Improving the Reliability of Artificial Intelligence Planning Systems by Analyzing their Failure Recovery*

Adele E. Howe
Computer Science Department
Colorado State University
Fort Collins, CO 80523

email: howe@cs.colostate.edu
telephone: 303-491-7589

Index Terms: Artificial Intelligence, Planning, Failure Recovery, Reliability, Debugging

March 31, 1994

Abstract

As planning technology improves, Artificial Intelligence planners are being embedded in increasingly complicated environments: ones that are particularly challenging even for human experts. Consequently, failure is becoming both increasingly likely for these systems (due to the difficult and dynamic nature of the new environments) and increasingly important to address (due to the systems' potential use on real world applications). This paper describes the development of a failure recovery component for a planner in a complex simulated environment and a procedure (called Failure Recovery Analysis) for assisting programmers in debugging that planner. The failure recovery design is iteratively enhanced and evaluated in a series of experiments. Failure Recovery Analysis is described and demonstrated on an example from the Phoenix planner. The primary advantage of these approaches over existing approaches is that they are based on only a weak model of the planner and its environment, which makes them most suitable when the planner is being developed. By integrating them, failure recovery and Failure Recovery Analysis improve the reliability of the planner by repairing failures during execution and identifying failures due to bugs in the planner and failure recovery itself.

*This research was supported by a DARPA-AFOSR contract F49620-89-C-00113, the National Science Foundation under an Issues in Real-Time Computing grant, CDA-8922572, and a grant from the Office of Naval Research under the University Research Initiative N00014-86-K-0764. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. This research was conducted as part of my PhD thesis research at the University of Massachusetts. I would like to thank my thesis advisor, Paul Cohen, and my thesis committee for their advice, guidance and supervision of this research. I also wish to thank the anonymous reviewers for their comments, which helped clarify much of the presentation in the paper.

1 Introduction

As planning technology improves, Artificial Intelligence planners are being embedded in increasingly complicated environments: ones that are particularly challenging even for human experts. Consequently, failure is becoming both increasingly likely for these systems and increasingly important to address. Failure is increasingly likely because of the difficult and dynamic nature of the new environments; failure is increasingly important to address because of the systems' potential use on applications such as scheduling manufacturing production lines [26] and Hubble space telescope time [17], and controlling robots [16].

AI planners determine a course of action; it may be the next action to be taken or may be a long sequence of actions. Plan failures may be caused by actions not having their intended effects, by unexpected environmental changes or by inadequacies in the planner itself. Plan failures may be discovered as the plan is being developed or during its execution. This paper describes an integrated approach to dealing with both types of failure, which uses feedback from failure recovery to help debug plan failures and improve failure recovery. This approach was developed as part of my thesis research [10] on improving the reliability of the Phoenix planner.

1.1 Approaches to Improving Planner Reliability

In general, software failures have been addressed in two ways: automated failure recovery and debugging. The first involves designing the software to detect and repair its own failures. The second method is to debug the software to remove the causes of failure.

The first part of this paper describes the design and the design methodology of automated failure recovery for an AI planner. The second part describes a procedure (called *Failure Recovery Analysis* or FRA) for analyzing the performance of failure recovery and identifying how the planner's knowledge base might influence the occurrence of particular failures. The two parts are tightly integrated; as Figure 1 shows, failure recovery is like a loop within FRA. Failure recovery repairs failures that arise during normal planning and acting. FRA "watches" the performance of failure recovery for clues to bugs in the planner and informs the designer of the bugs. The designer can then attempt to repair the plan knowledge base to prevent the failure from occurring again and, using FRA, can evaluate whether the repair was successful.

The two loops are coordinated by common access to the knowledge base of plans and failure recovery methods. During the inner loop, plans are instantiated to achieve goals in the target environment and are repaired using failure recovery methods stored in the plan knowledge base; during the outer loop, plans are analyzed to understand the observed behaviors and are repaired to avoid detrimental behaviors.

1.2 The Target Planner and its Environment

This research was developed using the Phoenix system [4][1]. The Phoenix system consists of a simulated environment, a set of agents that operate in that environment and an experimental interface for running experiments and collecting data.

The simulated environment is forest fire fighting in Yellowstone National Park. Fires spread at rates and directions that are influenced by weather (e.g., temperature, wind speed and direction, and humidity) and by terrain (e.g., ground cover, variations in elevation, moisture content in the foliage and natural and artificial boundaries, firelines). Agents work together to contain forest fires by removing fuel from their paths (a process called "building fireline). A single agent, the fireboss, directs field agents as to where to build fireline and how to navigate through the burning forest. The fireboss also directs support agents (e.g., fuel carriers, watchtowers and helicopters) and integrates the information gathered by them. Figure 2 shows the interface to the Phoenix system. The map in the upper part of the display shows Yellowstone National Park north of Yellowstone Lake and prior to the famous fires. Features such as wind speed and direction are shown in the window in

[1]The basic Phoenix planner as described in [4] included no failure recovery component; the research described in this paper augments the original system.

the upper left, and geographic features such as rivers, roads and terrain types are shown as light lines or grey shaded areas. Four bulldozers are building fireline around a fire near the center of the figure. A watchtower is visible at the top near the center.

Due mostly to the dynamics of the environment (e.g., weather changes and fire starts), the environment is challenging for current planning technology. To address the dynamics, Phoenix agents possess an architecture that consists of two layers of control, as well as sensors and effectors. The lowest layer, reflexes, address changes that occur quickly. Reflexes take pre-programmed action in reaction to simple, easily recognized situations. At a higher, more time consuming and deliberative level, the planner coordinates actions and avoids detrimental plan interactions. The planner is the focus of the efforts to improve reliability because the reflexes are extremely limited in their abilities and scope.

Plans in this environment and with this planner tend to fail fairly often. Plans can fail because they are based on assumptions of environment change or lack thereof that prove to be overly optimistic. Plans can fail because they are based on obsolete, uncertain or limited information. Phoenix plans also fail because they include bugs and could not possibly have not been tested in all possible situations.

2 Failure Recovery

The purpose of failure recovery is to repair, as efficiently as possible, the plan so as to resume progress toward the failed plan's goal. The failures can be either planning time (e.g., inability to produce a plan or problems detected during off-line checking) or execution time (e.g., due to changes in world state or actions not achieving their desired effect). Most of the planning literature has addressed planning time failure, while literature about robotics and more general software tends to address execution time failure.

The two basic approaches to failure recovery are backward and forward recovery [14]. In backward recovery, the system is returned to some previous correct state and resumes execution from there. Backward recovery requires that actions can be undone and that the system has full control over its environment; these requirements preclude backward recovery for many robotics and planning systems. Forward recovery transforms the failure state to a correct state by repairing the failure [6]. Approaches to forward recovery differ on how they decide on the appropriate transformation or repair: through formal analysis, normal planning, or heuristics.

Formal analysis approaches decide how to repair failures by referencing a complete model of failures and their causes. Time Petri net models, real-time logic and fault tree analysis are common techniques for modeling the causes of failures and for guiding recovery in software (e.g., [14,19]). Formal theories of planning and replanning have been proposed to guide plan modification (e.g., [12,18]).

Failure recovery can be treated as a normal part of the planning process. Lesser's Functionally Accurate, Cooperative (FA/C) paradigm for distributed problem solving [13] and Ambros-Ingerson et al.'s IPEM [1] treat failure recovery as just another planning or problem solving task.

Both recovery through formal analysis and as part of planning require that the planner employs a strong model of what to do in any situation, including failures. Heuristic approaches allow for gaps in knowledge and apply recovery methods to repair the failure. Typically, heuristic approaches operate by "retrieve-and-apply" [21] which maps observed failures to suggested responses. The most comprehensive and commonly used domain-independent strategy is replanning (e.g., [15,30]), which involves restating the planning problem and re-initiating planning. Other domain-independent recovery methods are based on an informal model of how that planner's plans fail and can be modified (e.g., [3,30]). Robotics and other areas that treat planning as a subtask have favored domain-dependent recovery methods (e.g., [16,29]).

Many different approaches have been proposed for failure recovery; most depend either on the domain or on the planner design. However, the literature offers few suggestions about how to design failure recovery for a *new* planner or domain. For constructing domain-dependent recovery programs, Nof et al. [20] propose a four-step framework: analyze the task, develop alternative recovery strategies, determine a selection strategy, and update based on experience with the system. Similarly, Simmons advocates starting the system with basic competence at its task (i.e., no failure recovery) and then adding execution monitoring and failure recovery methods as needed [23]. Wilkins [30] advocates combining both domain-independent and domain-specific methods; the system can try the more efficient domain-specific methods when they are available, but fall back on the domain-independent, when necessary.

2.1 Designing Failure Recovery for Phoenix

Most of the previously mentioned approaches to failure recovery classify the failure and select from a set of methods for adapting the plan in progress, as shown in Figure 3. The system continues to execute (planning and acting) until it recognizes that its actions are failing. Then, the system deals with the failure by taking some corrective action. The approaches differ in their classification of failures and their recovery methods; the designer needs to decide on the failure classification and the set of recovery methods for a new domain. The approach to these decisions that is adopted in this research is to begin with a flexible method selection mechanism and a core set of recovery methods and then refine the set by evaluating failure recovery performance in the host environment. This section will describe the basic design of failure recovery for the Phoenix planner and the methodology that directed the re-design and evaluation of that failure recovery component.

Design of Failure Recovery: In Phoenix, failures can be detected during construction of the plan or during its execution, and can be due to anything from rapid change in the environment to bugs in the plans. At present, the Phoenix fireboss detects ten types of failures, and bulldozers detect five types, as shown in Table 1. The classifications of failure types are largely domain-dependent. Failures are classified by what is known about what finally blocked the plan from continuing successfully; the actual cause of the failure is unknown.

In Phoenix, failure recovery is initiated in response to detecting a failure, an event that precludes successful completion of some plan. Failure recovery iteratively tries recovery methods until one works, at which point the plan is resumed. For example, Figure 4 depicts the flow of control between failure recovery and the rest of the plan for an *insufficient progress* failure (a failure detection mechanism determined that the plan is taking too long to complete). Failure recovery deals with this failure by searching a library of recovery methods for those applicable to the failure type. It selects one method from the possible set and executes it. If the recovery method succeeds, then failure recovery completes and the rest of the plan executes; otherwise, it abandons the current attempt, selects another method and tries again. This process continues until a method succeeds or it runs out of methods to try.

The recovery methods make mostly simple repairs to the structure of the failed plan. Consequently, these methods can be used in different situations, do not require expensive explanation, and provide a response to any failure. This strategy sacrifices efficiency for generality and results in a planner capable of responding to most failures, but perhaps in a less than optimal manner. An ideal set of methods should be a combination of general and domain-specific methods where the domain-specific methods provide the most efficient response to failure and the general methods provide a fallback for cases never before encountered. Failure recovery was started with the six methods listed in Table 2, and domain-specific methods were added based on evaluating the performance of the existing set.

The first four methods make changes local to the failed action and surrounding actions; the last two replan at either the next higher level of plan abstraction or at the top level. All of these methods make structural changes to plans in progress and are applicable to nearly all failures. These recovery methods, or ones very like them, have been used in other recovery systems. WATA is similar to the "retry" method described in [8]. RV and RA are Phoenix specific forms of SIPE's Reinstantiate [30]. SA is similar to "try a different action" in SWITCH [22] and action substitution used for debugging in GORDIUS [24]. The two replan methods, RT and RP, are constrained forms of the more general replanning done in many failure recovery systems. These recovery methods update the failed plan to better reflect new environment conditions (appropriate if the failure was due to the environment) or replace parts of the plan to circumvent the problem (appropriate if the plan failure was due to a bug in the plan).

2.2 Methodology: Tailoring Failure Recovery to Phoenix

Failure recovery was added to the Phoenix planner by gradually broadening the classification of failures and systematically adding new recovery methods to address them, evaluating the recovery methods as they were added. The key to the design process was understanding the effect of new methods by evaluating the performance of the entire set.

The methodology was to define performance in terms of an expected cost model and use the model to direct improvements to the design[2]. The assumption underlying the cost model is that not all methods will predictably succeed. The model assesses the *total* cost of applying a sequence of methods. The expected cost model accumulates the cost of trying enough methods to repair the failure; this amounts to the cost of trying the first method plus the cost of trying a second method if the first method fails plus the cost of trying the third and so on until no methods remain. When no methods remain, then the cost is the cost of outright failure. This combination of costs are captured in the following equation:

$$\begin{aligned} C(S_i) &= C(M_a) + (1 - P(M_a|S_i))[C(M_b) + ...(1 - P(M_y|S_i))[C(M_z) + \\ &\quad (1 - P(M_z|S_i))[C_F]]...] \end{aligned} \tag{1}$$

where $C(S_i)$ is the expected cost of recovery for failure S_i; $C(M_a)$ is the cost of employing an applicable method a; C_F is the cost of failing to recover; and $P(M_a|S_i)$ is the probability of method a succeeding in failure S_i.

The model is based on three assumptions about the independence of the parameters:

[2]The cost model and experiments testing improvements were described originally, but in less detail in [11].

1. **The cost of each method $C(M_m)$ is independent of the situation S_i.** Because the recovery methods are designed to be domain-independent, intuition suggests that their costs may be independent of when they are used.
2. **$C(M_m)$ is independent of the order of execution of the recovery methods.** Having tried one recovery method which failed should not cause other methods to be more or less expensive to execute.
3. **$P(M_m|S_i)$ is independent of the order of execution of the recovery methods.** If this assumption is true, then whether a recovery method is tried after another fails should have no effect on whether the new methods succeeds.

These assumptions will not be true in all environments and planners. They were tested for whether they were true in Phoenix.

2.2.1 Experiment 1: Baselines for Performance

This experiment gathered baselines for the parameters in the performance model and tested the assumptions of that model in Phoenix. It consisted of 116 trials resulting in 2462 failure situations and 5558 attempts to recover from the failures. The three fires were set at intervals of eight simulation hours. Approximately once an hour, the wind speed and direction were varied by up to ± 3 kph and up to ± 30 degrees. The agents were given failure recovery methods as described in Section 2.1. Recovery methods were selected randomly without replacement for each failure encountered (the one exception being that one failure type, `insufficient-progress`, could be repaired only by one of the two replan methods).

The experiment collected data on what failures occurred, what recovery methods were tried, the order in which recovery methods were tried, and the cost of executing the recovery methods. Cost is measured in seconds of simulation time required to repair a plan using the method. This measure was chosen because it provides a uniform, non-domain-specific assessment of what the agent loses by executing failure recovery: the agent loses time that might be spent generating other parts of the plan. This measure is not perfect, and other measures were considered as well.

One potential problem is that the cost measure assesses the environment only indirectly. The environment effect of the loss of time on the goal of containing forest fires is how much more of the forest burns while the fireboss is thinking, how much fuel is consumed by bulldozers idling, and how many bulldozers are trapped by fire while the fireboss is thinking. However, such environment measures were rejected because they have several problems as cost measures in Phoenix. First, they would have to be combined, but they have different units (i.e., acres, liters and count of bulldozers) and different relative values as well (e.g., what is the cost of losing a simulated bulldozer and how does it compare to acres of forest). Second, these cost factors are highly variable estimates biased by environment conditions that do not reflect the relative costs in the recovery methods. My knowledge of the planner suggests that no method in the basic set is more sensitive to the state of the environment (e.g., weather and terrain that also affect these environment cost factors) than the rest, and so to simplify comparison, the cost metric should be selected to minimize the environment's influence. As will be shown in the first experiment, the cost measure appears to be largely insensitive to context.

Another problem is that it is difficult to determine the scope of the time required to repair a plan. This time certainly should include the time to execute the recovery method itself, but how much of the time to execute the repair should be included: everything directly or indirectly added, just the obvious differences or just the repeated actions? For example, because the replan methods radically modify the plan, it is difficult to separate the newly added parts from the original. The cost measure was conservative and included parts of the original plan. This led to the paradox that replan methods that failed cost significantly less than those that succeeded. Unfortunately, alternative cost measures (including environment-based metrics) produced similar results. Five other measures were tested in a pilot experiment; all the measures either exhibited the same problems or neglected major factors related to cost. At least for Phoenix, the precision of a single comprehensive measure of cost may be asking too much, but continues to be examined.

In the meantime, the simulation time measure of cost was used to give a general measure of the effect of failure recovery design on cost.

On the question of whether the assumptions held on Phoenix, statistical tests on the data (ANOVAs for assumptions one and two and chi-square tests for assumption three) showed that the assumptions held for a subset of the methods[3]. In particular, the performance of the two replan methods was sensitive to the failure situation in which they are applied (assumption one) and to whether the replans follow other methods (assumptions two and three); the four remaining methods were insensitive to failure context and order of application. Hence, the local domain-independent methods appear to be insensitive to their context of application.

2.2.2 Experiment 2: Selecting Recovery Methods

Failure recovery, as implemented for experiment 1, selected recovery methods at random without replacement to repair each failure. The cost model can be used to guide the selection of recovery methods to minimize total cost. Simon and Kadane [25] showed that, for problems of the type described by equation 1, the expected cost of executing a sequence is minimized by the strategy of trying the methods in decreasing order of

$$\frac{P(M_m|S_i)}{C(M_m)}$$

which intuitively means "select the method that is most likely to succeed with the lowest cost". So if a single method has the best cost to success ratio, it may be the only method executed; if several methods are cheap but not guaranteed to succeed, then together they may still cost less than a single expensive, but certain of success method.

The first improvement was to add the selection strategy to failure recovery for Phoenix and then re-run the same experiment scenario for about the same number of trials as in experiment

[3]Because the bulldozers do far less planning than the fireboss, the bulldozer results tend to be similar, but less interesting. Consequently, only the results of the fireboss are reported.

1. The costs of recovering from each failure type in this experiment were compared to the results from experiment 1. Table 3 shows the mean costs of failure recovery for each failure type and the percentage overall for each failure type[4], for experiments 1 and 2. The mean recovery cost for the fireboss was 2943 for Experiment 1 ($sd = 3038$, $n = 1053$) and 2500 for Experiment 2 ($sd = 4024$, $n = 1026$). A z-test[5] on the differences between the mean recovery costs for the fireboss in the two experiments yielded a significant result ($z = -2.83, p < .0023$). Because not all of the assumptions of the model held, the selection strategy is not guaranteed to be optimal, but based on these results, it appears to significantly reduce the overall cost of failure recovery.

2.2.3 Experiment 3: Tailoring the Method Set

Intuition suggests that the best set of recovery methods are those tailored specifically to a domain and the failures encountered there. By tailoring, one should construct the cheapest and most effective methods for repairing the failure. This experiment tests that intuition by augmenting the recovery method set with methods designed specifically for those failures that were being handled inadequately by the current set. Two new methods were added for each of the agents. These methods were designed for failures that were both expensive and frequently occurring: `ip`, `prj` and `ner` for the fireboss. The new methods were based on existing methods and were tailored to particular failures encountered by the Phoenix fireboss planner.

For this experiment, 84 trials of Phoenix were run using the same experiment scenario as in the previous experiments; failure recovery incorporated the selection strategy used in experiment 2 with the two new methods added. As Figure 5 shows, the overall costs of failure recovery for the fireboss decreased in the three failure situations addressed by the two new methods, but the costs

[4] The large increase in the percentage of `prj` failures is due, at least in part, to the introduction of a programming bug that erroneously detected `prj` failures. Unfortunately, the problem was not detected until long after the experiment sequence.

[5] A z-test compares the distribution of two samples for whether the differences between the means, given the standard deviations, might have been due to noise.

increased in all but one of the other failure situations. Because the three targeted failures occurred frequently, this resulted in a reduction of the mean cost over all failure situations (from 2500 to 2355), but the reduction was not statistically significant.

2.3 Summary of Failure Recovery

The methodology followed for implementing failure recovery was to construct a basic set of general recovery methods and a model of expected cost and then run a series of controlled experiments to test ideas about how to improve performance, gradually refining failure recovery to address the requirements of the environment. Based on the results of these experiments, it appears that an untuned set of methods performs reasonably well at least in Phoenix, and the method set can be tuned, within limits, to suit the environment better. The advantage of constructing failure recovery iteratively from general methods is that the system achieves a basic level of performance quickly; the model guides the assessment of how much performance has improved.

In terms of constructing failure recovery for some other environment and planner, the most important result from these experiments is the insensitivity of certain properties of some general methods (the local methods) to aspects of their execution context: cost is independent of position and failure situation, and probability of success for a situation is independent of position of execution. The result is important both for design and methodological considerations. As designers, we know that if such independence assumptions hold in other planners, then we can use a similar ordering strategy and predict performance for these local methods. Furthermore, given the methodology of directing evaluation using a model, we can state and test some of the assumptions of our designs.

However, the interaction effects observed when comparing failure type distributions suggest that even domain-specific, specially designed methods can produce unforeseen interactions later in the plan. For example, two methods were designed specifically to repair three types of failures. Yet, these new methods had unexpected consequences: the type and frequency of failures changed (some increased, some decreased) and the cost of recovering from failures not addressed by the new recovery methods *increased*. The incidence of failures changed because the new recovery methods were preventing some failures, causing others, and permitting some plans to get further along before they failed. Because the recovery methods make mostly syntactic changes to the plan based on the content of the plan library, one suspects that it was not the recovery methods themselves causing the failures, but rather parts of the plan library. From this intuition developed the analysis method described in the next section.

3 Debugging the Planner

Failure recovery is a good approach to dealing with the unexpected source of failure. To improve reliability, *avoiding* failures is preferred over patching up the plans after the failures. One cannot avoid all failures; some failures are difficult to avoid because the environment is capricious or the contingencies are simply too expensive. The failures that seem most amenable to avoidance are those caused by bugs in the planner itself; debugging the planner removes such sources of failure.

Most planners are composed of two parts: control (decision making that guides plan construction) and a knowledge base. Improving or debugging control has received little attention in the literature and is not the focus here either. In general, the approach to debugging the knowledge base has been to debug the plans themselves, using knowledge intensive models of the domain, as the plans are being constructed for particular situations. One of the first approaches to debugging plans was Sussman's HACKER [28] which detects, classifies and repairs bugs in Blocksworld plans, using considerable knowledge about its domain. Simmons' GORDIUS [24] debugs faulty plans by regressing desired effects through a causal dependency structure constructed during plan generation from a causal model of the domain.

One enhancement to the basic approach is to keep track of what changes, made during the debugging of particular plans, were felicitous. Hammond's CHEF [7] backchains from failure to the states that caused it, applying causal rules that describe the effects of actions and map the information to canonical failures and repair strategies. Having repaired a plan satisfactorally, it then remembers the repair and the modified plan so that it can be used during later planning.

The approaches described so far are comprehensive; they all analyze a plan for what went wrong and repair the plan or knowledge base to avoid the failure in the future. Thus, they all assume that the analysis mechanism possesses a complete model of the domain and/or planner. Alternatively, the debugger could solicit outside help or information by asking a human user or by inferring obtained from failure recovery execution. Broverman [3] views failure recovery as an opportunity for knowledge acquisition and requests assistance from the human user of the planner to augment the system's model. Zito-Wolf and Alterman adapt plans when overly general plans fail [31]; these adaptations repair failures and then are used to augment the plan stored in long-term memory with choice points and alternative actions. These approaches may sacrifice completeness or degree of automation to broaden the types of failures addressed.

3.1 Failure Recovery Analysis: An Approach to Debugging the Plan Knowledge Base

Failure Recovery Analysis (FRA) exploits information about *observed* relationships between failures and repairs to help the designer discover how the planner or failure recovery may be causing failures [6]. This approach differs from the previously mentioned approaches in three ways. First, FRA uses execution time information to detect bugs in the planner's knowledge base. It does not debug plans as they are being developed, only after they have been applied in many situations. Second, FRA constructs a statistical model and compares it to a weak model of the planner's behavior. Thus, it integrates several types of models of behavior. Third, FRA involves a human user in the debugging process to judge the merits of possible changes, decide exactly how best to implement them and to compensate for the lack of complete model. As a consequence of

[6]This section expands on the presentation of FRA originally published in the short paper [9] by providing more detail on FRA, evaluating aspects of the procedure and relating it to improving planner reliability.

these differences, FRA is most appropriate when one cannot model behavior off-line (a model is unavailable or flawed) and when the bugs are intermittent and may involve subtle interactions.

FRA is a partially automated procedure, implemented as a set of loosely coupled Lisp functions, that is directed by a human designer. The designer guides every step in the process, deciding where to focus attention and ultimately how to fix the planner to repair the bug. FRA assists by uncovering possible causal relationships in execution data and matching them to structures in plans. The knowledge structures supporting FRA are intended to identify failures caused by bugs in the planner rather than environment induced failures. Planned expansion of the knowledge structures includes adding environment-based explanations.

In FRA, plan debugging proceeds as an iterative redesign process in which the designer tests the design, analyzes its failure recovery behavior, and modifies the planner to remove flaws (bugs and inadequate recovery). This cycle is depicted in Figure 6. The process continues until the designer is satisfied that the planner is reliable.

The designer starts the redesign cycle by running the planner in its environment and collecting execution traces of what failures occurred and how they were repaired. (Recall that this was part of the information collected in the experiments described in Section 2.2.) An *execution trace* is a collection of ordered events. In the current form of FRA, the execution traces include plan failures followed by recovery actions taken by the planner, as in the following trace:

$$F_{prj} \rightarrow R_{sa} \rightarrow F_{ner} \rightarrow R_{sp} \rightarrow F_{ip} \rightarrow R_{rp} \rightarrow F_{prj} \rightarrow R_{sp} \rightarrow F_{ip} \rightarrow R_{rt} \rightarrow F_{ru}$$

where F's are environment states and R's are actions. The subscripts indicate individuals from a set, so R_{sp} means recovery action of type sp.

Although one might wish to examine the contribution of other environment events and planner actions, the current analysis is limited to just failures and recovery actions for two reasons. First, the experiments described in section 2.2 suggested that recovery actions influence later failures and thus these traces, simple as they are, should help explain why. Second, expanding the set of influences significantly complicates the analysis. However, future work will focus on how to expand the set of factors included in the traces while managing the increased complexity of the analysis. Future versions of FRA should exploit the additional information.

Given the execution traces, we can now begin the analysis. The analysis has several parts and involves mapping from the evidence of patterns found in execution data through hypothesizing how the planner could have caused the patterns to arise. Figure 7 shows each of the steps, which will be described in detail in the remainder of this section.

Statistically Search for Significant Patterns The first step in analyzing failure recovery is to check the execution data for patterns that might indicate causal influences on the occurrence of failure. These patterns, called *dependencies*, are statistical models of significant co-occurrences between recovery efforts and failures. Dependencies tell the designer how the recovery actions influence the failures that occur and how one failure influences another.

From this step, we want to discover whether some set of recovery actions and failures occur with unusual frequency before a particular failure. To do so, we compare the frequency of the co-occurrence (combination of particular failures and recovery actions, called a *precursor*, followed by a particular failure) to the rest of the patterns in the execution traces. We count the number of times that the target precursor is followed by the failure, the number of times that the precursor is not followed by the failure, the number of times any other precursor is followed by the failure and the number of times that any other precursor is followed by any other failure. These four counts are arranged in a 2x2 contingency table (as in Table 4), and a G-test[7] is applied to the table to test the significance of the differences in the observed ratios. A G-test on this table yields $G = 42.9, p < .001$, which suggests that the contingency table in Figure 4 is extremely unlikely to have arisen by chance if R_{sp} and F_{ip} are independent.

[7]Roughly speaking, the G-test and its more familiar variant, the Chi-square, test whether two factors appear to be related by comparing ratios of their relative frequencies in the sample.

Dependencies are between any precursor and some failure. This analysis uses three types of precursors: recovery actions (R), failures (F) or pairs of a failure and the recovery action that repaired it (FR). If we test for dependencies in all three types of precursors, we shall find cases of overlap: we observe dependencies involving both a particular action itself (e.g., R_a) and the action in combination with some failure (e.g., $F_f R_a$). To distinguish whether the action itself or the combination best describe the relationship, we can apply a variant of the G-test to determine the contribution of each combination to the effect of the action itself. In this way, a large set of overlapping dependencies can be reduced to a smaller set of mutually exclusive dependencies.

The user looks over the set of dependencies and selects one for attention. The user may select on the basis of the frequency of particular failures, the cost of repairing particular failures, the expense of executing particular recovery methods, or even the skill of the programmers who worked on the code that detected the failures.

Match Selected Dependency to Suggestive Structures in the Plan KB The previous step provides a statistical model of the influence of certain factors on failures. However, we still do not know how these patterns might arise. The next two steps determine what in the plan knowledge base may be involved in a dependency selected for attention and hypothesize how the plan knowledge base may have caused the dependency.

The designer selects one of the dependencies for further attention. The dependency is mapped to actions in plans by associating recovery actions with plan changes and associating failures with the actions that detect them. Then, the plans are searched for structures that involve the actions and that are known to be susceptible to failure. These structures are called *suggestive structures* because, based on experience, they are suggestive of possible bugs; they are language structures used to coordinate actions and can be difficult to program properly. The set of suggestive structures forms part of an experiential model of what tends to break in the plans.

To find suggestive structures, the plan library is searched for all plans that contain some combination of the actions found in the dependency. Currently, FRA includes seven suggestive structures. Two such structures are *sequential ordering* and *shared variable*. *Sequential ordering* is an enabling condition; if one action is guaranteed to come before another, then the first action has the potential to influence the second. It is identified by searching backward in the temporal links of the plan from the second action in the dependency until either the beginning of the plan or the first dependency action is found. A *shared variable* requires that every action that references a particular variable agrees about how it is set and used. If some of the assumptions about the variable's use are implicit or under-specified, the variable might be a source of failures. For example, two actions may assume different units for a variable, leading to a major difference in their actual values. Shared variables are found by searching backward in the plan for all definitions or uses of the variables referenced in the second action in the dependency; if some precursor action references the same variable, then a shared variable structure is found.

Suggestive structures are similar in purpose to Thematic Organization Packages (or TOPs) in Chef [7]; however, while TOPs encompass the diagnosis of the failure, the interactions between the steps and states of the plan and the strategies for repair, suggestive structures identify only the interactions between the steps. Consequently, suggestive structures can be combined to form different explanations of failures and indicate different repairs.

Compare Structures and Dependency to Canonical Bugs and Fixes Finally, the statistical and experiential models are combined to find hypotheses about how common bugs might produce the observed failures. The designer compares the identified suggestive structures for the dependency to a set of possible explanations and modifications, indexed by the suggestive structures. Currently, FRA includes nine explanations. Two of these are:

Implicit Assumptions Two actions make different assumptions about the value of a plan variable to the extent that the later action's requirements for successful execution are violated. This can be fixed by adding new variables to the plan description to make the assumptions explicit or changing the plans so that the incompatible actions are not used in the same plans.

Band-aid Solutions A recovery action may repair the immediate failure, but that failure may be symptomatic of a deeper problem, which leads to subsequent failures. This can be fixed by limiting how the recovery action is used or substituting a new recovery action.

The explanations amount to hypotheses of what might have gone wrong. They do not precisely determine the cause, but rather attempt to provide enough evidence of flaws in the recovery actions or the planner to help the designer decide how to fix the bug. The modification is left to the designer.

3.2 Applying Failure Recovery Analysis to Debugging Phoenix

To demonstrate the utility of FRA, it was applied to find and fix a bug in the Phoenix planner. Additionally, the sensitivity of the statistics to the amount of execution data available was tested. The example demonstration shows that the modifications based on FRA are effective at reducing the incidence of a particular failure. This section will describe empirical results of the effects of sample size and provide advice on selecting the appropriate size.

3.2.1 Running a Cycle of FRA

The last experiment described in section 2.2 provided execution traces for demonstrating FRA. Those traces were searched for dependencies. A dependency that was both significant and included a failure that was expensive to repair was $[R_{sp}F_{ip}]$ (its contingency table is shown in Table 4). The numbers in the contingency table show that F_{ip} follows R_{sp} a significantly higher proportion of times ($52/85 = .61$) than it follows other recovery actions ($240/883 = .27$). So we conclude that F_{ip} depends on R_{sp}.

Next, the dependency is mapped to suggestive portions of the plan knowledge base. R_{sp} substitutes three types of fire prediction actions, and failure F_{ip} is detected by a monitoring action. Together, the two types of actions appear in three different fire fighting plans. Each such plan is checked for suggestive structures that involve the actions in the dependency. All three plans include the same suggestive structures: *sequential ordering* and *shared variable*, which were described previously.

Finally, the suggestive structures are used to explain the dependency. The shared variable can cause a failure if the substituted fire prediction action sets the variable differently than was expected by the envelope action (i.e., "shared assumptions" explanation). The prediction may not be specified well enough to be properly monitored or may violate monitoring assumptions about acceptable progress. Alternatively, the recovery action R_{sp} could lead to F_{ip} if the recovery action is repairing only a symptom of a deeper failure (i.e., "band aid solutions"). The fire may be raging out of control or the available resources may really be inadequate for the task.

Modifications to the Phoenix planner were implemented based on the example analysis and tested by starting another cycle of FRA. The primary modification was one of the two suggested by FRA. The selected modification was intended to fix *implicit assumptions*. This modification has two parts: first, check how the prediction calculation actions set and the monitoring action uses the variable `attack-projection` (which is a model of where the fire should be contained), and second, make explicit the differing assumptions of the three actions that set the variable so that the monitoring action can use the assumptions. The other potential modification, which fixes a *band-aid solution*, was rejected because it requires limiting the application of the suspect recovery action. In this case, the recovery action had been added to improve recovery performance in two expensive failures, removing it would set performance back to previous levels.

Checking the code for the prediction actions and the monitoring action showed that the values set by the three prediction actions varied widely. The monitoring action uses summaries of the resources' capabilities, set by the fire prediction actions, to construct expectations of progress for the plan. The three prediction actions differed in what capabilities were included in those summaries (e.g., rate of building fireline, rate of travel to the fire, startup times for new instructions, and refueling overhead). Because the monitoring action assumed that the summaries reflected *only* the rate of building fireline, the conditions for signaling failures effectively varied among the different prediction actions. To accommodate these differences, the prediction actions were restructured to set separate variables for each of the capabilities; the monitoring action then combines the separate variables to define expected progress. In addition, a few minor bugs were fixed in the calculation of the summaries.

Normally, in the next part of the redesign cycle, the designer would determine whether the modification achieved the desired effect: the observed dependency disappeared and the incidence of failure changed for the better. In this case, the modified planner was tested in 87 trials of the same experiment setup used for the earlier three experiments. Analyzing the resulting traces for dependencies showed that all four of the dependencies involving fire prediction actions and the monitoring action that were previously observed — $[R_{sp}, F_{ip}]$, $[F_{ner} R_{rp}, F_{ip}]$, $[F_{prj} R_{rp}, F_{ip}]$, and $[F_{prj} R_{ra}, F_{ip}]$ — were missing from the modified planner's execution traces. Additionally, the modifications led to a lower incidence of a general failure to calculate fire predictions (F_{prj}); F_{prj} accounted for 20.8% of the failures in the previous experiment and only .3% in this experiment. By repairing a hypothesized cause of failure and related bugs, one would also expect the overall rate of failures to decline. The data showed a decrease in the mean failures per hour from .41 in the previous experiment to .33 in this experiment.

From the standpoint of improving reliability of the Phoenix planner, this example was interesting because it was difficult to debug. Because there is a long time delay between the execution of the recovery action and the failure detection, no one had thought to check the much earlier event as a source of the failures. Also, the fire prediction actions (the actions added by R_{sp}) involve extremely complex code, which is difficult to debug. Consequently, the information provided by FRA was useful in tracking down bugs in the fire prediction code.

3.2.2 Collecting Enough Data?

Many factors will influence the utility of FRA. It cannot hypothesize a bug if it is not found in the plan knowledge base or in the knowledge base of experiences. Of course, neither can it attempt to explain a detrimental interaction or causal dependency if it has not been detected. Due to the statistical and heuristic nature of this procedure, it is impossible to make guarantees about what will be found and what will slip through. With respect to the experiential knowledge (i.e., the suggestive structures and the explanations), one can assume that increasing the number and diversity of the structures and explanations should eventually lead to capturing all but the most rare bugs, assuming that they can be observed as patterns in the execution traces. With respect to the statistics, one expects the patterns detected will actually be due to noise with probability 0.05 and that the likelihood of detecting a pattern will depend on the amount of data and the subtlety of the pattern. While I have not found a formal characterization of the relationship between the data and the pattern, its ramifications have been explored empirically.

Much of the effort required to detect dependencies is expended collecting execution traces. Fortunately, the G-test (the statistical basis for detecting dependencies) is additive, which means that G values for subsets of the sample can be added together to get a G value for the superset. If the ratios (the relative values of the top and bottom rows) remain the same but the total number of counts in the contingency table double, then the G value for the contingency table doubles as well. Consequently, given execution traces with few patterns, the G-test can find strong dependencies, but given more patterns, it will also find subtle dependencies. To select the sample size, one should conduct a pilot experiment to determine how much time is required to collect the data and what types of dependencies can be detected. If you can detect an interesting set of dependencies (dependencies that given more knowledge about the workings of the program indicate previously unsuspected interactions), then use that sample size in subsequent trials; otherwise, collect more samples.

The effect of simply increasing the amount of data in the contingency table is clear; the effect of the subtlety of the pattern (i.e., how different is it from the rest) is more complicated, given the mathematics underlying the G-test. One empirical way to assess the effect is to ask whether, in practice, dependencies tend to be subtle enough that if one or two fewer examples were found it would no longer be detected. The effect was tested by "tweaking" the frequency counts found in the data to see how many of the dependencies would not have been detected if the counts in row one in the contingency table varied by a small amount. For four sets of execution traces (those from the three experiments plus the set from the fourth experiment described in the last section), tweaking the values resulted in a loss of about 35% of the dependencies found previously. In other words, for the data from Phoenix, dependency detection is sensitive to small differences in the content of the execution traces. Most of the dependencies that were vulnerable to the tweaking were based on execution traces that included few instances of the precursor/failure pattern, 52% of the dependencies that disappeared were based on contingency tables in which one of the counts in the first row was less than five.

The implication of the "disappearing dependencies" is that rare or subtle patterns are especially sensitive to the amount of data and so should be viewed skeptically. Interpreting dependencies requires weighing false positives against misses. If one is trying to identify dependencies between precursors that occur rarely or failures that occur rarely, then additional effort should be expended to get enough execution traces to ensure that the dependency is not due to random chance.

3.3 Summary

The purpose of Failure Recovery Analysis is to identify possible cases in which plans may influence, exacerbate or cause failure. There are two reasons why analyzing failure recovery is a good way to determine why the planner fails. First, failure recovery influences which failures occur. Minor changes in the design of failure recovery produce significant changes in the number and types of failures. Second, failure recovery uses plans in ways not explicitly foreseen by its designers, but not forbidden or prevented by them either. Failure recovery repairs plans by adding or replacing portions of them. As a result, the plan may include plan fragments that are juxtaposed in orders and context not envisioned by the designers.

The current results are preliminary. FRA has been tested on a small number of examples from a single planner. FRA is based on a few assumptions about the planner and its environment: First, it should be possible to collect lots of execution information. Second, the planner should either include a plan library or have the capacity to store representative plan expansions. Case-based planners include a structure similar to the Phoenix plan library. For many classical or deliberative planners, collecting representative plans should be possible. Third, given some information about how failures are detected, it should be possible to relate failures to specific parts of the plan in order to localize the source of the failure. These assumptions differ from those of other approaches to debugging plans. As a consequence, FRA is not the method of choice when a complete model of the planner and environment is available, but is currently the only method proposed for cases in which one knows there are bugs, but is not sure where.

The most interesting feature of FRA is the integration of knowledge based reasoning with statistical reasoning to overcome some of the limitations of current comprehensive systems. Statistics provides useful summaries of performance and data, but requires interpretation. The knowledge based techniques in the later steps of FRA provide partial interpretation of statistical results.

4 Future Work

The failure recovery methodology and FRA procedure promise to improve planner reliability and to expedite the development of plan knowledge bases for new environments by assisting designers in tuning failure recovery and debugging knowledge bases. However, at its present stage of development, this research is limited in several ways: the procedure is only partially automated and is implemented as a loosely organized set of Lisp functions, execution traces contain only failures and recovery actions, dependencies include only temporally adjacent precursors and failures, the set of recovery actions is still small, and the procedure has been tested only on the Phoenix planner.

Future research will address these limitations by "closing the loop" of gathering and analyzing execution data and by generalizing to a broader range of bugs and to another planner. Closing the loop refers integrating all of the tools necessary to support complete testing, analysis and repair of a planner during its development process. The designer will still direct the process, but will do so by selecting from sets of pre-defined experiment scenarios and scripts for performing stereotypical analysis of the execution data. Generalization refers to expanding the set of recovery methods to be applied and the set of bugs that can be identified and applying the procedure to another planner. The set of suggestive structures and explanations needs to be enhanced, especially when FRA is applied to another planner and environment. Additionally, the analysis of dependencies will be expanded to include other features over longer periods of time.

5 Conclusion

Certain software systems, so called *ambitious systems* [5], are prone to failure. These include systems being developed for novel or unfamiliar tasks, systems in unpredictable environments, or systems with organizational complexity. Failure is a consequence of complexity in the environment or the software and the fact that our facility in constructing complex systems has surpassed our ability to understand their behavior. Consequently, the software most likely to fail is that which is also hardest to understand and to debug.

The goal of the described research is to reduce the impact and likelihood of failures that result from a lack of understanding about how an AI planner will perform. Failure recovery provides a safety net for catching failures that cannot be avoided easily; the incremental methodology expedites building failure recovery suited to a particular planner and its environment. Failure Recovery Analysis should help programmers to debug planners under development because it requires only a weak model of how they perform and relies on statistical analyses of the execution of the planner. Together, these approaches have been demonstrated to improve the reliability and the performance of the Phoenix planner. Future work should demonstrate the feasibility of applying the failure recovery design methodology and FRA to other planners as well.

References

[1] Jose A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 83–88, Minneapolis, Minnesota, 1988. American Association for Artificial Intelligence.

[2] Rodney A. Brooks. Symbolic error analysis and robot planning. *International Journal of Robotics Research*, 1(4):29–68, Winter 1982.

[3] Carol A. Broverman. *Constructive Interpretation of Human-Generated Exceptions During Plan Execution.* PhD thesis, COINS Dept, University of Massachusetts, Amherst, MA, February 1991.

[4] Paul R. Cohen, Michael Greenberg, David M. Hart, and Adele E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3), Fall 1989.

[5] Fernando J. Corbato. On building systems that will fail. *Communications of the ACM*, 34(9):72–81, September 1991.

[6] Maria Gini. Automatic error detection and recovery. Computer Science Dept. 88-48, University of Minnesota, Minneapolis, MN, June 1988.

[7] Kristian John Hammond. *Case-Based Planning: An Integrated Theory of Planning, Learning and Memory.* PhD thesis, Dept. of Computer Science, Yale University, New Haven, CT, October 1986.

[8] Steve Hanks and R. James Firby. Issues and architectures for planning and execution. In Katia P. Sycara, editor, *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 71–76, Palo Alto, Ca., November 1990. Morgan Kaufmann Publishers, Inc.

[9] Adele E. Howe. Analyzing failure recovery to improve planner design. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 387–393, July 1992.

[10] Adele E. Howe. *Accepting the Inevitable: The Role of Failure Recovery in the Design of Planners.* PhD thesis, University of Massachusetts, Department of Computer Science, Amherst, MA, February 1993.

[11] Adele E. Howe and Paul R. Cohen. Failure recovery: A model and experiments. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 801–808, Anaheim, CA, July 1991.

[12] Subbarao Kambhampati and James A. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence Journal*, 55(2-3), 1992.

[13] Victor R. Lesser. A retrospective view of FA/C distributed problem solving. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6):13471362, November/December 1991.

[14] Nancy G. Leveson. Software safety: Why, what, and how. *Computing Surveys*, 18(2):125–163, June 1986.

[15] D.M. Lyons, R. Vijaykumar, and S.T. Venkataraman. A representation for error detection and recovery in robot plans. In *Proceedings of SPIE Symposium on Intelligent Control and Adaptive Systems*, pages 14–25, Philadelphia, November 1989.

[16] David P. Miller. Execution monitoring for a mobile robot system. In *Proceedings of SPIE Symposium on Intelligent Control and Adaptive Systems*, pages 36–43, Philadelphia, PA, November 1989.

[17] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings*

of the Ninth National Conference on Artificial Intelligence, pages 17–24, Anaheim, CA, 1991. American Association for Artificial Intelligence.

[18] Leora Morgenstern. Replanning. In *Proceedings of the DARPA Knowledge-Based Planning Workshop*, pages 5–1 – 5–10, Austin, TX, December 1987.

[19] N. Hari Narayanan and N. Viswanadham. A methodology for knowledge acquisition and reasoning in failure analysis of systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-17(2):274–288, March/April 1987.

[20] S.Y. Nof, O.Z. Maimon, and R. G. Wilhelm. Experiments for planning error-recovery program in robotic work. In *Proceedings of the 1987 ASME International Computers in Engineering Comference*, pages 253–262, NY,NY, August 1987.

[21] Christopher Owens. Representing abstract plan failures. In *Proceedings of the Twelfth Cognitive Science Conference*, pages 277–284, Boston, MA, 1990. Cognitive Science Society.

[22] Harry J. Porta. Dynamic replanning. In *Proceedings of ROBEXS 86: Second Annual Workshop on Robotics and Expert Systems*, pages 109–115, June 1986.

[23] Reid Simmons. Monitoring and error recovery for autonomous walking. In *Proc. IEEE International Workshop on Intelligent Robots and Systems*, pages 1407–1412, July 1992.

[24] Reid G. Simmons. A theory of debugging plans and interpretations. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 94–99, Minneapolis, Minnesota, 1988. American Association for Artificial Intelligence.

[25] Herbert A. Simon and Joseph B. Kadane. Optimal problem-solving search: All-or-none solutions. *Artificial Intelligence Journal*, 6:235–247, 1975.

[26] Stephen F. Smith, Peng Si Ow, Nicola Muscettola, Jean-Yves Potvin, and Dirk C. Matthys. OPIS: an integrated framework for generating and revising factory schedules. In Katia P.

Sycara, editor, *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 497–507. Morgan Kaufmann Publishers, Inc, November 1990.

[27] Sankaran Srinivas. *Error Recovery in Robot Systems*. PhD thesis, California Institute of Technology, Pasadena, CA, 1977.

[28] Gerald A. Sussman. A computational model of skill acquisition. Technical Report Memo no. AI-TR-297, MIT AI Lab, 1973.

[29] Katia Sycara. Using case-based reasoning for plan adaptation and repair. In *Proceedings of a Workshop on Case-Based Reasoning*, pages 425–434. Morgan Kaufmann Publishers, Inc., 1988.

[30] David E. Wilkins. Recovering from execution errors in SIPE. Technical Report 346, Artificial Intelligence Center, Computer Science and Technology Center, SRI International, 1985.

[31] Roland Zito-Wolf and Richard Alterman. Ad-hoc, fail-safe plan learning. In *Proceedings of the Twelfth Cognitive Science Conference*, pages 908–913, Boston, MA, July 24-28 1990.

List of Figures

List of Tables

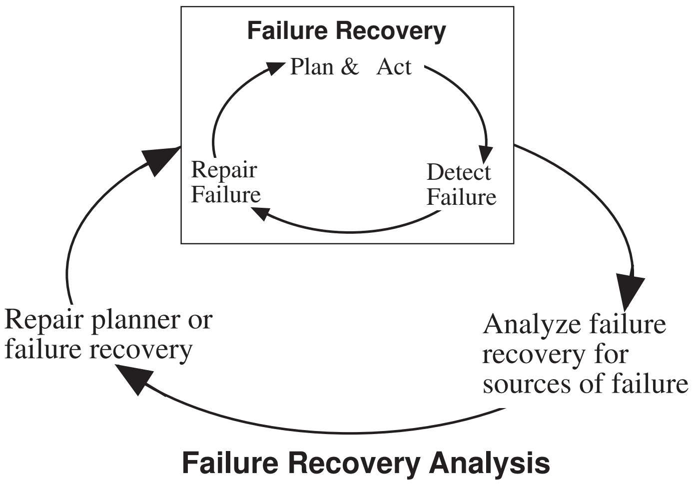

Failure Recovery
Plan & Act
Repair
Failure
Detect
Failure
Repair planner or
failure recovery
Analyze failure
recovery for
sources of failure
Failure Recovery Analysis

Figure 1:

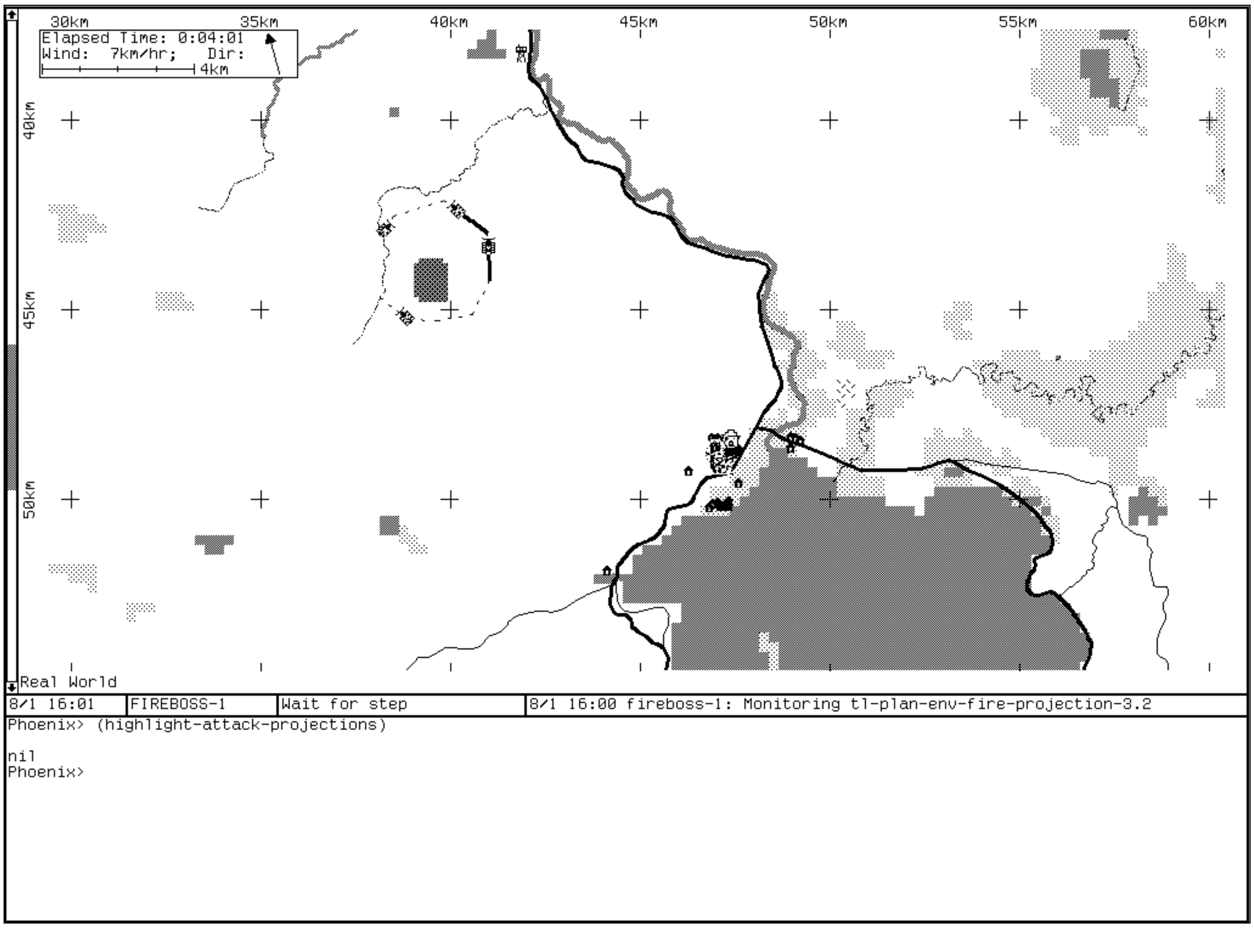

30km
35km
40km
45km
50km
55km
60km
Elapsed Time: 0:04:01
Wind: 7km/hr; Dir:
4km
40km
45km
50km
Real World
8/1 16:01
FIREBOSS-1
Wait for step
8/1 16:00 fireboss-1: Monitoring tl-plan-env-fire-projection-3.2
Phoenix> (highlight-attack-projections)
nil
Phoenix>

Figure 2:

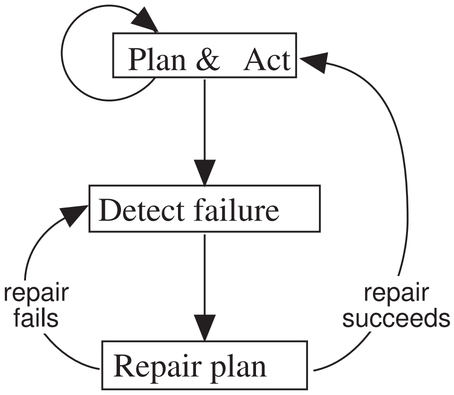

Plan & Act
Detect failure
Repair plan
repair
fails
repair
succeeds

Figure 3:

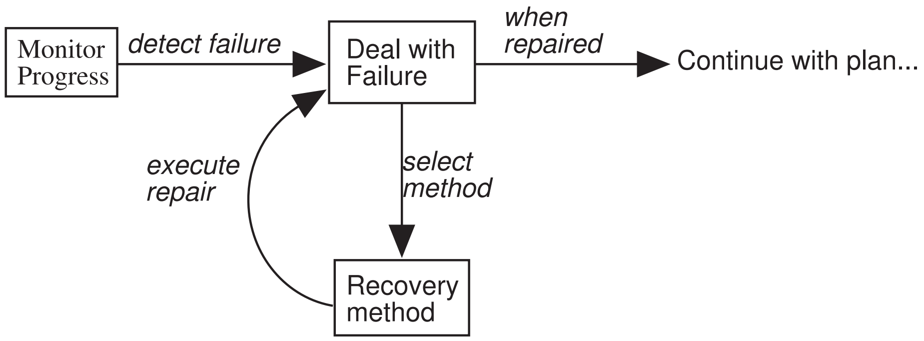
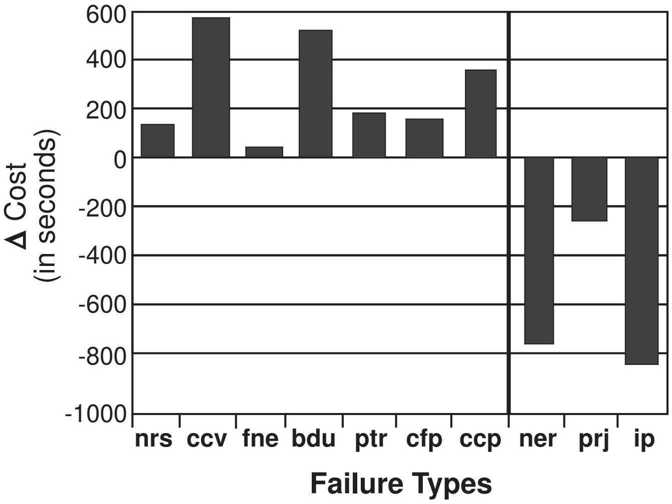

Monitor Progress
detect failure
Deal with Failure
when repaired
Continue with plan...
select method
execute repair
Recovery method

Figure 4:



600
400
200
0
-200
-400
-600
-800
-1000
Δ Cost
(in seconds)
nrs
ccv
fne
bdu
ptr
cfp
ccp
ner
prj
ip
Failure Types

Figure 5:

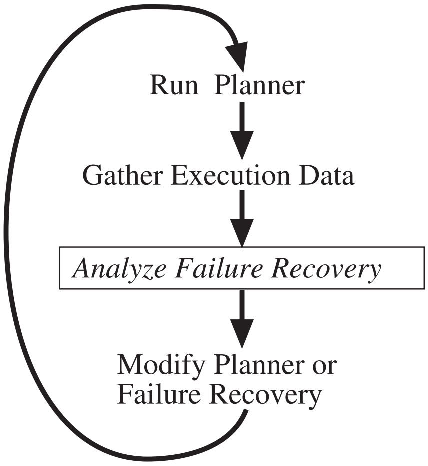

Run Planner
Gather Execution Data
Analyze Failure Recovery
Modify Planner or
Failure Recovery

Figure 6:

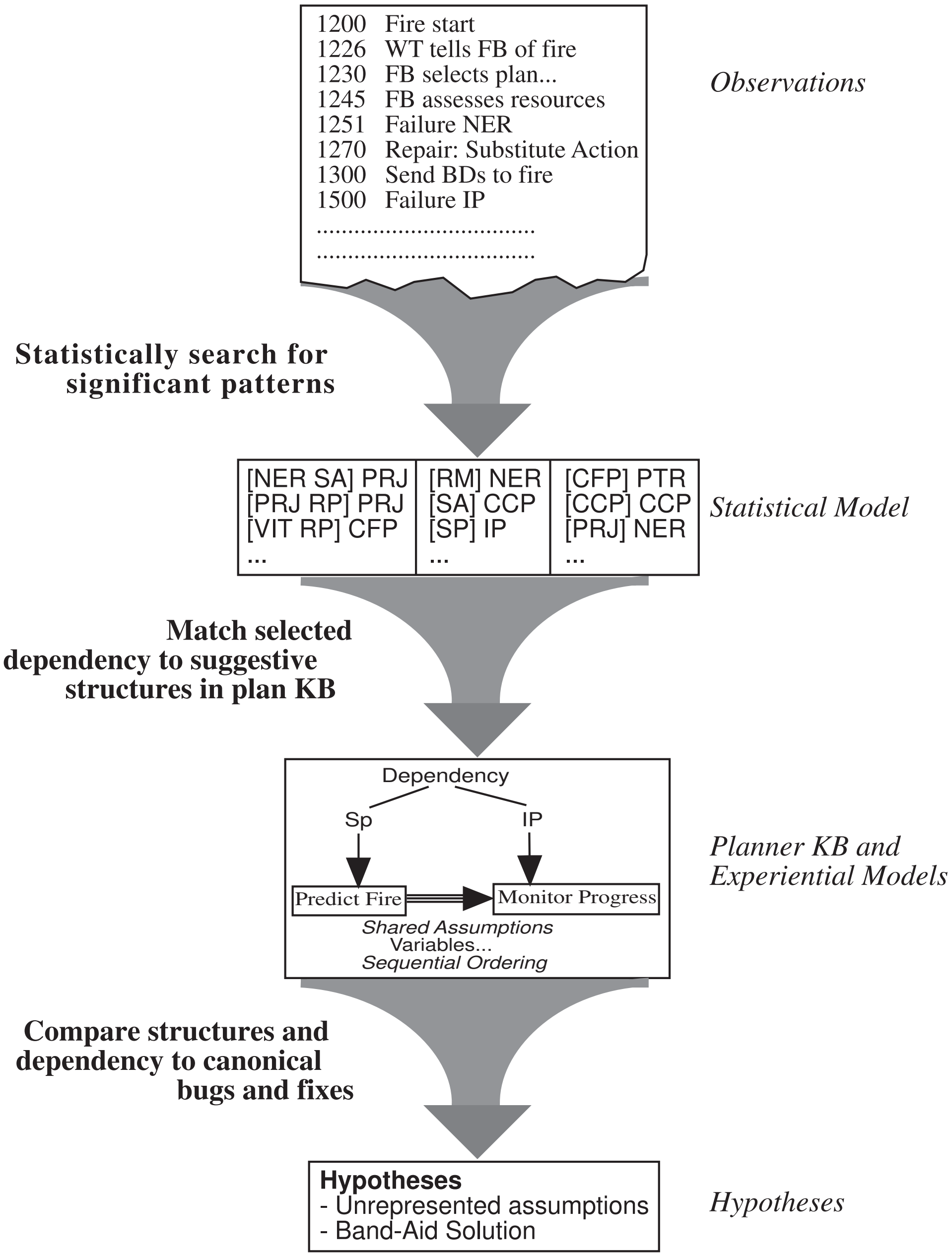

1200 Fire start
1226 WT tells FB of fire
1230 FB selects plan...
1245 FB assesses resources
1251 Failure NER
1270 Repair: Substitute Action
1300 Send BDs to fire
1500 Failure IP
Observations
Statistically search for significant patterns
[NER SA] PRJ
[PRJ RP] PRJ
[VIT RP] CFP
...
[RM] NER
[SA] CCP
[SP] IP
...
[CFP] PTR
[CCP] CCP
[PRJ] NER
...
Statistical Model
Match selected dependency to suggestive structures in plan KB
Dependency
Sp
IP
Predict Fire
Monitor Progress
Shared Assumptions
Variables...
Sequential Ordering
Planner KB and Experiential Models
Compare structures and dependency to canonical bugs and fixes
Hypotheses
- Unrepresented assumptions
- Band-Aid Solution
Hypotheses

Figure 7:

Agent	Failure Types
Fireboss	(CCP) Can't Calculate Path (CCV) Can't Calculate Variable (CFP) Can't Find Plan (FNE) Fire Not Encircled when it should be (IP) Insufficient Progress to contain the fire (NER) Not Enough Resources to contain the fire (NRS) No Remaining fireline Segments to build (PRJ) can't calculate PRoJection of fire (PTR) Can't Calculate Path to Road (RU) Resource Unavailable
Bulldozer	(CCP) Can't Calculate Path (DOP) Deadly Object in Path (NVV) No Variable Value (OOF) Out Of Fuel (PM) Position Mismatch

Table 1:

Method	Description
WATA	Wait and try the failed action again.
RV	Re-calculate one variable used in the failed action.
RA	Re-calculate all variables used in the failed action.
SA	Substitute a similar plan step for the failed action.
RP	Abort the current plan and re-plan at the parent level (i.e., the level in the plan immediately above this one).
RT	Abort current plan and re-plan at the top level (i.e., redo the entire plan).

Table 2:

	`ccp`	`ccv`	`cfp`	`fne`	`ip`	`ner`	`nrs`	`prj`	`ptr`	`ru`
Exp. 1 Costs (Random Strategy)	1932	5710	3030	1163	3883	3395	2904	2165	1373	3254
Exp. 1 $P(S_i)$	.268	.006	.118	.002	.213	.159	.002	.080	.075	.077
Exp. 2 Costs (Selection Strategy)	1056	1618	2707	474	3977	2838	414	2518	1041	2514
Exp. 2 $P(S_i)$	.212	.007	.084	.002	.203	.163	.004	.223	.043	.058

Table 3:

	F_{ip}	$F_{\overline{ip}}$
R_{sp}	52	33
$R_{\overline{sp}}$	240	643

Table 4: Contingency Table for $[R_{sp}, F_{ip}]$

Abstract

As planning technology improves, Artificial Intelligence planners are being embedded in increasingly complicated environments: ones that are particularly challenging even for human experts. Consequently, failure is becoming both increasingly likely for these systems (due to the difficult and dynamic nature of the new environments) and increasingly important to address (due to the systems' potential use on real world applications). This paper describes the development of a failure recovery component for a planner in a complex simulated environment and a procedure (called Failure Recovery Analysis) for assisting programmers in debugging that planner. The failure recovery design is iteratively enhanced and evaluated in a series of experiments. Failure Recovery Analysis is described and demonstrated on an example from the Phoenix planner. The primary advantage of these approaches over existing approaches is that they are based on only a weak model of the planner and its environment, which makes them most suitable when the planner is being developed. By integrating them, failure recovery and Failure Recovery Analysis improve the reliability of the planner by repairing failures during execution and identifying failures due to bugs in the planner and failure recovery itself.

Index terms: Artificial Intelligence, Planning, Failure Recovery, Reliability, Debugging

Index terms: Artificial Intelligence, Planning, Failure Recovery, Reliability, Debugging

Footnotes

Thanks This research was supported by a DARPA-AFOSR contract F49620-89-C-00113, the National Science Foundation under an Issues in Real-Time Computing grant, CDA-8922572, and a grant from the Office of Naval Research under the University Research Initiative N00014-86-K-0764. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. This research was conducted as part of my PhD thesis research at the University of Massachusetts. I would like to thank my thesis advisor, Paul Cohen, and my thesis committee for their advice, guidance and supervision of this research. I also wish to thank the anonymous reviewers for their comments, which helped clarify much of the presentation in the paper.

1. The basic Phoenix planner as described in [4] included no failure recovery component; the research described in this paper augments the original system.

2. The cost model and experiments testing improvements were described originally, but in less detail in [11].

3. Because the bulldozers do far less planning than the fireboss, the bulldozer results tend to be similar, but less interesting. Consequently, only the results of the fireboss are reported.

4. The large increase in the percentage of `prj` failures is due, at least in part, to the introduction of a programming bug that erroneously detected `prj` failures. Unfortunately, the problem was not detected until long after the experiment sequence.

5. A z-test compares the distribution of two samples for whether the differences between the means, given the standard deviations, might have been due to noise.

6. This section expands on the presentation of FRA originally published in the short paper [9] by providing more detail on FRA, evaluating aspects of the procedure and relating it to

improving planner reliability.

7. Roughly speaking, the G-test and its more familiar variant, the Chi-square, test whether two factors appear to be related by comparing ratios of their relative frequencies in the sample.

Address:

Adele E. Howe

Computer Science Department

Colorado State University

Fort Collins, CO 80523

email: howe@cs.colostate.edu