Abstract
Plans fail for many reasons. During planner development, failure can often be traced to actions of the planner itself. Failure recovery analysis is a procedure for analyzing execution traces of failure recovery to discover how the planner's actions may be causing failures. The four step procedure involves statistically analyzing execution data for dependencies between actions and failures, mapping those dependencies to plan structures, explaining how the structures might produce the observed dependencies, and recommending modifications. The procedure is demonstrated by applying it to explain how a particular recovery action may lead to a particular failure in the Phoenix planner. The planner is modified based on the recommendations of the analysis, and the modifications are shown to improve the planner's performance by removing a source of failure and so reducing the overall incidence of failure.

Introduction
Plans fail for perfectly good reasons: the environment changes unpredictably, sensors return flaky data [Lee et al., 1983], and effectors do not work as expected [Hayes, 1975]. During planner development, plans fail for not so good reasons: the effects of actions are not adequately specified [Atkinson et al., 1986], apparently unrelated actions interact [Sussman, 1973], and the domain model is incomplete and incorrect [Chien and Weissman, 1975]. planners should not cause their own failures, but figuring out what went wrong and preventing it later is not easy. Failures tell us what went wrong, but not why. The failure repair alleviates the immediate problem, but does not tell us how to fix the cause or even whether the repair itself might not cause failures later. This paper presents a procedure, called failure recovery analysis (FRA), for analyzing execution traces of failure recovery to discover when and how the planner's actions may be causing failures [Howe, 1992].

Most approaches to debugging planners are knowledge intensive. Sussman's HACKER [1973] detects, classifies and repairs bugs in blocks world plans, but it requires considerable knowledge about its domain. Hammond's CHEF [1987] backchains from failure to the states that caused it, applying causal rules that describe the effects of actions. Simmons's GORDIUS [1988] debugs faulty plans by regressing desired effects through a causal dependency structure constructed during plan generation from a causal model of the domain. Kambhampati's approach [1990] requires the planner to generate validation structures, explanations of correctness for the plan. His theory of plan modification compares the validation structure to the planning situation, detects inconsistencies, and uses the validation structure to guide the repair of the plan.

These approaches assume that the planner or debugger has a strong model of the domain. The approach presented in this paper, FRA, requires little knowledge to identify contributors to failures and only a weak model to explain how the planner might have caused failures. Complementary to the more knowledge intensive approaches, this approach is most appropriate when a rich domain model is not available or when the existing model might be incorrect or buggy, as when the system is under development.

The consequence of relying on a weak model is that while FRA can detect possible causes of the failure, it cannot identify the cause precisely enough to implement a repair. Debugging a planner requires judgment about what would be the best modification and whether the failure is worth avoiding at all. In repairing one failure, others might be introduced. In FRA, the designer decides how best to repair the failures.

The Planner and its Environment
Previous experiments and analyses of failure recovery in the Phoenix system (introduced below) showed that changing how the planner recovers from failures
changed the type and frequency of failures encountered [Howe and Cohen, 1991]. In these experiments, seemingly minor changes to the design of Phoenix's failure recovery component, such as adding two new failure recovery actions with limited applicability, had unexpected consequences. Failure recovery analysis of these experiments should explain why a well-justified modification to the planner produced such havoc.

The Phoenix system is a simulator of forest fire fighting in Yellowstone National Park and an agent architecture [Cohen et al., 1989]. A single agent, the fire boss, coordinates the efforts of field agents who build fireline to contain the spread of the fire. Its spread is influenced by weather and terrain, but even when these factors remain constant, the fire's spread is unpredictable. Plan failures are a natural result of this unpredictability of the environment, but they may also result from flaws in Phoenix's plans.

A plan failure is detected when a plan cannot execute to completion. Failures may be detected during plan generation or execution and are classified into 11 domain-specific types. For example, a violation—insufficient-time failure (abbreviated vit) is detected through execution monitoring when a plan will take longer to complete than it has been allotted.

To repair a failure, the planner applies one of a set of actions—usually six, but in one version of the system, eight. Most of the actions can be applied to any failure, but the scope and nature of their repairs varies. For example, replan-parent (abbreviated rp) is applicable to any failure and recomputes the plan from the last major decision point; while substitute-projection-action (abbreviated sp) repairs only two types of failures by replacing the failed action with another.

## Failure Recovery Analysis

Failure recovery analysis involves four steps. First, execution traces are searched for statistically significant dependencies between recovery efforts and subsequent failures. Second, dependencies are mapped to structures in the planner's knowledge base known to be susceptible to failure. Third, the interactions and vulnerable plan structures are used to generate explanations of why the failures occur. Finally, the explanations serve to separate occasional, acceptable failures from chronic, unacceptable failures, and recommend redesigns of the planner and recovery component. The first step is fully automated, and the second step is partially automated. The entire process will be illustrated with an example of how one of the recovery actions, sp, can influence vit failures.

### Step 1: Isolating Dependencies

The first step in FRA is to search failure recovery data for statistical dependencies between recovery efforts and failures. One failure, $F_p$, is dependent on another, $F_f$, if $F_p$ is observed to occur more often following $F_f$ than after any other failure. In general, the precursor, $F_f$, can be replaced with anything observable during execution—recovery actions, planning actions, events in the environment or some combination. For example, if $F_f$ denotes a failure and $R_i$ denotes the recovery action that repaired $F_f$, then $F_f - R_i - F_g$ is an execution trace leading to failure $F_g$, and $F_f R_i$ is the precursor. For any precursor, the question is the same: Does a failure depend on some action or event that preceded it?

Dependencies can be isolated by statistically analyzing execution traces. Execution traces can be viewed as transitions between failure types and actions, and then these transitions can be analyzed for dependencies. The statistical analysis is a two-step process: Combinations of failures and actions are first tested for whether they are more or less likely to be followed by each of the possible failures. Then the significant combinations are compared to remove overlapping combinations.

To determine whether failures are more or less likely after particular precursors, contingency tables are constructed for each precursor $P_a$ and each failure $F_f$ by counting: 1) instances of $F_f$ that follow instances of $P_a$, 2) instances of $F_f$ that follow instances of all precursors other than $P_a$ (abbreviated $P_{\neg a}$), 3) failures other than $F_f$ (abbreviated $F_{\neg f}$) that follow $P_a$ and 4) failures $F_{\neg f}$ that follow $P_{\neg a}$. These four frequencies are arranged in a 2x2 contingency table like the one in figure 1: the precursor in this table is $F_{\neg a} R_{sp}$, a failure and a recovery action. $F_{\neg a}$ is the failure not-enough-resources, which is detected when the fire cannot be fought with the available resources. $R_{sp}$ is the recovery action sp.

The targeted failure is $F_{vit}$. In this case we see a strong dependence or association between the precursor, $F_{\neg a} R_{sp}$, and the failure, $F_{vit}$: 42 cases of $F_{vit}$ follow $F_{\neg a} R_{sp}$ and only 21 failures other than $F_{\neg a}$ follow $F_{\neg a} R_{sp}$. But while $F_{\neg a} R_{sp}$ leads most frequently to failure $F_{vit}$, precursors other than $F_{\neg a} R_{sp}$ lead to $F_{vit}$ relatively infrequently (250 instances in 905). A G test on this table will detect a dependency between the failure and its precursor; in this case, $G = 41.4, p < .001$, which means that the contingency table in figure 1 is extremely unlikely to have arisen by chance if $F_{\neg a} R_{sp}$ and $F_{vit}$ are independent. So we conclude that $F_{vit}$ depends on $F_{\neg a} R_{sp}$ (abbreviated $[F_{\neg a} R_{sp}, F_{vit}]$).

Failure recovery analysis requires contingency tables for three types of precursors: failures ($F_f$), recovery actions ($R_i$) and pairs of a failure and the recovery action that repaired it ($F_f R_i$). Because these precursors are strongly related (recovery actions repair failures),

<table>
<thead>
<tr>
<th>$F_{neg} R_{sp}$</th>
<th>$F_{neg} R_{tp}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>250</td>
<td>655</td>
</tr>
</tbody>
</table>

Table 1: Contingency Table for $[F_{\neg a} R_{sp}, F_{vit}]$
any dependency could be due to \( F_j \) itself, \( R_e \) itself, or \( F_j R_e \) together. A statistical technique based on the G-test differentiates the three hypotheses by comparing the sum of the effects due to the pairs (e.g., \( F_j R_e \) for all possible \( R_e \)'s) to the effect due just to the grouped effect (e.g., \( F_j \)). The intuition behind the test is that if the pairs do not add much information about the effect then they can be disregarded; conversely, if the grouped effect, \( F_j \) or \( R_e \), masks differences between the pairs, then the grouped effect should be disregarded as misleading. For example, by comparing the example dependency, \([F_{ner} R_{sp}, F_{sit}]\), and related pairs to the grouped effect, \([R_{sp}, F_{sit}]\) using a variant of the G-test, we find that \([F_{ner} R_{sp}, F_{sit}]\) adds little information over knowing \([R_{sp}, F_{sit}]\), so \([F_{ner} R_{sp}, F_{sit}]\) is disregarded.

Step 2: Mapping to Suggestive Plan Structures

Step 1 tells us whether a failure depends on what precedes it, but not how the dependency relates to the planner's actions. The next step is to determine how the constituents of the dependencies (the recovery actions and failures) interact with each other in plans. This step has two parts: associating each dependency with actions in plans and finding structures in the plans that might lead to failures.

Associating Dependencies with Plans: The constituents of a dependency are associated with plan actions. The association is motivated by the following two relationships: Failures are detected by plan actions, and recovery actions transform failed plans by adding or replacing plan actions. So each dependency can be represented as sets of actions specifying all the ways the failures in the dependency are detected, and all the ways the recovery action in the dependency adds actions to plans. For example, to associate the dependency identified in step 1, \([R_{sp}, F_{sit}]\), with plan actions, we determine what actions are added by \( R_{sp} \) and what actions detect \( F_{sit} \), as displayed in figure 1. \( R_{sp} \) transforms a failed indirect attack plan (abbreviated \( P_{ia} \)) into a repaired plan \( P_{ia}^{r} \) by substituting a different type of fireline projection calculation action for the failed one. Fireline projections are the planner's blueprint for the placement of fireline to contain a forest fire; the Phoenix plan library includes three different actions for calculating projections: multiple-fixed-shell \((A_{p-mfs})\), tight-shell \((A_{p-ts})\), and model-based \((A_{p-mb})\). \( R_{sp} \) replaces one of these with another; so we know that \( R_{sp} \) adds one of these three projection actions. Failure \( F_{sit} \) is detected when plan monitoring indicates that progress against the fire has been insufficient and not enough time remains to complete the plan. \( F_{sit} \) is detected by an envelope action (a structure for comparing expected to actual progress [Hart et al., 1980]) called indirect-attack-envelope \((A_{env})\).

Identifying Structures that Lead to Failure: The plan library is searched for plans that govern the interaction between the actions of the dependency. These structures, called suggestive structures, are idioms in the plans that suggest causes of failure or that tend to be vulnerable to failure; they coordinate actions within plans and describe shared commitments to a course of action or shared expectations about the world. Suggestive structures can improve plan efficiency, but make the related actions sensitive to changes in the environment or intolerant of variations in the plan. Designers make trade-offs by using such structures; they intend that the efficiency gained from them outweighs the cost of occasional failure.

One example of a suggestive structure is a shared variable in a plan: as long as every action that uses the variable agrees about how it is set and used, shared variables can be invaluable for coordinating actions. But if some of these assumptions are implicit or under-specified, the variable might be a source of failures; for example, one action might assume that the variable’s units are minutes and another might assume seconds. Some suggestive structures from the Phoenix plan language are:

- **Shared Variables** One action sets some variable and another uses it.
- **Shared Resources** Two actions allocate and use the same resource.
- **Assuming Stability in the Environment** One action senses the environment and passes the result onto another or two actions share assumptions about the state of the environment.
- **Sequential Actions** One is guaranteed to follow another in some plan.
- **Iteration Constructs** Multiple actions are added to the plan by the same decision action. For example, Phoenix supports a rescheduling construct that duplicates actions until some condition is met.

To find suggestive structures, the plan library is searched for all plans that contain one of the possible
combinations of the actions in the dependency. Each such plan is checked for suggestive structures involving the dependency actions. In the example, the projection calculation actions \( A_{p-mfs}, A_{p-tes} \) and the envelope action \( A_{ens} \) appear together in three different indirect attack plans. All three indirect attack plans include the same suggestive structures: shared variable and sequential ordering. Figure 1 shows how the projection calculation actions and the envelope action are related in the indirect attack plans. All projection calculation actions set the variable attack-projection which is used by the envelope action. The envelope action always follows the projection calculation action in the indirect attack plans.

**Step 3: Explaining Dependencies**

Steps 1 and 2 determine what actions of the planner’s might lead to the observed failures. Step 3 completes the story of how the planner causes failures by constructing explanations of how the suggestive structures might have produced the observed dependencies. For example, the suggestive structure, *shared variables*, can cause failures when two actions in a plan use the variable differently, each making its own implicit assumptions about the value of the shared variable. Combinations of suggestive structures lead to many explanations; the two suggestive structures found in Step 2 for the dependency \([R_{sp}, F_{wit}]\) underlie two different explanations:

- **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.
- **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.

The explanations amount to sketches 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 motivate a redesign.

**Step 4: Recommending Redesigns**

Step 4 determines whether and how to repair the causes of failure. Each failure explanation translates directly to a set of possible plan modifications. The modifications are based on experience with repairing flaws of the types described by the explanations. In the example, the \([R_{sp}, F_{wit}]\) dependency is explained as due to two possible mechanisms: implicit assumptions and band-aid solutions. Each indicates a different problem with the plan library and each leads to a different modification:

- **Implicit Assumptions** Add new variables to the plan description to make the assumptions explicit or change the plans so that the incompatible actions are not used in the same plans.
- **Band-aid Solutions** Limit the application of the suspect recovery action or add new recovery actions to repair the failure.

The recommendation is not intended to be implemented by the system itself. Modifying a planner requires judgment about what would be the best modification and whether the failure is worth avoiding at all. In repairing one interaction effect, others might be introduced.

**Utility of Failure Recovery Analysis**

Failure recovery analysis is worthwhile only if it can tell us something about our planners that we didn’t already know, and if the effort required to perform the analysis is commensurate with the information gained. While the analysis of the Phoenix planner is ongoing, so far the results are promising. As this section describes, the recommendations of the example analysis in this paper have been tested in Phoenix and the modification has been shown to improve the planner’s performance.

**Diagnosing Failures in Phoenix**

The example analysis recommended two modifications. One required 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. The other modification, which was adopted, was to check how the projection calculations set and the envelope action uses the variable attack-projection and make explicit the differing assumptions of the three projection actions so that later actions could reason about the assumptions. The three actions differ in how they search for projections and in how they assess the resources’ capabilities. The envelope action uses summaries of the resources’ capabilities to construct expectations of progress for the plan. By examining the code, it became obvious that the summaries set by the three projection actions differed not only in how they were estimated but also in what capabilities were included (e.g., rate of building fireline, rate of travel to the fire, startup times for new instructions, and refueling overhead). Because the envelope assumed that the summaries reflected only the rate of building fireline, the conditions for signaling failures effectively varied among the different projection actions. To accommodate these differences, the projection actions were restructured to set separate vari-
led to a lower incidence of a general failure to calculate projections. The data showed a decrease in the mean failures per hour from \( .41 \) in the previous experiment to \( .33 \) in this experiment.

Because the dependency sets reflect the interaction of the planner and failure recovery, similar designs for the planner and failure recovery should result in similar dependency sets. We can test this intuition by examining the dependency sets derived from execution of different versions of the system and counting the number of significant dependencies shared by the different versions. Table 2 shows the number shared between the most similar previous system and the modified planner: about 30\% (4 out of 15 total) of the \([R,F]\) and \([F,F]\) dependencies from the first set appeared in the second.\(^1\) The more we change the system, the more the dependency sets should change. In fact, the dependency sets for these two planner and recovery configurations as well as two others showed moderate overlap between similar systems, but negligible overlap across systems that differed by more than one aspect of their design. Some of the implications of the overlap in dependencies will be discussed in the last section.

Applying FRA to Phoenix improved the planner's performance by removing the targeted dependency and reducing the overall incidence of failure. The analysis showed how failures depend on their immediate predecessors. The cost of this information is the effort required to perform the analysis and the computation time required for the experiments. The computational effort required for the analysis was minimal; calculating the first step and part of the second took less than five minutes for the two data sets. Generating the execution traces for the two data sets required about 45 hours of CPU time or about two days per data set. Considering the possible repercussions of even simple planner modifications, the turnaround time for results seems worth the information gained.

**Generalizing to Other Planners**

The experience with Phoenix should generalize to other planners. The primary requirement for FRA is lots of data about how the planner performs. Each experiment with Phoenix represents over 5000 hours of simulated time. Simulators expedite controlling the environment and gathering data; controlled testing of planners in “real” environments increases the effort required to collect the data. The consequence of getting too little data is that rare failures and their associated dependencies will be missing. The technique does not guarantee that all dependencies will be found; the confidence in the dependency relationship increases linearly with the amount of data.

Given the availability of the data, the first step in FRA is applicable to any planner and environment. The remaining steps have been tailored to Phoenix, but conceptually could be expanded for other planners. These three steps are based on explaining failures by matching patterns to explanations and modifications (as in the “retrieve-and-apply” approach [Owens, 1990]).

\(^1\)Many more dependencies appeared in the first than the second set. The reduction in overall number of dependencies between these sets was mostly due to the elimination of dependencies involving the failure \( F_{prj} \) which was hardly ever observed in execution traces of the modified planner.

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>A Total</th>
<th>Shared</th>
<th>B Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R,F )</td>
<td>15</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>( F,F )</td>
<td>15</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>( FR,F )</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2: Overlap in dependency sets for original planner (A) and modified planner (B)
from consideration which results in execution traces free from the interaction of the missing action. By comparing the dependencies for each action removed, one can infer which dependencies were due to interactions with the missing actions. For example, if an action, say \( R_{sp} \), is removed from the set and the frequency of \( F_{vit} \) relative to other failures decreases, then one could see whether dependencies in \([F, R_{sp}, F_{vit}]\) triples explain all the surplus \( F_{vit} \) failures when \( R_{sp} \) is in the set, or whether \( R_{sp} \) affects \( F_{vit} \) over longer intervals.

**Conclusion**

Analyses of failure recovery can contribute in several ways to our understanding of planner performance. As described, FRA can identify contributors to failure and assist in the debugging and evaluation of planners with incomplete or incorrect domain models. Additionally, the dependencies provide a measure of similarity between test situations. The more the environment and agent changes, the more one expects observed effects to change; thus, dependencies can be a kind of similarity measure across planners and environments.

The lesson from this analysis is that while design changes rarely have isolated effects, designers do not have to give up hope of analyzing the effects. They can track the effects: They make minor changes and havoc ensues, but they have a way to assess the havoc. Phoenix is an example of a system that can interleave plans in arbitrary ways, as dictated by situation. Debugging its failures by “watching the system” or by predicting all possible execution traces is simply not feasible, but running Phoenix many times and analyzing the data is feasible. Failure recovery analysis isolates indirect effects of design changes and proposes explanations and modifications based on a weak model of the planner and its environment; its primary contribution is in helping us understand how planning decisions and actions interact and assisting in debugging planners under development.

**References**


