j{+}1)|(s, j), a) = \sum_{a' \neq a} \xi(a'|s, a)T(s'|s, a')$$

Finally, the cost function and goal states remain the same (ignoring the fault count component of the state).

Proposition 3. *The solution produced by LAO* for the above augmented MDP provides an optimal solution for the k-fault-tolerant planning problem.*

We now establish and discuss several useful properties of k-fault-tolerant plans.

Lemma 1. *Let V_k^* denote the value function of an optimal k-fault-tolerant plan. When the error model is with undesirable faults, $\forall s : V_{k-1}^*(s, k-1) \le V_k^*(s, k-1)$.*

Proof. (Sketch) Note that this is equivalent to proving $\forall s : V^*(s) \le V_k^*(s, k-1)$ (it is easy to see that for any k, $V_k^*(s, k)$ and $V^*(s)$ correspond to the same problem). Suppose value iteration is used to compute value function V_k, and all $V_k(s, k-1)$ are initialized to $V^*(s)$. Then, due to the definition of undesirable faults and the fact that $V_k^*(s, k) = V^*(s)$, for all states $(s, k-1)$ the value $V_k(s, k-1)$ must converge to some value $V_k^*(s, k-1) \ge V^*(s)$. Since value iteration converges to a unique optimal solution regardless of the initial value, this completes the proof. □

Proposition 4. *When the error model is with undesirable faults, $\forall s : V_{k-1}^*(s, j) \le V_k^*(s, j)$, for all $0 \le j \le k-1$.*

The MDPs corresponding to the k-fault and $(k-1)$-fault-tolerant problems are exactly the same except for all states of the form $(s, k-1)$; Lemma 1 shows that $V_{k-1}^*(s, k-1) \le V_k^*(s, k-1)$. Moreover, there are no transitions from any state $(s, k-1)$ to (s', j) with $j < k-1$. Thus the optimal cost for the remaining states (which have the same transition function and costs) must be at least equal (or higher) in the k-fault-tolerant problem.

Proposition 5. *Let $h(s)$ be an admissible heuristic for the original MDP. When the error model is with undesirable faults, $h(s) \le V^*(s) \le V_k^*(s, j) \le \bar{V}^*(s)$, for all $0 \le j \le k$.*

The first inequality holds because the heuristic is admissible. The second and third inequalities result directly from Proposition 4 by noting that $V^*(s) = V_0^*(s, 0)$ and that $\bar{V}^*(s) = \lim_{k \to \infty} V_k^*(s, j)$ for all $0 \le j \le k$.

Theorem 1. *When the error model is with undesirable faults, a k-fault-tolerant plan π computed by LAO* is an optimal fault-tolerant plan, if for every reachable state (s, k) in the plan, s is fault-free.*

Proof. The k-fault-tolerant plan is created by solving an MDP that factors the *full* error model for all states (s, j) with $j < k$. Only when $j = k$, we "distort" the transition function to allow no more faults. But for every fault-free state, the "distorted" transition is identical to transition in the augmented MDP with the full error model. Hence, in that case, for every state covered by the plan we are in fact solving the problem with the full error model. However, this might not be the case for states in the explicit graph that are outside the best partial solution graph. That is, when considering the full error model, the value of these states can possibly change. But Proposition 5 shows that, with undesirable faults, considering the full error model can only increase the cost of these states, so they must not be part of the optimal plan. □

It is easy to show with a counterexample that Theorem 1 does not necessarily hold when faults are not undesirable.

Theorem 2. *A k-fault-tolerant plan π produced by LAO* is an optimal fault-tolerant plan with respect to initial state s_0, if for every state (s, k) expanded by LAO* during the planning process, s is fault-free.*

Proof. (Sketch) This follows by a similar argument to that used for Theorem 1. However, in this case faults need not be undesirable, since we are solving the exact same problem for all states expanded during the planning process. □

5 Continual Planning Paradigm

One problem with executing k-fault-tolerant plans is that the number of faults at runtime may exceed k. To address that, we propose a *continual planning paradigm* that can handle an *unbounded* number of faults by using multiple k-fault-tolerant plans. Continual planning refers to a variety of methods for interleaving planning and plan execution in situations where complete offline planning cannot address all runtime contingencies [Brenner and Nebel, 2009; desJardins *et al.*, 1999]. In our case, the idea is to create a new k-fault-tolerant plan once the k-th fault occurs. As long as we can produce the new plan before another fault occurs, planning will not delay execution. This incremental planning during execution continues until the goal is reached.

The continual planning paradigm is described in Algorithm 1. The general approach is to generate a k-fault-tolerant plan and execute it until the k-th fault occurs. At that point, two things happen in *parallel*: a new k-fault-tolerant plan is created (Line 6) while executing the next action of the existing plan (Line 5). In the experiments we conducted, the new plan was produced during the execution of one real world action (e.g., executing one driving maneuver), but the approach can be easily extended to address longer planning times.

There is one complication in this online planning process. Since the new plan will be activated from a start state that is yet unknown, all the possible start states need to be taken into account, including those reached as result of another fault. Rather than modify LAO* to handle multiple start states, we create a new dummy start state in Line 4 that leads via a single zero-cost action to all the start states we may encounter when planning is completed. For the sake of clarity, Algorithm 1 describes a straightforward implementation. Several possible improvements of this paradigm that make better use of idle time are discussed in the conclusion.

Evaluating the Continual Planning Paradigm Because it is often hard to assess precisely the quality of the intermediate plans, continual planning paradigms are often hard to evaluate analytically and require extensive experimentation. However, our paradigm is amenable to a precise analytical evaluation. Let $\pi_k(s,j)$ be a *universal* optimal k-fault-tolerant plan computed using value iteration over the entire state space. At runtime, we always execute $\pi_k(s,j)$ with the following rule. Whenever we reach a state (s,k) (once the k-th fault occurs), we execute one last action from the old plan and, if the outcome state is s' (as a result of non-faulty or faulty operation), we move to state $(s',0)$ and start executing the new plan.

More formally, consider the Markov chain defined over states of the form (s,j), with initial state $(s_0,0)$ and the following transition function, for any $s \in S, 0 \leq j < k$:

$$\begin{aligned} Pr((s',j)|(s,j)) &= \xi(\pi_k(s,j)|s,\pi_k(s,j))T(s'|s,\pi_k(s,j)) \\ Pr((s',j+1)|(s,j)) &= \sum_{a' \neq \pi_k(s,j)} \xi(a'|s,\pi_k(s,j))T(s'|s,a') \\ Pr((s',0)|(s,k)) &= \sum_{a'} \xi(a'|s,\pi_k(s,k))T(s'|s,a') \end{aligned}$$

The last transition probability from (s,k) to $(s',0)$ indicates the transition to a new plan. Let V_{cp} denote the value function defined over this Markov chain. Then we have:

Algorithm 1: Continual Planning Paradigm

```
  input: MDP problem $\langle S, A, T, C, s_0, G \rangle$, error bound $k$
1 $(s,j) \leftarrow (s_0,0)$;
2 $\pi \leftarrow$ Find-k-Fault-Tolerant-Plan($(s_0,0),k$);
  while $s \notin G$ do
      if $j \neq k$ then
3         $(s,j) \leftarrow$ ExecutePlan($s, \pi(s,j)$);
      else
4         Create new state $\hat{s}$ with one zero-cost action $\hat{a}$
          s.t. $\forall s' \in S : Pr((s',0)|\hat{s},\hat{a}) = T'(s'|s,\pi(s,j))$;
          do in parallel
5             $(s,j) \leftarrow$ ExecutePlan($s, \pi(s,j)$);
6             $\pi' \leftarrow$ Find-k-Fault-Tolerant-Plan($\hat{s},k$);
7         $\pi \leftarrow \pi'$; $(s,j) \leftarrow (s,0)$;
```

Proposition 6. *$V_{cp}(s_0,0)$ provides the expected value of the continual planning paradigm applied to the given problem.*

6 Experimental Results

We illustrate and evaluate the developed algorithms using two applications domains. One is a moderate size domain that extends the classical racetrack domain widely used as a reinforcement learning (RL) testbed [Sutton and Barto, 1998]. The other domain is based on a rich simulation of a driving scenario that we developed. We describe the two domains, their associated error models, and the obtained results.

6.1 Experiments with the Racetrack Problem

Domain Description Our first domain is a modified version of the racetrack problem described in [Sutton and Barto, 1998]. The original problem involves a simple simulation of a race car on a discrete track of some length and shape, where a starting line has been drawn on one end and a finish line on the opposite end of the track. The state of the car is determined by its location and its two-dimensional velocity. The car can change its speed in each dimension by at most 1 unit, giving a total of nine possible actions. After applying an action there is a probability p_{slip} that the resulting acceleration is zero, simulating failed attempts to accelerate/decelerate because of unpredictable slips on the track. The goal is to go from the start line to the finish line in as few moves as possible. For a complete description of the original problem, the interested reader can refer to [Sutton and Barto, 1998].

We modified the racetrack problem in several ways. Besides increasing the domain size, we use a threshold *MDS* (maximum deterministic speed) below which the actions of the car are deterministic. Additionally, we introduce the concept of a fault-prone location, where, with probability p_{er}, the action performed will be different from the intended action. Finally, we handle collisions with the boundary differently: if the projected path of the car is determined to cross the edge of the track at any square not on the finish line, the car halts in that location. Moreover, in any of these locations, the car can only perform recovery actions (each with a cost of 10) that bring it back to the racetrack in one move.

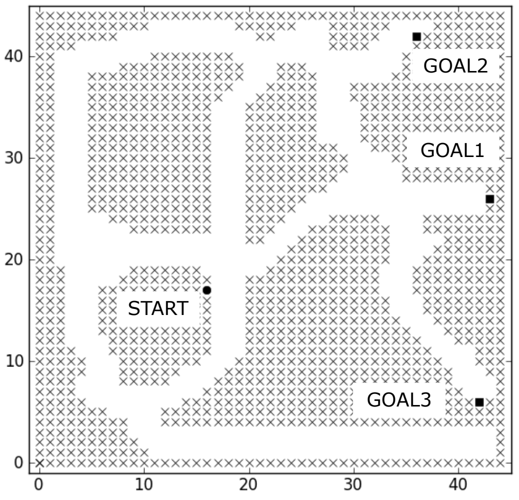

Figure 1: The racetrack domain with 3 different goal states

For the fault-prone locations, the error model allows only certain action selection errors: a faulty action may differ from the intended action by at most one unit in one of the dimensions (vertical or horizontal). For example, if the intended action is "accelerate horizontally by 1 unit and decelerate vertically by 1 unit", the possible faulty actions are: "accelerating horizontally by 1 unit" and "decelerate vertically by 1 unit."

Results Figure 1 shows the racetrack used in our experiments, which includes 20,147 states. We performed three sets of experiments, each one corresponding to a different goal location. In all cases $p_{slip} = 0.1$, $MDS = 3$, and $p_{er} = 0.05$ with all the locations inside the track being error-prone. Four planning approaches were evaluated. The first two are value iteration (VI) and LAO* (LF) using the full error model. The third (PE) is a continual planning approach in which planning ignores the error model, but when a run time error occurs, the car is brought to a stop and a new plan is computed from the current location. The last approach (CP) is our k-fault continual planning approach using $k = 1$. In all cases we used the deterministic shortest-path heuristic for LAO*.

Table 1 (left) shows the average CPU time spent on planning for the three set of experiments, computed based on 100 simulations for VI and LF, and 500 simulations for the continual planning approaches. Note that using LAO* with the full error model doesn't reduce planning time significantly with respect to value iteration (about 60% reduction). However, both continual planning approaches reduce planning time by an order of magnitude with respect to using LAO* with the full error model. Our proposed continual planning approach is faster because re-planning is done without stopping execution, therefore not having any added overhead each time an error occurs.

	CPU Time				Total Cost			
	VI	LF	PE	CP	VI	LF	PE	CP
GOAL1	7,452	2,939	242	113	24.954	15.929	12.603	10.665
GOAL2	7,452	3,124	553	262	31.334	22.678	20.532	17.573
GOAL3	7,452	3,366	613	456	30.194	22.020	19.555	17.093

Table 1: Average planning time (in milliseconds) and total cost of planning and execution for the racetrack domain

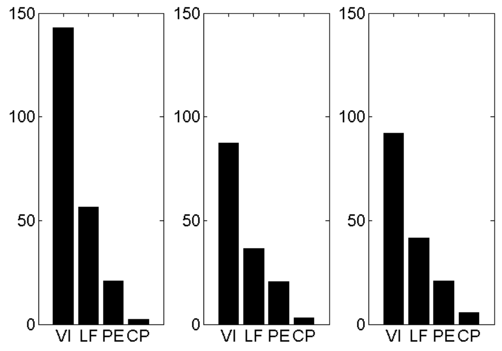

Figure 2: Relative difference in total cost with respect to the lower bound for the racetrack domain

We quantify the trade-off between execution and planning time by setting up a particular duration for each action in the real world, 500 milliseconds per action in this example. This is equivalent to associating a cost of 0.002 per millisecond of planning, allowing us to combine planning and execution costs and compute the total expected cost of all four approaches. These numbers are also shown in Table 1 (right). In all cases the total cost of using CP is more than 20% smaller than the cost of using LF (33% for GOAL1). Note that the PE approach loses some of its advantage with regards to the approaches using the full error model, since it has the additional cost of stopping the car each time an error happen (about 2.7 steps on average) as well as the re-planning overhead.

We compared the total expected cost of the four approaches with a theoretical (loose) lower bound equal to the optimal cost with the full error model, without considering planning time. Figure 2 shows the relative increase of expected cost per method with respect to this bound. In all cases, the CP approach differs from the bound by less than 5%, while the rest of the approaches yield a much greater increase of (at least) 20%, 35% and 85% for PE, LF and VI, respectively.

We also conducted experiments to measure the influence of the parameter k, up to $k = 3$, on the continual planning approach. These experiments showed that the decrease in cost resulting from a larger k was not enough to offset the increase in computational time needed to construct the plan with a larger set of reachable states. Note that the expected cost obtained with the CP approach using $k = 1$ (ignoring planning time) is already within 1% of the optimal cost. While this value can be further reduced with a larger k, the computational time could also increase by as much as an order of magnitude. For instance, for $k = 3$, the CPU time in the fourth column of Table 1 increases to 1,115, 2,996, and 7,154 for GOAL1, GOAL2, and GOAL3, respectively, more than an order of magnitude higher than the CPU time for $k = 1$. Although these results are specific to the racetrack domain, we expect that values of $k \leq 2$ would lead to the best overall performance in other domains.

6.2 Experiments with the Driving Domain

Domain Description We also evaluated the fault-tolerant planning approach in a simulated driving domain. The problem involves driving a car from an initial location to a des-

	CPU Time			Total Cost		
	VI	PE	CP	VI	PE	CP
GOAL1	256,219	3,128	2,782	311.02	82.33	63.08
GOAL2	256,219	4,030	3,905	346.72	119.03	98.41
GOAL3	256,219	4,102	3,877	348.42	117.40	102.78

Table 2: Average planning time (in milliseconds) and total cost of planning and execution for the driving domain

tination. The objective in this domain is to minimize the cost of travel, where the overall cost function involves both the cost of time and cost of human effort. To create the environment, we used the OpenStreetMap (OSM) package (http://wiki.openstreetmap.org), which is the result of a collaborative project to create a free editable map of the world. Each road is composed of a series of discrete locations and the underlying graph is available for manipulation by users via several existing tools and APIs. In our testbed, roads could have different traffic conditions and can be tagged as having various degrees of driving difficulty. Some roads can be traveled in both directions and others in one way only.

We created a user interface that allows us to upload a region of the map, identify start and destination locations, and display an icon of the vehicle on the map during plan execution. Different colors show the level of difficulty navigating different segments of the map. The state of the vehicle includes four main components: a controlled component describing the underlying state of the vehicle including its position, uncontrolled component that includes such features as traffic conditions, stability indication, and the driver state.

The available actions include moving the vehicle forward, turning at intersections, as well as continuing the currently executed action. When an action is performed, the vehicle enters an intermediate "unstable" state for the duration of the action, during which no new actions can be executed. There is uncertainty about the time it takes to restore stability (or complete an action). Faults in this domain represent driver errors such as missing a turn or taking the wrong exit.

Results We experimented with an instance of this domain that involves driving in a segment of Manhattan. The size of the state space was 2,198,400 states. We performed three sets of experiments, each one corresponding to a different start location. LAO* using the full error model ran out of memory, thus we only present results for the three other planning approaches. VI and CP are the same methods as described in the previous section. PE here is a continual planning approach in which planning ignores the error model and whenever an error occurs, the system completes the currently executed action and then waits for a new plan to be computed from the resulting state. In all cases we used a heuristic computed by A* applied to a deterministic version of the problem.

Table 2 (left) shows the average CPU time spent on planning by each method. The average was computed over 50 simulations for VI and 100 simulations for the continual planning approaches. Both continual planning approaches reduce planning time by two orders of magnitude with respect to VI. Moreover, in all cases the CP approach is better than the PE approach by approximately 5% to 10%.

Table 2 (right) shows the expected total cost of the three

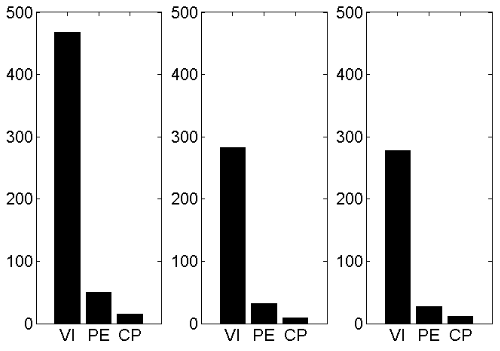

Figure 3: Relative difference in total cost with respect to the lower bound for the driving domain

planning approaches. In this case we trade-off execution and planning time by setting up a cost of 1.0 per second of planning. In all the test cases, the total cost of using CP is significantly lower than the cost of using VI, by more than 70%. Figure 3 shows the relative increase in cost of the three approaches with respect to the same lower bound used in the racetrack domain. In all cases, the CP approach is much closer to the theoretical optimal value, by (at least) a factor of 2 relative to PE and a factor of 20 relative to VI.

7 Conclusion

We present an effective way to handle runtime faults in planning under uncertainty. We target problem domains with a large number of states, where planning algorithms such as LAO* have a profound advantage over VI. We define formally the notion of execution time faults and confirm empirically that the advantage of LAO* diminishes in the presence of an error model that usually increases the number of reachable states by a large factor. Creating optimal plans that factor an unlimited number of execution-time errors is often intractable, even in modest-size domains.

To address this challenge, we propose and analyze an approach that is based on incrementally generating plans that can handle a bounded number of faults, referred to as k-fault-tolerant plans. When the fault bound is reached at runtime, a new plan is created online. A continual planning paradigm based on this approach is shown to produce near-optimal results in two complex stochastic domains.

In the future, we plan to consider ways to improve the performance of the continual panning paradigm by using an *anytime* version of LAO* [Zilberstein, 1996] to create the k-fault-tolerant plans. This, combined with a meta-level reasoning component that will optimize the time allocation to planning – particularly the offline planning needed to generate the initial plan that delays the start of plan execution – will help to further reduce the overhead of planning and increase the value of the plan. Employing an anytime algorithm will also guarantee the availability of the new plan when it is needed.

Finally, when $k > 1$, creating a new k-fault-tolerant plan could start before the k-th fault occurs, or even before the first fault occurs. Designing more effective ways to plan during execution in the spirit of the continual computation approach proposed by Horvitz [2001] could be beneficial in our setting.

References

[Bonet and Geffner, 2003] Blai Bonet and Hector Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 12–21, 2003.

[Brenner and Nebel, 2009] Michael Brenner and Bernhard Nebel. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331, 2009.

[desJardins *et al.*, 1999] Marie E. desJardins, Edmund H. Durfee, Charles L. Ortiz, and Michael J. Wolverton. Continual planning and acting in dynamic multiagent environments. *AI Magazine*, 20(4):13–22, 1999.

[Dubash *et al.*, 1992] Rumi M. Dubash, I-ling Yen, and Farokh B. Bastani. Fault tolerant process planning and control. In *Proceedings of the Sixteenth Annual International Computer Software and Applications Conference*, pages 188–193, 1992.

[Ginsberg, 1989] M. L. Ginsberg. Universal planning: an (almost) universally bad idea. *AI Magazine*, 10(4):40–44, 1989.

[Hansen and Zilberstein, 1998] Eric A. Hansen and Shlomo Zilberstein. Heuristic search in cyclic AND/OR graphs. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 412–418, Madison, Wisconsin, 1998.

[Hansen and Zilberstein, 2001] Eric A. Hansen and Shlomo Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

[Horvitz, 2001] Eric Horvitz. Principles and applications of continual computation. *Artificial Intelligence*, 126(1-2):159–196, 2001.

[Jensen *et al.*, 2004] Rune M. Jensen, Manuela M. Veloso, and Randal E. Bryant. Fault tolerant planning: Toward probabilistic uncertainty models in symbolic nondeterministic planning. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, pages 335–344, 2004.

[Meuleau and Smith, 2003] Nicolas Meuleau and David E. Smith. Optimal limited contingency planning. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, pages 417–426, 2003.

[Puterman, 1994] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[Schoppers, 1987] Marcel J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, 1987.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1998.

[Zilberstein, 1996] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.