ELSEVIER

Artificial Intelligence 76 (1995) 125-166

Artificial Intelligence

Understanding planner behavior

Adele E. Howe[a,*], Paul R. Cohen[b,1]

[a] *Computer Science Department, Colorado State University, Fort Collins, CO 80523, USA*

[b] *Computer Science Department, University of Massachusetts, Amherst, MA 01003, USA*

Received June 1993; revised March 1994

Abstract

As planners and their environments become increasingly complex, planner behavior becomes increasingly difficult to understand. We often do not understand what causes them to fail, so that we can debug their failures, and we may not understand what allows them to succeed, so that we can design the next generation. This paper describes a partially automated methodology for understanding planner behavior over long periods of time. The methodology, called *Dependency Interpretation*, uses statistical dependency detection to identify interesting patterns of behavior in execution traces and interprets the patterns using a weak model of the planner's interaction with its environment to explain how the patterns might be caused by the planner. Dependency Interpretation has been applied to identify possible causes of plan failures in the Phoenix planner. By analyzing four sets of execution traces gathered from about 400 runs of the Phoenix planner, we showed that the statistical dependencies describe patterns of behavior that are sensitive to the version of the planner and to increasing temporal separation between events, and that dependency detection degrades predictably as the number of available execution traces decreases and as noise is introduced in the execution traces. Dependency Interpretation is appropriate when a complete and correct model of the planner and environment is not available, but execution traces are available.

1. Introduction

AI planners have long been introspective. They sit thinking about whether their actions will interact or their plans are efficient, and then they issue a plan. Some planners execute their plans, sometimes before planning is complete. The introspective nature of planners can make it difficult for us to understand why they act as they do, especially when

* Corresponding author. E-mail: howe@cs.colostate.edu.

[1] E-mail: cohen@cs.umass.edu.

0004-3702/95/$09.50 © 1995 Elsevier Science B.V. All rights reserved

SSDI 0004-3702(94)00083-2

planning and acting are interleaved, and even more so when the planner is responsive to a dynamic environment. Although we know how our planners are designed, and we can record their behavior, we frequently have trouble connecting the two. A planner's design includes many interacting components, and actions and events in the environment combine to produce complex execution traces. To understand planner behavior, we must represent the internal workings of a planner and its external execution traces in a way that skips a staggering amount of detail and permits us to predict and explain the effects of modifications to the planner, its tasks, and its environment.

Our approach to understanding planner behavior is to find statistical regularities in execution traces and then use a weak model of the planner to explain the regularities. First, we find dependencies among actions—unexpectedly frequent or infrequent co-occurrences. Dependencies arise for many reasons, most very mundane. For example, one action might frequently follow another in an execution trace because a frequently-used subplan includes both actions in a strict order. Some dependencies, however, alert us to problems; for example, a long sequence of obstacle-avoidance actions suggests that the planner has become trapped. To explain dependencies in an execution trace, mundane or otherwise, we find out which plans were being executed during the trace, and we search the plans for structures that often result in co-occurrences of actions, such as a strict linear ordering between actions. When we have found all such structures, we use them to index a library of explanations of the dependencies. All these steps are completely automated or semi-automated, and they have been applied to help us debug the failure-recovery component of the Phoenix planner [12].

It is straightforward to find statistical dependencies in execution traces, and only slightly more difficult to identify relationships, such as overlap, between dependencies. However, most dependencies signify nothing of interest. The contribution of this paper is to show how weak knowledge about a planner and its environment can be used to interpret dependencies. Our approach is not to explain particular fragments of execution traces in terms of detailed histories of internal and external states, but, rather, to explain statistical tendencies, over many planning episodes, in terms of general structures in plans.

1.1. Related research

Much of the research effort into understanding planner behavior has focused on plan failures, in particular, understanding why plans fail so that the plan and the planner can be debugged. Researchers have taken two general positions on plan and planner debugging. One is that bugs or failures can be anticipated by looking at the structure of plans, the other approach is to uncover pathologies by simulating plan execution or actually executing plans. Sussman's HACKER is the earliest example of the first approach [19]: critics look at the structure of a plan at each level of its development and detect potential problems (and opportunities), which inform the next level of plan development. Instead of simulating execution to discover bugs, HACKER could recognize structures that would lead to bugs. For several years, researchers built planners to identify and satisfy increasingly complex constraints among actions; again, without actually simulating plan execution [18].

Many subsequent efforts relied on simulation or execution, however. Hammond's CHEF [10] simulates execution to produce an execution trace of the plan that includes relationships between plan actions and resulting states. CHEF chains backward through the execution trace to determine the steps that caused a failure, classifies the failure cause based on the explanation of what happened and indexes into a set of general repair strategies to fix the plan to avoid the observed failure. Like CHEF, Simmons' GORDIUS [16] debugs plans by simulating them with a causal model. GORDIUS traces plan assumptions by regressing desired outcomes (values of states) through a causal dependency structure (generated through simulation of the plan) and identifying mismatches. The system then repairs the faulty assumptions with one of a set of general repair strategies. Both Hammond's and Simmons' approaches assume that the simulator has a correct model of the domain; the approaches differ in the kinds of flaws they detect and the strategies they apply to repair the faults. Related efforts, all of which rely on causal models of the planner, include Hudlicka and Lesser's work on diagnosing failures during execution [14], and Birnbaum et al.'s proposal to enhance model-based diagnosis of plans [3].

Most of the research addresses debugging *plans*, rather than debugging the *planner*. While the two are related (after all, explaining why a plan would fail is a step toward explaining why a planner should not favor such a plan), little has been done on planner debugging. The most notable exception is Hammond's CHEF, which learned from its plan failures. CHEF would remember repaired plans and the bugs found in them so that subsequent planning could use the newly modified plans and account for the bugs found previously. Others have exploited the idea of model-based explanation of failures, bugs and errors that arise in execution traces to learn new plans (e.g., [2,5]).

Our position is an amalgam of the structural and simulation/execution approaches focused on debugging the planner. We think it is too difficult to debug a planner by identifying an individual bug and explaining how it arose in terms of a history of states, variable bindings, environmental events and other details. Instead, we rely on statistical dependencies to point us to pieces of plans, and we look for structural features of those pieces that might explain the dependencies. After we have made some modifications, we test whether our explanations were correct by executing plans and seeing whether particular dependencies disappear or are reduced. Our approach has some parallels in software testing and program analysis. For example, Bates and Wileden [1] developed a language for describing salient abstractions of a system's behavior; a module monitors the system's behavior and extracts event traces based on the desired abstractions. Gupta describes a knowledge-based system for selectively collecting and analyzing traces of interprocess messages [9]. In both of these systems, a human programmer must examine the resulting traces and localized failures to determine why the software failed and to debug it. As in our statistical approach, a technique for hardware fault diagnosis, called correspondence analysis, classifies failure modes into "causes" by analyzing contingency tables of system test results [15]. As in our knowledge-based approach, the DAACS (Dump Analysis And Consulting System) takes a snapshot of a particular type of fatal program error (the contents of a minidump) and matches information from the dump to a belief network of canonical diagnoses [4]. The result is a set of hypotheses about the source of the failure.

Most research in AI debugging explains particular failures in order to debug a plan; most research from software testing addresses finding patterns of behavior in order to debug a program. Our approach combines the two to explain patterns of behavior over time in order to debug a program, the planner. In particular, we use a statistical technique (called *dependency detection*) to identify patterns of behavior followed by a knowledge-based interpretation phase to construct explanations of how the planner produced those behaviors. The next two sections will describe the statistics and their interpretation.

2. Dependency detection

In this section we will describe how to find statistical dependencies in execution traces. A dependency is an unexpectedly frequent or infrequent co-occurrence. For example, let A, B and C be actions in a plan, and consider the execution trace

B A B C C C B B C B A B A B C A B A C.

One thing you will notice is that A almost never occurs without B following it, immediately (only the last occurrence of A is followed by C). We could represent this with a contingency table where the first action in the subsequence identifies the row and the second action identifies the column.

	B	$\overline{\text{B}}$	Totals
A	4	1	5
$\overline{\text{A}}$	3	10	13
Totals	7	11	18

The table shows that the subsequence AB occurred four times and the subsequence of A followed by something other than B (denoted A$\overline{\text{B}}$) occurred just once. In addition, the table shows 3 occurrences of $\overline{\text{A}}$B and 10 occurrences of $\overline{\text{A}}\overline{\text{B}}$. Apparently, the odds of seeing B as the second element in a two-element subsequence depends on whether the first element is A. If the first element is not A, then the odds of seeing B are 3:10, whereas the odds are 4:1 if the first element is A. In other words, the presence of A appears to make B more likely. This impression might be erroneous: the execution trace is short, the numbers in the contingency table are small, and the apparent relationship between A and B might be no more than an accidental pattern in a random sequence of letters. Statistical tests of contingency tables tell us the probability that apparent relationships, such as the dependency between A and B, are due to chance. (We will not describe the underlying probabilistic justification for these tests, here, but see [7, Chapter 2].)

The most common test of contingency tables is the chi-square test [17], but we will use the closely-related G test because it is *additive*, as described later. The test statistic for a contingency table is:

$$G = 2\sum_{\text{cells}} f_{ij} \ln\left(\frac{f_{ij}}{\hat{f}_{ij}}\right) \tag{1}$$

where f_{ij} is the number of occurrences, or frequency, in the cell i,j and $\hat{f}_{ij}$ is the expected frequency for cell i,j. Expected frequencies can be arrived at two ways: they can be specified extrinsically (as we might specify that the expected frequency of males to females in a sample is one to one) or they can be calculated from the contingency table under the assumption that the row and column variables are independent. In the first case, the G test is a test of goodness of fit to an extrinsic frequency distribution; in the second, it is a test of independence. Dependency detection is based on tests of independence.

Expected frequencies for tests of independence are derived from row and column sums. In the table, above, the total numbers of occurrences of B and $\overline{B}$ are 7 and 11, respectively, so our best estimate of the population probabilities of B and $\overline{B}$ are 7/(7+11) and 11/(7+11), respectively. By the same logic, our best estimates of the population probabilities of A and $\overline{A}$ are 5/(5+13) and 13/(5+13), respectively. Now, if the occurrence of B is independent of the precursor A, then the probability of the sequence AB is just the product of the probabilities of A and B. That is, assuming independence, Pr(AB) = Pr(A) $\times$ Pr(B) = (5/18) $\times$ (7/18). The expected frequency of AB is the probability of AB times 18, the number of items in our contingency table. This product, (5/18) $\times$ (7/18) $\times$ 18 simplifies to (5 $\times$ 7)/18. In general, the expected frequency for a cell in row i and column j, is $\hat{f}_{ij} = f_{i\bullet}f_{\bullet j}/f_{\bullet\bullet}$, where $f_{i\bullet}$ and $f_{\bullet j}$ are the totals for row i and column j, respectively, and $f_{\bullet\bullet}$ is the sum of all the cells. But remember: this formula gives the expected frequency for cells under the assumption that the column factor is *independent* of the row factor.

Equation (1) simply sums the deviations between the expected frequencies and the actual frequencies in a contingency table. The larger the value of G, the less we believe the independence assumption. Substituting expected and actual frequencies for our contingency table into Eq. (1), above, we obtain:

$$G = 2\left[4\ln\left(\frac{4}{\frac{5\cdot 7}{18}}\right) + 1\ln\left(\frac{1}{\frac{5\cdot 11}{18}}\right) + 3\ln\left(\frac{3}{\frac{7\cdot 13}{18}}\right) + 10\ln\left(\frac{10}{\frac{11\cdot 13}{18}}\right)\right] = 5.007.$$

When this value is referred to a chi-square distribution with one degree of freedom, we find that the probability of attaining a result greater than or equal to G, under the assumption that the occurrence of B is independent of the occurrence of A, is no more than 0.0265. Thus, if we claim that B depends on A, then the probability of being wrong is less than three in 100.

The G test for independence tells us whether the ratio of B to $\overline{B}$ is significantly different in one row of the contingency table than in another. Thus, it is sometimes called a *heterogeneity* test because it tells us whether the ratios of cell frequencies in each row are heterogenenous. In our example, heterogeneity is significant.

Although the dependency between A and B is probably not spurious, it is not the only dependency in the execution trace, and is perhaps not as easy to explain as other dependencies that might subsume it. Note, for example that the longer subsequence B*B occurs four times, three of which have A in place of the wildcard *.[2] Perhaps the

[2] We have used the symbol * to designate a *single* wildcard symbol.

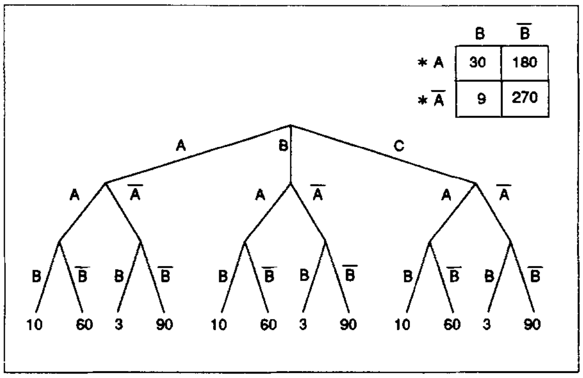


Fig. 1. An example of a dependency between A and B, where the dependency does not depend on the previous action.

execution trace is telling us that B tends to be followed after an intervening action by itself, and, by chance, A was the intervening action three times out of four. The question is whether B*B subsumes AB, that is, whether AB occurs primarily as a subsequence of BAB, or whether, instead, AB has some independent existence. Subsumption is one of the relationships we must consider when we look at longer sequences of actions. For example, if the execution trace includes an unexpectedly large number of sequences BAB, then we must consider three possibilities:

(1) There is a dependency between A and B, and the dependency does not depend on the preceding action, B.
(2) There is a dependency between B and B separated by one action, and the dependency does not depend on the intervening action, A.
(3) There is a dependency between B, A and B

An example of the first kind is shown in Fig. 1. The lower levels of the tree represent the four two-action sequences we considered earlier: AB, A$\bar{B}$, $\bar{A}$B, $\bar{A}\bar{B}$, and the upper level represents three possible precursors, A, B and C, of these sequences. With a tree like this we can see whether a dependency between A and B itself depends on a precursor. In Fig. 1, at least, the odds of seeing B after A are 1:6, whereas the odds of seeing B after $\bar{A}$ are 1:30, and these ratios do not depend on which action precedes the two-action subsequences. In other words, Fig. 1 shows an unexpectedly high co-occurrence of A and B that is not subsumed by AAB, BAB or CAB.

Fig. 2 shows an example of the second kind, an A*B dependency that does not depend on the intervening action that we substitute for the wildcard. In this case, the odds of seeing A*B are 1:6, whereas the odds of seeing $\bar{A}$*B are 1:30, and these odds do not depend on whether the intervening action is A, B or C.

Sometimes, dependencies can cancel each other out. Fig. 3 shows a case in which there is a strong propensity for B to follow AA in the three-action sequence AAB, and a strong propensity for B to *not* follow A in the sequences BAB and CAB. However, the overall contingency table shows no dependency between A and B: the probability of B following A is 0.5 and the probability of B following $\bar{A}$ is also 0.5.

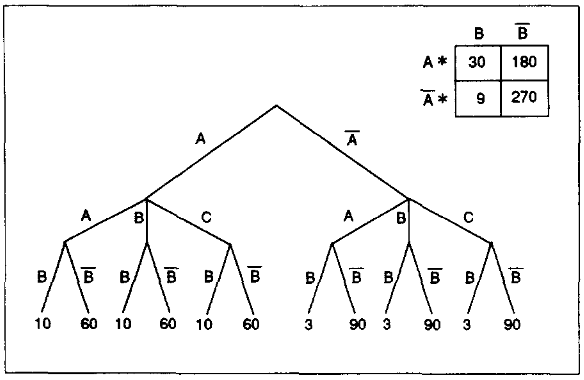


Fig. 2. An example of a dependency between A and B separated by one action, where the dependency does not depend on the intervening action.

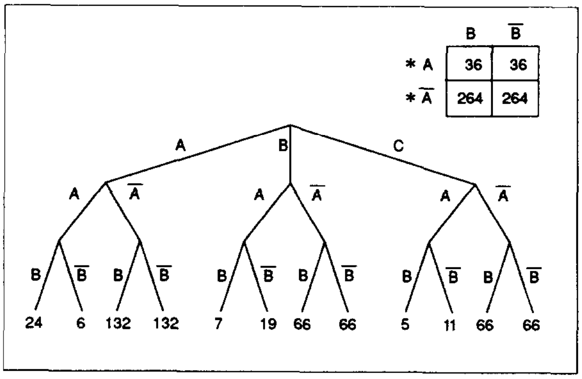

B $\bar{B}$
*A 36 36
*$\bar{A}$ 264 264
A B C
A $\bar{A}$ A $\bar{A}$ A $\bar{A}$
B $\bar{B}$ B $\bar{B}$ B $\bar{B}$ B $\bar{B}$ B $\bar{B}$ B $\bar{B}$
24 6 132 132 7 19 66 66 5 11 66 66

Fig. 3. Although the contingency table shows no dependency between A and B, B shows a strong propensity to follow A in the sequence AAB, but the opposite effect holds for the sequences BAB and CAB.

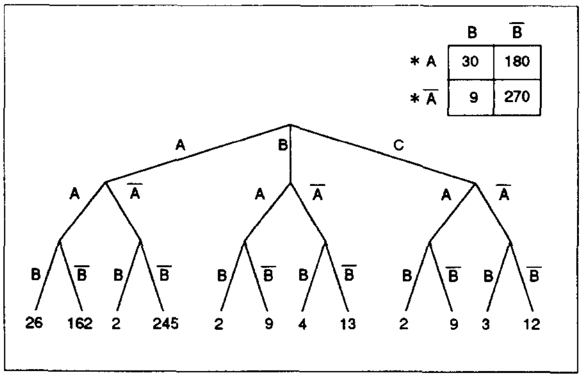

B $\bar{B}$
*A 30 180
*$\bar{A}$ 9 270
A B C
A $\bar{A}$ A $\bar{A}$ A $\bar{A}$
B $\bar{B}$ B $\bar{B}$ B $\bar{B}$ B $\bar{B}$ B $\bar{B}$ B $\bar{B}$
26 162 2 245 2 9 4 13 2 9 3 12

Fig. 4. The AB dependency appears only when the previous action is A.

Lastly, Fig. 4 shows a case in which the AB dependency shows up only when the first actions of a three-action sequence are AA, and this dependency dominates the data. There is no evidence in Fig. 4 for a general two-action AB dependency, because B does not appear to depend on A in three-action sequences BAB and CAB.

With G tests, we can differentiate the cases illustrated in Figs. 1, 2, 3 and 4. To do so, we must be able to test both two-item dependencies (e.g., $*$AB) and three-item dependencies (e.g., AAB). When it doesn't matter which item we substitute for $*$, only two-item dependencies will be significant. When dependencies cancel each other, only three-item dependencies will be significant. Usually, though, two- and three-item dependencies will all be significant. Let us show how to build contingency tables to test all these cases. We begin with the contingency table in Fig. 4, which we reproduce here as Table 1.

Table 1
The Fig. 4 contingency table

	B	$\bar{\text{B}}$
$*$A	30	180
$*\bar{\text{A}}$	9	270
Totals	39	450

Because Eq. (1) is a *sum* of differences between expected and actual frequencies, we can partition it into components. For instance, we can calculate one G statistic for the $*$A row of Table 1, and another G statistic for the $*\bar{\text{A}}$ row, and then add these statistics together to get G for the whole table. Noting that the row sums are 210, 279, and the column sums are 39, 450, respectively, we can calculate expected frequencies as above. The G statistic for the first row is:

$$G_{*\text{A}} = 2\left[30\ln\left(\frac{30}{\frac{(210\times 39)}{489}}\right) + 180\ln\left(\frac{180}{\frac{(210\times 450)}{489}}\right)\right] = 9.40.$$

The G statistic for the second row is calculated the same way:

$$G_{*\bar{\text{A}}} = 2\left[9\ln\left(\frac{9}{\frac{(279\times 39)}{489}}\right) + 270\ln\left(\frac{270}{\frac{(279\times 450)}{489}}\right)\right] = 10.88.$$

The sum of these statistics is 20.28, which is exactly what we get when we apply Eq. (1) to run a test of independence on the whole contingency table:

$$G_{\text{P}} = 2\left[30\ln\left(\frac{30}{\frac{(210\times 39)}{489}}\right) + 180\ln\left(\frac{180}{\frac{(210\times 450)}{489}}\right)\right.$$
$$\left. + 9\ln\left(\frac{9}{\frac{(279\times 39)}{489}}\right) + 270\ln\left(\frac{270}{\frac{(279\times 450)}{489}}\right)\right] = 20.28.$$

There's nothing magical about this *additivity* property: it is just a consequence of Eq. (1) being a sum. However, decomposing G tells us that $G_{*\mathrm{A}} = 9.40$ and $G_{*\overline{\mathrm{A}}} = 10.88$, that is, each row contributes roughly equally to the total G value of 20.28. Said differently, the frequencies 30 and 180 deviate from their expectations only a little less than do the frequencies 9 and 270. For convenience, G_{P} will denote the G statistic for Table 1; the P stands for "pooled" or "packed", in contrast to other G statistics we'll calculate later.

Comparing $G_{\mathrm{P}} = 20.28$ to a chi-square distribution with one degree of freedom, we find it is highly significant. This means the occurrence of B or $\overline{\mathrm{B}}$ depends on whether it is preceded by $*$A or $*\overline{\mathrm{A}}$, but it says little about possible effects of the first item in the precursor, denoted $*$. For instance, we have not tested the possibility, suggested by the tree in Fig. 4, that the dependency between B and $*$A reflects the strong propensity of $\overline{\mathrm{B}}$ to follow AA (in 162 of 188 cases), whereas the tendency of $\overline{\mathrm{B}}$ to follow BA or CA appears much weaker, due to relatively few cases. To test three-item dependencies, we must build slightly different contingency tables.

Note that a row in a contingency table might itself summarize *another* contingency table, the structure of which is lost when it is summarized. In Fig. 4, for example, the frequencies of $*$AB and $*$A$\overline{\mathrm{B}}$ summarize the frequencies of AAB and AA$\overline{\mathrm{B}}$, BAB and BA$\overline{\mathrm{B}}$, and CAB and CA$\overline{\mathrm{B}}$. We can "unpack" the 30 occurrences of $*$AB into 26 occurrences of AAB and two occurrences each of BAB and CAB. Similarly, we can unpack the 180 occurrences of $*$A$\overline{\mathrm{B}}$. We could also unpack the second row of Table 1—the nine occurrences of $*\overline{\mathrm{A}}$B and the 270 occurrences of $*\overline{\mathrm{A}}\overline{\mathrm{B}}$—but it isn't necessary for our immediate purposes. Unpacking the first row will suffice, and it yields Table 2.

Table 2
The "unpacked" contingency table

	B	$\overline{\mathrm{B}}$
AA	26	162
BA	2	9
CA	2	9
$*\overline{\mathrm{A}}$	9	270
Totals	39	450

The total G statistic for Table 2 will necessarily exceed $G_{\mathrm{P}} = 20.28$, calculated above. Remember that G measures *heterogeneity*, the degree to which the ratios of frequencies in rows differ. In general, heterogeneity for a table increases when a row is unpacked, because the row is replaced by two or more new rows that are apt to be heterogeneous. Only when the new rows contain frequencies in exactly the proportions of the unpacked row, will heterogeneity remain constant (we will see an example shortly). The G statistic for Table 2, denoted G_{U} for "unpacked", is easily computed with Eq. (1):

$$G_U = 2\left[26\ln\left(\frac{26}{\frac{(188\times39)}{489}}\right) + 162\ln\left(\frac{162}{\frac{(188\times450)}{489}}\right) + 2\ln\left(\frac{2}{\frac{(11\times39)}{489}}\right)\right.$$
$$+ 9\ln\left(\frac{9}{\frac{(11\times450)}{489}}\right) + 2\ln\left(\frac{2}{\frac{(11\times39)}{489}}\right) + 9\ln\left(\frac{9}{\frac{(11\times450)}{489}}\right)$$
$$\left. + 9\ln\left(\frac{9}{\frac{(279\times39)}{489}}\right) + 270\ln\left(\frac{270}{\frac{(279\times450)}{489}}\right)\right] = 20.57.$$

As expected, $G_U > G_P$, but not by much. We will let $G_H = G_U - G_P = 0.287$ denote the increase in heterogeneity due to unpacking. Because G_H is so small, we know that the new rows in Table 2, AA, BA and CA, differ little amongst themselves—the ratios of their frequencies are similar. In fact, the ratio in row AA is 26:162, or roughly 1:6; whereas the ratios for rows BA and CA are 1:4.5. The *raw* row frequencies, however, are very different: Row AA boasts 188 occurrences of B or $\overline{\text{B}}$, whereas the other rows contain just 11 occurrences each. The contributions of each row to G_U reflect these differences:

$$G_{AA} = 2\left[26\ln\left(\frac{26}{\frac{(188\times39)}{489}}\right) + 162\ln\left(\frac{162}{\frac{(188\times450)}{489}}\right)\right] = 7.327,$$

$$G_{BA} = 2\left[2\ln\left(\frac{2}{\frac{(11\times39)}{489}}\right) + 9\ln\left(\frac{9}{\frac{(11\times450)}{489}}\right)\right] = 1.18,$$

$$G_{CA} = 2\left[2\ln\left(\frac{2}{\frac{(11\times39)}{489}}\right) + 9\ln\left(\frac{9}{\frac{(11\times450)}{489}}\right)\right] = 1.18.$$

We have derived three results: First, a strong dependency exists between the precursors $*$A and $*\overline{\text{A}}$, and the subsequent occurrence of B or $\overline{\text{B}}$ ($G_P = 20.28$). Second, the heterogeneity introduced by unpacking $*$A into AA, BA and CA is negligible ($G_H = 0.287$). Third, the precursor AA contributes more to G_U than BA or CA does. How might we interpret these results?

We can see in Table 1 that B is more likely to follow $*$A than $*\overline{\text{A}}$: the probability of observing B is 1/6 following $*$A and 1/30 following $*\overline{\text{A}}$. We don't know whether

(1) the presence or absence of A in the precursor is responsible, or
(2) the presence of absence of a particular item that occurs in place of $*$ is responsible, or
(3) an interaction between a particular substitution for $*$ and the subsequent item in the precursor is responsible.

In fact, our results support the third interpretation: B follows $*$A more frequently than $*\overline{\text{A}}$, so we know that the presence or absence of A as the second item in the precursor increases the likelihood of observing B. But this is not the whole story, Unpacking the

∗A row of Table 1 affects heterogeneity little; the ratios of frequencies in the AA, BA and CA rows are similar, and if they are similar to each other, they must also be similar to the ratio of frequencies in the ∗A row. In other words, the propensity of B, say, to follow ∗A is similar to its propensity to follow AA, BA or CA. This suggests that the first item in a two-item precursor has little influence on the occurrence of B when the second item in the precursor is A. However, this ignores the fact that 188 of the 210 cases in the ∗A row of Table 1 are actually AA precursors, while BA and CA contribute very little. Thus, we conclude that the precursor AA strongly affects the occurrence of B; while the effects of the precursors BA and CA are consistent (observing B is more likely) but weaker. The conclusion that ∗A influences B is too general; we see now that AA influences B.

We are ready to quantify these conclusions. Keep in mind that G statistics are additive. As a result, we can construct a summary of our results like this:

Statistic	**Value**	**Significance**
G_{P}	20.28	significant
G_{AA}	7.327	significant
G_{BA}	1.18	not significant
G_{CA}	1.18	not significant
$G_{*\bar{\mathrm{A}}}$	10.88	significant
G_{U}	20.567	significant
G_{H}	0.287	not significant

The first result, $G_{\mathrm{P}} = 20.28$, tells us ∗A influences B, and if none of the more specific results were significant, we would conclude that ∗, the item that precedes A, doesn't matter. This test, on a 2×2 table, has one degree of freedom. The row and column sums are fixed because we use them to estimate cell frequencies, as is the total sum for the table. Thus, if a row contains C columns, only $C - 1$ are free to vary; the last being constrained to "make up" the row sum. Similarly, a column of R rows has $R - 1$ degrees of freedom. Thus an $R \times C$ table has $(R - 1) \times (C - 1)$ degrees of freedom. The result for the 4×2 unpacked table, $G_{\mathrm{U}} = 20.567$, has three degrees of freedom and is also significant. It tells us that the ratios of row frequencies in Table 2 are significantly different. Because G is additive, we can see which of the four rows in Table 2 are responsible for the heterogeneity. Clearly, rows AA and the unpacked row $*\bar{\mathrm{A}}$ contribute most. The rows BA and CA are insignificant: when we ask of each row whether its cell frequencies are different from their expectations, as measured by G, the answer is no. Each test on a row has $C - 1$ degrees of freedom, because the row sum (which is used to estimate expected frequencies) constrains one of the frequencies in a row. The tests of G_{AA}, G_{BA}, G_{CA}, and $G_{*\bar{\mathrm{A}}}$ each have one degree of freedom because our contingency tables have only two columns.

We get very different results when we calculate G statistics for other execution traces besides the one summarized in Fig. 1. For example, here are the statistics for Fig. 1:

Statistic	**Value**	**Significance**
G_P	9.40	significant
G_{AA}	3.133	not significant
G_{BA}	3.133	not significant
G_{CA}	3.133	not significant
$G_{*\bar{A}}$	10.88	significant
G_U	9.4	significant
G_H	0	not significant

The interpretation of these results is that there is a significant dependency between A and B in two-action subsequences, and this effect is identical ($G_H = 0$) for all substitutions—A, B and C—for $*$. In fact, none of the individual three-action dependencies is significant; for example, the ratio of the occurrences of B to $\bar{B}$ in the sequences AAB and AA$\bar{B}$ is not significantly different than the expected population ratio ($G_{AA} = 3.133$). In short, when the second action of a precursor is A, the first action does not influence the occurrence of B.

It turns out that the situation in Fig. 2 yields identical G statistics and has a very similar interpretation to the case in Fig. 1. The only difference is that we look for differences due to the precursors AA, AB and AC instead of the precursors AA, BA, CA. However, the ratio of B to $\bar{B}$ is not influenced by whether the first two actions are AA, AB or AC; in all cases, the ratio is 1:6. Thus, $G_H = 0$ and $G_P = G_T = 9.4$. We conclude that the data show a dependency between A and B, even when another action intervenes, and the intervening action does not influence this dependency.

Finally, consider the situation in which dependencies cancel each other out, as shown in Fig. 3. The G statistics for this case are:

Statistic	**Value**	**Significance**
G_P	0	not significant
G_{AA}	11.565	significant
G_{BA}	5.754	significant
G_{CA}	2.306	not significant
$G_{*\bar{A}}$	0	not significant
G_U	19.625	significant
G_H	19.625	significant

When the precursor is AA, then the ratio of B to $\bar{B}$ in the subsequent action is significantly different from its expectation ($G_{AA} = 11.565$). Similarly, if the precursor is BA, then the ratio of B to $\bar{B}$ is also significantly different from its expectation ($G_{BA} = 5.754$). If the precursor is CA, no effect is evident ($G_{CA} = 2.306$). Despite strong effects of two precursors, no effect is evident when all the data are pooled ($G_P = 0$). This is because there are really *two* three-item dependencies that cancel each other out. The first entails that AA is likely to be followed by B; the second says BA is likely to be followed by $\overline{\text{B}}$. These opposing tendencies yield a highly heterogeneous table when the $*$A row is unpacked ($G_{\text{H}} = 19.625$), but of course they are invisible in the $*$A row itself ($G_{*\text{A}} = 0$). We note in passing that the test of G_{H} has two degrees of freedom because three rows result from unpacking the $*$A row but their sums are constrained by the $*$A row sum. An alternative formulation would be to run an ordinary test of independence on a table with rows AA, BA and CA: this will produce G_{H}, and clearly has two degrees of freedom.

In sum, for any subsequence XYZ of three successive actions in an execution trace, we can differentiate three cases: Z depends on Y, irrespective of action X (Fig. 1); Z depends on X, irrespective of action Y (Fig. 2); and the dependency between Z and Y itself depends on X (Figs. 3 and 4).

3. Interpretation

Dependencies, by themselves, raise more questions than they answer. They might suggest relationships that we did not expect or they might only confirm what we knew before or be spurious. Interpretation distinguishes these situations and explains the relationships, suggested by the dependency, in terms of the planner and its environment.

Interpretation has two parts: identifying what plans are involved in the dependencies and constructing explanations of how the plans might have produced the dependencies. In contrast to detecting dependencies, both parts rely on knowledge about the planner and its environment.

3.1. Identifying suggestive plan structures

For the first part of interpretation, we search for the means of producing the dependency—how one event or action can influence the occurrence of another. The purpose of this step is to identify which planning structures, knowledge and mechanisms influence particular events (or *suggest* the means for a dependency). If one event depends on another, there must be a medium through which they interact; we search for it in the description of the plans. We can also detect unusually *low* co-occurrence of events, but at present, we do not try to explain any lack of desired co-occurrence.

Determining what might produce the dependencies requires knowledge about the planner and its environment. Yet, this conflicts with the goal of minimizing our reliance on a model of the planner and the environment. Our solution is to supplement the available execution information and knowledge structures available within the planner with a weak model of the planner and its interaction with the environment.

The process of identifying *suggestive plan structures* starts by selecting one dependency for attention. We select one dependency for pragmatic reasons. First, at present, the search for suggestive structures is supported by a set of Lisp functions, one for each type of suggestive structure; it would be tedious to run the functions for every dependency. Second, even if this step and the next were fully automated, the designer would have to wade through a lot of information about the structures and explanations. Focusing attention on a single dependency reduces the possible deluge of information.

One must be careful in selecting a dependency for interpretation. Spurious dependencies can be most easily avoided by selecting strong dependencies: those that have a low probability of being due to chance and that are based on many instances. Determining whether the dependency is of interest is a judgment call; only the designer can decide that the dependency is unexpected and significant. Dependencies can also be ranked by their tenacity (i.e., whether they appear impervious to changes in environmental conditions) and their utility (e.g., the desirability of the observed behavior or the value of the observed outcomes).

Having selected a dependency for attention, it remains to associate each dependency with actions in plans, and find structural descriptions of the interaction between the actions in the dependency. Roughly speaking, we first situate the dependency in the plans and then determine how the precursor might have influenced the dependent event.

Associating dependencies with actions

First, we need to locate the dependency in the plans to localize what produces the behavior. Dependencies are composed of plan actions and environment events. To determine the role of the planner in producing the dependencies, we need to relate both parts to the planner. Plan actions need only be associated with the plans in which they can appear. Environment events are detected, directly or indirectly, by plan actions; typically, plan actions initiate sensor activation or processing or at least relate sensory information to on-going plans. Consequently, each dependency can be mapped to sets of actions that specify all the ways that the events in the dependency are detected, and all the ways the plan actions in the dependency appear in plans.

The mapping from the dependency to the plans demarcates the portion of the plan space to be searched for possible interactions. For example, Fig. 5 shows a plan that includes both the predecessor from the dependency (Action P) and the antecedent (Action A) from a dependency (solid lines indicate temporal precedence, dotted lines represent a hierarchical relationship). From this point in the interpretation process, we no longer concentrate on the dependency, but rather on the parts of plans between the precursor and the antecedent of the dependency.

Finding suggestive structures

Having identified the actions that were involved in the dependency (call them dependency actions), we still do not know what they have to do with each other. We could search the planner's short- and long-term memory to find out. The problem with searching short-term memory (e.g., all the plan expansions, variable bindings, and histories of environmental events) is that, even if it could somehow be recorded for all the runs of the planner, the detail would be overwhelming and largely irrelevant. The problem with searching long-term memory (e.g., the planner's representation of plans, actions and strategies for planning) is that plans are described with the wrong amount of detail for inferring behavior over time—too much procedural detail about how actions are instantiated in plans and too little detail about how plan fragments and actions can be included in different plans.

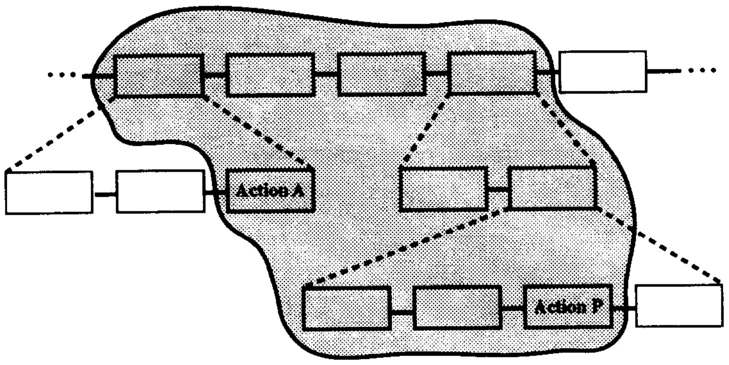


Fig. 5. A dependency has been mapped to two actions in a plan. The shaded region indicates the portion of the plan possibly relating to the dependency.

We supplement the available declarative plan representation with structural knowledge about how the planner combines plan fragments and actions to form executable plans and what relationships are formed between the plan fragments as a result of the combinations. The suggestive plan structures are language-based idioms that coordinate actions in plans and describe shared commitments to a course of action or shared expectations about the world. They are called suggestive structures because they suggest means by which events and the planners actions might interact. In many cases, such structures are included in plan languages because they support generalizing plans beyond specific variable instantiations (so promoting plan reuse), help coordinate the efforts of separate plans or agents, and constrain decision making (hopefully improving efficiency).

To identify suggestive plan structures for particular dependencies, we search the parts of plans demarcated by dependencies (as in Fig. 5) for structural relationships between the two actions. These structures are of three basic types: temporal, control and data interactions. In this case, the temporal structure is that the antecedent necessarily follows the predecessor; the suggestive structure is called *Sequential Actions*, in which one action is guaranteed to follow another in some plan. One cannot easily distinguish control and data relationships in this figure. A control structure is one in which some action determines whether a later action will be included in the plan. For example, the hierarchical relationship between actions in plans constitute a control structure because the expansion of the parent determines which actions are included at the lower level; this suggestive structure is called *Selection Constructs*, in which one action adds another to the plan. A data interaction structure is one in which plan actions share information. For example, some plan actions may set values of plan variables used by other plan actions; this structure is called *Shared Variables*, in which one action sets some variable and another uses it.

While most planners construct plans with these three types of structural relationships, the set of suggestive structures depends fundamentally on the plan language and the ways the planner incorporates information about the environment into its plans. The set of suggestive structures for one planner will be described in Section 4.2.

3.2. Explaining how suggestive structures produce dependencies

The second part of interpretation is to find explanations of how the suggestive structures might have produced the observed dependencies. The explanations amount to brief stories of what might have happened: how suggestive structures might combine to cause dependencies. They do not precisely determine the cause of the dependency, but rather are hypotheses about the source of the observed dependency.

The explanations are intended to explain the dependencies, not explain how the precursor caused the antecedent. This may seem a subtle distinction, but it is not. Dependencies are co-occurrences, not causal relationships. Two events may co-occur because they share the same cause or because they occurred together by chance. The explanations emphasize ways that the events can co-occur, rather than how one can cause the other.

An example of an explanation of a dependency between parts of different plans is *Resource Contention*, which describes what happens when several plans vie for the same resource with only one or none able to acquire or access it. Resource contention is common when multiple agents or plans compete for the same limited resources and can occur in plans without forced temporal sequencing or controlled resource management. An example of an explanation for a dependency between actions and the environment is *Overcommitment*, which describes why a plan may be inappropriate, and perhaps need to be retracted, when the environment changes in particular ways. Suggestive structures that suggest overcommitment are strong temporal sequences, data dependencies between the environment and plan actions, and control decisions based on possibly old information about the environment.

The result of the interpretation phase is a set of hypotheses about how the planner might have produced the observed dependencies. Given a set of suggestive structures and explanations, this phase may not find any explanations or may find too many; it is not guaranteed to find the "correct" one. Just as when human programmers try to describe the behavior of a program, this interpretation phase will be limited by its knowledge of the planner.

4. One application: Failure Recovery Analysis

We have applied dependency interpretation to understand why plans fail in Phoenix. This section describes how the target behavior (i.e., plan failure) and the target environment and planner (i.e., the Phoenix system which includes the environment and the planner) have led to a specialized form of dependency interpretation called Failure Recovery Analysis (or FRA). The purpose of Failure Recovery Analysis is to identify and explain cases in which plans may influence, exacerbate or cause failure; in other words, to assist in debugging the planner.

Failure recovery is the process by which the planner recognizes and repairs a plan or execution time failure. Most planners that operate in dynamic environments include some form of failure recovery. Even reactive systems include reactions for responding to less desirable situations.

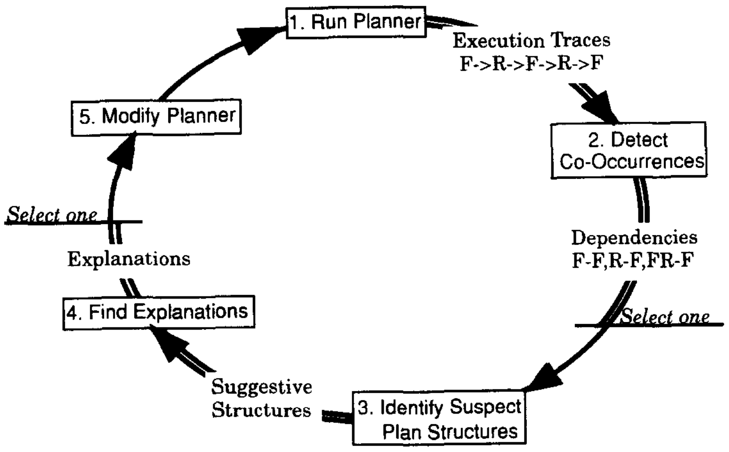


Fig. 6. The cycle for Failure Recovery Analysis: the designer iteratively debugs the planner by testing the planner, interactions.

We chose to focus on *failure recovery* as the aspect of the planner and environment to record and analyze for several reasons. 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 (as we discovered in a series of experiments with Phoenix [13]). 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 contexts not envisioned by the designers. Consequently, failure recovery may test the limits of the planner.

FRA proceeds as an iterative process in which the designer tests the planner, interprets the dependencies, and modifies the planner based on the dependency interpretation (as in Fig. 6). The process continues until the designer is satisfied with the resulting planner.

The designer starts by running the planner in its environment. The Phoenix system automates this first step; an automated experiment controller varies the environment within pre-defined ranges and collects execution traces of which failures occurred and how they were repaired. An example execution trace is:

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

where F's are failure types and R's are recovery methods. The subscripts indicate individuals from a set, so F_{ner} means failure of type `ner`.

Second, the execution traces are searched for dependencies between recovery efforts and failures. These dependencies tell the designer how the recovery actions influence the next failure[3] that occurs and how one failure influences the next. For FRA, dependencies consist of combinations of failure types and the recovery actions that repaired them. We

[3] The next failure refers to the failure that appears next in the execution trace. Because the execution trace removes time stamps, the temporal separation is unknown; it could be immediate or many hours later.

detect dependencies of three types: *F–F* (failure type followed by the next failure type), *R–F* (recovery action followed by the next failure type) and *FR–F* (failure type and the recovery method that repaired it followed by the next failure). Dependency detection (as described by the equations in Section 2) is implemented as Lisp functions that accept a list of execution traces as input and return a list of dependencies as output.

Third, the designer selects one of the dependencies for further attention and tries to determine how the planner's actions might have caused the observed dependency. The dependencies are mapped to actions in plans and then the plans are searched for suggestive structures that involve the actions and that are known to be susceptible to failure. To find suggestive structures, the designer runs a set of Lisp functions that check for each of the structures.

Fourth, the designer matches the suggestive structures for the dependency to a set of possible explanations and modifications. At present, this step is not automated; the designer must look through a set of explanations that are indexed by suggestive structures. The explanations and modifications are all described in general terms and are listed according to what structures indicate which explanations and modifications.

At the end of the cycle, the designer chooses the most likely explanation based on his or her understanding of the planner and modifies the planner to remove the suspected flaw. The cycle begins again with the designer running the planner. Next time around, the designer can search for more flaws to fix and can determine whether the modification achieved the desired effect, that is, the observed dependency disappeared and the incidence of failures changed for the better.

In contrast to other approaches to debugging, FRA is a procedure applied by a human designer, rather than a fully automated system. The designer decides where to focus attention and ultimately how to fix the planner to repair the bug. FRA trades power for generality. Because it uses a weak model, inherent in its suggestive structures and explanations, it can localize a broad range of bugs and should be appropriate for many planners, but it cannot guarantee that it will find all bugs or even find the actual cause of failure.

4.1. Phoenix: the target planner and its environment

The Phoenix system[4] [8] serves as the laboratory for this application of dependency interpretation. As shown in Fig. 7, Phoenix provides the simulated environment, an agent architecture with a set of plan knowledge bases for each type of agent, and an experimental interface for automatically controlling experiments and collecting data. Its environment is forest fire fighting in Yellowstone National Park.

The goal of forest fire fighting is to contain fires as efficiently as possible. Forest fires spread in irregular shapes, at variable rates, as a function of ground cover, elevation, moisture content, wind speed and direction, and natural boundaries (e.g., bodies of water and large roads). Fires are contained by removing fuel from their paths, causing them

[4] "Phoenix" refers to the entire system: simulator and all the components that comprise the agents, including the planner. Whenever possible, we use "Phoenix planner" to distinguish the planner from the other parts of the system.

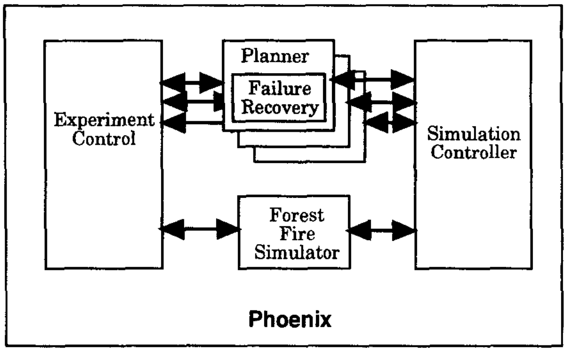


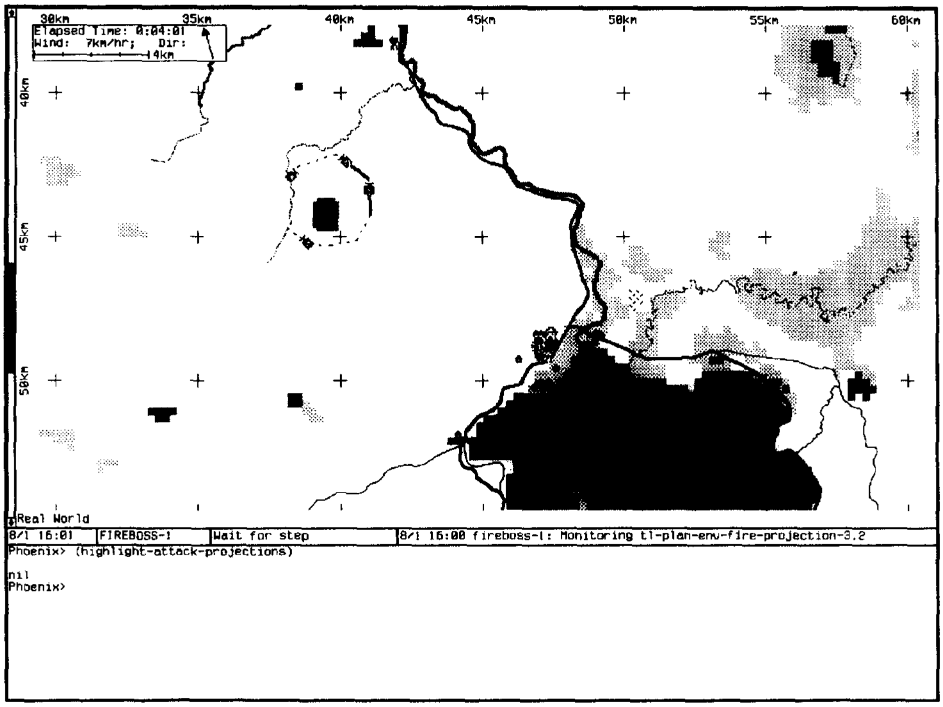
Fig. 7. Diagram of the separate processes that comprise the Phoenix system. The arcs between the processes indicate a transfer of data between processes.

to burn out, called building fireline. One agent, the fireboss, coordinates the activities of field agents, bulldozers, to surround the fire with fireline, exploiting natural boundaries whenever possible. Other agents, watchtowers, gasoline carriers, and helicopters, support the activities of the fireboss and bulldozers by gathering information and delivering gasoline to refuel agents in the field.

Fig. 8 shows the interface to the simulator. The map in the upper part of the display depicts Yellowstone National Park north of Yellowstone Lake. Features such as wind speed and direction are shown in the window in 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.

All Phoenix agents have the same agent architecture, which consists of sensors, effectors, reflexes and a planner. Sensors perceive the state of the environment local to the agent, and effectors change the environment. Together, reflexes and the planner form a two-layer control system. Each layer provides a particular level of competence. Reflexes address changes that occur faster than the cognitive component can respond, and the planner coordinates actions and avoids detrimental plan interactions. The planner plans by skeletal refinement; when new tasks are added to the timeline (a structure for developing and monitoring plans in progress), the planner searches its plan library for partially detailed, uninstantiated plans appropriate to the task and conditions of the environment. Plans are further expanded and instantiated as needed (e.g., as plans are executed or as information becomes available about the state of the environment).

By almost any measure, the Phoenix environment is challenging for AI planners. As a result, Phoenix plans fail for lots of reasons. The environment can change unpredictably, so if the planner bases a plan on slow or no change in the environment, the plan will fail. Phoenix agents are limited in their abilities to sense the environment; so plans can fail because they are based on obsolete or uncertain information. Phoenix plans also fail because they include bugs and have not been tested in all possible situations.



```
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>
```

Fig. 8. View from Phoenix simulator of bulldozers fighting a fire

4.2. *Phoenix-specific knowledge for FRA*

FRA uses three kinds of domain-specific knowledge: types of failures and recovery actions, suggestive structures, and explanations. At present, the Phoenix planner recognizes 11 types of failures and applies eight recovery actions to repair those failures. The failures types are listed in Table 3; the recovery actions in Table 4. While the failure types are obviously specific to the Phoenix planner, the recovery actions, suggestive structures and explanations are designed to generalize beyond Phoenix. We believe that many of them would apply to and characterize other planners as aptly as they do Phoenix, and we are in the process of acquiring other planners and analyzing them for whether these structures apply. Following Sussman's lead on plans [19], we envision defining canonical bugs and fixes for many classes of planners.

Suggestive structures

Suggestive structures are determined by the plan language. The Phoenix plan language is fairly impoverished in its representation of goals and effects of actions; it is largely a procedural language in which most of the reasoning is opaque. The suggestive structures for the Phoenix planner exploit what information is available in the structure of the plans. The following is a listing of the suggestive structures acquired so far for the Phoenix planner.

Table 3
Failure types for the Phoenix planner

CCP	Can't Calculate Path: The planner cannot find a safe path between two points on the map.
CCV	Can't Calculate Variable: This failure type is a catchall for being unable to assign a value to some variable.
CFP	Can't Find Plan: Planning operates by searching a plan library for one plan that fits the requirements of its context in the plan and is appropriate to the current state of the environment. This failure type is detected when no plan meets these criteria.
FNE	Fire Not Encircled: Each fire-fighting plan includes, as its last step, a check that the fire really has been contained; this failure is detected when the fire is not found to be fully encircled by fireline.
NER	Not Enough Resources: Fire-fighting plans estimate the resources required to contain particular fires; this failure is detected when the fire is too large for available resources.
NRS	No Remaining Segments: This failure indicates a mismatch in allocation of work to individual agents.
PRJ	Can't Calculate Projection: This failure is detected when the fire size cannot be projected for some future time, due to bugs in code or anomalies in available information.
PTR	Can't Calculate Path to Road: Normally, paths are calculated between two points; one variant on path calculation is to calculate a path from a point to the nearest road.
RU	Resource Unavailable: Resources can be assumed to be available during initial planning, but actually are unavailable when the plan later attempts to allocate them.
IP	Insufficient Progress: Envelopes detect differences between observed and expected progress in plans; this failure is detected when the observed progress is inadequate.
ZBT	Zero Build Time: In examining a particular fire, the planner may find that the fire is already contained and thus requires no fireline.

Table 4
Recovery actions for the Phoenix planner

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.
SP	Substitute one projection action for another which has failed.
AR	Assign additional resources to a plan.
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).

- *Shared Variables*: *One action sets some variable and another uses it.* Shared variables are vulnerable to failure for a variety of reasons: changes in value not getting propagated to all uses, mismatch in assumptions between how the value is set and used, and confusion about the units of a variable.
- *Shared Resources*: *Two actions allocate and use the same resource.* Separate plans may access the same resources. Resources refer both to physical resources such as bulldozers and fuel carriers and also to some types of data structures (global instance variables and shared frames). The Phoenix planner does not arbitrate resource use beyond first come first served.

- *Sequential Actions: One action is guaranteed to follow another in some plan.* Sequential orderings are vulnerable in that the progression from one action to the next is inexorable (barring the intercession of replanning). This structure is extremely weak as information about flaws in the planner; it is merely useful for determining whether one action typically precedes another in a plan as opposed to being completely unrelated.
- *Unordered Actions: Two actions are unordered by the plan.* Actions without ordering constraints in plans indicate that the designer expects the actions to be opportunistically interleaved by the planner at run time. Typically, the actions are independent (meaning their effects should be unrelated), but if for some reason (e.g., mistake on designer's part or modification by failure recovery) they are not independent, then the resulting order may produce interactions.
- *Iteration Constructs: Multiple actions are added to the plan by the same decision action.* Like most programming languages, the Phoenix plan language supports several iteration constructs. The assumption underlying several of these constructs is that we wish to take similar action over and over; for example, the language includes a `do-list` construct, which adds to the plan some number of the same action, each with a different variable binding. Because each of these is the same action, they share many of the same assumptions about the environment; if one action is vulnerable, the rest may be as well.
- *Repeated Actions: One action is included in many different plans and gets used in many different contexts.* Some planning actions are repeated many times in one or more plans. For example, path calculation actions are executed every time some agent is moved from one place to another. These oft repeated actions can be difficult to program (and so tend to be programmed incorrectly) because it may be difficult to predict in advance all the possible situations in which they may be used.
- *Environment References: One action senses the environment and passes the result onto another, or two actions share assumptions about the state of the environment.* Some plan actions sense the environment and calculate variables or make planning decisions based on the condition of the environment. If the decision or calculation assumes constancy of environmental conditions, then those calculations or decisions may be vulnerable to failure.

Explanations

Dependencies may result from one action causing a later failure or from incidental relations in plans. The catalog of explanations for the Phoenix planner identifies some of the structural influences on failure in Phoenix planner. As with the suggestive structures, this list is not intended to be complete, but rather to serve as a starting point for cataloging the ways that failure dependencies can be produced in Phoenix and other planners.

- *Overly Constrained Environment Assumptions: A sequence of plan actions assumes constancy of environmental conditions and thus is vulnerable to changes in those conditions.* Actions that assume stability in the environment may account for a higher frequency of certain types of failures: those resulting from changes in the environment.

A suggestive structure that suggests this explanation is environment references. Given this explanation, some planner modifications that might remove the dependency are: add monitoring actions to update the model of the environment more often, or reorder actions to coordinate actions that require the environment to stay the same.

- *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.* Some plan expectations are explicit, as in plan variables, and some are implicit, as in the assumptions underlying the values of the plan variables.

  Some suggestive structures that suggest this explanation are: sequential actions combined with shared variables, unordered actions combined with repeated actions, or iteration constructs combined with shared variables. Some modifications to the planner that might remove the dependency are: 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.
- *Resource Contention:Several plans may vie for the same resource with only one or none able to acquire or access it.* Resource contention is common when multiple agents or plans vie for the same limited resources. For the most part, the Phoenix planner treats on-going plans as independent, which minimizes the search for interactions and allows for reuse of plans, but makes resource contention more likely.

  A suggestive structure that suggests this explanation is parallel actions combined with shared resources. Some modifications to the planner that might remove the dependency are: institute a reservation policy (e.g., when a decision is made based on resource availability, immediately reserve the resources), create protection intervals so that plans cannot be interleaved between resource decisions and use, or sequence the contending plans.
- *Overcommitment:A set of actions may be determined so far in advance that the planner cannot change the sequence to accommodate change.* Plan designers trade-off efficiency and flexibility by moderating the amount of coordination built into plans. Sometimes the plan may be overly constrained such that it may discover information about the state of the world or may fail early on, but be so structured that the information cannot influence later decisions.

  Some suggestive structures that suggest this explanation are: parallel actions combined with shared variables or parallel actions combined with shared resources. Some planner and recovery modifications that might remove the dependency are: make the shared assumptions explicit so that the knowledge gleaned by one can be used in decision making by the other or change the planning so that a failure in one subplan means that a different choice is made for the later plan (i.e., do not try to do the same thing if it failed before).
- *Temporal Sequencing in Plans: When actions are strictly ordered in plans, failure occurrences may match the order.* A failure cannot occur if the action that detects it does not get executed. A failure that is detected early in the plan cannot follow one detected by later actions unless the plan is somehow restarted or several plans of the same type are interleaved.

A suggestive structure that suggests this explanation is sequential actions. Given this explanation, a planner modification that might remove the dependency is to replace the temporal structure with more opportunistic control.

- *High Base Frequency*: *Planning actions that have a high base frequency have more opportunity to detect failure.* Some planning actions are repeated many times in a plan. This can lead to a high base frequency for failures detected by these frequent actions and may lead to spurious dependencies or dependencies due to strict ordering effects.

  A suggestive structure that suggests this explanation is repeated actions. A planner modification that might remove the dependency is to create different versions of the action for different situations.
- *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.* In this sense, it causes the next failure because it makes it inevitable that some other symptom will be detected.

  Some suggestive structures that suggest this explanation are: R–F dependency combined with sequential actions and shared variables, F_xR–F_x dependency, or FR–F dependency combined with unordered actions and environment references. Some planner and recovery modifications that might remove the dependency are: limit the application of the suspect recovery action, force a replan earlier, add new recovery methods to repair the failure, or change the plan structure so that related failures can be identified.
- *Downstream Failures*: *Recovery actions may disrupt the flow of control between plan actions so that they cause later failures.* The recovery method may alter conditions or expectations of subsequent plan actions, thus causing them to fail. Recovery may cause later failures by making a poor repair or by not updating all plan variables that are related to the repair.

  Some suggestive structures that suggest this explanation are: R–F dependency combined with sequential actions, shared variables and a local repair method (R is one of WATA, RV, RAV, or SA) or FR–F dependency combined with sequential actions and environment references. Some planner and recovery modifications that might remove the dependency are: limit the application of the suspect recovery action, narrow the scope of the changes made by failure recovery to not produce the detrimental side effects or modify the plan description to make explicit the relationship between the repair and the subsequent failure.
- *Stealing*: *A recovery action that significantly modifies the expectations, constraints or assumptions of the plan may preclude a host of related failures, thereby making unrelated failures more likely to occur next.* When some failures are prevented or avoided by relaxing constraints in the plan or updating the internal model to match the environment, failures not related to these plan modifications may appear more frequent. In effect, the recovery method *steals* failures from the normal flow causing them to be replaced by others.

  Some suggestive structures that suggest this explanation are: R–F dependency or FR–F dependency where R is either a replan or a substitution action. The planner should not be modified to prevent it because, for the most part, this is a good effect.

5. Empirical evaluation

To determine whether FRA is feasible, we need to know whether the procedure works at all; in particular, can it help programmers of the Phoenix planner learn something about the Phoenix planner that they did not know before applying FRA? To determine whether the technique is *useful*, we need to assess how much information is gained by applying it and how much effort is required to gain that information. In this section, we demonstrate that FRA is feasible by applying FRA to the Phoenix planner, and we assess the usefulness of FRA by determining what information can be gained from different sets of execution traces for the Phoenix planner and by estimating the relationship between the amount of effort expended (i.e., how many execution traces are collected) and the sensitivity of the underlying statistics.

5.1. Demonstrating FRA in Phoenix

To demonstrate the feasibility of the FRA procedure, we applied it to help debug the current version of the Phoenix planner. First, we collected execution traces from 94 experiment trials (which included 968 failures) in which the Phoenix planner fought three fires over the course of about 60 simulation hours (we will refer to this set of execution traces as the "Base Case"). The fires were set at eight hour intervals and the wind speed and direction was allowed to change roughly every hour.

Second, we ran Lisp functions to analyze the execution traces for statistical dependencies between recovery efforts and failures. We found 46 dependencies: 15 R–F dependencies, 15 F–F dependencies, and 16 FR–F dependencies. From that set, we selected one for interpretation: $[R_{sp}, F_{ip}]$. We selected this dependency because the failure is a frequently occurring failure (F_{ip}) that is expensive to repair and the precursor includes a failure recovery method that had been previously added to improve recovery performance but seemed to be interacting detrimentally with other parts of the plan. This relationship (i.e., $[R_{sp}$ followed by $F_{ip}]$) was observed 52 times in the set of execution traces, which suggests that it was a common pattern of behavior for the planner.

Third, we ran Lisp functions to map the dependency to suggestive plan structures (as shown in Fig. 9). The dependency includes a recovery action as precursor, R_{sp}, and a failure as a successor, F_{ip}. R_{sp} transforms a failed indirect attack plan (abbreviated P_{ia}) into a repaired plan P'_{ia} by substituting a different type of fireline projection calculation action for the failed one. The Phoenix plan library includes three different actions for calculating projections: *multiple-fixed-shell* ($A_{p\text{-}mfs}$), *tight-shell* ($A_{p\text{-}ts}$), and *model-based* ($A_{p\text{-}mb}$). R_{sp} replaces one of these with another. Failure F_{ip} is detected when plan monitoring indicates that progress against the fire has been insufficient and not enough time remains to complete the plan. F_{ip} is detected by an envelope action (a structure for comparing expected to actual progress [11]) called *indirect-attack-envelope* (A_{env}). The three projection calculation actions and the envelope action appear together in three different indirect attack plans. All three indirect attack plans include the same suggestive structures: Shared Variable and Sequential Ordering. 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.

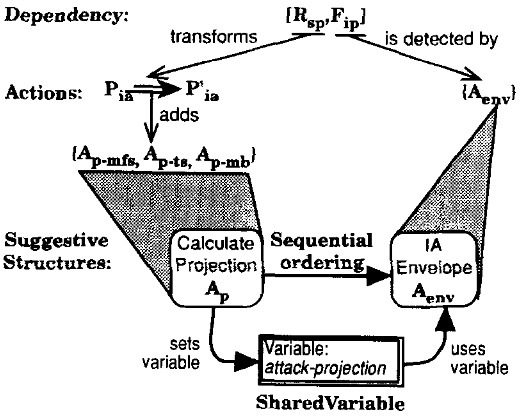


Fig. 9. Mapping a dependency to two suggestive structures.

Fourth, we looked up explanations of how the suggestive structures might have produced the observed dependency and identified possible modifications to repair the flaw. Combinations of suggestive structures lead to many explanations. The two suggestive structures found for the dependency $[R_{sp}, F_{ip}]$ (Shared Variable and Sequential Ordering) underlie two different explanations: Implicit Assumptions and Band-aid Solutions. The Shared Variable can cause a failure if the substituted projection calculation action sets the variable differently than was expected by the envelope action (thus, violating implicit assumptions of the envelope); the projection 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., is a band-aid, not a cure); the fire may be raging out of control or the available resources may really be inadequate for the task.

The results of the analysis are one dependency, two suggestive structures and two possible explanations for the dependency. At this stage in the FRA procedure, the designer decides what to do to change the planner. With FRA, the designer can look up stereotypical repairs, but still needs to select one and implement the repair. The stereotypical repairs for a band-aid solution are to limit the application of the recovery action R_{sp} or to add a new recovery action for the failure F_{ip}. In this case, the recovery action R_{sp} had been added to improve recovery performance in two expensive failures, F_{prj} and F_{ner}, removing it would set performance back to previous levels. Alternatively, adding a new recovery method might produce just as many new dependencies. For these reasons, we rejected these modifications as unlikely to be helpful.

The stereotypical repairs for implicit assumptions were more promising. In particular, the most constructive repair seemed to be to make the assumptions of the variable's use explicit by defining additional variables. We checked the projection code (the code involving the precursor of the dependency) and envelope code (the code that detects the failure type in the dependency) for the assumptions they make about the variable that they both use, `attack-projection`.

The projection code calculates an estimate or projection of the extent of the fire at some point in the future. The three versions of the projection calculation code differ in how they search for projections and in how they estimate the likely performance of the resources available to fight the fire (i.e., the combined time building fireline). The estimates of resource performance include many factors that are bundled together into a summary variable. The envelope code uses the summary variable to construct expectations of progress for the plan.

By examining the code, it became obvious that the performance estimates 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 variables for each of the capabilities; the envelope action then combines the separate variables to define expected progress.

As part of the changes just described, a few bugs were detected and fixed in the $A_{p\text{-}ts}$ code (one of the three projection actions used). The bugs almost certainly caused some of the F_{prj}'s previously observed in the data and had led to some projection actions being even more optimistic than intended (e.g., one of the activities had been estimated in minutes, but treated as seconds, leading to one parameter of the estimate being one sixtieth of its desired value). Because the parameters involved in these bugs were variables local to the projection code and were not recorded as part of the plan, it is unlikely that FRA could have found these bugs in the use of the parameters. FRA indicated that the projection code was buggy and suggested one way in which it was buggy—that the projection variables were not being set and used consistently. Consequently, we fixed these bugs because this was the aspect of the planner that FRA indicated needed to be fixed.

We can tell whether the modifications were effective if the R_{sp}–F_{ip} dependency is not detected in new execution traces and the overall rate of failures decreases. To test the modifications, we ran the modified planner in 87 trials in which three fires were set over 12 hour intervals and analyzed the execution traces for dependencies. The dependencies detected from the execution traces for these 87 trials included 12 F–F dependencies, 12 R–F dependencies and zero FR–F dependencies. The R_{sp}–F_{ip} dependency was not detected in the new execution traces, and the rate of failures went from 0.414 failures per hour[5] to 0.333 failures per hour (a z test on the differences in mean failures per hour yields a significant result, $z = -13.11$, $p < 0.0001$).

The modifications to the planner also resulted in different incidences of each failure type. Failure type F_{prj} accounted for 20.8% of the failures in the Base Case experiment and only 0.3% in the new execution traces. Similarly, the incidence of F_{ner} (a failure type that is also detected during projection calculation) decreased from 16% to 12%, and F_{ccp} decreased from 15% to 12%. Yet, the overall percentage of F_{ip} increased from 25% to 42%, and the counts of failures F_{nrs} and F_{ptr} in the execution traces increased significantly. Table 5 lists the incidence of failures for the Base Case and for

[5] Failures per hour was used as a measure because it was the most general measure of progress across all the activities of the agents.

Table 5
Failure counts for the Base Case and the modified planner execution traces

Failure type	Base Case		Modified planner	
	Number	%	Number	%
CCP	232	0.151	142	0.125
CCV	5	0.003	0	0
CFP	143	0.093	92	0.081
FNE	7	0.005	5	0.004
IP	381	0.247	475	0.418
NER	246	0.160	122	0.107
NRS	3	0.002	15	0.013
PFR	0	0	2	0.002
PRJ	321	0.208	1	0.001
PTR	76	0.049	147	0.129
RDU	122	0.079	129	0.113
ZBT	4	0.003	7	0.006
	1540	1.000	1137	1.000

the traces from the modified planner. Most of these changes in counts for different types of failures were probably due to removing the bugs in the code rather than implementing the changes to the projection calculation. An optimistic explanation for the increase in some failure types is that by removing the cause of some early plan failures (i.e., failures that can only be detected early in fire fighting plans), the plans are getting further and failures that are detected at later points in the plan are becoming more obvious. Without more data, it is difficult to know whether this optimistic explanation is correct. If the experiment were enhanced to record a measure of progress in the fire fighting plan when a failure is detected, then we might detect a general trend toward more progress.

This example shows that a designer can apply FRA to help identify and repair flaws in a planning system. Based on the dependencies, suggestive structures, explanations and modifications that were found while following the procedure for FRA, we were directed to examine and modify one portion of the code for the Phoenix planner. The modified planner detected fewer failures than previously and did not exhibit the dependency that was to be avoided by the modifications. Because we fixed bugs other than the one identified by FRA, it is impossible to conclude that the improvement is due to the change motivated by applying FRA. For a system of any complexity, it is virtually impossible to analyze code without finding some bugs. FRA pointed us to a source of problems, and we thought to fix every problem found there, whether or not it matched the explanation selected with the help of FRA. FRA was useful because it led us to examine a relationship that had been assumed previously to be correct.

As another example, we performed another cycle of FRA. First, we collected execution traces from 102 experiment trials (which included 1043 failures) under the same conditions as before. These new execution traces were collected because considerable time intervened between the previous example and this one during which one of the authors and the Explorer used for the experiment moved from University of Massachusetts to Colorado State University. The dependency sets in the new traces are similar (about 2/3 overlap), but the failure rate is significantly lower (went to 0.261 failures/hour from 0.333 failures/hour), due probably to some general fixes made in the course of installing the system in Colorado.

Second, we searched for dependencies and selected one for attention. We found 23 dependencies and from that set selected $[R_{rt}, F_{ner}]$. We selected the dependency by considering only those based on a reasonable amount of data (all cells in contingency table must be at least 5), ranking the dependencies by the value in their upper left cell, and then picking the most expensive repair cost of the top five in the ranking. So, $[R_{rt}, F_{ner}]$ is both common and expensive to repair. Third, we found suggestive structures for this dependency. R_{rt} is a replanning action which aborts the plan in progress and searches for a new plan to fight the fire. F_{ner} is detected during projection when the planner discovers that the available resources (the bulldozers) are inadequate to contain the fire. The suggestive structures found were: Shared Variables (setting and then using the variable describing the fire, also both reference the assessment of available bulldozers), Sequential Structure, and Environment References (both examine the state of the fire and the current weather conditions). Fourth, we found explanations: Overly Constrained Environment Assumptions and Implicit Assumptions. Because the variables involved referred to the state of the environment (e.g., the weather and the fire), we were inclined to combine the recommendations of the two explanations: make explicit the environment assumptions of the replanning decision and coordinate those assumptions with the projection code. Our intuition was that because the same criteria had been used to select a plan when the fire was first sighted as had been used in replanning, the replanning decision might be out of sync with the possibly more demanding state of the environment likely to hold during replanning.

Based on this analysis, we made a set of closely related changes to the Phoenix planner. We changed the criteria used to select plans, and we assigned variables based on some of the assumptions of the plan selection. Phoenix can select from five general fire fighting plans. Of those, two are already used only in extremely limited (easy) situations. Although they differed in their flexibility under varying environment conditions and their reliance on available resources, the remaining three had been chosen randomly. The selection criteria were modified so that the most stingy (able to use whatever resources were available) was selected when resources were tight, the most accurate (estimated many aspects of the state of the world) was selected when conditions differed considerably from average, and otherwise, one was selected at random. The characteristics of the three plans were assessed through pilot experiments, observation, and analysis of the code. Additionally, the assumption about the number of bulldozers available was made explicit at plan selection time as an assignment of bulldozers to the plan.

We ran the same conditions as before and collected execution traces from 82 experiment trials. We analyzed the results and found that the $[R_{rt}, F_{ner}]$ dependency had disappeared and the percentage of F_{ner} had reduced from 9% of the failures to 1%. However, the failure rate increased to 0.346 failures per hour. We have several choices at this point: We could back out this change and try a different solution; we could assume that the change needs to be better tuned and so run more pilot experiments to test the selection criteria, or we could examine the dependency set for ones that may have been introduced by this change and fix those. The decision is left to the programmer, but in all three cases, dependency detection can be used to assess the results and FRA can be used to suggest additional changes.

5.2. *Utility of dependencies*

We assess the utility of FRA by estimating the information gained relative to the effort required. We focus on dependency detection because it is the core technique of FRA; dependencies are the information that drives the rest of the procedure and the amount of effort required is dominated by collecting execution traces. We wished to determine the quality of the information: do dependencies capture weak or strong relationships and do dependencies summarize abstractly the interaction of the planner and its environment.

We estimate the information gained by describing how many and of what strength dependencies were found in a series of four experiments with Phoenix and by showing how the dependencies change as the planner is modified and as the time between the precursor and the antecedent is increased. In other words, we show that dependencies characterize strong effects and that the dependencies reflect what the planner does and how much time passes between the planner's actions and the dependent event. We estimate the effort required to detect dependencies by estimating the sensitivity of dependency detection to the amount of data available and the noise in the data and by summarizing the amount of computation required.

Our measurements are based on a set of four sets of execution traces of the Phoenix planner, which were gathered during a series of experiments. In the first three experiments, the failure recovery portion of the Phoenix planner was incrementally modified. The third experiment is the Base Case used in Section 5.1, the fourth experiment involved the modified planner described in the same section.

5.2.1. *Information gained*

As the planner and its environment changes (or is changed), the type and strength of dependencies changes as well. Prior to gathering the two sets of execution traces described in Section 5.1, we gathered execution traces from two previous versions of the planner (these two versions had successively simpler versions of failure recovery) and analyzed them for dependencies.

Amount of information

If dependency detection discovers too many dependencies, then the designer will be swamped with information that by its volume becomes useless. If most of the dependencies are strong (meaning the probability that they are due to noise or chance is low), then the designer can have more confidence in dependencies as an accurate characterization of a relationship between precursor and failure type.

Different dependencies were detected in the execution traces for each of the four versions of the planner. In terms of evaluating whether the designer is likely to be swamped in dependencies, it appears that, while the dependencies detected in each experiment's execution traces were different, the number of dependencies detected was not overwhelming. The most dependencies detected in any one experiment was 46 for the Base Case used in Section 5.1; the fewest dependencies detected was 24. The total

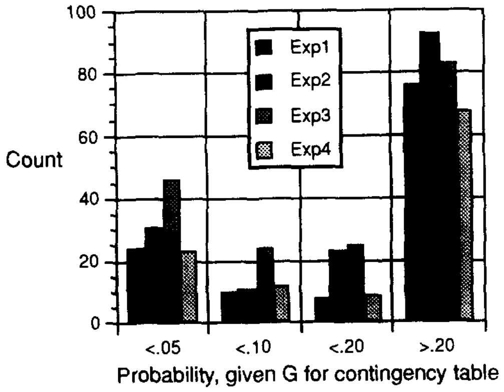


Fig. 10. Histogram of p values for all precursor and failure combinations (F–F, R–F and FR–F) in four data sets. p values are the probabilities that the G value for a contingency table was due to noise or chance.

for all four sets of execution traces was 125. To get a sense for how many dependencies are likely to be detected, Fig. 10 shows the distribution of p values for four data sets. We say that we detect a dependency between a precursor and a failure type if the p (probability that G was due to chance) is less than α, where $\alpha = 0.05$. As one would expect, the overwhelming majority of p values are $> \alpha$. In the figure, each column is the count of precursor–failure combinations with p between the upper limit of the previous column and the limit listed at the bottom of the column; so < 1.0 means the count of $0.05 < p < 1.0$.

The strength of a dependency can be measured in terms of the probability that the ratios observed for the dependency arose due to chance or noise. The majority of the dependencies detected in the four sets of execution traces were well below the α threshold. In fact, the histogram in Fig. 11 shows that over half of the dependencies had $p < 0.01$, indicating that the dependencies that were detected were highly significant.

Sensitivity of the dependencies to planner version

To determine whether the dependencies reflect the changes being made to the planner, we tested the overlap in the dependencies detected in the four sets of execution traces. Additionally, we tested the temporal persistence of dependencies by analyzing the execution traces for dependencies between a precursor and a failure that occurs much later in the execution trace.

The incidence of failure types changed across the four experiments. Consequently, we expect the dependencies detected in the execution traces for each experiment to change as well. We tested this expectation by comparing the sets of dependencies detected across combinations of the experiments and counting the overlap. The results are summarized in Table 6.

The planner used in experiment four is a variant of the planner used in experiment three, which is a variant of what was used in experiment two, which is a variant of what was used in experiment one. Thus, one would expect adjacent experiments (i.e., one and two, two and three, three and four) to share relatively many dependencies,

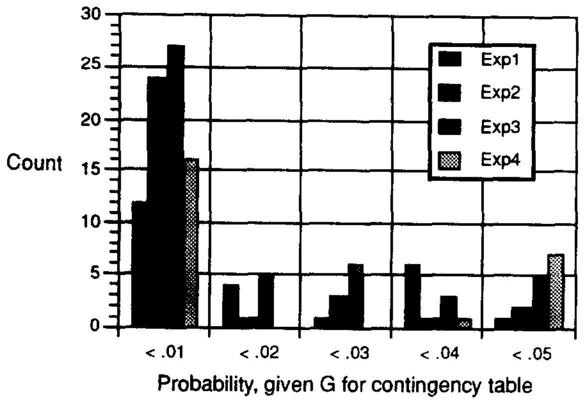


Fig. 11. Histogram of p values for dependencies. The dependencies were detected in execution traces from four experiments.

Table 6
Number of dependencies shared by combinations of experiments

	Experiment combinations								
	1&2	2&3	3&4	1&3	2&4	1&4	1,2&3	2,3&4	1,2,3&4
R-F	1	1	4	0	0	0	0	0	0
F-F	9	5	4	2	2	2	2	2	1
FR-F	1	1	0	0	0	0	0	0	0
Totals	11	7	8	2	2	2	2	2	1

and non-adjacent experiments to share relatively few dependencies. Data from different experiments should share dependencies because the dependencies capture the structure of some interactions within the planner and between the planner and the environment; thus, the more that the planner changes, the more the dependencies should change. In fact, the execution traces for the four experiments exhibit just this phenomenon: experiments one and two had the most dependencies in common; the full set (one, two, three and four) had the fewest in common.

Sensitivity of the dependencies to temporal separation

Dependencies represent downstream influences of precursors on *later* failures. A R_{sp}–F_{ip} dependency indicates the R_{sp} influences or makes more likely the occurrence of F_{ip} as the next failure. Detecting "downstream influences" implies that we expect the effects of precursors (i.e., failure types and recovery methods) to have some persistence, to last long after the actions that detected the failures or repaired them have finished. In evaluating whether dependencies capture those downstream influences, we need to determine for *how long* (i.e., how many steps in the execution traces) the precursors influence failures downstream.

FRA detects dependencies between a precursor and the failure that immediately follows it. The algorithm was designed to detect dependencies between the precursor and the failure immediately following it for pragmatic reasons—it limited the number of patterns that needed to be tested by the detection code. Dependency detection runs G tests on *all possible* patterns of precursors and failures. Thus, the number of patterns increases combinatorially as we add earlier failures and recovery methods to the precursors, making it computationally expensive to consider longer and earlier precursors.

Practically, we need to know whether we are missing some downstream influences when we limit the patterns to include only the previous failure. We can divide this concern about missing information into two parts: Should precursors include more than just a single pair of a failure and the recovery method that repaired it? Do precursors influence failures after the next failure? To answer the first question, we modified dependency detection to test much longer patterns, patterns of more than one failure and more than one recovery method, and we looked at whether dependencies are detected at all as the precursors become longer and more complicated. To answer the second question, we modified dependency detection to test dependencies between a failure and a much later failure, for example, two, three, or four failures later. As with the first question, we looked at whether dependencies still are detected as the temporal separation increases between the precursor and the failure.

FRA tests for dependencies of types F–F, R–F and FR–F for all failures and recovery methods that appear in the execution traces. For example, the execution traces for experiment 1 contained 10 different failure types and six different recovery methods, so there were 600 possible patterns of FR–F or 600 different patterns of length two to be tested. Given the number of different failure types in the execution traces and the number of different recovery methods, we can calculate the number of possible patterns for any length precursor. For example, a precursor of length three (up to RFR–F) produces 4620 possible patterns: 3600 RFR–F, 600 $RF*$–F, 360 $R*R$–F, and 60 $R**$–F. The $*$ stands for a wild card value, meaning it does not matter what value is in that position so long as something in the execution trace appears between the other values. $R**$–F means a particular recovery method R followed by any failure, then by any recovery method and finally by a particular failure F.

Testing all possible patterns up to $FRFR$–F would require 59,280 tests for the execution traces for the first experiment. The execution traces for all four experiments together included only 3900 failures (collected over 15,000 simulation hours), which suggests that even if computation time were not a concern, the execution traces could not even contain all the patterns. So, we need to determine whether it is even worth testing for the longer patterns in the execution traces. We did so by modifying the dependency detection code to test any length precursor and examining the rates of detection. Table 7 shows the results of testing for dependencies with precursor up to length four (i.e., $FRFR$–F) in the execution traces for experiment 1. Clearly, the number of possible combinations of failure types and recovery methods quickly outpaces the size of the execution traces, but as the length of the precursor increases, the percentage of dependencies detected for the patterns found in the execution traces decreases (23% for R–F to 9% for $FRFR$–F).

Table 7 suggests that, as one might expect, the influence of the precursors decays over time. A smaller percentage of long precursors are detected as dependencies than of short precursors. For example, G/Found (the number of significant combinations divided by the number of different combinations found in the execution traces) is

Table 7
Dependency search space for precursors up to *FRFR*

	Combinations	Found	*G* test	*G*/Combinations	*G*/Found
R–F	60	40	9	0.150	0.23
FR–F	700	160	36	0.050	0.23
RFR–F	4620	531	50	0.011	0.09
FRFR–F	53900	1737	162	0.003	0.09
Totals	59280	2468	257	—	—

Table 8
Dependency search space for temporally separated failures. Execution traces are from the first experiment

	Combinations	Non-empty	*G* test	*G*/Comb.	*G*/Non-empty
F–F	760	225	43	0.06	0.19
*F**–*F*	760	220	20	0.03	0.09
F **–*F*	760	207	33	0.04	0.16
F ***–*F*	760	199	32	0.04	0.16

0.23 for the short precursor *R* and decreases to 0.09 for the longest precursor *FRFR*; *G*/Combinations (the number of significant combinations divided by the number of combinations of the values for *F* and *R*) decreases from 0.15 to 0.003, indicating that the dependencies account for a small proportion of all the possible long combinations. The decrease in percentages suggests that by limiting the length tested to just two, we are not significantly limiting the proportion of dependencies found. Yet, the program did still detect 162 dependencies of length four, a count considerably higher than the 9 dependencies found for precursors of length 2. So while the numbers are relatively lower, we could still be missing dependencies if we do not test for longer dependencies.

Alternatively, we can evaluate whether the influence of precursors diminishes over time by looking at simpler precursors, just failures, gathered over longer temporal separations. The same data set (the earliest set of execution traces) was analyzed for dependencies of $F_xF^*\text{–}F_y$, meaning pairs of a particular failure F_x followed by some number of failures followed by a particular failure F_y. Table 8 shows the results for temporal separations of up to four failures. It appears that as the distance from the precursor to the failure increases, the number of patterns decreases slightly and the proportion of significant dependencies decreases slightly (19% for *F–F* and 16% for *F* ***–*F*). Because the decrease in the proportion of significant dependencies is so small, the only conclusion is that we are likely to be missing dependencies by not considering dependencies over longer intervals. The future work section describes some proposed approaches to finding dependencies over longer intervals.

5.2.2. *Effort required: sensitivity of dependency detection to size of data set*

The *G* test is a statistical test, which means that it is sensitive to the amount of data available. For the *G* tests performed as part of dependency detection, the amount of data refers to the number of patterns in the execution traces. For example, the execution trace,

$$F_{prj} \rightarrow R_{sp} \rightarrow F_{ip} \rightarrow R_{rp} \rightarrow F_{ner} \rightarrow R_{sp} \rightarrow F_{ip} \rightarrow R_{rt} \rightarrow F_{prj} \rightarrow R_{sp} \rightarrow F_{nrs},$$

includes five FR–F patterns. The contingency table for the pattern $F_{prj}R_{sp}$–F_{ip} constructed from this execution trace is shown in Table 9. The total number of patterns is five. The ratio for the pattern $F_{prj}R_{sp}$–F_{ip} in the execution trace is 1:1 (i.e., the ratio of the count of the precursor followed by the target failure to the count of the precursor not followed by the target failure) and the ratio for any other precursor ($\overline{F_{prj}R_{sp}}$) being followed by the failure (F_{ip}) is 1:2. The difference between those two ratios is not sufficient to detect a dependence between the precursor and the failure; a G test on this contingency table yields $G = 0.236$, $p < 0.627$. If we had 20 times more patterns in the execution traces (i.e., 100 patterns in the traces) with the same ratios (i.e., 20:20 and 20:40), a G test would be significant, $G = 4.711$, $p < 0.03$, and a dependency would be detected.

Table 9
Contingency table for the dependency R_{sp}–F_{ip}

	F_{ip}	$F_{\overline{ip}}$
$F_{prj}R_{sp}$	1	1
$\overline{F_{prj}R_{sp}}$	1	2

The total number of patterns and the ratios of the patterns in a contingency table influences the results of the G test. Imagine that you have a significant contingency table ($p < \alpha$), there are two ways to change the table that may change the result (i.e., whether p is still less than α): vary the total count of the table (i.e., n) and vary the ratios (top row to bottom row). The first change addresses the sensitivity of the test to the size of the execution traces; the second addresses the sensitivity to noise: how much of a difference is required to detect a dependency no longer? The two changes are not independent, but to understand the influence of each, we examine them separately. The sensitivity of the test to the size of the execution traces is examined analytically, and the sensitivity to noise is examined empirically in this section.

To determine the effect of the sample size on detecting dependencies, we will examine the equation underlying the G test used to detect dependencies, the heterogeneity test. The nature of the G test is that the G values for subsets of the sample can be added together to get a G value for the superset (this was the property exploited in pruning). If the ratios 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. For example, the G value for the contingency table in Table 9 is 0.236; the G value for the contingency table with 20 times more patterns is 4.711 or roughly (as close as one gets with round off error) 20 times 0.236.

To explain how this arises, consider an alternative, but mathematically equivalent form of the equation for heterogeneity:

$$G = 2\left[a\ln\left(\frac{a}{\hat{p}_f}\right) + b\ln\left(\frac{b}{\hat{p}_{\overline{f}}}\right) - (a+b)\ln(a+b)\right],$$

where a is the count from the upper left of the contingency table, b is the count from the upper right of the contingency table, c is the count from the lower left of the contingency table, d is the count from the lower right of the contingency table, $\hat{p}_f = c/(c+d)$ and $\hat{p}_{\bar{f}} = d(c+d)$. Assuming we hold constant the ratios (a:b and c:d) but vary the total count $(a+b+c+d)$, we can replace b with $x * a$ (making a ratio of a:$x * a$) and c with $y * d$ (making a ratio of c:$y * c$) and simplify the equation to:

$$G = 2a[(1+x)\ln(1+y) + x\ln\frac{x}{y} - (1+x)\ln(1+x)]. \tag{2}$$

which illustrates that if we hold the ratios constant (i.e., x and y) and vary only a to increase the total, then the value of G increases linearly with increases in a. This explanation is valid only for the simple version of G (heterogeneity) used to detect the dependencies in the first place.

Linearity is desirable because the effect of more data is predictable and because it tends to reduce the likelihood of surprising results if a few more trials are collected. We know that as more execution traces are collected, the more likely we are to detect dependencies, but that the new dependencies detected are likely to have been borderline before the addition of data. The bottom line is that the G test can find strong dependencies given execution traces with few patterns, but given more patterns it will also find rare dependencies. So if a user of FRA is interested in detecting any dependencies, then a few execution traces will be adequate to do so; if the user wishes to find rare or obscure dependencies, then it will be necessary to gather more execution traces. The level of effort expended in gathering execution traces depends on what kinds of dependencies one wishes to find.

The preceding analysis tells us how G changes if we expend the effort to gather more execution traces, but the analysis assumes that the ratios in the contingency table are constant. Consequently, we cannot say much about the effect of noise given only a few patterns in execution traces. To rephrase the concern, one of the problems with fewer patterns in execution traces is that any particular failure may be a random event. Formal analysis of this concern is difficult. Instead, we tested the effect of noise empirically by inserting random events into the execution traces and testing whether the dependencies found in the original traces remained. The counts are shown in Table 10. About 65% of the dependencies remain after introducing noise. This means that 35% of the dependencies detected would disappear if a few patterns more or less were included in the execution traces. As one would expect, most of the dependencies that were vulnerable to the tweaking were based on execution traces that included few instances of the precursor/failure pattern: 23 out of 44 or 52% of the dependencies that disappeared were based on contingency tables in which $f \leqslant 5$. The implication of the sensitivity of dependency detection to noise in the execution traces is that rare patterns (i.e., patterns based on few instances in the execution traces) are especially sensitive to noise and so should be viewed skeptically.

When evaluating the cost of executing dependency detection, we need to consider two factors. First, dependency detection is simple and fully automated; the calculations for each test are fast and can be run in batch mode. The computation time required to collect the dependencies in the four sets of execution traces was far shorter than the

Table 10
Dependencies remaining after tweaking contingency tables. Contingency tables were tested for sensitivity to minor changes in ratios. The table includes the number of dependencies remaining after tweaking over the total number of dependencies found in the execution traces from the four experiments.

	Experiment 1	Experiment 2	Experiment 3	Experiment 4
R–F	0/4	4/8	10/15	7/12
F–F	9/13	15/19	7/15	10/12
FR–F	5/7	3/4	11/16	0/0
Total	14/24	22/31	28/46	17/24

time required to gather the execution traces in the first place (less than ten minutes for dependency detection, but about two weeks for the data collection). Second, the complexity of the dependency detection algorithm is mitigated by a practical reduction in the number of patterns to be considered at each step. The only patterns considered are those that appeared in the execution traces; for the four experiments, the number of patterns observed was only four-fifths of those possible. Of those that remained, the first step in dependency detection reduced the set by two thirds, and the second step by another half. From a practical standpoint, the cost of executing dependency detection is fairly low.

6. Future work

We have described a technique for identifying and explaining dependencies between a planner and its behavior and have applied it to explain sources of failure in the Phoenix planner. Obviously, this is just the beginning. We envision three directions for future research. First, we intend to expand the definition of dependencies to encompass longer time intervals and longer combinations of actions and events. Second, we will examine how dependencies characterize environments and explore whether they can be used as "markers" for identifying similarities between apparently different environments. Third, we will apply dependency interpretation to planners other than Phoenix and behaviors other than plan failures.

6.1. Detect longer dependencies

Dependency detection is based on the assumption that the most recent precursors are most likely to influence which failure occurs next, or to phrase it differently, that a precursor's influence does not persist beyond the next failure. Empirical evidence suggests that we probably are missing dependencies by not extending the temporal extent of precursors. Unfortunately, the combinatorial nature of dependency detection appears to preclude identifying arbitrarily long sequences of significant precursors. Unless we gather incredibly long execution traces, we quickly run out of instances of each pattern of precursor and failure. Additionally, considering more patterns means consuming more computation time.

However, the combinatorics of dependency detection are based on searching for *arbitrarily* long sequences. We can manage the complexity by focusing the search for long sequences; rather than searching the execution traces for any pattern of precursor and failure, we search for particularly interesting ones. Sets of longer dependencies can be accumulated either by controlling the collection of data to selectively test for particular dependencies (i.e., experiment design) or by heuristically controlling the construction and comparison of dependencies (i.e., supplementing the G test).

A new experiment design would selectively eliminate recovery methods or plan actions to test whether each precipitates or avoids particular failures [6]. For the analysis, we would remove an action or recovery method from consideration, which results in execution traces free from interactions with the missing action. Consequently, rather than examining all possible chains of which some method is a member, dependency detection would involve comparing dependency sets generated from execution traces with and without each action. By comparing the dependency sets generated in this way, one can infer which dependencies were due to interactions with the missing methods. For example, if an action, say R_{sp}, is removed and the frequency of F_{ip} relative to other failures decreases, then the analysis should determine how the R_{sp} might have produced the additional F_{ip} failures: Was R_{sp} itself producing the failures? Does R_{sp} in conjunction with other recovery methods, (e.g., R_xR_{sp}–F_{ip} dependencies) account for the surplus F_{ip} failures? Does R_{sp} influence F_{ip} over longer intervals?

Briefly, with the new experiment design, we can determine which of the possible explanations account for the additional F_{ip} failures by comparing the counts of particular failures, with and without the action, for whether changes in the counts differ uniformly and depend on the previous failure. If the counts differ uniformly, then we can conjecture that the difference is due solely to the action that was removed; otherwise, we check further for how the failure counts compared to the counts when other actions were removed, looking for cases where actions behaved similarly (if action X and action Y interact, then removing either one should produce similar results).

Longer combinations require two changes to the application of the G test: determining pools and partitions for longer precursors and comparing the results of separate G tests. The pool and partitions represent different hypotheses about what is producing the observed dependency; for example, does F_x influence F_y or does F_x in conjunction with particular recovery methods, say R_a and R_b, influence F_y? If we consider all possible pools and their partitions, then we are faced with a combinatorial algorithm. If we consider pools and partitions based on whether one or the other was significant in other applications of the G test, then we may use results on smaller chains to determine which larger chains to explore. Our intuition suggests that if a set of partitions is found to account for little of the variance in a pool, then it is probably not worth looking at longer precursors that include the partitions. For example, if R_{sp}–F_{ip} is favored over F_xR_{sp}–F_{ip} based on a G test, then longer precursors, such as $F_vR_wF_xR_{sp}$–F_{ip}, are unlikely to account for much variance either. To further develop this approach, we need to run the G test on pools of pairs (e.g., FR–F) and partitions of triples (e.g., RFR–F and FFR–F) (and probably longer pools and partitions) and test whether the intuition is supported by the data. Then, we need to derive heuristics for selecting pools and partitions based on the results of previous G tests.

Another approach to finding longer dependencies is to rephrase the problem from finding all possible long dependencies to finding many highly significant dependencies. In this case, we use local search techniques to explore the large space of possible long dependencies. By using local search, we will find the strongest of the related dependencies (and so need not do the pruning step), can easily tune the amount of time spent searching by modifying the number of random starts, and can define search operators that match our intuition about what constitutes a neighborhood of dependencies.

Working out the logistics for detecting longer dependencies is the first step toward broadening the set of events included in execution traces. Execution traces for FRA include only failures and recovery methods; yet, many other events and actions influence the types of failures that occur. Instrumentation is available in the Phoenix system to collect other influences (e.g., changes in the weather, initiating new plans, and observations of new fires), but currently FRA cannot analyze such execution traces. Enhancing dependency detection to consider longer chains means that more events can be added to the execution traces without being inundated with possible dependencies.

6.2. *Construct equivalence classes of environments*

Execution traces from the four experiments yielded different dependencies. The dependencies were not only different, but the degree of difference appears to depend on *how much* failure recovery and the planner differed from one experiment to the next. Based on analyzing the execution traces for the Phoenix planner, similar versions, i.e., versions that differed by a "single" change in the implementation, of failure recovery result in similar sets of dependencies.

Comparisons of dependencies detected from execution traces of pilot studies suggest that dependencies are also sensitive to the degree of similarity in the environment. Perhaps, some dependencies function as markers for particular characteristics of environments. For example, severely resource-constrained environments may be characterized by many repetitions of resource contention failures, producing the observed dependency that one resource contention failure leads to another. Additionally, failure recovery methods that perform short-term load balancing to reduce the contention for one constrained resource may simply cause a different type of resource contention failure, producing a dependency between short-term load balancing recovery methods and particular types of resource contention failures. One would expect these dependencies to appear in any type of resource-constrained environment, however superficially different, whether it is a transportation planner, an air traffic control system, or a forest fire fighter dispatcher. Looking for such markers requires a lingua franca so that dependency sets for different environments can be compared. One option is to describe a hierarchy of failures and methods from general classes, such as "resource contention", to domain specific instances (e.g., "bulldozer unavailable" in Phoenix).

The benefit of constructing equivalence classes of environments is that we can predict more easily how a given planner will perform in any environment in the equivalence class. From a design standpoint, we can design planners for new environments by borrowing heavily from previous designs known to work well in similar environments. From a scientific standpoint, we will be able to tell when differences in environments are superficial, allowing us to compare planners with knowledge bases designed for different environments.

6.3. Apply dependency interpretation to other systems

Applying Dependency Interpretation to other systems means enhancing the other systems to collect execution traces and acquiring supportive knowledge specific to the other systems. For planners embedded in simulated environments, collecting execution traces should be supported by the simulator already; for planners that are not in simulated environments, the planner itself may need to be augmented to collect the execution traces. Phoenix-specific knowledge is applied at each step of FRA; for a new system, we need to acquire two types of knowledge:

- environment event types and plan actions for the planner,
- suggestive plan structures and explanations for the new planner.

The first should be easy because most planners have pre-defined methods for recognizing salient environment events. The second is the more difficult. The intuition behind suggestive structures is that people who program particular systems have structures that they check first when trying to track down bugs or understand why a program behaves as it does. For example, with resource contention failures, one would probably look first at the structures involved in managing the resources: pairings of reserve/release actions, actions that consume resources and decisions about resources and the preconditions on those actions. We have recorded some of the structures that are most suspect for the Phoenix planner and believe that many of these structures apply equally well to other planners. We intend to test this conjecture.

7. Conclusion

Part of the challenge of designing new planners is understanding when and why they work. The lessons of previous systems are the basis of future designs. Unfortunately, as our planners and their host environments become increasingly complex, they become increasingly difficult to understand.

This paper describes our first experiments with a method for understanding the behavior of planners. The approach combines a domain-independent, syntactic technique for summarizing behavior and identifying interesting patterns, with a domain-dependent, semantic technique for interpreting those patterns. The strength of the approach lies in the application of statistical techniques to reduce the tremendous amounts of execution information down to salient patterns and in the reliance on weak domain/planner knowledge for interpretation. Statistical dependency detection prunes an overwhelming and largely uninteresting space of behavioral data, and dependency interpretation exploits knowledge of the interactions of small parts of the planner to explain behavior of the system as a whole.

The future of planning depends on our ability to explain the behavior of increasingly complex systems. For us, this has meant changing our focus from explaining individual decisions in particular planning episodes to explaining statistical anomalies across many episodes. Also, our explanations are not in terms of highly specific state information, but, rather, in terms of general plan structures and programming idioms. Consequently, we cannot debug the Phoenix planner given only the execution trace of a single planning episode, nor can we pinpoint the source of a bug to a particular line of code. Nor can anyone else, however. We think it unlikely that anyone will write a program to automatically localize bugs in arbitrary complex systems, given execution traces that may or may not be buggy. Dependency detection and interpretation, on the other hand, find and explain statistical anomalies that might indicate bugs or planners that are particularly well designed for their environments, well enough for a programmer to easily find previously unsuspected bugs and a designer to extrapolate a good design to a new environment.

Acknowledgments

This research was supported by ARPA-AFOSR contracts F49620-89-C-00113 and F30602-93-C-0100, a grant from the Office of Naval Research under the University Research Initiative N00014-86-K-076, the National Science Foundation under an Issues in Real-Time Computing grant, CDA-8922572, and National Science Foundation Research Initiation Award #RIA IRI-9308573. 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 the first author's Ph.D. Thesis research at the University of Massachusetts. We wish to thank Cynthia Loiselle for carefully reading and translating portions of the document to LaTex and David Hart and David Westbrook for helping run experiments with Phoenix. We also wish to thank the reviewers for their careful review and insightful comments.

References

[1] P.C. Bates and J.C. Wileden, High-level debugging of distributed systems: the behavioral abstraction approach, Department of Computer and Information Science 83-29, University of Massachusetts, Amherst, MA (1983).

[2] S.W. Bennett, Learning approximate plans for use in the real world, in: A. Segre, ed., *Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, NY (Morgan Kaufmann, San Mateo, CA, 1989) 224-228.

[3] L. Birnbaum, G. Collins, M. Freed and B. Krulwich, Model-based diagnosis of planning failures, in: *Proceedings AAAI-90*, Boston, MA (1990) 318-323.

[4] L.J. Burnell and S.E. Talbot, Incorporating probabilistic reasoning in a reactive program debugging system, in: *Proceedings Ninth Conference on Artificial Intelligence for Applications*, Orlando, FL (1993) 321-327.

[5] S.A. Chien, Learning by analyzing fortuitous occurences, in: A. Segre, ed., *Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, NY (Morgan Kaufmann, San Mateo, CA, 1989) 249-251.

[6] P.R. Cohen, An experiment to test additivity of effects, Technical Report Memo No. 20, University of Massachusetts, Experimental Knowedge Systems Laboratory, Amherst, MA (1991).

[7] P.R. Cohen, *Empirical Methods for Artificial Intelligence* (MIT Press, Cambridge, MA, 1993).

[8] P.R. Cohen, M. Greenberg, D.M. Hart and A.E. Howe, Trial by fire: understanding the design requirements for agents in complex environments, *AI Mag.* **10** (3) (1989) 32-48.

[9] N.K. Gupta and R.E. Seviora, An expert system approach to real time system debugging, in: *Proceedings IEEE Computer Society Conference on AI Applications*, Denver, CO (1984) 336–343.

[10] K.J. Hammond, Explaining and repairing plans that fail, in: *Proceedings IJCAI-87*, Milan, Italy (1987) 109–114; also: *Artif. Intell.* **45** (1990) 173–228.

[11] D.M. Hart, P.R. Cohen and S.D. Anderson, Envelopes as a vehicle for improving the efficiency of plan execution, in: K.P. Sycara, ed., *Proceedings Workshop on Innovative Approaches to Planning, Scheduling and Control* (Morgan Kaufmann, San Mateo, CA, 1990) 71–76

[12] A.E. Howe, Accepting the inevitable: the role of failure recovery in the design of planners, Ph.D. Thesis, University of Massachusetts, Department of Computer Science, Amherst, MA (1993).

[13] A.E. Howe and P.R. Cohen, Failure recovery: a model and experiments, in: *Proceedings AAAI-91*, Anaheim, CA (1991) 801–808.

[14] E. Hudlicka and V. Lesser, Modeling and diagnosing problem-solving system behavior, *IEEE Trans. Syst. Man Cybern.* **17** (3) (1987) 407–419.

[15] L.F. Pau, *Failure Diagnosis and Performance Monitoring*, Control and Systems Theory **11** (Marcel Dekker, New York, 1981).

[16] R.G. Simmons, A theory of debugging plans and interpretations, in: *Proceedings AAAI-88*, Minneapolis, MN (1988) 94–99.

[17] R.R. Sokal and F.J. Rohlf, *Biometry: The Principles and Practice of Statistics in Biological Research* (Freeman, New York, 2nd ed., 1981).

[18] M. Stefik, Planning with constraints (MOLGEN: Part 1), *Artif. Intell.* **16** (1981) 111–139.

[19] G.A. Sussman, A computational model of skill acquisition, Technical Report Memo no. AI-TR-297, MIT AI Lab, Cambridge, MA (1973).