Synthesis of Fault Tolerant Plans for Non-Deterministic Domains*

Rune M. Jensen, Manuela M. Veloso, and **Randal E. Bryant**
Computer Science Department,Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh,PA 15213-3891, USA
{runej,mmv,bryant}@cs.cmu.edu

Abstract

Non-determinism is often caused by infrequent errors that make otherwise deterministic actions fail. In this paper, we introduce fault tolerant planning to address this problem. An n-fault tolerant plan is guaranteed to recover from up to n errors occurring during its execution. We show how optimal n-fault tolerant plans can be generated via the strong universal planning algorithm. This algorithm uses an implicit search technique based on the reduced Ordered Binary Decision Diagram (OBDD) that is particularly well suited for non-deterministic planning and has outperformed most alternative approaches. However, the OBDDs used to represent the blind backward search of the strong algorithm often blow up. A heuristic version of the algorithm has recently been proposed but is incapable of dynamically guiding the recovery part of the plan toward error states. To address this problem, we introduce two specialized algorithms 1-FTP (blind) and 1-GFTP (guided) for 1-fault tolerant planning that decouples the synthesis of the recovery and non-recovery part of the plan. Our experimental evaluation includes 7 domains of which 3 are significant real-world cases. It verifies that 1-GFTP efficiently can handle non-local fault states and demonstrates that it due to this property can outperform guided fault tolerant planning via strong planning. In addition, 1-FTP often outperforms strong planning due to an aggressive expansion strategy of the recovery plan.

Introduction

As often noted, classical planning with its deterministic actions, static environments, and fully observable states is too restricted to represent most real-world domains. A simple but effective extension is to consider non-deterministic domains where actions may lead to one of several possible next states. In this way, dynamic environments and alternative action behavior can be represented. Compared to MDPs, non-deterministic models have two important advantages. First, they are strictly less expressive which makes it possible to solve larger problems, and second, they avoid the problem of gathering statistical data to estimate probability distributions.

Until recently, though, efficient non-deterministic planning algorithms did not exist. Conditional planning (e.g. Olawsci & Gini 1990) suffers from an exponential growth of the plan with the number of sensing actions, and classical universal planning (Schoppers 1987) lacks an efficient plan representation. One way to by-pass this problem is to use reactive planners (e.g. Koenig & Simmons 1995). However, such approaches are generally incomplete. The introduction of implicit but complete search methods based on the reduced Ordered Binary Decision Diagram (OBDD, Bryant 1986) has changed this picture. In contrast to classical search approaches, OBDD-based search is particularly well suited for non-deterministic domains. In addition, the OBDD constitutes a very compact data structure for representing universal plans. The revived research on non-deterministic planning has led to a large body of novel work on universal planning (Cimatti, Roveri, & Traverso 1998; Jensen & Veloso 2000), adversarial planning (Jensen, Veloso, & Bowling 2001), conformant planning (Cimatti & Roveri 2000), planning with extended goals (Pistore, Bettin, & Traverso 2001), and planning under partial observability (Bertoli, Cimatti, & Roveri 2001). Despite the general success of this research, it faces two challenges: 1) Non-deterministic models of real-world problems are often too abstract to allow solutions of high quality, 2) OBDDs representing the search frontier of blind backward search tend to have a high growth rate in many planning domains.

In this paper, we address both of these problems. With respect to the first, a key observation is that non-determinism in real-world domains often is caused by infrequent errors that make actions fail. In many cases, no actions can be guaranteed to succeed. For such domains, it may be hard or even impossible to generate plans that recover from any combination of errors. We propose a new framework called *fault tolerant planning* to handle this kind of non-determinism. Fault tolerant planning assumes that actions have primary and secondary effects. The primary effect models the usual deterministic behavior of the action, while the secondary ef-

*We would like to thank Sylvie Thiebaux, Piergiorgio Bertoli, Reid Simmons, and Anders P. Ravn for providing case study material. We also wish to thank Bruce Krogh, Nicola Muscettola, and reviewers for rewarding discussions and suggestions. The research is sponsored in part by the Danish Research Agency and the United States Air Force under Grants Nos F30602-00-2-0549 and F30602-98-2-0135. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, the Air Force, or the US Government.

fect models the error effects. Fault tolerant plans are robust to errors occurring during the execution of the plan. However, since errors are assumed to be rare, only a limited number of errors are allowed to happen during any execution of the plan.

We are not aware of any previous work in AI-planning where faults are modeled explicitly as secondary effects. Fault tolerance, however, has been studied in control theory. Basically, two different fault models have been developed: a transition based (e.g. Chen & Patton 1999) and a state based (e.g. Klein & Wehlan 1996). The focus in this work, however, is on theoretical foundation rather than practical application. Universal planning and fault tolerant planning are also related. A fault tolerant plan is not as restricted as a *strong universal plan* that requires that the goal can be achieved in a finite number of steps independent of the number of errors. In many cases, a strong plan does not exist because all possible errors must be taken into account. This is not the case for fault tolerant plans, and if errors are infrequent, they still may be very likely to succeed. A fault tolerant plan is also not as restricted as a *Strong cyclic plan*. An execution of a strong cyclic plan may be infinite due to cycles, but it will never reach states not covered by the plan. Thus, strong cyclic plans also have to take all error combinations into account. The only previous class of universal plans being more relaxed than fault tolerant plans are *weak universal plans*. An execution of a weak plan may reach states not covered by the plan, it is only guaranteed that *some* (maybe just one) execution exists that reaches the goal from each state covered by the plan. Fault tolerant plans are almost always preferable to weak plans. Weak plans for most non-deterministic domains are useless, because they give no guarantees for *all* the possible outcomes of actions. For a fault tolerant plan, any action may fail, but only a limited number of fails can occur.

In this paper, we concentrate on n-fault tolerant planning. An n-fault tolerant plan is guaranteed to reach a goal state as long as at most n faults happen during execution. The question is how to generate these plans. One might suggest to use a classical planning system. Consider for instance synthesizing a 1-fault tolerant plan in a domain where there is a non faulting plan of length k and at most f error states of any action. It is tempting to claim that a 1-fault tolerant plan then can be found using at most kf calls to a classical planning algorithm. This analysis, however, is flawed. It only holds for evaluating a given 1-fault tolerant plan. It neglects that many additional calls to the classical planning algorithm may be necessary in order to *find* a valid solution. Instead, we need an efficient approach for finding plans for many states simultaneously. This can be done by reducing fault tolerant planning to strong universal planning by adding a fault counter to the domain.

However, this approach does not address the second problem of OBDD-based planning which is that the blind backward search used by the strong universal planning algorithm tends to be inefficient. A fruitful idea seems to be to guide the search using a recent best-first version of strong planning (Jensen, Veloso, & Bryant 2003). The approach works well if error states are local and falls within the fraction of the state space traversed by the best-first search. However, faults are often caused by permanent mal-functions that make error states non-local. In this case, the search must be actively guided toward the error states. For this reason, we introduce two specialized algorithms 1-FTP (blind) and 1-GFTP (guided) for 1-fault tolerant planning that decouples the synthesis of the recovery and non-recovery part of the plan. Our experimental results show that 1-GFTP efficiently can handle non-local error states and may have dramatic performance gains compared to guided 1-fault tolerant planning via strong planning. In addition, our experiments indicate that even an unguided version of 1-GFTP called 1-FTP often outperforms 1-fault tolerant planning via strong planning, due to its aggressive expansion strategy of the recovery part of the plan. The experimental evaluation includes 7 domains: DS1, PSR, BeamWalk, PowerPlant, LV, 8-puzzle, and SIDMAR. Of these DS1, PSR, and SIDMAR are real-world case studies.

The paper is organized as follows. We first give preliminaries on heuristic OBDD-based search techniques. We then define n-fault tolerant planning and describe fault tolerant planning via strong planning and the two algorithms 1-FTP and 1-GFTP. Finally, we present experimental results and draw conclusions.

OBDD-based Search Techniques for Non-Deterministic Planning

An OBDD is rooted DAG representing a Boolean function on a set of linearly ordered Boolean variables. It has one or two terminal nodes labeled 1 or 0, and a set of variable nodes. Each variable node is associated with a Boolean variable and has two outgoing edges *low* and *high*. Given an assignment of the variables, the value of the Boolean function is determined by a path starting at the root node and recursively following the high edge, if the associated variable is true, and the low edge, if the associated variable is false. The function value is $true$, if the label of the reached terminal node is 1; otherwise it is $false$. The graph is ordered such that all paths in the graph respect the ordering of the variables. An OBDD representing the function $f(x_1, x_2, x_3) = \neg x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_2 \wedge x_3 \vee x_1 \wedge x_3$ is shown in Figure 1. OBDDs are canonical due to two reduction rules that remove unnecessary tests and reuse structure. Given a "good" ordering of the variables, the reductions may lead to an exponential space saving compared to the truth-table representation of the function. Such orderings are often easy to find in practice. Interestingly, the compactness of OBDDs is inexpensive in terms of their accessibility. Equivalence and satisfiability tests on OBDDs take constant time and binary synthesis $x \otimes y$ has time and space complexity $O(|x||y|)$ (Bryant 1986). Robust software packages exist for manipulating OBDDs. In these packages, graphs of several OBDDs are represented by a multi-rooted OBDD.

OBDDs were originally applied for verification of combinational circuits. Later McMillan 1993 introduced an OBDD-based method coined *symbolic model checking* for verification of sequential circuits and software. The latter technique forms the foundation of OBDD-based non-

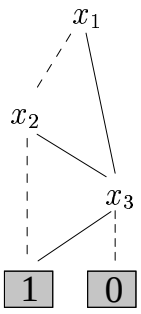

Figure 1: An OBDD representing the function $f(x_1, x_2, x_3) = \neg x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_2 \wedge x_3 \vee x_1 \wedge x_3$. High and low edges are drawn with solid and dashed lines, respectively.

deterministic planning. However, in contrast to symbolic model checking, non-deterministic planning is a synthesis problem rather than a decision problem. This is an important reason for choosing OBDDs rather than SAT approaches to solve these problems. While SAT techniques recently have been very successful in formal verification, they lack an efficient data structure for representing non-deterministic plans.

By using Boolean vectors to represent states and actions, an OBDD can encode a set of states and the transition relation of a search space by representing their *characteristic function*. Assume two sets of Boolean vectors $\vec{v}$ and $\vec{a}$ are used to represent the states and actions of a non-deterministic domain. Any subset of states P and actions Q can then be represented by Boolean functions $P(\vec{v})$ and $Q(\vec{a})$. Similarly, the characteristic function $T(\vec{v}, \vec{a}, \vec{v}')$, where unprimed and primed variables denote current and next states, can be used to represent the transition relation of a search space. Union, intersection, and complement of sets corresponds to disjunction, conjunction and negation on their characteristic function. In the sequel, we will not distinguish between set operations and their corresponding Boolean operations.

The core operation is to find the set of state-action pairs (SAs) where the action applied in the state may cause a transition to a state in C. This can be done by computing the *preimage* of C

$$\textsc{PreImg}(C) = \exists \vec{v}' \,.\, T(\vec{v}, \vec{a}, \vec{v}') \wedge C(\vec{v}').$$

A common problem when computing the preimage is that the intermediate OBDDs tend to be large compared to the OBDD representing the result. Another problem is that the transition relation may be very large if it is represented by a single OBDD. In symbolic model checking, one of the most successful approaches to solve this problem is *transition relation partitioning*. For planning problems, where each transition normally only modifies a small subset of the state variables, the suitable partitioning technique is *disjunctive partitioning* (Clarke, Grumberg, & Peled 1999). In a disjunctive partitioning, unmodified next state variables are unconstrained in the transition expressions and the abstracted transition expressions are partitioned such that each partition only modifies a small subset of the variables. Let $\vec{m}_i$ denote the modified next state variables of partition P_i in a partition $P_1, P_2, \cdots, P_n$. The preimage computation may now skip the quantification of unchanged variables and operate on smaller expressions

$$\textsc{PreImg}(C) = \bigvee_{i=1}^{n} \Big(\exists \vec{m}_i' \,.\, P_i(\vec{v}, \vec{a}, \vec{m}_i') \wedge C(\vec{v})[\vec{m}_i / \vec{m}_i'] \Big),$$

where $[\vec{m}_i / \vec{m}_i']$ substitutes $\vec{m}_i$ with $\vec{m}_i'$ in $C(\vec{v})$.

In guided non-deterministic planning, the SAs in the preimage are divided according to a heuristic function h. For a state s, $h(s)$ estimates the minimum number of actions necessary to reach s from the initial state. Each partition of the preimage contains SAs with identical h-value. It has been shown how this can be accomplished by associating each transition with the change δh it causes in the value of the heuristic function (in forward direction) and constructing a *disjunctive branching partitioning* where each partition only contains transitions with identical δh (Jensen, Veloso, & Bryant 2003).

Let δh_i denote δh of partition P_i. Further, let C be a set of states with identical h-value h_c. By computing the preimage separately for each partition

$$\textsc{PreImg}_i(C) = \exists \vec{m}_i' \,.\, P_i(\vec{v}, \vec{a}, \vec{m}_i') \wedge C(\vec{v})[\vec{m}_i / \vec{m}_i'],$$

we split the preimage of C into n components $\textsc{PreImg}_1, \cdots, \textsc{PreImg}_n$ where the value of the heuristic function for all states of SAs in $\textsc{PreImg}_i$ equals $h_c - \delta h_i$.

Fault Tolerant Planning

A fault tolerant planning domain is similar to a classical planning domain. However, in addition to the primary effect of actions, we add a secondary effect that describes the outcome of a failure. Since an action often can fail in many different ways, we allow the secondary effect to lead to one of several possible next states. Thus, secondary effects are non-deterministic.

Definition 1 (Fault Tolerant Planning Domain) *A fault tolerant planning domain is a tuple $\langle S, Act, \rightarrow, \rightsquigarrow \rangle$ where S is a finite set of states, Act is a finite set of actions, $\rightarrow \subseteq S \times Act \times S$ is a deterministic transition relation of primary effects, and $\rightsquigarrow \subseteq S \times Act \times S$ is a non-deterministic transition relation of secondary effects. Instead of $(s, a, s') \in \rightarrow$ and $(s, a, s') \in \rightsquigarrow$, we write $s \xrightarrow{a} s'$ and $s \overset{a}{\rightsquigarrow} s'$, respectively.*

An action a is *applicable* in a state s iff $s \xrightarrow{a} s'$ for some state s'. An n-fault tolerant planning problem is similar to a classical planning problem with a single initial state and a set of goal states.

Definition 2 (N-Fault Tolerant Planning Problem) *An n-fault tolerant planning problem is a tuple $\langle \mathcal{D}, s_0, G, n \rangle$ where $\mathcal{D}$ is a fault tolerant planning domain, $s_0 \in S$ is an initial state, $G \subseteq S$ is a set of goal states, and $n : \mathbb{N}$ is an upper bound on the number of faults that can occur during a plan execution.*

An n-fault tolerant plan is obviously not a single sequence of actions since faults cause the plan to branch. Instead, we define it to be a function $\pi : S \times \{0, 1, \cdots, n\} \rightarrow 2^{Act}$, where $a \in \pi(s, e)$ implies that a is applicable in s. The intuition is that for a current state s where e faults have occurred, $\pi(s, e)$ is a set relevant actions for reaching a goal state. Notice that this definition assumes that both states and faults are observable. In order to define valid fault tolerant plans, we introduce the *execution* of an n-fault tolerant plan.

Definition 3 (Execution) *Let $\mathcal{P} = \langle \mathcal{D}, s_0, G, n \rangle$ be a n-fault tolerant planning problem and let π be an n-fault tolerant plan for $\mathcal{P}$. An execution of π is a possibly infinite sequence $\langle q_0, e_0 \rangle \ \langle q_1, e_1 \rangle \ \langle q_2, e_2 \rangle \cdots$ of pairs in $S \times \{0, \cdots, n+1\}$ such that, $q_0 = s_0$, $e_0 = 0$, and for all $\langle q_i, e_i \rangle$ in the sequence*

- *either $\langle q_i, e_i \rangle$ is the last pair in the sequence in which case no action a exists such that $a \in \pi(q_i, e_i)$, or*
- *$a \in \pi(q_i, e_i)$, and*
  - *$q_i \xrightarrow{a} q_{i+1}$, $e_{i+1} = e_i$, or*
  - *$q_i \overset{a}{\rightsquigarrow} q_{i+1}$, $e_{i+1} = e_i + 1$.*

We define the length of a finite execution $p_1 p_2 \cdots p_m$ to be $m + 1$. An execution is called *successful* iff $e \leq n$ for any pair $\langle s, e \rangle$ in the execution sequence.

Definition 4 (Valid N-Fault Tolerant Plan) *An n-fault tolerant plan for a planning problem $\langle \mathcal{D}, s_0, G, n \rangle$ is valid iff all successful executions are finite and terminate in a goal state.*

Definition 5 (Optimal N-Fault Tolerant Plan) *An n-fault tolerant plan for a planning problem $\langle \mathcal{D}, s_0, G, n \rangle$ is optimal iff it is valid and its longest successful execution is minimal.*

Fault Tolerant Planning Algorithms

As described in the introduction, n-fault tolerant planning can be reduced to strong universal planning. A *universal planning domain* is a tuple $\langle S, Act, \rightarrow \rangle$ where S and Act have their usual meaning and $\rightarrow \subseteq S \times Act \times S$ is a non-deterministic transition relation of action effects. Thus, in a universal planning domain we do not distinguish between primary and secondary effects of actions. A *universal planning problem* is a triple $\langle \mathcal{D}, s_0, G \rangle$, where $\mathcal{D}$ is a universal planning domain, $s_0 \in S$ is an initial state, and $G \subseteq S$ is a set of goal states. A strong universal plan is a mapping of states to sets of actions guaranteed to reach a goal state in a finite number of steps. An algorithm for synthesizing strong universal plans is shown in Figure 2. The function STATES projects the actions in a set of SAs

$$\text{STATES}(Q) = \{s \mid \exists a \in Act \,.\, \langle s, a \rangle \in Q\}$$

The algorithm builds a strong universal plan incrementally during a blind backward search from the goal states to the initial state. In each iteration (l.2-8), a precomponent of the covered states C is computed and added to the plan. The precomponent is the SAs in the precomponent of C pruned for SAs in the precomponent of the complement of C. These SAs are pruned since they might lead to a state outside of C and thus do not guarantee progress.

function SP(s_0, G)
1 $U \leftarrow \emptyset$; $C \leftarrow G$
2 **while** $s_0 \notin C$
3 $\quad U_p \leftarrow \text{PREIMG}(C) \setminus \text{PREIMG}(\overline{C})$
4 $\quad U_p \leftarrow U_p \setminus C \times Act$
5 $\quad$ **if** $U_p = \emptyset$ **then return** failure
6 $\quad$ **else**
7 $\quad\quad U \leftarrow U \cup U_p$
8 $\quad\quad C \leftarrow C \cup \text{STATES}(U_p)$
9 **return** U

Figure 2: A strong universal planning algorithm.

An n-fault tolerant planning problem $\langle \mathcal{D}_f, s_{0_f}, G_f, n_f \rangle$ where $\mathcal{D}_f = \langle S_f, Act_f, \rightarrow_f, \rightsquigarrow_f \rangle$ is transformed into a universal planning problem $\langle \mathcal{D}, s_0, G \rangle$ where

- $\mathcal{D} = \langle S, Act, \rightarrow \rangle$, s.t.
  - $S = S_f \times \{0, \cdots, n\}$
  - $Act = Act_f$
  - $\langle s, e \rangle \xrightarrow{a} \langle s', e' \rangle \equiv$
    $(s \xrightarrow{a}_f s' \wedge e' = e) \ \vee \ (s \overset{a}{\rightsquigarrow}_f s' \wedge e' = e + 1)$
- $s_0 = \langle s_{0_f}, 0 \rangle$
- $G = \{\langle g, e \rangle \mid g \in G_f, \ e \leq n_f\}$.

Notice that the expression $e' = e + 1$ is false if $e = n$ due to the restriction on the domain of the fault counter. Thus, the universal planning domain exactly models all successful executions of any possible n-fault tolerant plan. A strong universal plan U for the transformed problem is a valid n-fault tolerant plan $\pi(s, e) = \{a \mid \langle \langle s, e \rangle, a \rangle \in U\}$ since it is guaranteed to have finite executions terminating in a goal state. The solution is also optimal since the strong algorithm returns plans with minimum worst case execution length.

The blind search algorithm in Figure 2 applied to an n-fault planning problem is called n-FTP$_S$. As discussed in the introduction the performance of blind OBDD-based search in many practical non-deterministic planning domains is limited. For this reason, we also consider a guided version of n-FTP$_S$ called n-GFTP$_S$. This algorithm is substituting the blind search algorithm of n-FTP$_S$ with the best-first heuristic search algorithm described in (Jensen, Veloso, & Bryant 2003). This algorithm uses a heuristic estimate of the distance to the initial state to guide the search. In each iteration, a complete but partitioned precomponent is computed using a disjunctive branching partitioning. Only the SAs in the precomponent with lowest h-value are added to the strong plan.

We may expect this algorithm to work well when secondary effects are local. In practice, however, secondary effects may be permanent mal-functions requiring considerable recovery activity. Indeed in theory, secondary effects may be uncorrelated with primary effects. This problem calls for specialized algorithms where the planning for primary and secondary effects is decoupled. We concentrate on 1-fault tolerant planning and introduce two algorithms 1-FTP using blind search and 1-GFTP using guided search.

The 1-FTP algorithm is shown in Figure 3. Notice that

function 1-FTP(s_0, G)
1 $F^1 \leftarrow \emptyset$; $C^1 \leftarrow G$
2 $F^0 \leftarrow \emptyset$; $C^0 \leftarrow G$
3 **while** $s_0 \notin C^1$
4 $\quad f_c^1 \leftarrow \text{PREIMG}(C^1) \setminus C^1 \times Act$
5 $\quad f^1 \leftarrow f_c^1 \setminus \text{PREIMG}_e(\overline{C^0})$
6 $\quad$ **while** $f^1 = \emptyset$
7 $\quad\quad f^0 \leftarrow \text{PREIMG}(C^0) \setminus C^0 \times Act$
8 $\quad\quad$ **if** $f^0 = \emptyset$ **then return** *failure*
9 $\quad\quad F^0 \leftarrow F^0 \cup f^0$
10 $\quad\quad C^0 \leftarrow C^0 \cup \text{STATES}(f^0)$
11 $\quad\quad f^1 \leftarrow f_c^1 \setminus \text{PREIMG}_e(\overline{C^0})$
12 $\quad F^1 \leftarrow F^1 \cup f^1$
13 $\quad C^1 \leftarrow C^1 \cup \text{STATES}(f^1)$
14 **return** $\langle F^1, F^0 \rangle$

Figure 3: The 1-FTP algorithm.

it takes a 1-fault tolerant planning problem as input (not its universal dual). The functions PREIMG and PREIMG_e compute preimages of the primary and secondary effects, respectively. 1-FTP returns a valid 1-fault tolerant plan represented by two plans F^1 and F^0. F^1 is robust to one fault while F^0 is a recovery plan. Let $\pi^i(s) = \{a \mid \langle s, a\rangle \in F^i\}$ for $i = 0, 1$. The corresponding fault tolerant plan is then

$$\pi(s,e) = \begin{cases} \pi^0(s) & : \quad e = 0 \\ \pi^1(s) & : \quad e = 1 \end{cases}$$

1-FTP performs a backward search from the goal states that alternate between blindly expanding F^1 and F^0 such that failure states of F^1 always can be recovered by F^0. Initially F^1 and F^0 are assigned to empty plans (l.1-2). The variables C^1 and C^0 are states covered by the current plans in F^1 and F^0. They are initialized to the goal states since these states are covered by zero length plans. In each iteration of the outer loop (l.3-13), F^1 is expanded with SAs in f^1 (l. 12-13). First, a candidate f_c^1 is computed. It is the preimage of the states in F^1 pruned for SAs of states already covered by F^1 (l.4). The variable f^1 is assigned to f_c^1 restricted to SAs for which all error states are covered by the current recovery plan (l.5). If f^1 is empty the recovery plan is expanded in the inner loop until f^1 is nonempty (l.6-11). If the recovery plan at some point has reached a fixed point and f^1 is still empty, the algorithm terminates with failure, since in this case no recovery plan exists (l.8).

1-FTP expands both F^0 and F^1 blindly. An inherent strategy of the algorithm, though, is not to expand F^0 more than necessary to recovery faults of F^1. This is not the case for n-FTP_S that for $n = 1$ at least will expand states with $e = 1$ as much as states with $e = 0$. The aggressive strategy, on the other hand, makes 1-FTP suboptimal as the example in Figure 4 shows. In the first two iterations of the outer loop, (p_2, b) and (p_1, b) are added to F^1 and nothing is added to F^0. In the third iteration of the outer loop, F^0 is extended with (p_2, b) and (q_2, a) and F^1 is extended with (q_2, a). In the last two iterations of the outer loop, (q_1, a) and (s_0, a) are added to F^1. From this, only a single 1-fault tolerant plan can be extracted with F^1 equal to the sequential plan aaa. However, this plan has a worst case execution length of 4 while the plan bbb with a proper recovery plan has a worst case length of 3.

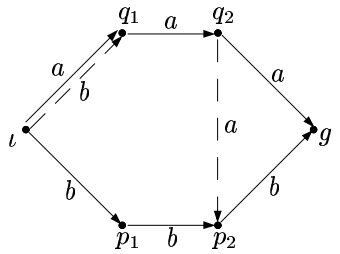

Figure 4: A problem with a single goal state g showing that 1-FTP may return suboptimal solutions. Dashed lines indicate secondary effects. Notice that action a and b only have secondary effects in q_2 and s_0, respectively. In all other states, the actions are assumed always to succeed.

Despite the different search strategies applied by 1-FTP and 1-FTP_S they both perform blind search. A more interesting algorithm is a guided version of 1-FTP called 1-GFTP. The over all design goal of 1-GFTP is to guide the expansion of F^1 toward the initial state and guide the expansion of F^0 toward the failure states of F^1. However, this can be accomplished in many different ways. Below we evaluate three different strategies. For each algorithm, F^1 is guided in a best-first manner toward the initial state using the approach employed by n-GFTP_S.

The first strategy is to assume that failure states are local and guide F^0 toward the initial state as well. The resulting algorithm is similar to 1-GFTP_S and has low performance. The problem is that the best-first approach causes F^0 only to cover a narrow beam of states in the search space. Any faults causing just a slight state change tend not to be covered by F^0. The strategy can be improved by widening the beam by taking the search depth into account. However, this does not provide a satisfactory solution for non-local states.

The second strategy is ideal in the sense that it dynamically guides the expansion of F^0 toward error states of the precomponents of F^1. This can be done by using a specialized OBDD operation that splits the precomponent of F^0 according to the Hamming distance to the error states. The theoretical complexity of the specialized algorithm for an OBDD E representing the error states and an OBDD P representing the precomponent of F^1 is $\exp(|E||F|)$. Due to the dynamic programming used by the OBDD package the average complexity may be exponentially lower. Unfortunately, this does not seem to be the case in practice.

The third strategy is chosen for 1-GFTP and employs an indirect guidance. It expands F^0 blindly but prunes SAs from the precomponent of F^0 not used to recover error states of F^1. We expect this strategy to work well even if the *absolute position* of error states is non-local. However, the strategy assumes that the *relative position* of error states is local in the sense that the SAs in F^0 in expansion i of F^1 are rel-

evant for recovering error states in expansion $i+1$ of F^1. In addition, we still have an essential decision problem to solve: to expand F^1 or F^0. There are two extremes: 1) compute a complete partitioned backward precomponent of F^1, expand F^0 until some SAs in the precomponent of F^1 has recovered error states, and add the SAs with recovered errors from the partition with least h-value to F^1, or 2) compute a complete partitioned backward precomponent of F^1, expand F^0 until some SAs in the partition with lowest h-value has recovered error states and add these SAs to F^1. It turns out that neither of these extremes work well in practice. Instead, we consider a strategy somewhere in between. The idea is to spent half of the last expansion time on recovering error states of the SAs in the partition with lowest h-value and, in case no such SAs exist, iteratively add SAs from partitions with higher h-value. The resulting algorithm is shown in Figure 5. By convention bold variables denote maps with

function 1-GFTP(s_0, G)
1 $F^1 \leftarrow \emptyset$; $\mathbf{C}^1[h_g] \leftarrow G$
2 $F^0 \leftarrow \emptyset$; $C^0 \leftarrow G$
3 $t \leftarrow 0$
4 **while** $s_0 \notin C^1$
5 $\quad t_s \leftarrow t_{CPU}$
6 $\quad \mathbf{PC} \leftarrow$ PRECOMP$(\mathbf{C}^1)$
7 $\quad f^1 \leftarrow \emptyset$; $f_c^1 \leftarrow \emptyset$
8 $\quad \mathbf{f}_c^0 \leftarrow emptyMap$
9 $\quad i \leftarrow 0$
10 $\quad$ **while** $f^1 = \emptyset \wedge i < |\mathbf{PC}|$
11 $\quad\quad i \leftarrow i+1$; $t \leftarrow t/2$
12 $\quad\quad f_c^1 \leftarrow f_c^1 \cup \mathbf{PC}[i]$
13 $\quad\quad \langle \mathbf{f}_c^0, f^1 \rangle \leftarrow$ EXPANDTIMED$(f_c^1, \mathbf{f}_c^0, C^0, t)$
14 $\quad$ **if** $f^1 = \emptyset$ **then**
15 $\quad\quad \langle \mathbf{f}_c^0, f^1 \rangle \leftarrow$ EXPANDTIMED$(f_c^1, \mathbf{f}_c^0, C^0, \infty)$
16 $\quad t \leftarrow t_{CPU} - T_s$
17 $\quad$ **if** $f^1 = \emptyset$ **then return** *failure*
18 $\quad f^0 \leftarrow$ PRUNEUNUSED$(\mathbf{f}_c^0, f^1)$
19 $\quad F^0 \leftarrow F^0 \cup f^0$; $C^0 \leftarrow C^0 \cup$ STATES(f^0)
20 $\quad F^1 \leftarrow F^1 \cup f^1$
21 $\quad$ **for** $j = 1$ **to** i
22 $\quad\quad \mathbf{C}^1[h_j] \leftarrow \mathbf{C}^1[h_j] \cup$ STATES$(f^1 \cap \mathbf{PC}[h_j])$
23 **return** $\langle F^1, F^0 \rangle$

Figure 5: The 1-GFTP algorithm.

either h-values or integers as keys and sets of states or SAs as values. The keys of maps are sorted ascendingly. For a map $\mathbf{M}$, $\mathbf{M}[k]$ is the value associated with key k, $|\mathbf{M}|$ is the current number of entries in $\mathbf{M}$ and M is the union of all the entries in $\mathbf{M}$. All variables of a function are local including its arguments. The instantiation of F^1 and F^0 of 1-GFTP is similar to 1-FTP except that the states in C^1 are partitioned with respect to their associated h-value. Initially, $\mathbf{C}^1[h_{goal}]$ is assigned to the goal states. [1] In each iteration of the main loop (l.4-22), the precomponents f^1 and f^0 are computed and added to F^1 and F^0. First, the start time t_s is logged by reading the current time t_{CPU} (l.5). Then a complete partitioned precomponent candidate $\mathbf{PC}$ of F^1 is computed by PRECOMP (l.6). For each entry in $\mathbf{C}^1$, PRECOMP adds the preimage for each branching partition (we assume there are m of these) to $\mathbf{PC}$.

function PRECOMP$(\mathbf{C}^1)$
1 $\mathbf{PC} \leftarrow emptyMap$
2 **for** $i = 1$ **to** $|\mathbf{C}^1|$
3 $\quad$ **for** $j = 1$ **to** m
4 $\quad\quad SA \leftarrow$ PREIMG$_j(\mathbf{C}^1[h_i])$
5 $\quad\quad SA \leftarrow SA \setminus C^1 \times Act$
6 $\quad\quad \mathbf{PC}[h_i + \delta h_j] \leftarrow \mathbf{PC}[h_i + \delta h_j] \cup SA$
7 **return** $\mathbf{PC}$

The inner loop (l.10-13) of 1-GFTP expands the two candidates f_c^1 and f_c^0 for f^1 and f^0. In each iteration, a new partition of $\mathbf{PC}$ is added to f_c^1 (l.12).[2] The function EXPANDTIMED then expands f_c^0. In iteration i, the time out bound of the expansion is $t/2^i$. EXPANDTIMED returns early if 1) a precomponent f^1 in the candidate f_c^1 is found where all error states are recovered (l.5 and l.12), or 2) f_c^0 has reached a fixed point. The preimages added to f_c^0 is kept in a map $\mathbf{f}_c^0$ in order to prune SAs not used for recovery.

function EXPANDTIMED$(f_c^1, \mathbf{f}_c^0, C^0, t)$
1 $t_s \leftarrow t_{CPU}$
2 $Oldf_c^0 \leftarrow \perp$
3 $i \leftarrow |\mathbf{f}_c^0|$
4 $recovS \leftarrow$ STATES$(f_c^0) \cup C^0$
5 $f^1 \leftarrow f_c^1 \setminus$ PREIMG$_e(\overline{recovS})$
6 **while** $f^1 = \emptyset \wedge Oldf_c^0 \neq f_c^0 \wedge t_{CPU} - t_s < t$
7 $\quad Oldf_c^0 \leftarrow f_c^0$
8 $\quad i \leftarrow i+1$
9 $\quad recovS \leftarrow$ STATES$(f_c^0) \cup C^0$
10 $\quad p \leftarrow$ PREIMG$(recovS)$
11 $\quad \mathbf{f}_c^0[i] \leftarrow p \setminus recovS \times Act$
12 $\quad f^1 \leftarrow f_c^1 \setminus$ PREIMG$_e(\overline{recovS})$
13 **return** $\langle \mathbf{f}_c^0, f^1 \rangle$

Eventually f_c^1 may be equal to $\mathbf{PC}$ but still not contain a recoverable precomponent f^1. In this case 1-GFTP expands f_c^0 (l.15) untimed. If f_c^0 has reached a fixed point but no recoverable precomponent f^1 exists, no 1-fault tolerant plan exists and 1-GFTP returns with failure (l. 17). Otherwise, f_c^0 is pruned for SAs of states not used to recover the SAs in f^1 (l.18). This pruning is computed by PRUNEUNUSED that traverses backward through the preimages of $\mathbf{f}_c^0$ and marks states that either are error states of SAs in f^1, or states needed to recover previously marked states.

function PRUNEUNUSED$(\mathbf{f}_c^0, f^1)$
1 $err \leftarrow$ SAIMG$_e(f^1)$
2 $img \leftarrow \emptyset$; $marked \leftarrow \emptyset$
3 **for** $i = |\mathbf{f}_c^0|$ **to** 1
4 $\quad \mathbf{f}_c^0[i] \leftarrow \mathbf{f}_c^0[i] \cap \big((err \cup img) \times Act\big)$
5 $\quad marked \leftarrow marked \cup$ STATES$(\mathbf{f}_c^0[i])$

[1] To simplify the presentation, we assume that all goal states have similar h-value. A generalization of the algorithm is trivial.

[2] Recall that $\mathbf{PC}$ is traversed ascendingly such that the partition with lowest h-value is added first.

6 $\quad img \leftarrow \text{SAIMG}(\mathbf{f}_c^0[i])$
7 **return** $f_c^0 \cap (marked \times Act)$

The function SAIMG computes next states of primary effects of a set of SAs

$$\text{SAIMG}(SA) = \Big(\exists \vec{v}, \vec{a}\,.\, SA(\vec{v}, \vec{a}) \wedge T(\vec{v}, \vec{a}, \vec{v}')\Big)[\vec{v}'/\vec{v}].$$

Similarly, SAIMG_e computes next states of secondary effects (error states) of a set of SAs. The updating of F^1 and F^0 of 1-GFTP (l.19-22) is similar to 1-FTP, except that $\mathbf{C}^1$ is updated by iterating over $\mathbf{PC}$ and picking SAs in f^1. Notice that in this iteration h_j refers to the keys of $\mathbf{PC}$.

Experimental Evaluation

The algorithms 1-FTP, 1-GFTP, 1-FTP$_S$, and 1-GFTP$_S$ have been implemented in a new OBDD-based search engine called OBS (Jensen 2003). OBS is implemented in C++/STL and uses the BuDDy OBDD-package (Lind-Nielsen 1999). All experiments are carried out on a Linux

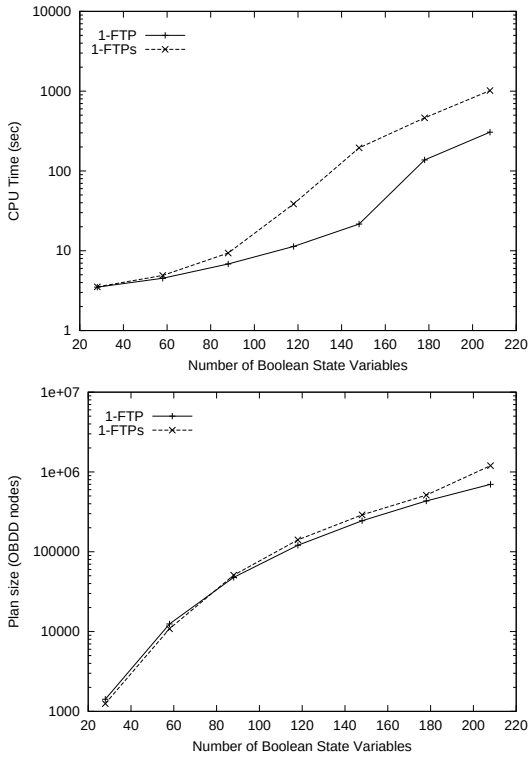

Figure 6: Results of the PSR problems.

RedHat 7.1 PC with kernel 2.4.16, 500 MHz Pentium III CPU, 512 KB L2 cache and 512 MB RAM. We represent the parameter setting of the BuDDy package by the tuple $\langle n, c, t\rangle$, where n is the number of allocated OBDD nodes in the unique table, c is the number of allocated OBDD nodes in the operator caches, and t is the total time in seconds spent by the package on memory allocation. We measure the size of plans and transition relations as the total number of OBDD nodes used by the OBDDs to represent them.

Unguided Search

DS1 is a description of the SMV code representing the Livingstone model used by the Remote Agent for the Deep Space 1 probe. It is a model of the electrical system of the spacecraft. An action is a bus command. Primary effects model the state change given that all units work correct. Secondary effects assumes that two of the four failures tested in the experiment happens. The domain has been written for 1-FTP. The encoding has 84 Boolean state variables. The BuDDy parameters of the experiment are $\langle 1M, 100K, 0.31\rangle$. A partitioned transition relation of size 104881 is computed in 0.42 seconds. The size of the solution is 535 and the total CPU time is 1.15 seconds. The experiment shows that OBDD-based fault tolerant planning is mature to be applied on signifi cant real-world problems. In addition, it demonstrates that even 1-fault tolerance impose strong restrictions on a physical system. No 1-fault tolerant plan exists for the problem if all of the original four failures are considered.

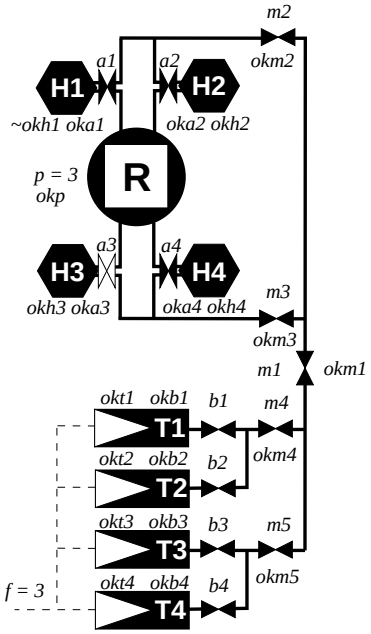

Figure 7: The power plant domain. An open valve is drawn solid and allows water or steam to flow through it. In the depicted state, a failure of heat exchanger 1 is a assumed just to have happened.

PSR is another real-world domain for power supply restoration. It is a power grid with feeders and switches. The actions consist of opening and closing switches. The secondary effect is that they break and get stuck in their current position. We compare the performance of 1-FTP and 1-FTP$_S$ in two versions of the domain. The fi rst is the *simple* domain described in (Bertoli *et al.* 2002). With the OBDD package initialization $\langle 1M, 700K, 0.98\rangle$, 1-FTP and 1-FTP$_S$ solve this problem in 6.8 and 11.25 seconds, respectively. The second version is parameterized and is an $n \times 2$ matrix of switches and feeders connected by lines.

	1-FTP		1-FTP$_S$	
$\#vars$	T_{total}	$\|sol\|$	T_{total}	$\|sol\|$
40	6.1	65K	8.7	62K
80	157.8	1.2M	189.4	1.5M

Figure 8: Results of the power plant experiment. The total CPU time and plan size is given by T_{total} and $|sol|$, respectively. The number of Boolean state variables is given by $\#vars$.

For the result shown in Figure 6, the OBDD package parameters are $\langle 15M, 500K, 3.38\rangle$. 1-FTP performs significantly better than 1-FTP$_S$ for this problem. Interestingly, the performance difference is not reflected in the plan sizes. However, this may be an artifact caused by the fact that the plan size for 1-FTP is a sum of the size of two OBDDs, while the plan size for 1-FTP$_S$ is the size of a single OBDD.

The powerPlant domain is shown in Figure 7. It is a simple model of a nuclear power plant consisting of turbines, heat exchangers, and valves. All units may break permanently. The task is to execute the control actions in order to satisfy the safety and progress requirements of the plant. A 1-fault tolerant plan exists but only for simple mal-functions. We compare the performance of 1-FTP and 1-FTP$_S$ in two versions of the domain. The first considers controlling a single power plant, while the second considers controlling two power plants simultaneously. The results are shown in Figure 8. In both experiments, the parameters of the OBDD package are $\langle 15M, 500K, 3.4\rangle$. 1-FTP has a slightly better performance than 1-FTP$_S$. However, both algorithms suffer from a large growth rate of the OBDDs representing the frontier of the backward search.

The beamWalk domain considers a robot walking on a beam. However, it can fall down in each step on the beam. It represents a worst case scenario for 1-FTP and 1-FTP$_S$ since a fault in the last step to reach the goal causes a transition to the state furthest away from the goal. Both algorithms must iterate over all states before a solution is found. The results are shown in Figure 9. As expected, both algorithms

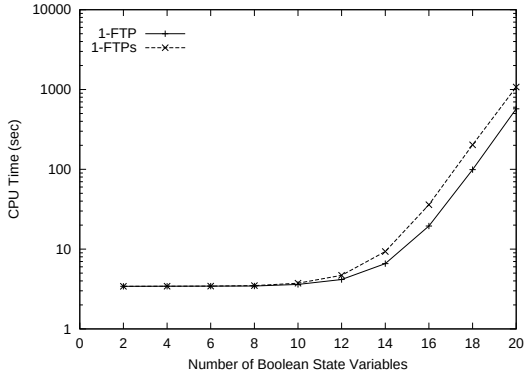

Figure 9: Results of the BeamWalk experiments.

have a limited performance in this domain. Again, however, we observe a slightly better performance of 1-FTP.

Guided Search

The LV domain has been designed to demonstrate the difference between 1-GFTP and 1-GFTP$_S$. It is an $n \times n$ grid world with initial state $(0, n-1)$ and goal state $(n/2, n/2)$. The actions are up, down, left, and right. Above the $y = x$ line actions may fail causing x and y to be swapped. Thus, error states are mirrored in the $y = x$ line. An 9×9 problem of the domain is shown in Figure 10. The essential

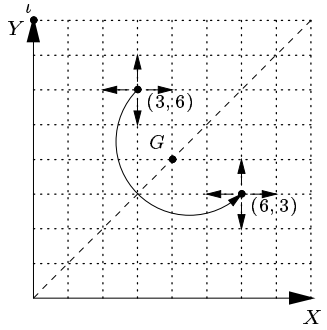

Figure 10: The LV domain.

property is that the relative site of error states is local while their absolute site is not. This is the assumption made by 1-GFTP, but not 1-GFTP$_S$ that requires error states to have local absolute site. The heuristic value of a state is the Manhattan distance to the initial state. The BuDDy parameters are $\langle 5M, 500K, 1.4\rangle$. The results are shown in Figure 11. As depicted, the performance of 1-GFTP$_S$ degrades very

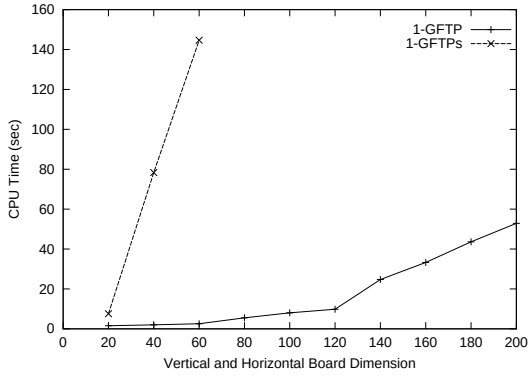

Figure 11: Results of the LV experiments.

fast with n, due to the misguidance of the heuristic. Its total CPU time is more than 500 seconds after the first three experiments. 1-GFTP$_S$ is fairly unaffected by the error states. To explain this consider how the backward search proceeds from the goal state. The precomponents of F^1 will cause this plan to beam out toward the initial state. Due to the relative locality of error states, the pruning of F^0 will cause F^0 to beam out in the opposite direction. Thus both F^1 and F^0 remains small during the search.

The 8-puzzle further demonstrates this difference between 1-GFTP and 1-GFTP$_S$. We consider a non-

deterministic version of the 8-puzzle where the secondary effects are self loops. Thus, error states are the most local possible. We use the usual sum of Manhattan distances of tiles as an heuristic for the distance to the initial state. The experiment compares the performance of 1-FTP, 1-GFTP, 1-FTP$_S$, and 1-GFTP$_S$. The BuDDy parameters are $\langle 1M, 100K, 0.29\rangle$. The number of Boolean state variables is 35 in all experiments. The results are shown in Figure 12. Again, 1-FTP performs substantially better than 1-FTP$_S$.

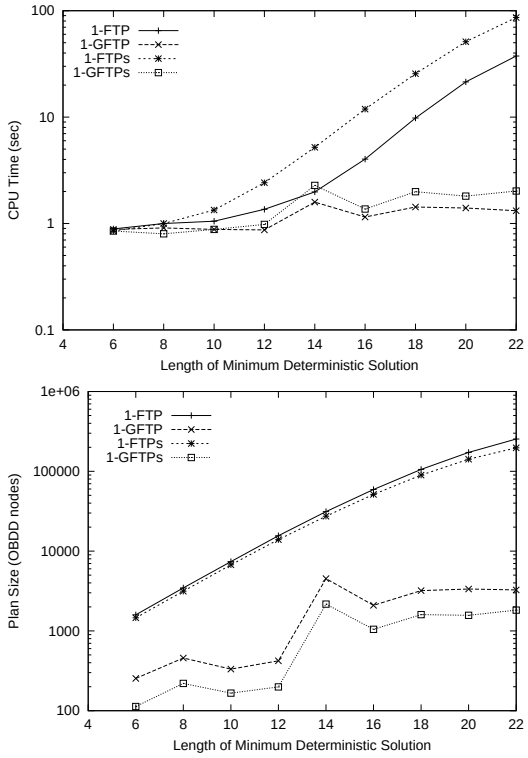

Figure 12: Results of the 8-puzzle experiments.

The guided algorithms 1-GFTP and 1-GFTP$_S$ have much better performance than the unguided algorithms. Due to local error states, however, there is no substantial performance difference between these two algorithms.

In our final domain SIDMAR, we study the robustness of 1-GFTP and 1-GFTP$_S$ to the kind of non-local errors found in real-world domains. The domain is shown in Figure 13 and is an abstract model of a steel producing plant of SIDMAR in Ghent, Belgium. The primary effects of actions are to move, lift and perform treatments of ladles on machines. The secondary effects are that machines break permanently and moves fail. We consider casting two ladles of steel. The heuristic is the sum of machine treatments carried out on the ladles. The experiment compares the performance of 1-FTP, 1-GFTP, 1-FTP$_S$, and 1-GFTP$_S$. The BuDDy parameters are $\langle 5M, 500K, 1.41\rangle$. The number of Boolean state variables is 47 in all experiments. The results are shown in Figure 14. Missing data points indicates that the associated algorithm spent more than 500 seconds trying

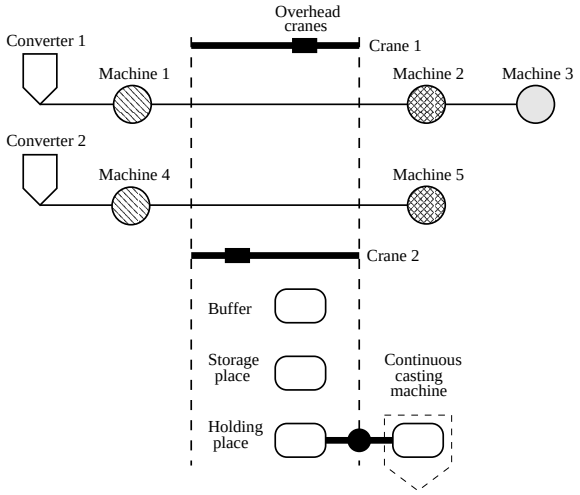

Figure 13: Layout of the SIDMAR steel plant.

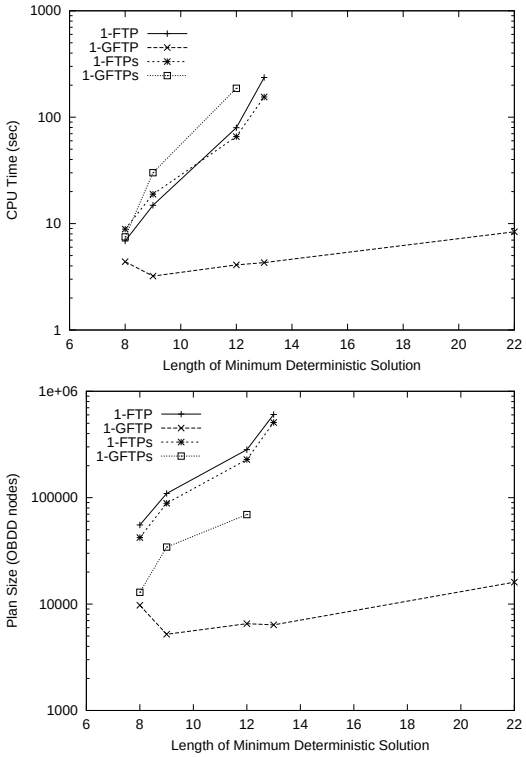

Figure 14: Results of the SIDMAR experiments.

to solve the problem. The only algorithm with good performance is 1-GFTP. Thus, in practical applications 1-GFTP$_S$ can be highly sensitive to non-local error states. Also notice that this is the only domain where 1-FTP does not outperform 1-FTP$_S$.

Conclusions and Future Work

In this paper, we have introduced a new planning framework called fault tolerant planning. An n-fault tolerant plan can handle up to n faults during execution caused by rare secondary effects of actions. Strong and strong cyclic solutions seldom exists for such problems and weak solutions are mostly useless in practice. By adding a fault counter to the domain, we show how optimal n-fault tolerant planning can be reduced to strong universal planning. A major disadvantage of the strong algorithm is that it performs a blind search. Both in artificial and real-world domains this approach often leads to a blow-up of the OBDDs representing the search frontier. A guided version of the strong algorithm has recently been developed. However, for fault tolerant planning the recovery part of the plan should be dynamically guided toward error states while the non-recovery part should be statically guided toward the initial state. In two specialized algorithms for 1-fault tolerant planning 1-FTP and 1-GFTP, we decouple the synthesis of the recovery and non-recovery part of the plan. 1-FTP uses a blind search strategy, while 1-GFTP is guided. Our experimental results show that 1-GFTP consistently outperforms its strong algorithm counter part 1-GFTP$_S$ and in particular is robust to non-local error states. Our investigation of real-world domains suggests that such error states are frequent and caused by permanent failures. Despite the blind search of 1-FTP, it often outperforms its strong algorithm counter part 1-FTP$_S$ since it may avoid producing large recovery plans.

In the near future, we plan to formally prove completeness of 1-FTP and 1-GFTP. More long term goals includes a generalization of fault tolerant domains to discrete event systems with exogenous and endogenous events and a development of tools to analyse fault tolerant systems.

References

Bertoli, P.; Cimatti, A.; Slanley, J.; and Thiébaux, S. 2002. Solving power supply restoration problems with planning via symbolic model checking. In *Proceedings of the 15th European Conference on Artificial Intelligence ECAI'02*.

Bertoli, P.; Cimatti, A.; and Roveri, M. 2001. Conditional planning under partial observability as heuristic-symbolic search in belief space. In *Pre-Proceedings of the 6th European Conference on Planning (ECP-01)*, 379–384.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 8:677–691.

Chen, J., and Patton, R. J. 1999. *Robust Model-Based Fault Diagnosis for Dynamic Systems*. Kluwer Academic Publishers.

Cimatti, A., and Roveri, M. 2000. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research* 13:305–338.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Strong planning in non-deterministic domains via model checking. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning System (AIPS'98)*, 36–43. AAAI Press.

Clarke, E.; Grumberg, O.; and Peled, D. 1999. *Model Checking*. MIT Press.

Jensen, R. M., and Veloso, M. M. 2000. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research* 13:189–226.

Jensen, R. M.; Veloso, M. M.; and Bowling, M. 2001. Optimistic and strong cyclic adversarial planning. In *Preproceedings of the 6th European Conference on Planning (ECP'01)*, 265–276.

Jensen, R. M.; Veloso, M. M.; and Bryant, R. E. 2003. Guided symbolic universal planning. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling ICAPS-03*. To Appear.

Jensen, R. M. 2003. The OBS software package version 0.7. `http://www.cs.cmu.edu/~runej`.

Klein, E., and Wehlan, H. 1996. Systematic design of a protective controller in process industries by means of the boolean differential calculus. In *Proceedings of WODES-96*.

Koenig, S., and Simmons, R. G. 1995. Real-time search in non-deterministic domains. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1660–1667. Morgan Kaufmann.

Lind-Nielsen, J. 1999. BuDDy - A Binary Decision Diagram Package. Technical Report IT-TR: 1999-028, Institute of Information Technology, Technical University of Denmark. `http://cs.it.dtu.dk/buddy`.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publ.

Olawsci, D., and Gini, M. 1990. Deferred planning and sensor use. In *Proceedings, DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*. Morgan Kaufmann.

Pistore, M.; Bettin, R.; and Traverso, P. 2001. Symbolic techniques for planning with extended goals in non-deterministic domains. In *Pre-Proceedings of the 6th European Conference on Planning (ECP-01)*, 253–264.

Schoppers, M. J. 1987. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-87)*, 1039–1046. Morgan Kaufmann.