Fragment-based Conformant Planning

James Kurien
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 93404
jkurien@parc.com

P. Pandurang Nayak
Stratify, Inc.
501 Ellis Street
Mountain View, CA 94043
nayak@stratify.com

David E. Smith
NASA Ames Research Center
MS 269-2
Moffett Field, CA 94035
desmith@arc.nasa.gov

Abstract

With complex systems such as spacecraft, we often need to achieve goals even though failures prevent the exact state of the system from being determined. Conformant planning is the problem of generating a plan that moves a system from any of a number of possible initial states to a goal state, given that actions may have uncertain outcomes and sensing is unavailable. Two existing approaches to conformant planning are to consider the effects of actions in all worlds simultaneously, or to generate a plan in one world and test it in the remaining worlds. In contrast, in this work we attempt to find a plan for one world and extend it to work in all worlds. This approach is motivated by the desire to find conformant plans when one exists and partially conformant plans when one does not. It can be implemented with many underlying planning approaches and search strategies, and can be used in an anytime manner. We show that on a familiar conformant planning domain this approach is competitive with all but the fastest planners on serial problems and dominant on problems where a parallel plan is required.

Introduction

With complex systems such as spacecraft, we are often faced with situations where we need to achieve goals even though there are faults present. Unfortunately, these systems are only partially observable, or observations may be costly. For example, sensors dedicated to measuring the internal state of spacecraft are usually quite minimal due to power and weight constraints. Thus, if we diagnose a fault aboard a spacecraft, we might do no better than finding a small set of failures that are equally likely given the limited set of observations. Our problem then is that we need to generate plans that achieve goals even though the exact failure that occurred, and thus the exact state of the spacecraft, cannot be determined. This is a *conformant planning problem*.

Conformant planning is a generalization of deterministic planning wherein the task is to generate a plan that moves a system from any one of a number of possible initial states to a state that satisfies a goal predicate. In addition, actions may have uncertain outcomes and sensing actions are not available. The computational challenge of conformant planning lies in the fact that the effects of a plan when executed

in one state may be different and highly undesirable when the plan is executed in a different state. Thus one cannot choose an action based on its desired effect given one possible initial state of the system (called a *world*) without in some way considering its unintended effects when it is executed in all other possible initial states.

One approach to conformant planning has been to consider the effects of each action under consideration across all worlds simultaneously. The CGP planner (Smith & Weld 1998) creates a GraphPlan-style planning graph (Blum & Furst 1995) for each world and adds mutual exclusion constraints between them. When an action is selected for inclusion in the plan, its effects across all worlds are simultaneously captured by the multiple planning graphs. CMBP (Cimatti & Roveri 2000) encodes the possible initial states of the world into a binary decision diagram (BDD). A plan action maps a BDD that represents a set of worlds onto a new BDD that represents the outcome of the action on each world in the initial BDD. Actions are applied to the initial world BDD and all resulting BDDs until a BDD containing only goal states is found. The path of actions leading to the goal BDD is the conformant plan. GPT(Bonet & Geffner 2001) also considers how an action maps a set of possible states onto a set of resulting states, but relies upon search heuristics rather than compact state set encodings to achieve efficiency. It uses A* search in the space of world sets rather than breadth-first search as used by CMBP. An admissible heuristic for the A* search is formed using a fully observable version of the planning problem. Intuitively, the cost of reaching a goal state from a set of states is approximated by the maximum cost of reaching the goal state from any state in the set. HSCP (Bertoli, Cimatti, & Roveri 2001) is a successor to CMBP that uses both a compact, BDD encoding of the belief state and a heuristic search. HSCP searches backwards from the goal, attempting to find a set of actions that lead to a belief state that contains the initial states. Rather than an admissible heuristic, HSCP selects the action that yields the belief state with the highest cardinality. Intuitively, this action leads the largest number of states to the goal given the plan constructed thus far.

Techniques with a generate and test flavor have also been attempted. In *C*-PLAN (Castellini, Giunchiglia, & Tacchella 2001) a possible plan is a sequence of actions that reaches the goal from at least one initial world. A valid plan

Time	Action	Time	Action
1	dunk p1	7	dunk p4
2	flush	8	flush
3	dunk p2	9	dunk p5
4	flush	10	flush
5	dunk p3	11	dunk p6
6	flush		

Figure 1: Plan for Bomb In the Toilet with 6 Packages

is a sequence of actions that achieves the goal from each initial world, and for which every action's preconditions are met in each world. The valid conformant plans are thus a subset of the possible plans. Intuitively, *C*-PLAN encodes the planning domain and goal as a propositional formula and allows a satisfiability procedure to choose an initial world and possible plan for that world. The plan is then tested for validity as a conformant plan. The main effort of this approach is in limiting the number of possible but invalid plans the planner generates.

In this work, we take a different approach to conformant planning in that we attempt find a plan that works in a single world and *extend* it to work in all worlds. To illustrate our approach, consider the bomb in the toilet problem (McDermott 1987), an oft-used benchmark in the conformant planning literature. In this domain, we are presented with a set of packages. One of the packages contains a bomb. In order to defuse the bomb, we must dunk it in a toilet. Once a toilet has been used to dunk a package, it clogs and must be flushed before another package can be dunked. Figure 1 illustrates a successful plan for the six package, one toilet problem requiring 11 time steps, the minimal number. Note that given 11 time steps there are 7^{11} possible action sequences and 6! conformant plans. Note also that the first action of this conformant plan is itself a plan to achieve the goal in the world wherein the bomb occupies p1. Similarly, the conformant plan contains a one-step plan for every other world. It also contains actions, in this case flushing the toilet, that reconcile the post-conditions of the plan for each world with the preconditions for another world.

These characteristics are not specific to this particular conformant plan or even this planning domain. By definition, if a conformant plan exists for a goal and a set of worlds, then the conformant plan must contain a set of actions that achieve the goal in each individual world. This property can be made more useful if stated slightly differently. If a conformant plan exists for a set of worlds, there must exist some plan for each world that can be augmented to form a conformant plan. In particular, we must add actions that achieve the goal in other worlds and ensure the pre- and post- conditions of all actions are satisfied. This insight forms the basis of our conformant planning approach, which we refer to as *fragment-based* conformant planning.

In fragment-based conformant planning, we choose a world from the set of possible initial states and find a plan to achieve the goal in that world. We refer to this single-world plan as a *fragment*, as it may be a fragment of a conformant plan. We then choose a second world and plan. During this second planning attempt, the planner is forced to include the

actions from the first fragment. In effect, we are attempting to extend the initial fragment into a plan that is conformant for both worlds. If we are successful for these two worlds we try to expand the plan to a third world, and so on. If we can find a plan for the final world that includes the fragments found for all other worlds, we have a conformant plan. Interestingly, the plan assembled after considering only a fraction of the worlds may be conformant. In this case, each remaining planning attempt very quickly discovers the plan succeeds in the next world and returns an empty fragment.

Note that while every conformant plan has a fragment for each world, not every fragment for a world can be extended into a conformant plan. The chosen fragment for achieving the goal in one world may conflict with every fragment for achieving the goal in a world that has not yet been considered. Numerous search strategies are possible. A simple strategy is to perform complete backtracking on the possible fragments of each world before backtracking to the previous world. Alternatively, we can use stochastic sampling of the fragments for each world. In either case, the order in which we select worlds is a consideration, since some worlds may be tougher than others. We have implemented a fragment based planner, *fragPlan* and have performed experiments that compare different search strategies on problems with different structures. We have also compared *fragPlan* against other planners. The current implementation of *fragPlan* is not as expressive as some of these planners, but it is better adapted to finding partially conformant plans in an anytime fashion. The planners are therefore compared on the bomb in the toilet problem where their capabilities clearly overlap. For this problem, *fragPlan* is competitive with planners other than HSCP on serial problems and is able to generate parallel plans that HSCP cannot.

The remainder of the paper is organized as follows. In the next section, we describe the fragment-based conformant planning approach. We then we discuss a variety of search strategies for controlling a fragment-based planner. In the following section, we discuss our implementation of *fragPlan* as a SAT-based planner, and we present the results on problems from the conformant planning literature. Finally we present areas for future research and conclusions.

The Fragment Planning Approach

In fragment-based conformant planning, we attempt to create a plan that achieves the goal in all worlds by progressively assembling plan fragments that achieve the goal in each world. The approach is based upon the observation that, by definition, a conformant plan achieves the goal in all worlds and thus, for each world, must contain a set of actions that achieves the goal. Intuitively, we can construct a conformant plan by finding the appropriate fragment for each world and assembling them. We will do this by finding a plan for an initial world, w_i, asserting this plan, referred to as a *fragment*, into the domain theory, then planning for the next world w_j. Consider the following bomb in the toilet example with three packages.

Example 1 We have three packages, p1, p2 and p3. To ensure none has a live bomb, each must be dunked in the toilet.

Column 1	Column 2	Column 3	Column 4	Column 5	Column 6
Plan For w_1	Fragments for w_2	Plan For $\{w_1, w_2\}$	Extracted Fragment	Fragments for w_3	Plan For $\{w_1, w_2, w_3\}$
1 dunk p1	1 dunk p1	1 dunk p1	1	1 dunk p1	1 dunk p1
2	2	2	2	2	2 flush
3	3	3 flush	3	3	3 dunk p3
4	4	4	4	4	4 flush
5	5	5 dunk p2	5 dunk p2	5 dunk p2	5 dunk p2

Figure 2: Generating A Plan for Defusing a Bomb in One of Three Routes

Figure 2 illustrates one possible assembly of fragments for this example. In the figure, we refer to the world wherein the bomb is in package i as world w_i. Note in this example we choose to use a planning style wherein plan actions are assigned to fixed points in time, as is done in GraphPlan and SATPlan (Kautz & Selman 1996) style planners. This is not a requirement for fragment-based planning.

Column 1

In world w_1, the bomb is in p1. We first create a plan for w_1, which is simply to dunk p1.

Column 2

We then use this plan for w_1 as a fragment of the conformant plan that must appear in all subsequent plans. This determines the initial conditions for planning for the second world, w_2, where p2 has the bomb.

Column 3

We then plan for the situation wherein the bomb is in p2 and the first action of the plan must be to dunk p1. This results in a plan that succeeds in w_2. We then check that the plan still achieves the goal in w_1. In Column 3, we now have a plan that achieves the goal in all worlds considered thus far. In the next column, we extract a fragment for w_2 from this plan.

Column 4

Conceptually, we can think of the plan in Column 3 as consisting of the fragments that were required for previous worlds (here, *dunk p1*), a fragment required to achieve the goal in w_2 (here, *dunk p2*), and a set of repair actions that allow these fragments to coexist (here *flush*). In fact, there are 11 sets of repair actions that allow the chosen fragments for w_1 and w_2 to coexist[1]. Only four of these allow us to later add a fragment for w_3.

In order to avoid constraining all future plans to the extent possible, we would like to avoid asserting into our fragment set any repair actions that can easily be re-derived later. Currently we use a simple procedure to remove repair actions from a plan. We first remove the fragments from previous worlds. They will be added back in in the next step. We then perform a search that, given w_2, removes any action whose post-conditions are true before it is executed. This generally removes repair actions. If all repair actions are not removed, we may simply encounter additional backtracking. In this example, the plan is reduced to *dunk p2*.

---

[1]If we use only flushes, we can fit 8 variations in the three available time steps. To include a dunk, time 2 and time 4 must contain flushes while time 3 can dunk of any of 3 packages.

```
proc simpleFragPlan(Domain, Remain, Done) {
    select ω from Remain
    Plan=plan(Domain,ω)
    if(no Plan)
      return Failed
    if(Plan does not achieve goal in Done)
      return Failed
    if(Remain − ω = ∅)
      return Plan
    frag=ExtractFragment(Plan,ω)
    return simpleFragPlan(Domain+frag, Remain − ω, Done+ω) }
```

Figure 3: Simple Fragment Planner

Column 5

The fragment extracted from the plan for w_2 is asserted into the planning domain along with the fragment for w_1. Subsequent plans must include these fragments.

Column 6

Finally, we plan for the situation in which the bomb is in p3, the first action must be to dunk p1 and the last action must be to dunk p2. We find a fragment that achieves the goal given the bomb is in p3 and choose a set of repair actions that fit it within the fragments previously asserted for w_1 and w_2. We now have a conformant plan.

Figure 3 illustrates a simple but incomplete fragment-based planner to serve as a strawman. It produces plans in the manner illustrated in Figure 2. We begin with a domain model in $Domain$, the set of initial worlds in $Remain$, and $Done$ equal to the empty set. We select a world ω, and use a deterministic planner to find a plan that succeeds in ω. If no such plan is found, or if the plan does not succeed in the worlds we have previously considered, the procedure gives up. If a suitable plan is found, it is reduced to the actions needed to achieve the goal in world ω. The variable frag then contains a plan that will achieve the goal in world ω. We must now find a fragment for the next world given the fragments we have found for the worlds selected thus far. This is accomplished with a recursive call that adds the current fragment to the planning domain. This forces the next call to the planner to find a plan that integrates achievement of the goal in the next world selected with the pre- or post- conditions of the fragments that achieve the goal within the previously explored worlds.

As stated, the algorithm in Figure 3 does not do any backtracking. Unfortunately, not every plan for a single world ω can be extended into a conformant plan over a set of worlds that includes ω. There are two ways that a set of fragments

```
proc completePlan(Domain, Remain, Done) {
    select ω from Remain
    Choose Plan from Plan(Domain, ω)
    if(no Plan found)
       return false
    if(Plan satisfies all worlds in Done) {
       if(Remain − ω=∅) {
          report Plan as conformant plan
          return true }
       frag=ExtractFragment(Plan,ω);
       return completePlan(Domain+frag,Remain-ω,Done+ω) }
    return false }
```

Figure 4: A Recursive, Complete Fragment Planner

might fail to extend into a conformant plan. First, it may be the case that the fragment we arbitrarily choose for the current world, w_i, may be incompatible with all fragments for achieving the goal in some subsequent world w_j. Second, when we extend the fragment for w_i to achieve the goal under w_j by choosing additional actions, there is no guarantee the extended plan still accomplishes the goal when executed in world w_i. We must check the augmented plan in w_i to ensure it maintains this property. For either one of these failures, we will be forced to undo some subset of the existing choices before any further progress can be made.

Figure 4 illustrates a complete, backtracking planner. Intuitively, a suitable fragment for a conformant plan is one that results in a plan that satisfies the goal in the new world ω plus all worlds considered thus far (the set $Done$), and allows a suitable fragment to be found for the remaining worlds (the set $Remain$). We initially call the planner with $Domain$ equal to the planning domain, $Remain$ equal to the initial world set, and $Done$ empty. As the algorithm recurs, an additional world ω is moved from $Remain$ to $Done$, and the fragment that satisfies the goal in world ω and does not disrupt the existing fragments for $Done$ is added to $Domain$. The non-deterministic operator **Choose** ensures we eventually consider all fragments for a given world. This is a backtrack point if the chosen fragment does not work out. Consider the operation of the algorithm after some number of worlds have have been added to $Done$ and the corresponding fragments added to $Domain$. If no plan can be found for the next world ω, we must backtrack to the fragment choice point at the previous recursion level. If a plan is found and no worlds remain, we have found a conformant plan. Otherwise, we extract from the plan a fragment that represents the actions needed to achieve the goal in ω and add the fragment to $Domain$. We then attempt to complete the conformant plan for the remaining worlds.

Note that the world ordering selected and the order fragments are chosen by **Choose** have no impact on completeness in the procedure of Figure 4. However, they may have a significant impact upon the amount of backtracking that is performed. In Section 3, we discuss strategies for these choices in the context of this complete planning algorithm and an incomplete, randomized variation. Options for implementing Plan are discussed in Section 4.

Finally, we have not yet discussed uncertainty in actions. If we assume Plan returns only plans that achieve the goal in world w under all possible non-deterministic executions, two additional modifications of Figure 4 are necessary. First, we must also check each possible execution of the plan in all worlds that have previously been planned for. Second, when extracting a fragment for a world from a plan, we can discard an action only if it is not required by any non-deterministic execution of the remaining fragment. Since our primary area of concern is on-board spacecraft systems where actions are considered deterministic, in the remainder of the paper we will only consider uncertainty in the initial conditions.

Search Strategies

The complete planning algorithm of Figure 4 uses chronological backtracking. Figure 5 illustrates *fragPlan*, a fragment-based planner with the flexibility to accommodate a variety of search strategies, both complete and incomplete, systematic or randomized. In particular, we consider incomplete, randomized search strategies, given their effectiveness in comparison to systematic algorithms in many domains (Gomes *et al.* 1998; Selman, Kautz, & Cohen 1996). Intuitively, we can think of planning with fragments as analogous to a constraint satisfaction problem. We have a set of variables (worlds) for which we must choose assignments (fragments) so as to satisfy a set of constraints (the conformant planning domain and goal). Decisions about how variables are ordered and which assignments are chosen will directly impact how much backtracking will be required to retract infeasible variable assignments and how quickly a solution will be found. Therefore, each of our search strategies will have to specify the following characteristics.

- *Variable Ordering:* In what order will the search consider the worlds?

- *Frustration Level:* How many unsuccessful fragment combinations will the search consider before backtracking to a previously considered world?

- *Backtracking Distance:* When the search backtracks, how many fragment choice points will it backtrack over?

Note that in the complete case, either systematic or randomized, the answers to these questions have no impact upon completeness. All fragment sets will eventually be attempted, and each conformant plan is one of those sets, so ordering choices impact efficiency only. With an incomplete planner, we explicitly assume only a portion of the possible fragment sets will be considered. The order in which we consider the worlds and fragment choices will have a significant impact upon which subsequent fragments are considered. The goal is to develop search strategies that consider fragment sets where a conformant plan is likely to be found.

The algorithm of Figure 5 differs from the complete, systematic planner of Figure 4 in several respects. First, since the search procedure will be examining and modifying the world ordering, we store it in an explicit stack rather than through recursion. Second, the search need not consider all fragments for a new world given a set of worlds and fragments. Instead, the search gives up on the current fragment

```
proc fragPlan(Domain, Worlds) {
   select ω from Worlds
   worldStack=empty
   loop{
     Plan=Plan(Domain,ω)
     if(Plan≠ ∅ & Plan satisfies all worlds on worldStack) {
       if(Worlds ⊆ worldStack)
         return Plan
       newFrag=ExtractFragment(Plan,ω)
       Domain= Domain + newFrag
       push(ω,worldStack)
       select ω from Worlds }
     if (Plan=∅ or Frustrated() ) {
       // For each world removed from the stack, we must remove
       // the corresponding fragment from Domain
       stack = adjustStack(worldStack,Domain,failures)
       select ω from Worlds }
   }
}
```

Figure 5: Flexibile Fragment Planner

Time	Action	Time	Action
1	dunk p5	7	
2	flush	8	dunk p2
3	dunk p4	9	flush
4	flush	10	
5	dunk p3	11	dunk p1
6	flush		

Figure 6: A Simple Fragment-based Plan That Failed

choices and world selection whenever it becomes frustrated with the number of fragments it has generated for the current world without finding one consistent with fragments it previously chose. Third, when the search gives up, it may undo as many previous world selections as desired and continue the search from there. Finally, the Plan subroutine need not be complete and may employ randomized search procedures. This will be critical to the effectiveness of our search procedures. We have investigated several search procedures that make different choices for variable ordering, frustration level and backtracking distance. Of these, chronological backtracking and two randomized searches that provided interesting experimental results are described below.

Chronological Backtracking

In chronological backtracking, upon failure we undo the last choice made and replace it with its successor until all choices have been considered. To implement backtracking, Plan must be implemented such that successive calls with the same $Domain$ and $ω$ return successive plans according to some ordering, and Frustrated must always be false. Thus, Plan will return plans given the current fragments until all such plans have been exhausted. The adjustStack procedure must then remove the last world from the stack, along with its current choice of fragment, and this world must be selected as the current world $ω$. We then resume generating possible plans for the fragments associated with the worlds remaining on the stack. This strategy considers every possible extension of the current fragment set before reconsidering its previous choice. Consider the situation of Figure 6. Suppose the first fragment chosen was the action *dunk p2* at time step 8. No conformant plan can result from extending this fragment. However the planner does not fail until it attempts to place the dunk action for the sixth package, as illustrated in the figure. The planner must then systematically rule out the exponential number of extensions of the *dunk p2* fragment before reconsidering that fragment.

Stochastic Probing (Langley 1992)

In this context, a probe consists of one selection of the world order and a choice of fragment for each world. To implement stochastic probing, we select a world at random and use a randomized planner to find one possible fragment. We subsequently select worlds randomly from the set of unconsidered worlds, and find fragments that are consistent with the existing fragment set. If we reach a point where we cannot find the next fragment for the world ordering, we throw out all fragments and the world sequence and begin again. To implement this search strategy, we must randomly select worlds that are not on the stack, Plan must be a randomized planner, Frustrated must always be true, and adjustStack must completely empty the world stack and $Domain$. We expect this strategy to work well in situations similar to Figure 6 wherein the problematic choice was made far back in the stack of decisions, but is not detectable until much later.

Bubbling

Bubbling refers to the motion of worlds for which the search is having difficulty finding a plan toward the top of the stack. In bubbling, Plan is randomized and Frustrated becomes true after some small, fixed number of planning attempts. When the planner becomes frustrated in its attempts to find a plan for $ω$ given all of the worlds on the stack, adjustStack pops the last world off the stack. The variable $ω$ is left with its selected value. Thus the search continues for a fragment for $ω$, but within the context of a smaller set of worlds and associated fragments. This continues until we find a plan that satisfies $ω$ and the stacked worlds, or until the stack consists only of $ω$ and we find a plan for it. Intuitively, the problematic value for $ω$ bubbles up the stack until a fragment is found. The search then returns to finding fragments for the worlds not on the stack. We can refine the variable ordering strategy of bubbling to prefer that heavily constrained worlds are solved first, in a manner analogous to squeaky-wheel optimization(Joslin & Clements 1999). We approximate this by by introducing a notion of difficulty. Each time Plan fails to find a fragment, the difficulty of each world on the stack is incremented. After $ω$ has been satisfied, we can select the next world to attempt based upon this estimate of its difficulty. We expect this search to do well in domains where a small subset of the worlds are significantly more difficult to satisfy than the remaining worlds.

Implementation Using a SAT Planner

The fragment-based approach does not require a specific planning approach be used to implement the Plan procedure that is used on individual worlds. It requires that we are able to force the procedure to include actions for previous fragments in its plan for the current world. Partial-order planners begin with an empty plan and add actions that link the initial state to the goal state and remove conflicts between actions. We can enforce the inclusion of fragments by initializing the procedure with a non-empty initial plan consisting of the fragments. In GraphPlan-style planners, the backtracking search selects actions that lead from the goal back to the initial conditions. At each level of this search, we can simply force the search to include the appropriate fragment actions in its set of actions. SAT-plan based procedures build a propositional representation of the possible plans. We can simply assert that the proposition corresponding to selecting each fragment action must be true.

To implement our Plan procedure, we chose to work within the framework of planning as propositional satisfiability and chose Blackbox(Kautz & Selman 1999) as its basis. Blackbox compiles a planning domain into a Graphplan-style plan graph(Blum & Furst 1995) then converts the graph into a propositional formula that describes the planning problem. A satisfying assignment to the variables of the formula corresponds to a plan. Blackbox then calls a SAT procedure to find a satisfying assignment for the propositional formula. In order to represent uncertainty in the initial state of the world, we assume that the set of worlds can be described by a vector of finite domain variables. For example, in the bomb in the toilet domain, each member of the vector represents the presence or absence of a bomb in one package. Given the vector we can easily enumerate a desired set of worlds, such as the case where at most one package has a bomb or the case where any number of packages may have bombs, by enumerating the corresponding assignments to the vector. We consider the ramifications of a world being the actual world by asserting the propositional representation of the corresponding vector assignment into the propositional planning representation. Blackbox then finds a plan that achieves the goals given that the conditions of the world are true. Note that we cannot create a conformant plan by specifying the initial world as a disjunction of the possible worlds, as a SAT solver will simply pick the most convenient mixture of vector assignments from multiple worlds. Even if the case where the vector is of length one, this would only guarantee that there exists a world where the resulting plan reaches the goal. We seek a plan that reaches the goal in every world.

Representing the planning domain as a propositional formula and planning in a single world as propositional satisfiability has two advantages. First, we can employ fast, randomized propositional satisfiability engines. Second, addition of fragments to the planning domain is trivial. We simply assert that the propositional variable corresponding to each action in the fragment must be true. The SAT procedure is then constrained to find only plans wherein those actions are taken. Using the Blackbox style of propositional encoding has two primary disadvantages, both of which involve Blackbox's use of a planning graph to generate the encoding. First, each occurrence of an operator in the graph is encoded with the point in time at which it appears in the graph. Thus the planning process assigns actions to specific points in time. Since we do not know how we will need to extend the existing fragments, this is constraining. Consider the bomb in the toilet plan generated in Figure 6. We have considered worlds w_1 through w_5 and have generated a plan that is conformant for those worlds. It can be made conformant for w_6 simply by adding a dunk and flush action before any dunk action. However, the way the planner has laid out the actions for the first five worlds, the two remaining time steps are not adjacent, and the two necessary actions cannot be inserted. The planner will have to backtrack, backjump or restart to remove the dunk action from time 8. We could partially address this issue with an encoding trick in the propositional representation that would allow us to move fragments so as to coalesce or create places to insert actions. However, since the core issue is flexibility in ordering actions, it might be more fruitful to consider mapping fragment-based planning into a partially-ordered planning framework. This is beyond the scope of this paper. Second, each occurrence of an action in the graph is encoded with a distinct proposition for each possible instantiation of its arguments and clauses that enforce its pre- and post-conditions. Thus, actions with conditional effects cannot be handled directly. In deterministic domains, conditional effects can be represented in Graphplan by splitting each operator and allowing the planner to choose the operator that represents the appropriate conditional effect (Gazen & Knoblock 1997). Intuitively, this approach does not work in the conformant planning case, as the correct action to represent a conditional effect may depend upon the world being considered. This can be addressed by producing a SAT encoding of the plan operators without the intermediate planning graph. This is also beyond the scope of this paper.

Experimental Results

The planning system described has been implemented in C++, making use of existing functionality from Blackbox, satz (Li & Anbulagan 1997), and Graphplan. We illustrate operation of the planner on variations of the bomb in the toilet problem, a simplification of the RING domain[2] (Cimatti & Roveri 2000) and a conformant logistics problem wherein packages must be delivered via a set of roads that might contain mines. In the first subsection below, we compare performance of *fragPlan* on the bomb in the toilet problem against planners from the literature. In the next subsection, we illustrate scalability as the number of possible worlds increases. Finally, we illustrate how the performance of different search strategies varies with the domain.

Performance Comparison on the BTC Domain

The bomb in the toilet domain with clogging, or *BTC*, is a conformant planning benchmark for which performance

---

[2]The full domain requires actions with conditional effects which, as noted above, our implementation does yet not support.

P-T	Steps	GPT	CMBP	HSCP	fragPlan	
					Time	Calls
6-1	11 / 11	0.07	0.01	0.01	0.11	23.68
8-1	15 / 15	0.11	0.20	0.01	0.47	40.90
10-1	19 / 19	1.31	0.71	0.01	2.89	124.52
6-4	8 / 3	1.44	0.41	0.01	0.03	7.5
8-4	12 / 3	8.78	2.74	0.03	0.23	66.43
10-4	16 / 5	59.97	14.42	0.03	0.44	45.70
6-6	6 / 1	8.69	3.29	0.03	0.02	6.90
8-6	10 / 3	68.43	20.71	0.05	0.05	8.23
10-6	14 / 3	486.97	111.83	0.08	0.27	19.68

Figure 7: *fragPlan* and serial planners on BTC

Figure 8: Time versus parallelism on BTC

information is available for many planners. Figure 7 illustrates performance of planners that consider only serial actions versus *fragPlan* on variations of the BTC problem. The first column lists the number of packages and toilets in the problem. The second column lists the minimum number of time steps in a conformant plan for planners that consider a single action at a time, and the minimum number of time steps required for planners that allow parallel actions. The next three columns of the table show results for three planners from the literature. GPT (Bonet & Geffner 2001) is a planner that uses A* search and dynamic programming to solve conformant, contingent and probabilistic planning problems. CMBP (Cimatti & Roveri 2000) is a conformant planner that uses binary decision diagrams (BDD) to represent the outcome of candidate actions in all worlds simultaneously. HSCP (Bertoli, Cimatti, & Roveri 2001) is an impressive successor to CMBP that uses a heuristic to control the actions considered during search. Figures for these planners were taken from (Bertoli, Cimatti, & Roveri 2001) and were generated on an 300Mhz Pentium II PC running Linux with 512M of memory. The final two columns illustrate the performance of *fragPlan* using stochastic probing as the search strategy. The first column lists the time to solve the problem. Since we are using a randomized procedure, timing figures are averages over thirty runs. The next column shows the average number of calls to the Plan procedure to find a fragment. *fragPlan* was run on a 266Mhz Pentium II laptop running Windows 2000 on 288M of memory. Figure 9 illustrates performance of *C*-plan (Castellini, Giunchiglia, & Tacchella 2001), a conformant planner that like *fragPlan* uses a propositional representation that allows parallel actions. *C*-plan generates possible plans that may be conformant and tests whether each is in fact conformant. The label TIME indicates *C*-plan did not find a plan in 1200 seconds. The second column under *C*-plan lists the number of possible plans that are tested for each problem. Figures for *C*-plan were taken from (Castellini, Giunchiglia, & Tacchella 2001) and were generated on an 850Mhz Pentium III PC running Linux with 512M of memory. Relative to these experimental runs, we have made several observations.

Observation 1 *On serial BTC problems,* fragPlan *is competitive with GPT and CMBP but is dominated by HSCP.*

Consider the first three rows of Figure 7 which represent problems with multiple packages and one toilet, {6-1, 8-1,
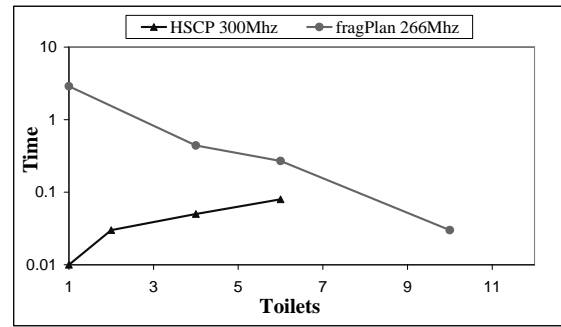
10-1}. These are completely serial problems, in that there is a single toilet, and only one package can be dunked at a time. The task for *fragPlan* is to order the dunking fragments such that the final plan alternates dunking and flushing within the minimal number of time steps. In the six package problem, probing generates an average of 23.68 calls to Plan rather than the minimal 6. This indicates there is some amount of misplacement of fragments which is being addressed by restarting the algorithm. As we increase the number of steps by increasing the number of packages, the amount of restarting required increases. This is intuitive, as there are more opportunities for fragments to be misplaced. However, the number of fragments generated and the difficulty of finding those fragments, as judged by the total runtime of the algorithm, appear to combine to scale at approximately the same rate as problem complexity in the GPT and CMBP planners. HSCP uses the same BDD representation as CMBP, but uses a heuristic function rather than blind search to select actions. This heuristic appears to do an excellent job of focusing the search, and HSCP is significantly faster than *fragPlan* on these serial problems.

Observation 2 *On parallel BTC problems,* fragPlan *dominates CMBP and GPT, and eventually surpasses HSCP.*

Like CMBP, HSCP cannot generate parallel plans in domains where they are required, but will produce serialized plans. Consider for example the performance of CMBP on the problems with ten packages. CMBP encodes the set of initial states of the world in a BDD, then considers the outcome of the possible actions on each world in the set. The result of the action on each possible world is a set of worlds that is also encoded in a BDD. When each world in a BDD satisfies the goal, a plan has been found. As the number of toilets increases, so does the number of actions that must be applied to each BDD. The search branching factor in the 6 toilet problem is 6 times larger than in the 1 toilet problem, while the search depth is less effected since CMBP does not consider parallel actions. This leads to an explosion in the number of BDD's that CMBP must generate. Thus execution time increases from 0.71 seconds to 111.83 seconds. In the case of GPT, we are not certain of the exact mechanism behind the increase in time. We suspect it is a similar explosion in the number of possible partial plans and resulting states of the world it must consider as it performs an

$P-T$	Steps	fragPlan		CPlan	
		Time	Calls	Time	Plans
6-1	11	0.11	23.68	221.55	52561
8- 1	15	0.47	40.90	TIME	-
10-1	19	2.89	124.52	TIME	-
6-10	1	0.02	6	0.01	1
8-10	1	0.02	8	0.01	1
10-10	1	0.03	12.19	0.04	1

Figure 9: *fragPlan* and *C*-plan on BTC

Figure 10: Effect of Worlds on Iterations Per World

A* search over action sequences. The search heuristic in HSCP again does an excellent job of focusing the search, so execution time increases by only an order of magnitude between 1 and 6 toilets. In contrast, the execution time for *fragPlan decreases* by two orders of magnitude between 1 and 10 toilets. Figure 8 illustrates the drop in runtime for *fragPlan* on the 10 package problem as the number of toilets increases, and the corresponding growth in HSCP. Intuitively, *fragPlan* is attempting to assemble the fragments necessary to achieve the goal in each world along with the necessary inter-fragment repair actions, within the allotted number of time steps. Allowing parallel actions significantly simplifies the problem of aligning the necessary fragments to the appropriate time steps.

Observation 3 *On BTC,* fragPlan *dominates* C-*plan.*

Like *fragPlan*, *C*-plan generates parallel plans. Turning to Figure 9, *C*-plan appears to be at a severe performance disadvantage. Conceptually, in order to find a plan of length n, *C*-plan tests every possible plan of length less than n as well as every possible plan of length n. *C*-plan adopts several strategies to reduce the number of possible plans it considers but the number remains considerable. It is not able to find a plan in a competitive amount of time for a serial problem or moderately parallel problems, where the length n and the number of plans at each length are relatively high. See (Castellini, Giunchiglia, & Tacchella 2001) for additional performance figures. However, when the parallelism is sufficient to reduce the number of planning steps to one, *C*-plan does exceedingly well. The first and only possible plan it generates dunks all packages simultaneously, which is a valid conformant plan.

Performance with an exponential set of worlds

A common criticism of planners that explicitly enumerate worlds is that there may be be exponential number of such worlds. In the BTC domain, a problem of n packages has 2^n worlds if we allow that every package may contain a bomb. In the modified RING domain, MRING, a maze contains n rooms. Each room has a window that may be open, closed or locked, yielding 3^n worlds. The goal of MRING is for all windows to be locked, and a robot placed at a known location may close or lock the window in the current room, or move to the next room. Thus the maximum number of worlds is dictated by the number of bombs in BTC and the number of unlocked or open windows in MRING. We have performed a number of experiments that vary the number
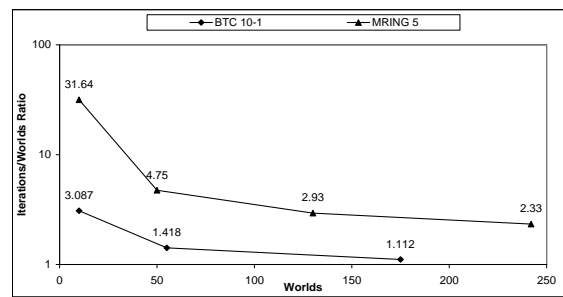
of worlds in the BTC and MRING domains and make the following observations.

Observation 4 *The representation size is constant*

Unlike planners such as CGP, fragPlan does not duplicate its planning representation for each world. The propositions capturing each world (*e.g.,* whether or not package *i* has a bomb) are asserted into the representation in turn. Thus the memory required by fragPlan is dominated by its single world Blackbox representation and does not increase during search. For problems where the worlds are combinations of a fixed number of properties, the only increase in memory to consider additional worlds is a few bytes per world to represent the corresponding fragment.

Observation 5 *Plan Calls Approach The Number of Worlds*

As the number of possible worlds increases for a domain, there tends to be a great deal of overlap between worlds - that is, conformant plans for some of the worlds will also be conformant plans for other worlds. For example, any fragment that solves the BTC world where an entire set of packages have bombs, is a conformant plan for worlds where any subset of those packages have bombs. Similarly for the Ring domain - a fragment that solves the world where a set of rooms have open windows will be a conformant plan for worlds where any subset of those rooms have open windows[3]. As a result, even though the number of possible worlds is growing exponentially with the number of independent sources of uncertainty, the planner tends to discover a conformant plan after considering only a few of these worlds. This observation is confirmed by Figure 10, which shows the ratio of plan calls required for the BTC problem with 10 packages and MRING problem with 5 rooms as the number of possible worlds increases. For all domains that we have tested, this ratio approaches 1 as the amount of uncertainty in the domain is increased.

In other words, for most of the worlds, the planner is just verifying that the current plan works on the current world. We note that this verification problem is polynomial, whereas the planning problem for a single world is

---

[3]This characteristic is not limited to artificial domains such as BTC and Ring. Consider a problem where there are N candidate faults in a spacecraft, and one must plan to achieve a goal. Plans for worlds in which several of the faults are present typically work for worlds in which a subset of those faults are present.
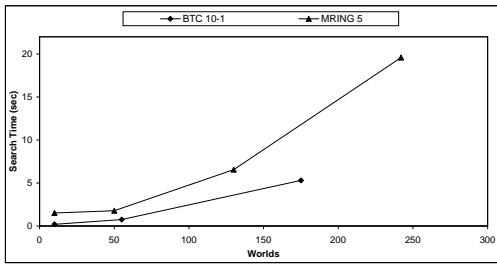
Figure 11: Effect of Worlds on Search Time

Worlds	Relevant Mines	Irrelevant Worlds	Average Calls to Plan	
			Probing	Bubbling
6	8 in w_1	5	33.52	11.39
6	4 each in w_1, w_2	4	44.81	12.61
70	1 each in $w_1 - w_5$	65	510.83	128.93

Figure 12: Probing vs. Bubbling on Asymmetric Worlds

NP-complete. As a result, there is no a priori reason to expect that verification on an exponential number of worlds is computationally worse than planning on a single world. However, our current implementation does not take advantage of the polynomial nature of verification. As shown in Figure 11, we do see growth in search time as the number of worlds increases, but this growth is very slow.

Comparison of Search Strategies for FragPlan

Of the several search strategies we implemented for Frag-Plan, our experiments yielded the most interesting results for stochastic probing and bubbling. Stochastic probing in essence selects a world ordering, chooses a fragment for each world, and restarts as soon as a suitable fragment cannot be found. Bubbling attempts to solve the most difficult worlds first, then add back in worlds that appear increasingly difficult. In general in all of our experiments, it did not prove easy to beat stochastic probing by a significant margin. In cases where the initial worlds displayed significant differences in difficulty, bubbling was able to best probing by a small multiple in performance. The details of these experiments and further performance insights are given below.

Observation 6 *Stochastic Probing Dominates on BTC*

A fragment set may fail to extend into a conformant plan because of a global property of the partial plan rather than a property of any particular fragment. In these cases, we expect bubbling's attempt to solve the most difficult worlds first based upon plan failures will not to lead to a conformant plan. As illustrated in Figure 6, the BTC problem has this property. When the fragment for the final world w_j cannot be placed, it is because of the placement of all of the existing fragments. Bubbling removes the fragment for the previous world considered, w_i. This allows the fragment for w_j to replace the fragment for w_i. The roles then reverse, making bubbling completely ineffective on the BTC problem. Outfitting bubbling with a random restart or asserting previously attempted fragment sets as nogoods would prevent this type of cycle from developing. More generally, locally re-ordering the worlds does not guarantee a plan.

Observation 7 *Bubbling Dominates on Asymmetric Worlds*

In order to further compare search strategies, we defined a logistics problem with uncertainty in its initial conditions. We used this problem to investigate the advantage of probing on problems where a small subset of the worlds are more difficult to plan for than the rest.

Example 2 Consider the problem of delivering relief packages to refugee camps. A depot with packages is located in one location and a number of camps are at distinct locations. Two locations may be connected by an incoming and an outgoing route. One delivery truck and one minesweeper are available. A subset of the routes may be mined.

A plan for this problem must run the minesweeper between the truck's initial location and the depot, drive the truck to the depot, load the truck, and so on. Figure 12 illustrates performance on three cases of this problem. In each problem, five camps require packages and one does not. The first column specifies the number of worlds, where a world is an assignment of mines to routes. In order to create structure in the worlds, not all worlds specify that routes needed to achieve the goal are mined. In the second column world w_1 represents the belief that 8 relevant routes are mined. In the remaining worlds, specified in the third column, the mines are on routes irrelevant to the goal. The final row is an exaggerated case wherein 65 worlds specify mines on a route that is not needed in the plan, and camps are in a linear arrangement that maximizes the number of mines that must be cleared in order to reach the most difficult (farthest) camp. This problem is particularly suited for bubbling, as we must clear the mine on the first route before considering driving to the second route and clearing it. The fourth column denotes the average number of calls to Plan that are made before finding a conformant plan, averaged over 30 plans. While probing does not perform as well as bubbling, even on the extreme case wherein 5 of 70 worlds must be considered in order, it does not do as poorly as we expected. The next observation provides further insight into this behavior.

Observation 8 *Lazy Conformance*

Given the number of worlds in Figure 12, we were surprised that probing did not perform significantly worse. We discovered that plans for world w_i returned by Plan often work in some other world w_j. We think of this as *lazy conformance*. Our SAT encoding specifies the impact of every world upon reaching the goal, even though only one world is asserted when planning. We believe this biases the heuristics of the SAT procedure to activate actions that remove threats to the goal under from w_j when finding a plan for w_i. Intuitively, the heuristics of the SAT procedure see a threat to a precondition of the goal without seeing that the threat is only entailed if w_j is asserted. Even if *extractFragment* removes the extra actions for world w_j from the fragment inserted into the planning domain, they ensure that there is space for such actions to be easily added when w_j is considered. The result is that in this domain fragment-based planning is less sensitive to irrelevant worlds, fragment placement and world ordering than we would have expected.
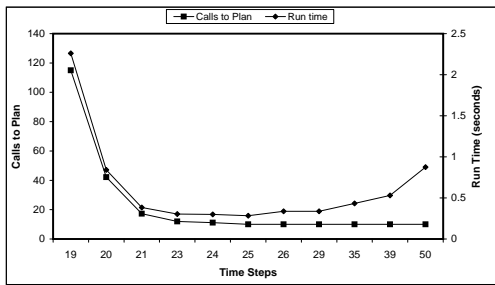
Figure 13: Effect of Time Steps on fragPlan

Observation 9 *Probing improves with longer horizons*

When we use a randomized search within fragPlan, we expect performance to be more sensitive to how tightly the fragments constrain each other than by the total number of action sequences. Figure 13 illustrates the performance on BTC with 10 packages and 1 toilet as the planning horizon expands from 19 steps to 50. Note the calls to Plan decrease to the minimum of one per world while the number of action sequences increases exponentially. Intuitively, placing 10 dunk fragments becomes significantly easier with a few steps of slack. Eventually, the penalty for manipulating a larger representation outweighs the reduction in search. However, even at 50 steps performance is good. We have documented this phenomenon on all domains we considered. For control applications where some plan must be found, this suggests quickly generating a plan given a conservative planning horizon then iteratively shortening the horizon to find shorter plans in the time that remains.

Future Work and Conclusions

As we noted earlier, when there are many independent sources of uncertainty and consequently many possible worlds, it is usually the case that there are a small number of worlds that are sufficient for finding a conformant plan. For example, in the RING domain it is sufficient to solve the problem where all windows are open. Likewise in diagnosis a plan that corrects the simultaneous occurrence of every failure is often a plan that corrects any failure. Of course, this is not guaranteed to yield a conformant plan. For example, if either of a pair of redundant devices is failed, a plan is possible, but no plan is possible if both are failed. This suggests future work on search heuristics that attempt to find or construct worlds for which the plan is the conformant plan.

In this paper we have described a conformant planning method that attempts to incrementally grow the set of worlds in which a plan is valid. The capabilities of this approach are different than other conformant planners in order to address a different set of problems. For non-artificial planning problems, no conformant plan may exist or we may not be able to find one before we are required to act. In both cases, conformant planners return no plan. *FragPlan* can be interrupted at any point and return the most conformant plan it has generated thus far. *FragPlan* is a subroutine of *SCOPE*, the Safe, Conformant, Optimizing Planning Engine(Kurien, Nayak, & Smith 2002). Motivated by our experience with

autonomous systems, SCOPE finds plans that are pareto optimal in terms the number of worlds in which a single goal is met (conformancy), the number of goals of achievement met across all worlds (expected performance) or the number goals of maintenance respected across all worlds (safety).

We have described how the fragment planning method can be implemented to take advantage of many different deterministic planning technologies and search strategies. We have also shown that the search is not as sensitive to growth in the number of worlds as previous possible worlds approaches such as CGP. *fragPlan* is competitive with CMBP on problems within its current expressive power, but is outperformed by HSCP. However, if parallel plans are required, HSCP, CMBP and GPT are not applicable, and *fragPlan* dominates. *FragPlan* does not currently handle actions with uncertain or conditional outcomes as HSCP, CMBP and GPT are able. In exchange, it is better adapted when the existence of a conformant plan, minimal planning horizon or time available for planning time are not known a priori.

References

Bertoli, P.; Cimatti, A.; and Roveri, M. 2001. Heuristic search + symbolic model checking = efficient conformant planning. In *Procs. IJCAI-01*.

Blum, A. L., and Furst, M. L. 1995. Fast planning through planning graph analysis. In *Proceedings of IJCAI-95*, 1636–1642.

Bonet, B., and Geffner, H. 2001. Gpt: A tool for planning with uncertainty and partial information. In *Workshop on Planning with Uncertainty and Partial Information,IJCAI-2001*.

Castellini, C.; Giunchiglia, E.; and Tacchella, A. 2001. Improvements to sat-based conformant planning. In *ECP2001*.

Cimatti, A., and Roveri, M. 2000. Conformant planning via symbolic model checking. In *JAIR*, volume 13, 303–338.

Gazen, B. C., and Knoblock, C. A. 1997. Combining the expressivity of UCPOP with the efficiency of graphplan. In *ECP*, 221–233.

Gomes, C. P.; Selman, B.; McAloon, K.; and Tretkoff, C. 1998. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *Procs. AIPS-98*, 208–213.

Joslin, D. E., and Clements, D. P. 1999. Squeaky wheel optimization. *JAIR* 10.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Procs. AAAI-96*.

Kautz, H., and Selman, B. 1999. Unifying sat-based and graph-based planning. In *Procs. IJCAI-98*.

Kurien, J.; Nayak, P. P.; and Smith, D. E. 2002. Maximizing safety and achievement under time constraints. In preparation.

Langley, P. 1992. Systematic and nonsystematic search strategies. In *Procs. AIPS-92*, 145–152.

Li, C. M., and Anbulagan. 1997. Heuristics based on unit propagation for satisfiability problems. In *Procs. IJCAI-97*, 366–371.

McDermott, D. 1987. A critique of pure reason. *Computational Intelligence* 3:151–160.

Selman, B.; Kautz, H.; and Cohen, B. 1996. Local search strategies for satisfiability testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 26.

Smith, D. E., and Weld, D. S. 1998. Conformant graphplan. In *Proceedings of AAAI-98*.