Journal of Artificial Intelligence Research 1 (200?) ? Submitted ?; published ?

Fault Tolerant Planning: Moving Toward Probabilistic Uncertainty Models in Symbolic Non-Deterministic Planning

Rune M. Jensen RUNEJ@CS.CMU.EDU
Manuela M. Veloso MMV@CS.CMU.EDU
Randal E. Bryant BRYANT@CS.CMU.EDU
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA

Abstract

Symbolic non-deterministic planning represents action effects as sets of possible next states. In this paper, we move toward a more probabilistic uncertainty model by distinguishing between likely primary effects and unlikely secondary effects of actions. We consider the practically important case, where secondary effects are failures, and introduce n-fault tolerant plans that are robust for up to n faults occurring during plan execution. Fault tolerant plans are more restrictive than weak plans, but more relaxed than strong cyclic and strong plans. We show that optimal n-fault tolerant plans can be generated by the usual strong algorithm. However, due to non-local error states, it is often beneficial to decouple the planning for primary and secondary effects. We employ this approach for two specialized algorithms 1-FTP (blind) and 1-GFTP (guided) and demonstrate their advantages experimentally in significant real-world domains.

1. Introduction

MDP solving (e.g., Sutton & Barto 1998) and Symbolic Non-Deterministic Planning (SNDP) (e.g., Cimatti *et al.* 2003) can be regarded as two alternative frameworks for solving planning problems with uncertain outcomes of actions. Both frameworks are attractive, but for quite different reasons. The main advantage of MDP solving is the high expressive power of the domain model: for each state in the MDP, the effect of an action is given by a probability distribution over next states. The framework, however, is challenged by a high complexity of solving MDPs. The main advantage of SNDP is its scalability. The domain model has less expressive power than an MDP. Action effects are modeled as sets of possible next states instead of probability distributions over these states. This allows powerful symbolic search methods based on Binary Decision Diagrams (BDDs, Bryant 1986) to be applied. SNDP, however, is challenged by its coarse uncertainty model of action effects. The current solution classes are suitable when a pure disjunctive model of action effects is sufficient (e.g., for controlling worst-case behavior). However, when this is not the case, they often become too relaxed (weak plans) or too restrictive (strong cyclic and strong plans).

A large body of work in MDP solving addresses the scalability problem (e.g., Sutton & Barto 1998; Tesauro 1995). In particular, symbolic methods based on Algebraic Decision Diagrams (ADDs, Bahar *et al.* 1993) have been successfully applied to avoid explicitly enumerating states (Hoey, St-Aubin, & Hu, 1999; Feng & Hansen, 2002).

A dual effort in SNDP, where the uncertainty model of action effects is brought closer to its probabilistic nature, is still lacking. In this paper, we take a first step in this direction

©200? AI Access Foundation and Morgan Kaufmann Publishers. All rights reserved.

by introducing a new class of fault tolerant non-deterministic plans. Our work is motivated by two observations:

1. Non-determinism in real-world domains is often caused by infrequent errors that make otherwise deterministic actions fail.
2. Normally, no actions are guaranteed to succeed.

Due to the first observation, we propose a new uncertainty model of action effects in SNDP that distinguishes between primary and secondary effects of actions. The primary effect models the usual deterministic behavior of the action, while the secondary effect models error effects. Due to the second observation, we introduce *n-fault tolerant plans* that are robust for up to n errors or faults occurring during plan execution. This definition of fault tolerance is closely connected to fault tolerance concepts in control theory and engineering. Every time we board a two engined aircraft, we enter a 1-fault tolerant system: a single engine failure is recoverable, but two engines failing may lead to an unrecoverable breakdown of the system.

An n-fault tolerant plan is not as restrictive as a strong plan that requires that the goal can be reached 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 may still be very likely to succeed. A fault tolerant plan is also not as restrictive as a strong cyclic plan. An execution of a strong cyclic plan will never reach states not covered by the plan unless, it is a goal state. Thus, strong cyclic plans also have to take all error combinations into account. Weak plans, on the other hand, are more relaxed than fault tolerant plans. Fault tolerant plans, however, are almost always preferable to weak plans because they give no guarantees for *all* the possible outcomes of actions. For fault tolerant plans, any action may fail, but only a limited number of failures are recoverable.

One might suggest using a deterministic planning algorithm to generate n-fault tolerant plans. 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 deterministic 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 with the BDD-based approach of SNDP.

The paper contributes a range of unguided as well as guided algorithms for generating fault tolerant plans. We first observe that an n-fault tolerant planning problem can be reduced to a strong planning problem and solved with the strong planning algorithm. The resulting algorithm is called n-FTP$_S$. Since the performance of blind strong planning is limited, we also consider a guided version of n-FTP$_S$ called n-GFTP$_S$ using the approach introduced in Jensen, Veloso & Bryant (2003). The n-GFTP$_S$ algorithm is efficient, when secondary effects are local in the state space, because they then will be covered by the search beam of n-GFTP$_S$. In practice, however, secondary effects may be permanent malfunctions that due to their impact on the domain cause a transition to a non-local state. To solve this problem, we decouple the planning for primary and secondary effects. We restrict our investigation to 1-fault tolerant planning and introduce two algorithms: 1-FTP and 1-GFTP using blind and guided search, respectively. The algorithms have been implemented in the BIFROST search engine (Jensen, 2003b) and experimentally evaluated on a range of domains including three real-world domains: DS1 (Pecheur & Simmons, 2000), PRS (Thiébaux & Cordie, 2001), and SIDMAR (Fehnker, 1999). The experiments illustrate the natural connection between the existence of fault tolerant plans and the redundancy characteristics of the modeled system. Moreover, they show that even 1-fault tolerant plans impose much stronger requirements on the system than weak plans. Finally, they indicate that faults in real-world domains often cause non-local transitions that require specialized planning algorithms to be handled efficiently.

Previous work explicitly representing and reasoning about action failure is very limited. Some reactive planning approaches take action failure into account (e.g. Georgeff & Lansky 1986; Williams *et al.* 2003), but do not involve producing a fault tolerant plan. The `MRG` planning language (Giunchiglia, Spalazzi, & Traverso, 1994) explicitly models failure effects. However, this work does not include planning algorithms for generating fault tolerant plans. To our knowledge, the n-fault tolerant planning algorithms introduced in this paper are the first automated planning algorithms for generating fault tolerant plans given a description of the domain that explicitly represents failure effects of actions.

In the following section, we present necessary SNDP terminology and results. We then define n-fault tolerant plans and describe the developed fault tolerant planning algorithms. Finally, we present our experimental evaluation and draw conclusions.

2. Related Work

Disjunctive action effects in conditional (e.g., Peot & Smith, 1992; Weld, Anderson, & Smith, 1998) and symbolic non-deterministic planning (e.g., Cimatti et al., 2003) are often caused by action failures. This is also the case for the large body of work in AI on fault diagnosis (e.g., (Kleer & Williams, 1987; Hammond, 1990; Senjen & De Beler, 1993; Doyle, 1995)). However, work explicitly representing and reasoning about success and failure effects of actions is very limited. The Elmer system (McCalla & Ward, 1982) uses error transitions from abstract actions to detect and recover from failures. In the Procedural Reasoning System (PRS) (Georgeff & Lansky, 1986), the procedure descriptions defines the effect of successful and unsuccessful execution of a procedure. Similarly, the Reactive Model Based Programming Language (RMPL) (Williams et al., 2003) and its underlying executor Titan can handle faults at runtime. The approach, however, does not involve computing a fault tolerant plan. The `MRG` (Giunchiglia et al., 1994) planning language explicitly models failure effects. However, this work does not include planning algorithms for generating fault tolerant plans. To our knowledge, the n-fault tolerant planning algorithms introduced in this thesis are the first automated planning algorithms for generating fault tolerant plans given a description of the domain that explicitly represents failure effects of actions.

Similarly to AI, there has been substantial amount of work on fault diagnosis in DES control theory. This work has mainly focused on analysing event sequences in order to determine if a fault has happened, and if so, which kind of fault (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995, 1996; Sampath, Sengupta, Lafortune, &

Teneketzis, 1998; Su, 2001). However, there has also been a considerable amount of work on fault models. These models can be characterized as either *transition based* or *state based*. Most work (e.g., (Chen & Patton, 1999; Cho & Lim, 1998; Cin, 1997)) use the transition based model and regard *faults* as unexpected changes in a system that tends to degrade the overall system performance rather than causing a total breakdown. The term *failure* suggests a complete breakdown of a system component or function. The transition based model is also used in supervisory control (Ramadge & Wonham, 1987) where faults usually are considered uncontrollable events (Balemi, Hoffmann, Gyugyi, Wong-Toi, & Franklin, 1993; Cho & Lim, 1998). Within this frame, an approach to fault tolerant control has been considered that is closely related to n-fault tolerant planning. The work in (Perraju, Rana, & Sarkar, 1997), specifies fault tolerance for mission critical systems. A *masking fault tolerant system* can recover from any fault. A *t-fault tolerant system* can recover from up to t faults occurring during its life time. The system is modeled by an automaton with start states, but no goal states. In addition, no algorithms or theory for controller synthesis are provided.

The state based models usually divides the state space into ranges of operation of some system (e.g., "normal operation range", "admissible error range", and "non-admissible error range" (Klein & Wehlan, 1996), or "good" and "bad" states (Özveren & Willsky, 1991)). In Özveren's work (Özveren & Willsky, 1991), *Stability* is defined to be to visit the good states infinitely often. Thus, a controller is *stable* if it from any reachable bad state can force a trajectory that in a finite number of steps reaches the good states. *Stabilizability* is defined to choosing state feedback such that the closed loop system is stable. A related approach (Passino, 1994) defines *Lyapunov stability* of a class of DES. Consider a set of states X_m that are invariant in the plant. That is, any execution starting in any state in X_m stays within X_m. X_m is stable in the sense of Lyapunov if for any $\epsilon > 0$ a max distance $\delta > 0$ (given by some metric) can be found such that any execution starting at a state within δ from X_m ends up in a state less than ϵ from X_m.

We are not aware of work in any other field than AI and DES control theory that reason explicitly about failures in order to automatically synthesize fault tolerant plans or fault tolerant discrete controllers.

3. Symbolic Non-deterministic Planning

A *non-deterministic planning domain* is a tuple $\langle S, Act, \rightarrow \rangle$, where S is a finite set of states, Act is a finite set of actions, and $\rightarrow \subseteq S \times Act \times S$ is a non-deterministic transition relation of action effects. Instead of $(s, a, s') \in \rightarrow$, we write $s \xrightarrow{a} s'$. The set of next states of an action a applied in state s is given by $\textsc{Next}(s, a) \equiv \{s' : s \xrightarrow{a} s'\}$. An action a is called *applicable* in state s iff $\textsc{Next}(s, a) \neq \emptyset$. The set of applicable actions in a state s is given by $\textsc{App}(s) \equiv \{a : \textsc{Next}(s, a) \neq \emptyset\}$.

A *non-deterministic planning problem* is a tuple $\langle \mathcal{D}, s_0, G \rangle$, where $\mathcal{D}$ is a non-deterministic planning domain, $s_0 \in S$ is an initial state, and $G \subseteq S$ is a set of goal states. Let $\mathcal{D}$ be a non-deterministic planning domain. A *state-action pair* $\langle s, a \rangle$ of $\mathcal{D}$ is a state $s \in S$ associated with an applicable action $a \in \textsc{App}(s)$. A *non-deterministic plan* is a set of state-action pairs (SAs) defining a function from states to sets of actions relevant to apply in order to reach a goal state. States are assumed to be fully observable. An execution of a non-deterministic plan is an alternation between observing the current state and choosing an action to apply from the set of actions associated with the state. The set of states *covered* by a plan π is given by $\textsc{States}(\pi) \equiv \{s : \exists a \,.\, \langle s, a\rangle \in \pi\}$. The set of possible end states of a plan is given by $\textsc{Closure}(\pi) \equiv \{s' \notin \textsc{States}(\pi) : \exists \langle s, a\rangle \in \pi \,.\, s' \in \textsc{Next}(s, a)\}$.

Following Cimatti *et al.* (2003), we use CTL to define weak, strong cyclic, and strong plans. CTL specifies the behavior of a system represented by a *Kripke structure*. A Kripke structure is a pair $\mathcal{K} = \langle S, R\rangle$, where S is a finite set of states and $R \subseteq S \times S$ is a total transition relation. An *execution tree* is formed by designating a state in the Kripke structure as an initial state and then unwinding the structure into an infinite tree with the designated state as root.

We consider a subset of CTL formulas with two *path quantifiers* $\texttt{A}$ ("for all execution paths") and $\texttt{E}$ ("for some execution path") and one *temporal operator* $\texttt{U}$ ("until") to describe properties of a path through the tree. Given a finite set of states S, the syntax of CTL formulas are inductively defined as follows

- Each element of 2^S is a formula,
- $\neg\psi$, $\texttt{E}(\phi\,\texttt{U}\,\psi)$, and $\texttt{A}(\phi\,\texttt{U}\,\psi)$ are formulas if ϕ and ψ are.

In the following inductive definition of the semantics of CTL, $\mathcal{K}, q \models \psi$ denotes that ψ holds on the execution tree of the Kripke structure $\mathcal{K} = \langle S, R\rangle$ rooted in the state q

- $\mathcal{K}, q_0 \models P$ iff $q_0 \in P$,
- $\mathcal{K}, q_0 \models \neg\psi$ iff $\mathcal{K}, q_0 \not\models \psi$,
- $\mathcal{K}, q_0 \models \texttt{E}(\phi\,\texttt{U}\,\psi)$ iff there exists a path $q_0 q_1 \cdots$ and $i \geq 0$ such that $\mathcal{K}, q_i \models \psi$ and, for all $0 \leq j < i$, $\mathcal{K}, q_j \models \phi$,
- $\mathcal{K}, q_0 \models \texttt{A}(\phi\,\texttt{U}\,\psi)$ iff for all paths $q_0 q_1 \cdots$ there exists $i \geq 0$ such that $\mathcal{K}, q_i \models \psi$ and, for all $0 \leq j < i$, $\mathcal{K}, q_j \models \phi$.

We will use three abbreviations $\texttt{AF}\,\psi \equiv \texttt{A}(S\,\texttt{U}\,\psi)$, $\texttt{EF}\,\psi \equiv \texttt{E}(S\,\texttt{U}\,\psi)$, $\texttt{AG}\,\psi \equiv \neg\texttt{EF}\neg\psi$. Since S is the complete set of states in the Kripke structure, the CTL formula S holds in any state. Thus, $\texttt{AF}\,\psi$ means that for all execution paths a state, where ψ holds, will eventually be reached. Similarly, $\texttt{EF}\,\psi$ means that there exists an execution path reaching a state, where ψ holds. Finally, $\texttt{AG}\,\psi$ holds, if every state on any execution path satisfies ψ.

The *execution model* of a plan π for the problem $\langle \mathcal{D}, s_0, G\rangle$ of the domain $\mathcal{D} = \langle S, \mathit{Act}, \rightarrow\rangle$ is a Kripke structure $\mathcal{M}(\pi) = \langle S, R\rangle$, where

- $S = \textsc{Closure}(\pi) \cup \textsc{States}(\pi) \cup G$,
- $\langle s, s'\rangle \in R$ iff $s \notin G$, $\exists a \,.\, \langle s, a\rangle \in \pi$ and $s \xrightarrow{a} s'$, or $s = s'$ and $s \in \textsc{Closure}(\pi) \cup G$.

Notice that all execution paths are infinite which is required in order to define solutions in CTL. If a state is reached that is not covered by the plan (e.g., a goal state or a dead end), the postfix of the execution path from this state is an infinite repetition of it. Given a problem $\mathcal{P} = \langle \mathcal{D}, s_0, G\rangle$ and a plan π for $\mathcal{D}$ we then have

- π is a weak plan iff $\mathcal{M}(\pi), s_0 \models \texttt{EF}\,G$,

- π is a strong cyclic plan iff $\mathcal{M}(\pi), s_0 \models \texttt{AGEF}\, G$,
- π is a strong plan iff $\mathcal{M}(\pi), s_0 \models \texttt{AF}\, G$.

Weak, strong cyclic, and strong plans can be synthesized by the NDP algorithm shown below. The algorithm performs a backward breadth-first search from the goal states to the initial state. The set operations can be efficiently implemented using BDDs. For a detailed description of this approach, we refer the reader to Jensen (2003a). In each iteration (l.2-7), NDP computes a precomponent P_c of the plan from the states C currently covered by the plan. In a guided version of the algorithm (Jensen et al., 2003), the set of state-action pairs (SAs) of the precomponent is partitioned according to a heuristic measure (e.g. an estimate of the distance to the initial state).

```
function NDP($s_0, G$)
1 $P \leftarrow \emptyset$; $C \leftarrow G$
2 while $s_0 \notin C$
3    $P_c \leftarrow$ PreComp($C$)
4    if $P_c = \emptyset$ then return "no plan exists"
5    else
6        $P \leftarrow P \cup P_c$
7        $C \leftarrow C \cup$ States($P_c$)
8 return $P$
```

The strong, strong cyclic, and weak planning algorithms only differ by the definition of the precomponent. Let $\textsc{PreImg}(C)$ denote the set of SAs, where the action applied in the state may lead into the set of states C. That is $\textsc{PreImg}(C) \equiv \{\langle s, a\rangle \,:\, \textsc{Next}(s,a) \cap C \neq \emptyset\}$. The weak and strong precomponent are then defined by

$$\begin{aligned} PC_w(C) &\equiv \textsc{PreImg}(C) \setminus C \times Act, \\ PC_s(C) &\equiv (\textsc{PreImg}(C) \setminus \textsc{PreImg}(\overline{C})) \setminus C \times Act. \end{aligned}$$

The strong cyclic precomponent depends a fixed point computation. We refer the reader to Jensen (2003a) for details.

Due to the breadth-first search carried out by NDP, weak solutions have minimum length best-case execution paths and strong solutions have minimum length worst-case execution paths (Cimatti et al., 2003). Formally, for a non-deterministic planning domain $\mathcal{D}$ and a plan π of $\mathcal{D}$ let $\textsc{Exec}(s,\pi) \equiv \{q \,:\, q \text{ is a path of } \mathcal{M}(\pi) \text{ and } q_0 = s\}$ denote the set of execution paths of π starting at s. Let the length of a path $q = q_0 q_1 \cdots$ with respect to a set of states C be defined by

$$|q|_C \equiv \begin{cases} i & : \text{ if } q_i \in C \text{ and } q_j \notin C \text{ for } 0 \leq j < i \\ \infty & : \text{ otherwise.} \end{cases}$$

Let $\textsc{Min}(s, C, \pi)$ and $\textsc{Max}(s, C, \pi)$ denote the minimum and maximum length of an execution path from s to C of a plan π

$$\begin{aligned} \textsc{Min}(s, C, \pi) &\equiv \min_{q \in \textsc{Exec}(s,\pi)} |q|_C, \\ \textsc{Max}(s, C, \pi) &\equiv \max_{q \in \textsc{Exec}(s,\pi)} |q|_C. \end{aligned}$$

Similarly, let Π denote the set of all plans of $\mathcal{D}$ and let $\text{WDist}(s, C)$ (weak distance) and $\text{SDist}(s, C)$ (strong distance) denote the minimum of $\text{Min}(s, C, \pi)$ and $\text{Max}(s, C, \pi)$ for any plan $\pi \in \Pi$ of $\mathcal{D}$

$$\begin{aligned} \text{WDist}(s, C) &\equiv \min_{\pi \in \Pi} \text{Min}(s, C, \pi), \\ \text{SDist}(s, C) &\equiv \min_{\pi \in \Pi} \text{Max}(s, C, \pi). \end{aligned}$$

Let Weak and Strong denote the NDP algorithm, where $\text{PreComp}(C)$ is substituted with $PC_w(C)$ and $PC_s(C)$. For a weak plan $\pi_w = \text{Weak}(s_0, G)$ and strong plan $\pi_s = \text{Strong}(s_0, G)$, we then have

$$\begin{aligned} \text{Min}(s_0, G, \pi_w) &= \text{WDist}(s_0, G), \\ \text{Max}(s_0, G, \pi_s) &= \text{SDist}(s_0, G). \end{aligned}$$

4. N-Fault Tolerant Planning Problems

A fault tolerant planning domain is a non-deterministic planning domain, where actions have primary and secondary effects. The primary effect is deterministic. However, 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.*

The planning language NADL$^+$ described in Appendix 7 may be used to represent fault tolerant planning domains. An n-fault tolerant planning problem is a deterministic planning problem extended with the fault limit n.

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 $n : \mathbf{N}$ is an upper bound on the number of faults the plan must be able to recover from.*

An n-fault tolerant plan is defined via a transformation of an n-fault tolerant planning problem to a non-deterministic planning problem. The transformation adds a fault counter f to the state description and models secondary effects only when $f \leq n$.

Definition 3 (Induced Non-Deterministic Planning Problem) *Let $\mathcal{P} = \langle \mathcal{D}, s_0, G, n \rangle$ where $\mathcal{D} = \langle S, Act, \rightarrow, \rightsquigarrow \rangle$ be an n-fault tolerant planning problem. The non-deterministic planning problem induced from $\mathcal{P}$ is $\mathcal{P}^{nd} = \langle \mathcal{D}^{nd}, \langle s_0, 0 \rangle, G \times \{0, \cdots, n\} \rangle$ where $\mathcal{D}^{nd} = \langle S^{nd}, Act^{nd}, \rightarrow^{nd} \rangle$ and is given by*

- $S^{nd} = S \times \{0, \cdots, n\}$,
- $Act^{nd} = Act$,

- $\langle s, f\rangle \xrightarrow{a}^{nd} \langle s', f'\rangle$ *iff*
  - $s \xrightarrow{a} s'$ *and* $f' = f$*, or*
  - $s \overset{a}{\rightsquigarrow} s'$*,* $f < n$*, and* $f' = f + 1$*.*

Definition 4 (Valid N-Fault Tolerant Plan) *A valid* n*-fault tolerant plan for the* n*-fault tolerant planing problem* $\mathcal{P}$ *is a non-deterministic plan* π *for the non-deterministic planning problem induced from* $\mathcal{P}$ *where* $\mathcal{M}(\pi), s_0^{nd} \models \mathtt{AF}\, G^{nd}$*.*

Thus, an n-fault tolerant plan is valid if any execution path where at most n failures happen eventually reaches a goal state. An n-fault tolerant plan is optimal if it has minimum worst case execution length.

Definition 5 (Optimal N-Fault Tolerant Plan) *An optimal* n*-fault tolerant plan is a va-lid* n*-fault tolerant plan* π *where* $\textsc{Max}(s_0^{nd}, G^{nd}, \pi) = \textsc{SDist}(s_0^{nd}, G^{nd})$

5. N-Fault Tolerant Planning Algorithms

It follows directly from Definition **??** that the strong algorithm returns a valid n-fault tolerant plan, if it exists, when given the induced non-deterministic planning problem as input. Moreover, if the blind strong algorithm is used to generate the solution, it follows from Theorem **??** that the returned n-fault tolerant plan is optimal. Let n-FTP$_S$ denote the Strong algorithm applied to an n-fault tolerant planning problem. Since the performance of blind strong planning is limited, we also consider solving n-fault tolerant planning problems with the guided version of strong planning defined in previous chapter. Let n-GFTP$_S$ denote the GuidedStrong algorithm applied to an n-fault tolerant planning problem. Due to the pure heuristic search approach, n-GFTP$_S$ may return suboptimal solutions.

We may expect n-GFTP$_S$ to be efficient when secondary effects are local in the state space because they then will be covered by the search beam of n-GFTP$_S$. In practice, however, secondary effects may be permanent malfunctions that due to their impact on the domain cause a transition to a non-local state. That is a state from which no short path of primary effects exists to the source state. Indeed, in theory, the location of secondary effects may be completely uncorrelated with the location of primary effects. To solve this problem, we develop a specialized algorithm where the planning for primary and secondary effects is decoupled. We constrain our investigation to 1-fault tolerant planning and introduce two algorithms: 1-FTP using blind search and 1-GFTP using guided search. The input to these algorithms is a 1-fault tolerant planning problem, not its induced non-deterministic planning problem.

The 1-FTP algorithm is shown in Figure 1. The function PreImgSA$_f$ computes the preimage of secondary effects. 1-FTP returns two non-deterministic plans F^0 and F^1 for the fault tolerant domain, where F^0 is robust to one fault while F^1 is a recovery plan.

Example 1 An example of the non-deterministic plans F^0 and F^1 returned by 1-FTP is shown in Figure 2 $\diamond$

1-FTP performs a backward search from the goal states that alternate between blindly expanding F^0 and F^1 such that failure states of F^0 always can be recovered by F^1. Initially

```
function 1-FTP($s_0, G$)
1   $F^0 \leftarrow \emptyset$; $C^0 \leftarrow G$
2   $F^1 \leftarrow \emptyset$; $C^1 \leftarrow G$
3   while $s_0 \notin C^0$
4       $f_c^0 \leftarrow \textsc{PreImgSA}(C^0) \setminus C^0 \times Act$
5       $f^0 \leftarrow f_c^0 \setminus \textsc{PreImgSA}_f(\overline{C^1})$
6       while $f^0 = \emptyset$
7           $f^1 \leftarrow \textsc{PreImgSA}(C^1) \setminus C^1 \times Act$
8           if $f^1 = \emptyset$ then return "no solution exists"
9           $F^1 \leftarrow F^1 \cup f^1$
10          $C^1 \leftarrow C^1 \cup \textsc{States}(f^1)$
11          $f^0 \leftarrow f_c^0 \setminus \textsc{PreImgSA}_f(\overline{C^1})$
12      $F^0 \leftarrow F^0 \cup f^0$
13      $C^0 \leftarrow C^0 \cup \textsc{States}(f^0)$
14  return $\langle F^0, F^1 \rangle$
```

Figure 1: The 1-FTP algorithm.

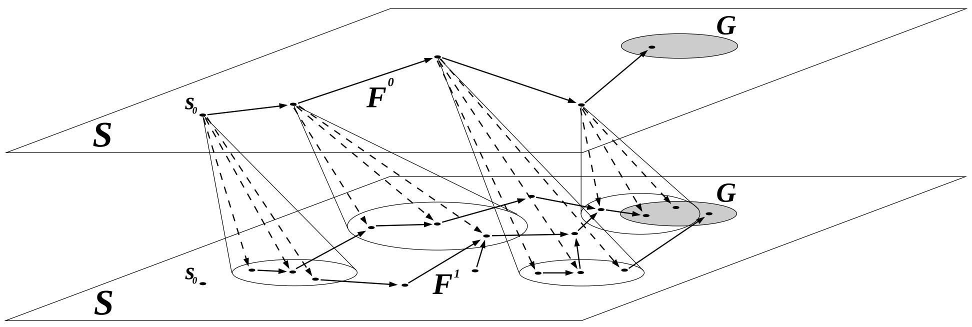

Figure 2: An example of the non-deterministic plans F^0 and F^1 returned by 1-FTP. Primary and secondary effects of actions are drawn with solid and dashed lines, respectively. In this example, we assume that F^0 forms a sequence of actions from the initial state to a goal state, while F^1 recovers all the possible faults of actions in F^0.

F^0 and F^1 are assigned to empty plans (l. 1-2). The variables C^0 and C^1 are states covered by the current plans in F^0 and F^1. 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^0 is expanded with SAs in f^0 (l. 12-13). First, a candidate f_c^0 is computed. It is the preimage of the states in F^0 pruned for SAs of states already covered by F^0 (l. 4). The variable f^0 is assigned to f_c^0 restricted to SAs for which all error states are covered by the current recovery plan (l. 5). If f^0 is empty the recovery plan is expanded in the inner loop until f^0 is nonempty (l. 6-11). If the recovery plan at some point has reached a fixed point and

f^0 is still empty, the algorithm terminates with failure, since in this case, no recovery plan exists (l. 8). We claim without proof that 1-FTP is *sound*, *complete*, and *terminating.*

1-FTP expands both F^0 and F^1 blindly. An inherent strategy of the algorithm, though, is not to expand F^1 more than necessary to recover the faults of F^0. This is not the case for n-FTP$_S$ that does not distinguish states with different number of faults. The aggressive strategy of 1-FTP, however, makes it suboptimal as the example in Figure 3 shows. In

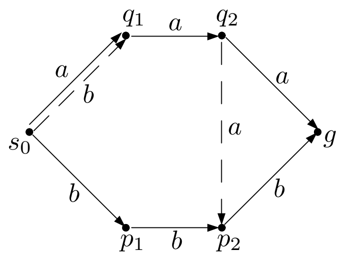

Figure 3: A problem with a single goal state g showing that 1-FTP may return suboptimal solutions. Dashed lines indicate secondary effects. Notice that action a and b only have secondary effects in q_2 and s_0, respectively. In all other states, the actions are assumed always to succeed.

the first two iterations of the outer loop, $\langle p_2, b\rangle$ and $\langle p_1, b\rangle$ are added to F^0 and nothing is added to F^1. In the third iteration of the outer loop, F^1 is extended with $\langle p_2, b\rangle$ and $\langle q_2, a\rangle$ and F^0 is extended with $\langle q_2, a\rangle$. In the last two iterations of the outer loop, $\langle q_1, a\rangle$ and $\langle s_0, a\rangle$ are added to F^0. The resulting plan is

$$\begin{aligned} F^0 &= \{\langle s_0, a\rangle, \langle q_1, a\rangle, \langle q_2, a\rangle, \langle p_1, b\rangle, \langle p_2, b\rangle\} \\ F^1 &= \{\langle p_2, b\rangle, \langle q_2, a\rangle\}. \end{aligned}$$

The worst case length of this 1-fault tolerant plan is 4. However, a 1-fault tolerant plan

$$\begin{aligned} F^0 &= \{\langle s_0, b\rangle, \langle p_1, b\rangle, \langle p_2, b\rangle\} \\ F^1 &= \{\langle q_1, a\rangle, \langle q_2, a\rangle\} \end{aligned}$$

with worst case length of 3 exists.

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 based on the non-deterministic state-set branching framework introduced in previous chapter. The over all design goal of 1-GFTP is to guide the expansion of F^0 toward the initial state and guide the expansion of F^1 toward the failure states of F^0. However, this can be accomplished in many different ways. Below we evaluate three different strategies. For each algorithm, F^0 is guided in a pure heuristic 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^1 toward the initial state as well. The resulting algorithm is similar to 1-GFTP$_S$ and has poor performance. The problem is that the pure heuristic approach causes F^1 only to cover a narrow beam of states in the state space. Error states not within close distance to the primary effects tend not to be covered by F^1. 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^1 toward error states of the precomponents of F^0. This can be done by using a specialized BDD operation that splits the precomponent of F^1 according to the Hamming distance to the error states. The complexity of this operation, however, is exponential in the size of the BDD representing the error states and the size of the BDD representing the precomponent of F^0. Due to the dynamic programming used by the BDD package, the average complexity may be much lower. However, this does not seem to be the case in practice.

The third strategy is the one chosen for 1-GFTP. It expands F^1 blindly, but then prunes SAs from the precomponent of F^1 not used to recover error states of F^0. Thus, it uses an indirect approach to guide the expansion 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^1 in expansion i of F^0 are relevant for recovering error states in expansion $i+1$ of F^0. In addition, we still have an essential problem to solve: to expand F^0 or F^1. There are two extremes.

1. *Expand F^1 until first recovery of f^0.* Compute a complete partitioned backward precomponent of F^0, expand F^1 until some partition in f^0 has recovered error states and add the partition with least h-value to F^0.

2. *Expand F^1 until best recovery of f^0.* Compute a complete partitioned backward precomponent of F^0, expand F^1 until the partition of f^0 with lowest h-value has recovered error states and add this partition to F^0. If none of these error states can be recovered then consider the partition with second lowest h-value and so on.

It turns out that neither of these extremes work well in practice. The first is too conservative. It may add a partition with a high h-value even though a partition with a low h-value can be recovered given just a few more expansions of F^1. The second strategy is too greedy. It ignores the complexity of expanding F^1 in order to recover error states of the partition of f^0 with lowest h-value. Instead, we consider a mixed strategy: spend half of the last expansion time on recovering error states of the partition of f^0 with lowest h-value and, in case this is impossible, spend one fourth of the last expansion time on recovering error states of the partition of f^0 with second lowest h-value, and so on.

The 1-GFTP algorithm is shown in Figure 4. The keys in maps are sorted ascendingly. The instantiation of F^0 and F^1 of 1-GFTP is similar to 1-FTP except that the states in C^0 are partitioned with respect to their associated h-value. Initially the map entry, $\mathbf{C}^0[h_{goal}]$ is assigned to the goal states. [1] The variable t stores the duration of the previous expansion. Initially, it is given a small value ϵ. In each iteration of the main loop (l. 4-22), the precomponents f^0 and f^1 are computed and added to F^0 and F^1. First, the start time t_s is logged by reading the current time t_{CPU} (l. 5). Then a map $\mathbf{PC}$ holding a complete partitioned precomponent candidate of F^0 is computed by PreCompFTP (l. 6).

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

```
function 1-GFTP($s_0, G$)
1  $F^0 \leftarrow \emptyset;\ \mathbf{C}^0[h_g] \leftarrow G$
2  $F^1 \leftarrow \emptyset;\ C^1 \leftarrow G$
3  $t \leftarrow \epsilon$
4  while $s_0 \notin C^0$
5     $t_s \leftarrow t_{CPU}$
6     $\mathbf{PC} \leftarrow$ PreCompFTP($\mathbf{C}^0$)
7     $f^0 \leftarrow \emptyset;\ f_c^0 \leftarrow \emptyset$
8     $\mathbf{f}_c^1 \leftarrow emptyMap$
9     $i \leftarrow 0$
10    while $f^0 = \emptyset \wedge i < |\mathbf{PC}|$
11       $i \leftarrow i + 1;\ t \leftarrow t/2$
12       $f_c^0 \leftarrow f_c^0 \cup \mathbf{PC}[i]$
13       $\langle \mathbf{f}_c^1, f^0 \rangle \leftarrow$ ExpandTimed($f_c^0, \mathbf{f}_c^1, C^1, t$)
14    if $f^0 = \emptyset$ then
15       $\langle \mathbf{f}_c^1, f^0 \rangle \leftarrow$ ExpandTimed($f_c^0, \mathbf{f}_c^1, C^1, \infty$)
16    $t \leftarrow t_{CPU} - t_s$
17    if $f^0 = \emptyset$ then return "no solution exists"
18    $f^1 \leftarrow$ PruneUnused($\mathbf{f}_c^1, f^0$)
19    $F^1 \leftarrow F^1 \cup f^1;\ C^1 \leftarrow C^1 \cup$ States($f^1$)
20    $F^0 \leftarrow F^0 \cup f^0$
21    for $j = 1$ to $i$
22       $\mathbf{C}^0[h_j] \leftarrow \mathbf{C}^0[h_j] \cup$ States($f^0 \cap \mathbf{PC}[h_j]$)
23 return $\langle F^0, F^1 \rangle$
```

Figure 4: The 1-GFTP algorithm.

For each entry in $\mathbf{C}^0$, PreCompFTP inserts the preimage in $\mathbf{PC}$ of each partition of a disjunctive branching partitioning of the transition relation of primary effects. We assume that this partitioning has m subrelations $R_1, \cdots R_m$ where the transitions represented by R_i are associated with a change δh_i of the h-value (in forward direction). The inner loop (l. 10-13) of 1-GFTP expands the two candidates f_c^0 and f_c^1 for f^0 and f^1. In each iteration, a partition of the partitioned precomponent $\mathbf{PC}$ is added to f_c^0 (l. 12).[2] The function ExpandTimed expands f_c^1. In iteration i, the time out bound of the expansion is $t/2^i$. ExpandTimed returns early if

1. a precomponent f^0 in the candidate f_c^0 is found where all error states are recovered (l. 5 and l. 11), or
2. f_c^1 has reached a fixed point.

The preimage added to f_c^1 in iteration i of ExpandTimed is stored in the map entry $\mathbf{f}_c^1[i]$ in order to prune SAs not used for recovery.

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

```
function PreCompFTP($\mathbf{C}^0$)
1  $\mathbf{PC} \leftarrow \mathit{emptyMap}$
2  for $i = 1$ to $|\mathbf{C}^0|$
3      for $j = 1$ to $m$
4          $\mathit{SA} \leftarrow \textsc{PreImgSA}_j(\mathbf{C}^0[h_i]) \setminus C^0 \times \mathit{Act}$
5          $\mathbf{PC}[h_i - \delta h_j] \leftarrow \mathbf{PC}[h_i - \delta h_j] \cup \mathit{SA}$
6  return PC
```

Eventually f_c^0 may contain all the SAs in **PC** without any of these being recoverable. In this case 1-GFTP expands f_c^1 (l. 15) untimed.

```
function ExpandTimed($f_c^0, \mathbf{f}_c^1, C^1, t$)
1  $t_s \leftarrow t_{CPU}$
2  $\mathit{Oldf}_c^1 \leftarrow \bot$
3  $i \leftarrow |\mathbf{f}_c^1|$
4  $\mathit{recovS} \leftarrow \textsc{States}(f_c^1) \cup C^1$
5  $f^0 \leftarrow f_c^0 \setminus \textsc{PreImgSA}_f(\overline{\mathit{recovS}})$
6  while $f^0 = \emptyset \wedge \mathit{Oldf}_c^1 \neq f_c^1 \wedge t_{CPU} - t_s < t$
7      $\mathit{Oldf}_c^1 \leftarrow f_c^1$
8      $i \leftarrow i + 1$
9      $\mathbf{f}_c^1[i] \leftarrow \textsc{PreImgSA}(\mathit{recovS}) \setminus \mathit{recovS} \times \mathit{Act}$
10     $\mathit{recovS} \leftarrow \textsc{States}(f_c^1) \cup C^1$
11     $f^0 \leftarrow f_c^0 \setminus \textsc{PreImg}_f(\overline{\mathit{recovS}})$
12 return $\langle \mathbf{f}_c^1, f^0 \rangle$
```

If f_c^1 has reached a fixed point but no recoverable precomponent f^0 exists, no 1-fault tolerant plan exists and 1-GFTP returns with "no solution exists" (l. 17). Otherwise, f_c^1 is pruned for SAs of states not used to recover the SAs in f^0 (l. 18). This pruning is computed by PruneUnused that traverses backward through the preimages of $\mathbf{f}_c^1$ and marks states that either are error states of SAs in f^0, or states needed to recover previously marked states.

```
function PruneUnused($\mathbf{f}_c^1, f^0$)
1  $\mathit{err} \leftarrow \textsc{SAimg}_f(f^0)$
2  $\mathit{img} \leftarrow \emptyset$; $\mathit{marked} \leftarrow \emptyset$
3  for $i = |\mathbf{f}_c^1|$ to 1
4      $\mathbf{f}_c^1[i] \leftarrow \mathbf{f}_c^1[i] \cap ((\mathit{err} \cup \mathit{img}) \times \mathit{Act})$
5      $\mathit{marked} \leftarrow \mathit{marked} \cup \textsc{States}(\mathbf{f}_c^1[i])$
6      $\mathit{img} \leftarrow \textsc{SAimg}(\mathbf{f}_c^1[i])$
7  return $f_c^1 \cap (\mathit{marked} \times \mathit{Act})$
```

The function SAimg(π) and SAimg$_f$(π) computes the image states of a set of SAs π for primary and secondary effects respectively.

$$\textsc{SAimg}(\pi) \equiv \{s' : \exists \langle s, a \rangle \in \pi \,.\, s \xrightarrow{a} s'\} \quad (1)$$

$$\textsc{SAimg}_f(\pi) \equiv \{s' : \exists \langle s, a \rangle \in \pi \,.\, s \overset{a}{\rightsquigarrow} s'\} \quad (2)$$

The updating of F^0 and F^1 of 1-GFTP (l. 19-22) is similar to 1-FTP, except that $\mathbf{C}^0$ is updated by iterating over $\mathbf{PC}$ and picking SAs in f^0. Notice that in this iteration h_j refers to the keys of $\mathbf{PC}$. We claim without proof that 1-GFTP is *sound*, *complete*, and *terminating.*

The specialized algorithms can be generalized to n faults by adding more recovery plans $F^n, F^{n-1}, \cdots, F^0$. For n-GFTP all of these recovery plans would be indirectly guided by the expansion of F^n. The algorithm is illustrated in Figure 5

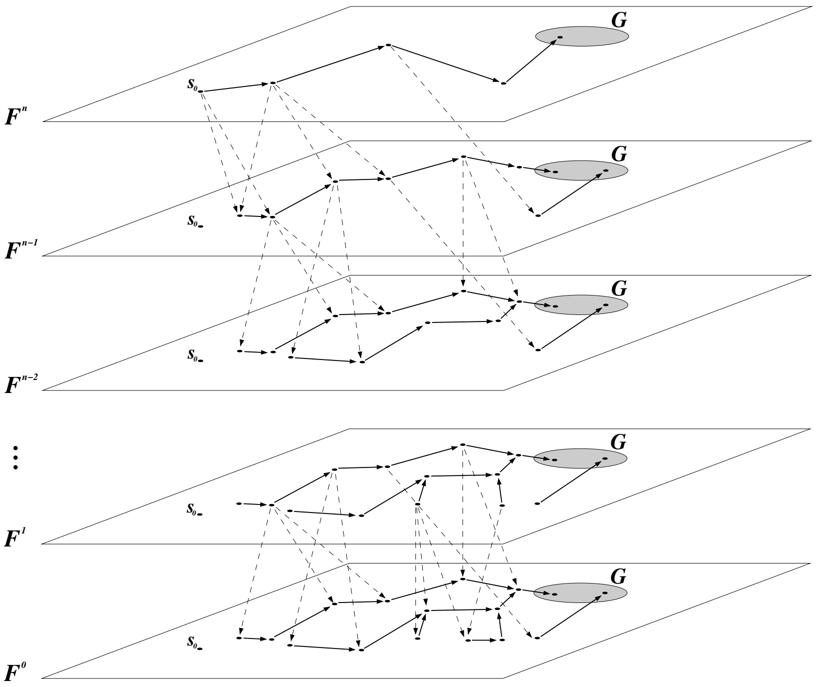

Figure 5: An example of $F^n, \cdots, F^0$ produced by a specialized n-fault tolerant planning algorithm. Primary and secondary effects of actions are drawn with solid and dashed lines, respectively.

6. Experimental Evaluation

The experimental evaluation has two major objectives: to get a better intuition about the nature of fault tolerant plans and to compare the performance of the developed algorithms. 1-FTP, 1-GFTP, 1-FTP$_S$, and 1-GFTP$_S$ have been implemented in the BDD-based BIFROST 0.7 search engine. The domains are defined in an extended version of the Non-deterministic Agent Domain Language (NADL) (Jensen & Veloso, 2000b) called NADL$^+$. NADL$^+$ is described in Appendix 7.

All experiments have been executed on a Redhat Linux 7.1 PC with kernel 2.4.16, 500 MHz Pentium III CPU, 512 KB L2 cache and 512 MB RAM. Since the number of allocated BDD nodes in the unique table (n) and the number of allocated BDD nodes in the operator caches (c) of the BuDDy[3] BDD package (Lind-Nielsen, 1999) may cause an exponential performance difference of BIFROST, we state the settinge of these parameters for each experiment. In general, the sizes of the operator caches and the unique table are adjusted to fit the memory requirements of the most demanding algorithm in an experiment. Thus, performance differences between algorithms are not due to relative differences in cache misses or page faults.

6.1 Unguided Search

We first focus on unguided search and study four fault tolerant planning domains. Two of these, DS1 and PSR, are models of real-world domains.

DS1

DS1 is based on an SMV encoding (Pecheur & Simmons, 2000) of the Livingstone model (Williams & Nayak, 1996) used by the Remote Agent for NASA's Deep Space One probe. The Livingstone model describes the electrical system of the spacecraft. It consists of a system bus and a number of units connected to the bus. These units include a power distribution subsystem, a Ion Propulsion System (IPS), Propulsion Drive Electronics (PDE), a Reaction Control System (RCS), Attitude Control System (ACS), Star Tracker Unit (SRU), and a MICAS camera. We recast the SMV encoding as a fault tolerant planning problem in NADL$^+$. Each bus-command is an action. The primary effect of the command is the changes it causes on the electrical system given that all units work correct. The secondary effect of an action is one of the two faults F2 and F4 considered in the Remote Agent Experiment (Muscettola, Nayak, Pell, & Williams, 1998).

F2 : camera or pasm switch is recoverably stuck on/off.

F4 : an x-z thruster valve is permanently stuck closed.

In addition to these two faults, the Remote Agent Experiment considered two other errors. We are not modeling these, since no 1-fault tolerant plan exists when taking all four faults into account. The following simplifications have been made in the NADL$^+$ model of the SMV description

1. we assume that the state of components is known,
2. attitude errors are assumed to be deterministically computable,
3. relative thrust is assumed to be low or nominal if a valve is stuck otherwise nominal,
4. redundant state variables in the SMV model have been removed. [4]

The NADL$^+$ encoding of the domain has 84 Boolean state variables. We consider generating a 1-fault tolerant plan from an initial state where the IPS is in standby mode, the MICAS

3. Comparison experiments with the CUDD package (Somenzi, 1996) has not shown a significant performance difference (Jensen, 2002).
4. An automatic approach for doing this has been developed in (Yang, Simmons, Bryant, & O'Hallaron, 1999).

camera is "off", and the pasm switch is "on". The goal is to reach a state where the IPS is in thrusting mode, the MICAS camera is "on", and the pasm switch is "off". The BDD package parameters are $n = 1M$ and $c = 100K$. The threshold for merging partitions of a disjunctive transition relation partitioning is 5000. The total size of the transition relation is 104881 and 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 a BDD encoding is very efficient for the kind of constraints modeled by DS1. Despite a fairly large and dense model, a disjunctive transition relation is fast to compute. In addition, a 1-fault tolerant plan for a non-trivial problem in this domain is small and can be generated in less than a second. The experiment demonstrates that BDD-based fault tolerant planning is mature to be applied on significant real-world problems.

For DS1 even 1-fault tolerance imposes a strong restriction on a physical system. No 1-fault tolerant plan exists for the problem, if all of the original four failures are considered. This result is encouraging since it shows that fault tolerant plans are substantially more restricted than weak plans and in addition illustrates the natural connection between the existence of fault tolerant plans and the redundancy characteristics of the modeled system.

PSR

The Power Supply Restoration domain (PSR) is a network of electric lines connected via switching devices (SDs), and fed via circuit-breakers (CBs). Switching devices and circuit breakers can either be open or closed. A circuit-breaker supplies power, when it is closed, and a switching device stops the power propagation if it is open. Consumers may be located on any line and are supplied only when the line is supplied. We assume that each closed circuit-breaker forms a feeder. A feeder is a tree consisting of closed switching devices and lines reachable downstream from the circuit breaker. The leafs are open switching devices and dead end lines. The "simple" PSR domain investigated in (Bertoli, Cimatti, Slanley, & Thiébaux, 2002) is shown in Figure 6. In the depicted configuration, it only has a single feeder rooted in CB_2. In the original definition of PSR domains, each unit in the system

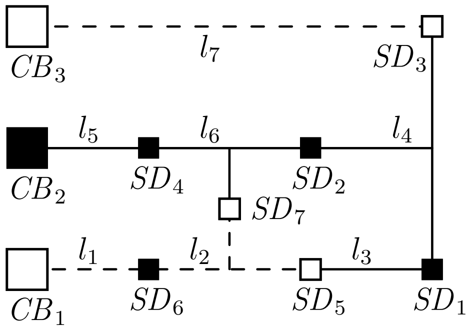

Figure 6: The "simple" PSR domain studied in (Bertoli et al., 2002). A filled box denote that the associated circuit-breaker or switching device is closed. Supplied and unsupplied lines are drawn solid and dashed, respectively.

may fail. Lines may short circuit, and switches may get stuck in one of their two positions. In addition, states are assumed only to be partially observable. We consider a simplified version of the domain where states are fully observable and lines do not fail. The actions of the simplified domain is to open and close switching devises and circuit breakers. The primary effect of actions is that they open and close their associated units. The secondary effect is that the units break permanently and get stuck in their current position.

A specialized linear version of the domain shown in Figure 7 with n ranging from 5 to 35.

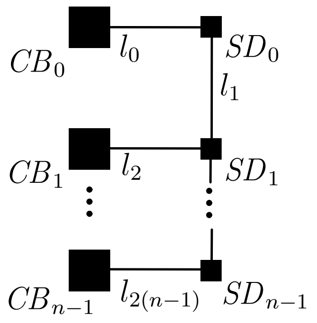

Figure 7: The linear PSR networks used for experiments.

We compare the performance of 1-FTP and 1-FTP$_S$. In the initial state, all switches are open and the goal is to feed all lines. 1-FTP and 1-FTP$_S$ solve the simple network in 6.8 and 11.25 seconds, respectively (0.98 seconds is used on memory allocation, n = 1M and c = 700K). The results of the linear network are shown in Figure 8. The BDD package parameters are $n = 15M$ and $c = 500K$ and 3.38 seconds are used on memory allocation. 1-FTP performs significantly better than 1-FTP$_S$ on this problem. Interestingly, the performance difference is not reflected by 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 BDDs, while the plan size for 1-FTP$_S$ is the size of a single BDD. Similarly to the DS1 domain, 1-fault tolerance imposes a strong constraint on the PSR domain. For most configurations, where a few units already have failed, no 1-fault tolerant plan exists.

Power Plant

The power plant domain is shown in Figure 9 and originates in (Jensen & Veloso, 2000b). The task is to execute the control actions in order to bring the plant from some bad state, where the plant is unsafe or not working properly, to some good state, where the plant satisfies its safety and activity requirements. A single reactor R is surrounded by four heat exchangers H1, H2, H3 and H4. The heat exchangers produce high pressure steam to the four electricity generating turbines T1, T2, T3 and T4. The heat exchangers can fail and leak radioactive substances from the internal water loop to the external steam loop. If this happens, the blocking valve ($a1$, $a2$, $a3$ or $a4$) of the heat exchanger must be closed. However, these valves can fail too, in which case the valves $m2$, $m3$ or $m1$ are used.

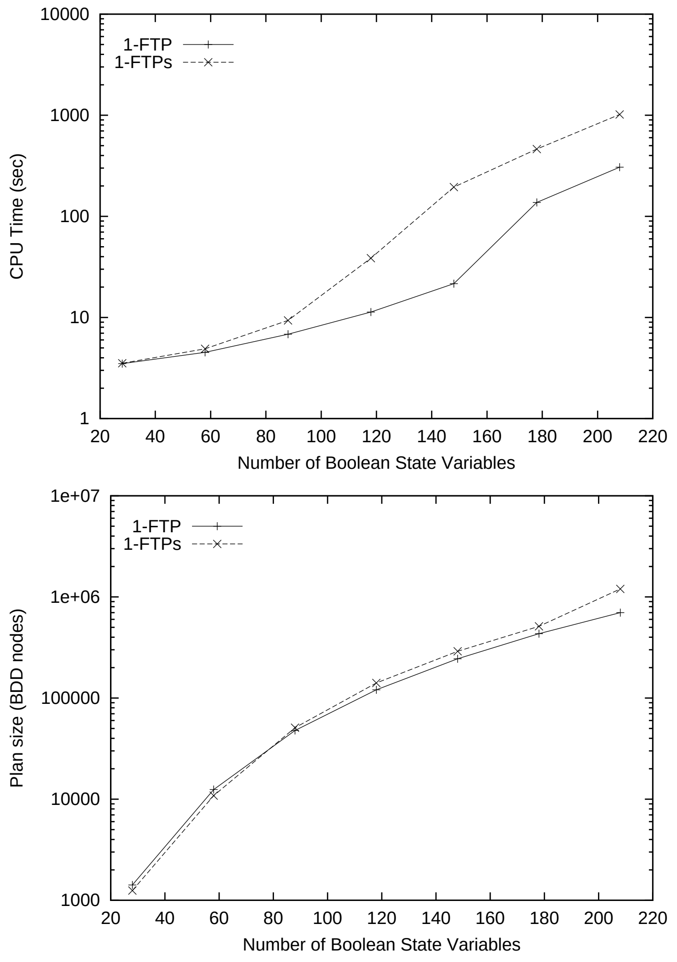

Figure 8: Results of the PSR problems.

Similarly, if turbines fail, they must be shut down by closing one of the valves $b1$, $b2$, $b3$ or $b4$, or $m4$, $m5$ and $m1$. The energy production p of the plant can either be 0,1,2,3 or 4 units of energy per time unit. The production must be adjusted to fit the demand f, if possible. A heat exchanger can only transfer enough energy to a single turbine, and a single turbine can only produce one unit of energy per time unit. The initial state is shown in Figure 9. A failure of heat exchanger 1 is a assumed to have just happened.

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. The second considers controlling two power plants simultaneously. The results are shown in Figure 10. In both experiments, the parameters of the BDD package are $n = 15M$ and $c = 500K$. The time spent on memory allocation is 3.4 seconds. 1-FTP has a slightly better performance than 1-FTP$_S$. However, both algorithms suffer from a large growth rate of the BDDs representing the frontier of the backward search. Again, 1-fault tolerant plans turns out to be hard to

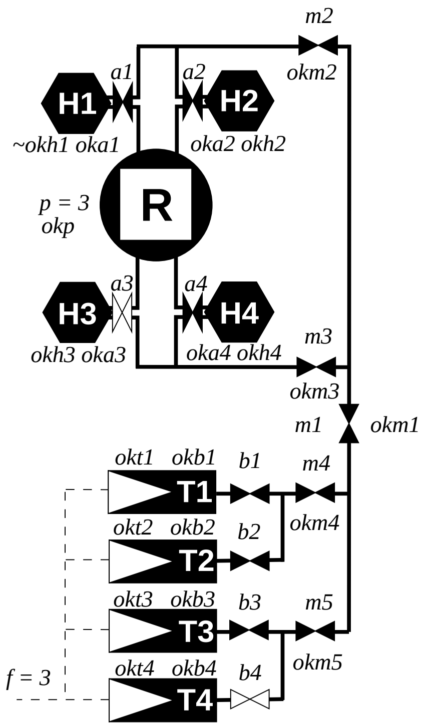

Figure 9: The power plant domain. An open valve is drawn solid and allows water or steam to flow through it. In the depicted state, a failure of heat exchanger 1 is assumed just to have happened.

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

Figure 10: Results of the power plant experiment. The total CPU time and plan size is given by t_{total} and $|sol|$, respectively. The size of the problem is the number of Boolean state variables.

generate. Even though the system is highly redundant, 1-fault tolerant plans only exist for simple malfunctions like the one investigated in this experiment.

Beam Walk

The Beam Walk domain was introduced in (Cimatti, Roveri, & Traverso, 1998) and consists of a robot walking on a beam. The primary effect of the move action is that the robot moves one step forward on the beam. The secondary effect is that it falls down from the beam. The domain is shown in Figure 11. The BeamWalk domain 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

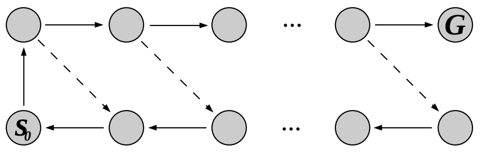

Figure 11: The Beam Walk domain. Solid edges denote primary effects of the move action, while dashed edges denote secondary effects.

a solution is found. The results are shown in Figure 12. As expected, both algorithms

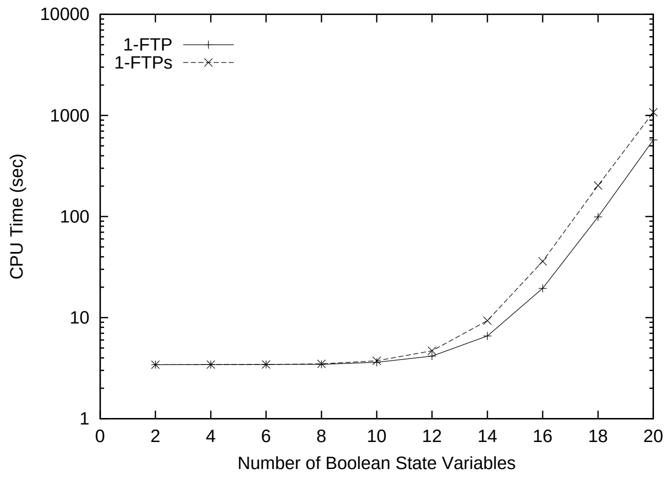

Figure 12: Results of the BeamWalk experiments.

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

6.2 Guided Search

The main purpose of the experiments in this section is to study the difference between 1-GFTP and 1-GFTP$_S$. In particular, we are interested in investigating how sensitive these algorithms are to non-local error states and to what extent we may expect this to be a problem in practice. We study 3 domains, of which SIDMAR descends from a real-world study.

LV

The LV domain is an artificial domain and has been designed to demonstrate the different properties of 1-GFTP and 1-GFTP$_S$. It is an $m \times m$ grid world with initial state $(0, m-1)$ and goal state $(\lfloor m/2 \rfloor, \lfloor m/2 \rfloor)$. The actions are Up, Down, Left, and Right. Above the $y = x$ line, actions may fail causing the x and y position to be swapped. Thus, error states are mirrored in the $y = x$ line. A 9×9 instance of the problem is shown in Figure 13. The essential property is that error states are non-local, but that two states close to each other

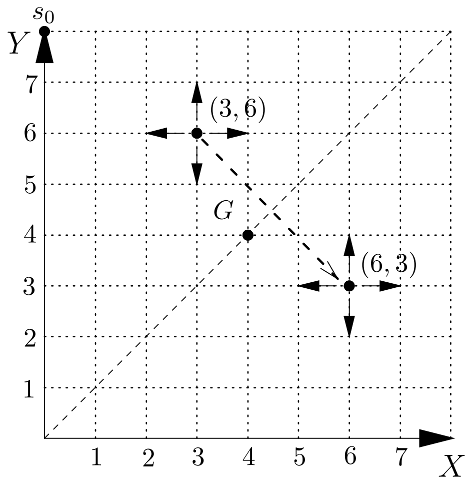

Figure 13: The 9×9 instance of the LV domain.

also have error states close to each other. This is the assumption made by 1-GFTP, but not 1-GFTP$_S$ that requires error states to be local. The heuristic value of a state is the Manhattan distance to the initial state. The BDD package parameters are $n = 5M$ and $c = 500K$. Memory allocation takes 1.4 seconds. The results are shown in Figure 14. As

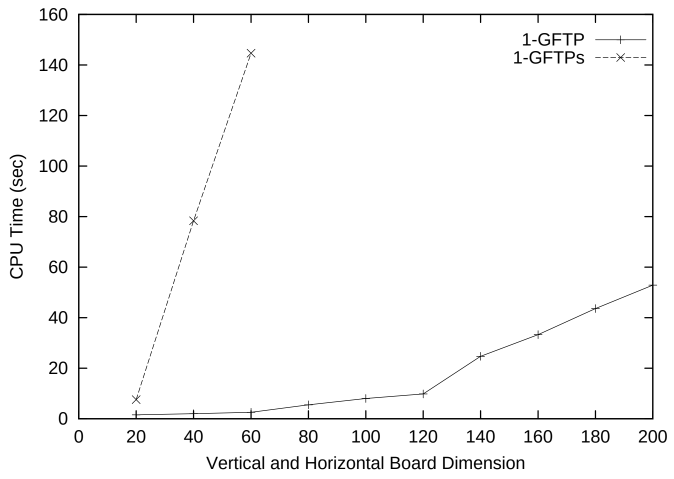

Figure 14: Results of the LV experiments.

depicted, the performance of 1-GFTP$_S$ degrades very fast with m due to the misguidance of the heuristic for the recovery part of the plan. 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 guided precomponents of F^0 will cause this plan to beam out toward the initial state. Due to the relative locality of error states, the pruning of F^1 will cause F^1 to beam out in the opposite direction. Thus, both F^0 and F^1 remain small during the search.

Non-Deterministic 8-Puzzle

The 8-Puzzle consists of a 3×3 board with 8 numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The goal is to reach a configuration, where the tiles are ordered ascendingly left to right, top to bottom. We consider a non-deterministic version of the 8-Puzzle, where the secondary effects are self loops as shown in Figure 15. Thus, error states are the most local possible. We use the usual sum of

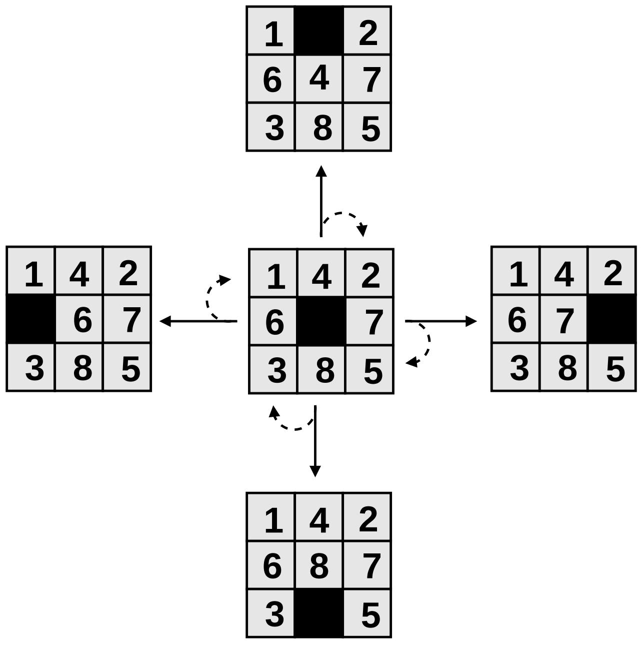

Figure 15: Primary (solid) and secondary (dashed) effects of the non-deterministic 8-Puzzle domain..

Manhattan distances of tiles as a 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 BDD package parameters are $n = 1M$ and $c = 100K$. Memory allocation takes 0.29 seconds. The number of Boolean state variables is 35 in all experiments. The results are shown in Figure 16. The results of the 8-Puzzle experiment further demonstrate the difference between 1-GFTP and 1-GFTP$_S$. Again, 1-FTP performs substantially better than 1-FTP$_S$. 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. As depicted, 1-FTP is slightly faster than 1-GFTP$_S$ in the experiment with a minimum deterministic solution length of 14. For such small problems, we may expect to see this since 1-FTP only expands the recovery plan when needed while 1-GFTP$_S$ expands the recovery part of its plan in each iteration.

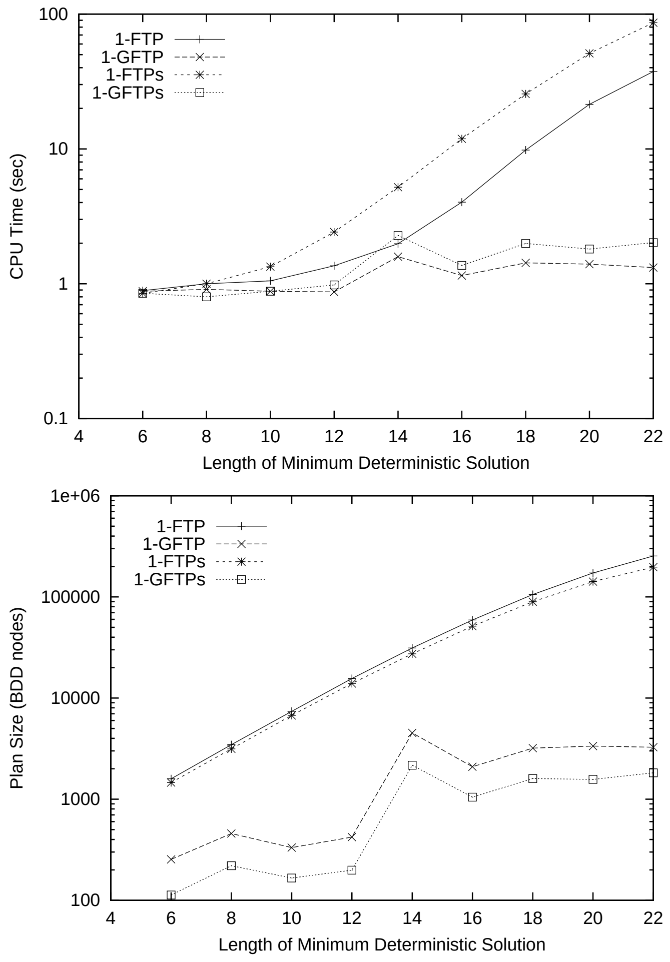

Figure 16: Results of the 8-Puzzle experiments.

SIDMAR

The final experiments are on the SIDMAR domain introduced in Section **??**. The purpose of these experiments is to study the robustness of 1-GFTP and 1-GFTP$_S$ to the kind of errors found in real-world domains. The SIDMAR domain is an abstract model of a real-world steel producing plant in Ghent, Belgium used as an ESPRIT case study (Fehnker, 1999). The layout of the steel plant is shown in Figure 17. The goal is to cast steel of different qualities. Pig iron is poured portion-wise in ladles by the two converter vessels. The ladles can move autonomously on the two east-west tracks. However, two ladles can not pass each other and there can at most be one ladle between machines. Ladles are moved in the north-south direction by the two overhead cranes. The pig iron must be treated differently to obtain steel of different qualities. There are three different treatments: 1) machine 1 and 4, 2) machine 2 and 5, and 3) machine 3. Before empty ladles are moved to the storage place, the steel is cast by the continuous casting machine. A ladle can only leave the casting machine, if there already is a filled ladle at the holding place. We assume that actions of

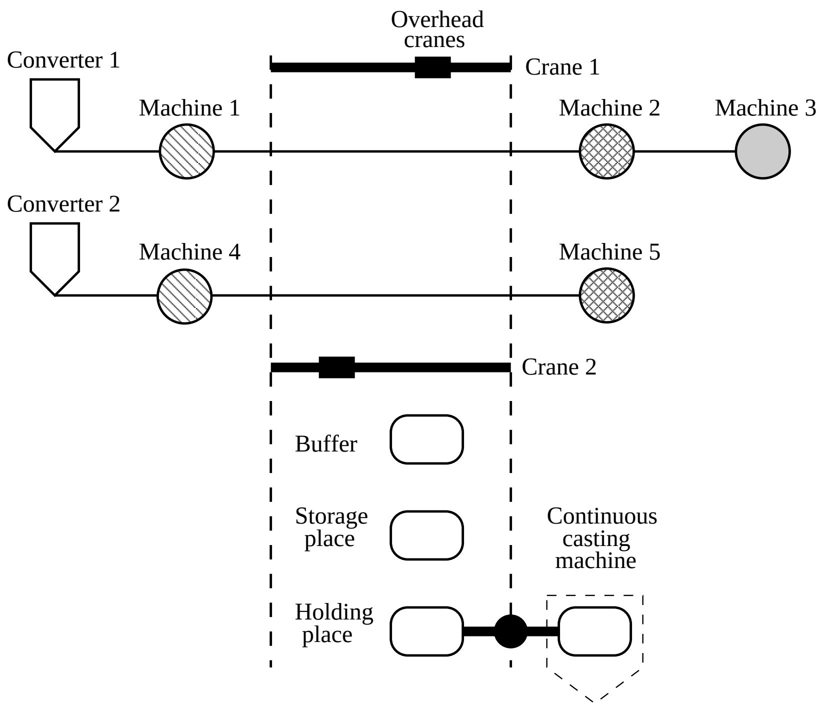

Figure 17: Layout of the SIDMAR steel plant.

machine 1,2,4, and 5 and move actions on the track may fail. The secondary effect of move actions is that nothing happens for the particular move. Later moves, however, may still succeed. The secondary effect of machine actions is that no treatment is carried out, and the machine is broken down permanently.

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 BDD package parameters are $n = 5M$ and $c = 500K$. Memory allocation takes 1.41 seconds. The number of Boolean state variables is 47 in all experiments. The results are shown in Figure 18. Missing data points indicates that the associated algorithm spent more than 500 seconds trying to solve the problem. The only algorithm with good performance is 1-GFTP. The experiment indicates that real-world domains may have non-local error states that limits the performance of 1-GFTP$_s$. Also notice that this is the only domain where 1-FTP does not outperform 1-FTP$_S$. In this domain, 1-FTP seems to be finding complex plans that fulfills that the recovery plan is minimum. Thus, the strategy of 1-FTP to keep the recovery plan as small as possible does not seem to be an advantage in general.

7. Conclusion

In this paper, we have introduced n-fault tolerant plans as a new solution class of SNDP. Fault tolerant plans reside in the gap between weak plans and strong cyclic and strong plans. They are more restrictive than weak plans, but more relaxed than strong cyclic and strong

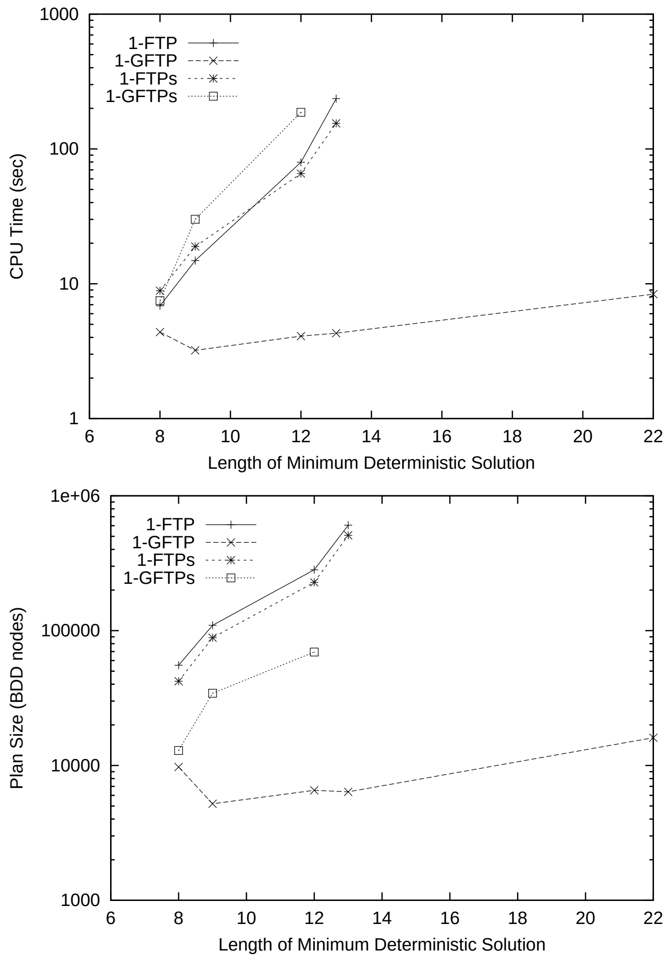

Figure 18: Results of the SIDMAR experiments.

plans. Optimal n-fault tolerant plans can be generated by the strong planning algorithm via a reduction to a strong planning problem. Our experimental evaluation shows, however, that due to non-local error states, it is often beneficial to decouple the planning for primary and secondary effects of actions.

Fault tolerant planning is a first step toward probabilistic uncertainty models in SNDP. A fruitful direction for future work is to move further in this direction and consider fault tolerant plans that are adjusted to the likelihood of faults or to consider probabilistic solution classes with other transition semantics than faults.

Acknowledgments

The authors wish to thank Paolo Traverso, Reid Simmons, and Bruce Krogh for fruitful discussions and feed back on this work. This research was carried out while the first author was at Carnegie Mellon University. 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.

Appendix A. NADL$^+$

NADL was developed as a part of the UMOP project (Jensen, 1999; Jensen & Veloso, 2000b, 2000a; Jensen, Veloso, & Bowling, 2001). However, despite providing a very general framework for modeling non-deterministic planning problems, NADL does not allow additional information about transition costs, heuristic estimates, and failure effects of actions. NADL$^+$ adds these features to the language. There are are three main differences between the two languages

1. NADL$^+$ has three new optional action description components **dg**, **dh**, and **err**. In addition, it uses the entry **heu** to define the value of the heuristic estimate in the initial state and the goal states,
2. An action description may concist of descriptions of several *transition groups*,
3. NADL$^+$ assumes that the system and environment are described by as set of actions instead of a set of agents.

The action component **dg:** *int* associates a transition cost or weight with the action. The component **dh:** *int* describes the change of a heuristic estimate associated with each transition represented by the transition group. The change is always given in forward direction even if the heuristic guides a backward search. Finally, **err:** *formula* defines a set of next states reached by the action given that its execution fails.

An NADL$^+$ problem description consists of: a set of *state variables*, a set of *system* and *environment actions*, and an *initial* and *goal condition*. The set of state variable assignments defines the state space of the domain. The set of system actions must be non-empty while the set of environment actions may be empty if no active environment exists. System and environment actions are assumed to be *synchronous*. At each step, exactly a single system and environment action is performed. The resulting action is called a *joint action*. Only the system actions are controllable. An action has three main parts: a set of *modified* state variables, a *precondition* formula, and an *effect* formula. The set of modified variables are the state variables which may have their value changed by the action. In order for an action to be applicable, the precondition formula must be satisfied in the current state. The effect of the action is defined by the effect formula. The value of state variables not modified by a joint action is unchanged. The initial and goal condition are formulas that must be satisfied in the initial state and the goal states, respectively.

Example 2 An NADL$^+$ planning problem is shown in Figure 19. The problem has two state variables *pos* and *power*. The position is a natural number that can be represented by three Boolean variables. This gives *pos* the domain $\{0, 1, 2, 3, 4, 5, 6, 7\}$. The power is a proposition and is represented by a single Boolean state variable. The system is a robot moving between the eight positions. It has two actions *Right* and *Left*. The cost of both actions is 1. The heuristic is for guiding a backward search from the goal states to the initial state. It therefore estimates the distance to the initial state. This estimate is simply the value of the position. Thus, a successful Left action changes the heuristic estimate with -1, while a successful Right action changes it with $+1$. The effect of the Right action, depends on the *power* variable. If the power is *true* then the position is increased, otherwise nothing happens. For this reason, the transitions of the Right action are partitioned into two transition groups where the first describes the successful outcome of the action where **dh:** is 1, and the second describes the unsuccessful outcome of the action where **dh:** is 0. The Left action is assumed to succeed independent of the value of *power*. It can therefore be described by a single transition group. The environment controls the power with two actions *On* and *Off*. Since the system and environment must apply exactly one action at each step, there are four joint actions *Left-On*, *Left-Off*, *Right-On*, and *Right-Off*. Initially, the power is on and the robot is at position 0. The goal is to reach position 7. The value of the heuristic estimate must be given for the goal states in order to use a branching partitioning to propagate the value of the heuristic estimate to other states. This is done by adding the entry **heu:** 7 to the goal condition. ◇

Syntax of NADL$^+$

Below is the BNF syntax of NADL$^+$. The syntax of formulas is given separately.

```
⟨NADL+⟩        ::= variables ⟨VarDecl⟩ {⟨VarDecl⟩}
                   system ⟨ActionDecl⟩ {⟨ActionDecl⟩}
                   environment {⟨ActionDecl⟩}
                   initially ⟨Formula⟩ [heu : ⟨Number⟩]
                   goal ⟨Formula⟩ [heu : ⟨Number⟩]

⟨VarDecl⟩      ::= ⟨VarType⟩ ⟨IdLst⟩

⟨VarType⟩      ::= bool
                |  nat( ⟨Number⟩ )

⟨IdLst⟩        ::= ϵ
                |  ⟨Id⟩
                |  ⟨Id⟩ {,⟨Id⟩}

⟨ActionDecl⟩   ::= ⟨Id⟩ ⟨TranDecl⟩ {⟨TranDecl⟩}

⟨TranDecl⟩     ::= [dg : ⟨Number⟩]
                   [dh : ⟨Number⟩]
                    mod : ⟨IdLst⟩
                    pre : ⟨Formula⟩
                    eff : ⟨Formula⟩
                   [err : ⟨Formula⟩]
```

```
variables
  nat(3) pos
  bool power
system
    Right
      dg:   1
      dh:   1
      mod:  pos
      pre:  pos < 7 ∧ power
      eff:  pos' = pos + 1

      dg:   1
      dh:   0
      mod:  pos
      pre:  pos < 7 ∧ ¬power
      eff:  pos' = pos
    Left
      dg:   1
      dh:   −1
      mod:  pos
      pre:  pos > 0
      eff:  pos' = pos − 1
environment
    On
      mod:  power
      pre:  ¬power
      eff:  power'
    Off
      mod:  power
      pre:  power
      eff:  ¬power'
initially
  pos = 0 ∧ power
goal
  pos = 7
  heu: 7
```

Figure 19: An NADL^{+} planning problem.

An identifier is a sequence of numbers, letters and the character "_" that does not begin with a number. The syntax of formulas is given below. The $->$ operator is an *if-then-else* operator. The relation operator <> denotes *not equal to.* The Boolean operators => and <=> denote logical implication and bi-implication, respectively. The other operators have their usual semantics.

$$\begin{array}{lcl} \langle \mathit{Formula} \rangle & ::= & \langle \mathit{Formula} \rangle - > \langle \mathit{Formula} \rangle \; , \langle \mathit{Formula} \rangle \\ & | & \langle \mathit{Formula} \rangle \; \langle \mathit{BoolOp} \rangle \; \langle \mathit{Formula} \rangle \\ & | & \langle \mathit{NumExp} \rangle \; \langle \mathit{RelOp} \rangle \; \langle \mathit{NumExp} \rangle \end{array}$$

```
                |    ~ ⟨Formula⟩
                |    ( ⟨Formula⟩ )
                |    true
                |    false
                |    ⟨Id⟩

⟨BoolOp⟩       ::=  => | <=> | /\ | \/

⟨RelOp⟩        ::=  = | <> | > | <

⟨NumExp⟩       ::=  ⟨Id⟩
                |    ⟨Number⟩
                |    ⟨Number⟩ ⟨NumOp⟩ ⟨Number⟩
⟨NumOp⟩        ::=  + | −
```

References

Bahar, R., Frohm, E., Gaona, C., Hachtel, E., Macii, A., Pardo, A., & Somenzi, F. (1993). Algebraic decision diagrams and their applications. In *IEEE/ACM International Conference on CAD*, pp. 188–191.

Balemi, S., Hoffmann, G. J., Gyugyi, P., Wong-Toi, H., & Franklin, G. F. (1993). Supervisory control of a rapid thermal multiprocessor. *IEEE Trans. on Automatic Control*, *38*(7).

Bertoli, P., Cimatti, A., Slanley, J., & 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*.

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

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

Cho, K.-H., & Lim, J.-T. (1998). Synthesis of fault tolerant supervisor for automated manufacturing systems: A case study on photolithographic process. *IEEE Trans. on Robotics and Automation*, 348–351.

Cimatti, A., Pistore, M., Roveri, M., & Traverso, P. (2003). Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, *147*(1-2). Elsevier Science publishers.

Cimatti, A., Roveri, M., & Traverso, P. (1998). Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pp. 875–881. AAAI Press.

Cin, M. D. (1997). Verifying fault-tolerant behavior of state machines. In *Proceedings of the Second IEEE High-Assurance Systems Engineering Workshop HASE 97*, pp. 97–99.

Doyle, R. J. (1995). Determining the loci of anomalies using minimal causal models. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 1821–1827.

Fehnker, A. (1999). Scheduling a steel plant with timed automata. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press.

Feng, Z., & Hansen, E. (2002). Symbolic LAO* search for factored markov decision processes. In *Proceedings of the AIPS-02 Workshop on Planning via Model Checking*, pp. 49–53.

Georgeff, M., & Lansky, A. L. (1986). Procedural knowledge. *Proceedings of IEEE*, *74*(10), 1383–1398.

Giunchiglia, F., Spalazzi, L., & Traverso, P. (1994). Planning with failure. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems.*

Hammond, K. (1990). Explaining and repairing plans that fail. *Artificial Intelligence*, *40*, 173–228.

Hoey, J., St-Aubin, R., & Hu, A. (1999). SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*, pp. 279–288.

Jensen, R. M. (1999). OBDD-based universal planning in multi-agent, non-deterministic domains. Master's thesis, Technical University of Denmark, Department of Automation. IAU99F02.

Jensen, R. M. (2002). A comparison study between the CUDD and BuDDy OBDD package applied to AI-planning problems. Tech. rep., Computer Science Department, Carnegie Mellon University. CMU-CS-02-173.

Jensen, R. M. (2003a). *Efficient BDD-Based Planning for Non-Deterministic, Fault-Tolerant, and Adversarial Domains.* Ph.D. thesis, Carnegie Mellon University. CMU-CS-03-139.

Jensen, R. M. (2003b). The BDD-based InFoRmed planning and cOntroller Synthesis Tool BIFROST version 0.7. `http://www.itu.edu/people/rmj`.

Jensen, R. M., & Veloso, M. M. (2000a). OBDD-based deterministic planning using the UMOP planning framework. In *Proceedings of the AIPS-00 Workshop on Model-Theoretic Approaches to Planning*, pp. 26–31.

Jensen, R. M., & Veloso, M. M. (2000b). 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., & Bowling, M. (2001). Optimistic and strong cyclic adversarial planning. In *Pre-proceedings of the 6th European Conference on Planning (ECP'01)*, pp. 265–276.

Jensen, R. M., Veloso, M. M., & Bryant, R. E. (2003). Guided symbolic universal planning. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling ICAPS-03*, pp. 123–132.

Kleer, J., & Williams, B. C. (1987). Diagnosing multiple faults. *Artificial Intelligence*, *32*(1), 97–130.

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

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

McCalla, G., & Ward, B. (1982). Error detection and recovery in a dynamic planning environment. In *Proceedings of the 2nd National Conference on Artificial Intelligence (AAAI'82)*, pp. 172–175.

Muscettola, N., Nayak, P. P., Pell, B., & Williams, B. C. (1998). Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, *103*(1-2), 5–47.

Özveren, C. M., & Willsky, A. S. (1991). Stability and stabilizability of discrete event dynamic systems. *Journal of ACM*, 730–752.

Passino, K. M. (1994). Lyapunov stability of a class of discrete event systems. *IEEE Trans. on Automatic Control*, 269–279.

Pecheur, C., & Simmons, R. (2000). From livingstone to SMV. In *FAABS*, pp. 103–113.

Peot, M., & Smith, D. (1992). Conditional nonlinear planning. In *Proceedings of the 1'st International Conference on Artificial Intelligence Planning Systems (AIPS'92)*, pp. 189–197. Morgan Kaufmann.

Perraju, T. S., Rana, S. P., & Sarkar, S. P. (1997). Specifying fault tolerance in mission critical systems. In *Proceedings of High-Assurance Systems Engineering Workshop, 1996*, pp. 24–31. IEEE.

Ramadge, P. J., & Wonham, W. M. (1987). Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, *25*(1), 206–230.

Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., & Teneketzis, D. (1995). Diagnosability of discrete-event systems. *IEEE Trans. on Automatic Control*, *40*(9), 1555–1575.

Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., & Teneketzis, D. (1996). Failure diagnosis using discrete-event models. *IEEE Trans. on Control Systems Technology*, *4*(2), 105–123.

Sampath, M., Sengupta, R., Lafortune, S., & Teneketzis, D. (1998). Active diagnosis of discrete-event systems. *IEEE Trans. on Automatic Control*, *43*(7), 908–929.

Senjen, R., & De Beler, M. (1993). Hybrid expert systems for monitoring and fault diagnosis. In *Proceedings of the 9th IEEE Conference on Artificial Intelligence Applications*, pp. 235–241.

Somenzi, F. (1996). CUDD: Colorado university decision diagram package.. `ftp://vlsi.colorado.edu/pub/`.

Su, R. (2001). Decentralized fault diagnosis for discrete-event systems. Master's thesis, Dept. Electl. Engrg., Univ. of Toronto.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: an Introduction.* MIT Press.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, *38*(3), 58–68.

Thiébaux, S., & Cordie, M. O. (2001). Supply restoration in power distribution systems – a benchmark for planning under uncertainty. In *Pre-Proceedings of the 6th European Conference on Planning (ECP-01)*, pp. 85–96.

Weld, D. S., Anderson, C. R., & Smith, D. E. (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98).*

Williams, B., & Nayak, P. (1996). A model-based approach to reactive self-configuring systems. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96).*

Williams, B. C., Ingham, M., Chung, S. H., & Elliott, P. H. (2003). Model-based programming of intelligent embedded systems and robotic space explorers. In *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, Vol. 9, pp. 212–237.

Yang, B., Simmons, R., Bryant, R. E., & O'Hallaron, R. O. (1999). Optimizing symbolic model checking for constraint-rich models. In *Proceedings of Computer-Aided Verification (CAV'99)*, pp. 328–340.