8
Detecting and Explaining Dependencies in Execution Traces

Adele E. Howe and Paul R. Cohen

Computer Science Department
Colorado State University
Fort Collins, CO 80523
howe@cs.colostate.edu and

Experimental Knowledge Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
cohen@cs.umass.edu

ABSTRACT AI systems in complex environments can be hard to understand. We present a simple method for finding dependencies between actions and later failures in execution traces of the Phoenix planner. We also discuss failure recovery analysis, a method for explaining dependencies discovered in the execution traces of Phoenix's failure recovery behavior.

Dependencies are disproportionately high co-occurrences of particular precursors and later events. For the execution traces described in this paper, the precursors are failures and failure recovery actions; the later events are later failures. In complicated environments, it can be difficult to know whether actions produce long-term effects, in particular, whether certain actions cause or contribute to later plan failures. Statistical techniques such as those discussed in this paper can help designers determine how recovery actions affect the long-term function of a plan and whether recovery actions are helping or hindering the progress of plans.

8.1 Identifying Contributors to Failure in Phoenix

Phoenix is a simulated environment populated by autonomous agents. It is a simulation of forest fires in Yellowstone National Park and the agents that fight the fires. Agents include watchtowers, fuel trucks, helicopters, bulldozers and, coordinating (but not controlling) the efforts of all, a fireboss. Fires burn in unpredictable ways due to wind speed and direction, terrain and elevation, fuel type and moisture content, and natural boundaries such as rivers, roads and lakes. Agents behave unpredictably, too, because they instantiate plans as they proceed, and they react to immediate, local situations such as encroaching fires.

Lacking a perfect world model, neither a Phoenix planner nor its designers can be absolutely sure of the long term effects of actions: Does an action interact detrimentally with a later action in the plan? Will an action provide short term gain with long term loss? Are failures caused by

[0]This research was supported by DARPA-AFOSR contract F49620-89-C-00113, the National Science Foundation under an Issues in Real-Time Computing grant, CDA-8922572, and a grant from the Texas Instruments Corporation.

[1]*Selecting Models from Data: AI and Statistics IV.* Edited by P. Cheeseman and R.W. Oldford.

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

TABLE 8.1. Contingency table for testing the pattern $R_{sp} \rightarrow F_{ip}$.

a mismatch between the planning system and its environment? These questions are extremely difficult to answer for large, complex systems, and yet, these are precisely the systems in which detrimental interactions and failures are most likely [Corbato 91]. To identify the sources of failure and expedite debugging, we have developed a technique called *Failure Recovery Analysis* (FRA) [Howe 92]. FRA detects dependencies between failure recovery actions – those taken to recover from plan failures – and later failures. FRA also explains how some failure recovery actions might have caused later failures.

FRA involves four steps. First, execution traces are analyzed for statistically significant dependencies between failure recovery actions and subsequent failures; we call this step dependency detection. The remaining three steps explain failures by using the dependencies to focus the search for flaws in the planner that may have caused the observed failures.

8.1.1 Detecting Dependencies

Dependency detection is syntactic and requires little knowledge of the planner or its environment; thus, it can be applied to any planner in any environment. To begin, we gather execution traces of the planner. To determine how the planner's actions might lead to failure, the execution traces include failures and the recovery actions that repaired them, as in the following short trace:

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

F's are failures (e.g., F_{ip} is the Insufficient Progress failure in Phoenix) and R's are recovery actions (e.g., R_{sp} is the Substitute Projection action in the Phoenix Planner's recovery action set). It appears from this short trace that the failure *ip* is always preceded by the recovery action *sp*. We call disproportionately high co-occurrences between failures and particular precursors *dependencies*. In this example, the precursor of *ip* is the action *sp*, but conceptually, it could be any combination of predecessors in the trace: the preceding failure, the combination of the failure and the recovery action that repaired it, or even longer combinations of previous actions and failures. Currently, the analysis looks at only singles (i.e., failures or recovery actions) and pairs (i.e., failures and recovery actions) as precursors.

With more data, we can test whether the observed relationship between *sp* and *ip* is statistically significant. We build a contingency table of four cells: one for each combination of precursor and failure and their negations. For example, the contingency table in Table 8.1 tests whether F_{ip} depends on the precursor R_{sp}.

We test the significance of the observed relationship, $R_{sp} \rightarrow F_{ip}$, with a G-test on the contingency table. A G-test on this table is highly significant, $G = 42.86, p < .001$, meaning that it is highly unlikely that the observed dependence is due to chance or noise.

We construct contingency tables for three types of immediate precursors: failures, recovery actions, and a combination of a failure and the recovery action that repaired it. We denote these cases $F \rightarrow F$, $R \rightarrow F$, and $FR \rightarrow F$, respectively. The three types overlap. In particular,

$FR \rightarrow F$ is a special case of both $F \rightarrow F$ and $R \rightarrow F$ (because they subsume all possible values of the missing member), so if the former dependency is significant, we do not know whether it is truly a dependency between a F_1R and the subsequent F_2, or between F_1 irrespective of the intervening action, or between R with the initial failure playing no role. In practice, all three dependencies might be present to varying degrees.

We sort out the strengths of the dependencies by running a variant on the G-test called the Heterogeneity G-test [Sokal and Rohlf 81]. The intuition is that we compare the contributions of subsets to that of the superset; one can imagine looking at a Venn diagram (as in Figure 1) to gauge whether the failure *ner*, the recovery action *sp* or the combination seems to account for most of the area in the intersection with the subsequent failure *ip*. Both F_{ner} and R_{sp} overlap with part of the subsequent F_{ip}, but it is easy to see that a larger proportion of R_{sp} than F_{ner} overlaps with F_{ip}. Thus, R_{sp} is a more reliable precursor for F_{ip}.

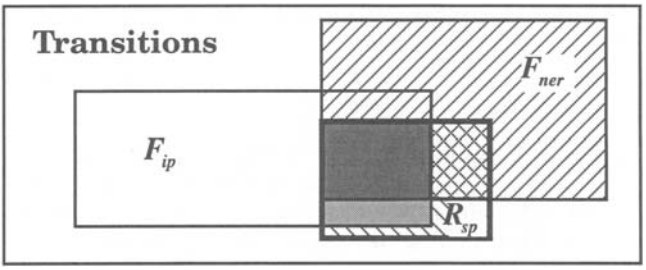

FIGURE 1. Venn diagram representing the data for the precursors and the following failure.

Similarly, but somewhat harder to see, the proportion of $F_{ner}R_{ip}$ (the darker shaded area plus the cross hatched area) that overlaps with F_{ip} (just the darker shaded area) is about the same as the proportion of R_{sp} (the area in the heavy box) that overlaps with F_{ip} (the two shaded areas), suggesting that we can generalize the relationship without loss of information. To compute the overlap, we add the G values for each of the subsets together (e.g., for $F \rightarrow F$, we add G values for all possible R's) and compare the result to the G value for the superset; if the difference is significant, then the subsets account for more of the variance in the dependency and so cannot be generalized. For this example, the G values for the subsets add to 45.89; the difference between that and G for the superset R_{sp} is 3.035, which is not a significant difference at the .05 level.

8.1.2 Explaining Dependencies

The first step in failure recovery analysis identifies potential problems in the planner's interaction with its environment; the remaining three steps explain how those problems may have been produced by the planner's actions and suggest redesigns to avoid the resulting failures. The statistical dependencies are mapped to structural dependencies in the planning knowledge bases suspected to be vulnerable to failure. Then, the interactions and vulnerable plan structures are used to generate explanations of how the observed failures might occur. Finally, the explanations serve to recommend redesigns of the planner and recovery component. These steps do not rely on statistical techniques or arguments, so we will not describe them in detail here. Interested readers should consult [Howe 92].

8.1.3 Sensitivity of Dependency Detection to the Size of Execution Traces

Execution traces are often expensive to collect. Consequently, much of the effort required to execute dependency detection is expended collecting execution traces. We expect that the results of dependency detection will vary based on how many execution traces we collect; the total number of patterns (i.e., possible combinations of different types of precursors and failures) and the ratios of the patterns (i.e., the ratio of the counts in the first column to the counts in the second column) in a contingency table influences the results of the G-test. To determine how the size and number of execution traces collected influences the results of the G-test, we need to answer two questions: How does the value of G change as the number of patterns in the execution traces increases? How does the value of G change as the precursor to failure co-occurrence (i.e., the ratio of the upper right to upper left cells in the contingency table) varies from the rest of the execution traces (i.e., the ratio of the lower left to the lower right cells in the contingency table)? The first question 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?

G-Test Sensitivity to Execution Trace Size.

We selected the G-test over the more common Chi-square test because the G-test is *additive*. Additivity means that G values for subsets of the sample can be added together to get a G value for the superset. 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 1 is 42.86; the G value for the contingency table with 10 times fewer counts (i.e., a contingency table with 5, 3, 24 and 64 in its cells) is 4.319 or roughly (as close as one gets when rounding the counts to the nearest integer) one tenth of 42.86. Additivity means that the value of G increases linearly with the amount of data (or in this case, the number of patterns in the execution traces).

A linear relationship between the number of patterns in the execution traces and the value of G is convenient for several reasons. First, the additivity property is exploited for the second step in dependency detection: pruning overlapping dependencies. We can divide the patterns into their subparts (e.g., a precursor with both a failure and recovery method in it) and add the resulting G values to get the same value as if we had calculated a G for all the subsets together. Second, a linear relationship is predictable. We know that the more patterns in the execution traces, the more likely we are to detect dependencies. Linearity is convenient because we are unlikely to be surprised by new dependencies suddenly showing up if we gather a few more execution traces (meaning the new dependencies were not even close to being dependencies before the additions). The bottom line is that given execution traces with few patterns, the G-test can find strong dependencies, but given more patterns, it will also find rare dependencies. 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.

Empirically Testing for Data Sensitivity.

We know that the value of G increases linearly with increases in the number of patterns in the execution traces, but only if the ratios in the contingency table remain the same, as the number of

	Exec. Traces 1	Exec. Traces 2	Exec. Traces 3	Exec. Traces 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

TABLE 8.2. Dependencies remaining after tweaking contingency tables. 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.

patterns increases. In trying to decide how many execution traces to gather, we also need to know whether the results will be vulnerable to noise, which is more apparent with few patterns. Unlike the sensitivity to total number of patterns, the sensitivity of the G-test to noise is complicated.

We can evaluate empirically whether, in practice, getting slightly more or fewer execution traces would have significantly changed which dependencies were detected in execution traces gathered from Phoenix. We do so by seeing how many of the dependencies would not have been detected if the counts in row one in the contingency table varied by a small amount. For example, if the contingency table in Table 8.1 had [52,35,240,643] instead of [52,33,240,643], then $G = 40.42$, which is not much different than the value for Table 8.1 of $G = 42.86$. To determine whether the dependencies detected in the execution traces are vulnerable to noise, we can do the following test: 1) construct the contingency table for dependencies detected in execution traces, 2) vary, one at a time, the counts of row one, column one and row one, column two by ± 2, and 3) run a G-test on the resulting contingency tables. By tweaking the contingency table cell values in this manner, we check the sensitivity of G to noise in the data. Both columns of the first row were varied because some of the first column counts were 1, which makes it impossible to test whether a lower ratio of first column to second column might not have influenced the value of G more than a higher ratio. We tweaked the counts by ± 2 because many contingency tables contained cell counts of less than 5, varying by ± 2 spans that range.

Table 8.2 shows how many of the dependencies found in each of four sets of execution traces for Phoenix would remain if their contingency tables are so tweaked. About 65% of the dependencies remain after tweaking their contingency table values, meaning that the counts in the first row of the contingency table can be changed by ± 2 without dropping the significance of G below the level of α. So, 35% of the dependencies detected would disappear if a few patterns more or less were included in the execution traces. Based on this testing of execution traces from Phoenix, dependency detection is sensitive to small differences in the content of the execution traces. Most of the dependencies that were vulnerable to the tweaking were based on execution traces that included few instances of the precursor/failure pattern, 23 out of 44 or 52% of the dependencies that disappeared were based on contingency tables in which one of the counts in the first row was less than five.

The implication of the sensitivity of dependency detection to noise in the execution traces is that rare patterns are especially sensitive to noise and so should be viewed skeptically. One must interpret the results of dependency detection with care: if "sensitive" dependencies are discarded, then rare events may remain undetected; at the same time, one does not wish to chase chimeras. Interpreting dependencies requires weighing false positives against misses. If we are trying to identify dependencies between precursors that occur rarely or failures that occur rarely, then additional effort should be expended to get enough execution traces to ensure that the

dependency is not due to noise.

8.2 Applications and Extensions of Dependency Detection

We discovered by chance that dependencies can be used to track modifications to planners and their environments. We ran Phoenix in one configuration, call it A, and collected execution traces from which we derived a set of significant dependencies, D_A. Then we modified Phoenix slightly–we changed the strategy it used to select failure recovery actions–and ran the modified system and collected execution traces, and, thus, another set of dependencies, D_B. Finally, we added two new failure recovery actions to the original set, ran another experiment, and derived another set of dependencies D_C. To our surprise, the intersections of the sets of dependencies were small. However, both $D_A \cap D_B$ and $D_B \cap D_C$ contained more dependencies than $D_A \cap D_C$, suggesting that the size of an intersection mimics the magnitude of modifications to a system.

These results are only suggestive, but they raise the possibility that particular planner-environment pairs can be characterized by sets of significant failure-action dependencies. If true, this technique might enable us to identify classes of environments and planners. Today, we assert on purely intuitive grounds that some environments are similar; in the future we might be able to measure similarity in terms of the overlap between sets of dependencies derived from a single planner running in each environment. Conversely, we might measure the similarity of *planners* in terms of dependencies common to several planners in a single environment.

8.2.1 Further Work

More Complex Dependencies.

The dependencies examined so far have been limited to temporally adjacent failures, actions or the combination of each. The combinatorial nature of dependency detection precludes arbitrarily long sequences of precursors. More complex dependencies can be discovered either by controlling the collection of data to selectively test for particular dependencies (through experiment design) or by heuristically controlling the construction and comparison of dependencies (through enhancements to the Heterogeneity G-Test). A new experiment design would selectively eliminate actions from the available set to test whether each precipitates or avoids particular failures (i.e., an ablation or lesion study). Rather than examining all possible chains of which some action is a member, the new analysis removes the action from consideration, which results in execution traces free from the interaction of the missing action. Dependency sets from the different execution traces would be compared to assess the influence of the missing action.

Alternatively, dependency sets can be built iteratively from subsets; the Heterogeneity G-Test suggests a method of doing so for singletons and pairs, but cannot be applied in a straightforward fashion to longer combinations. We need to enhance the technique to compare longer combinations and use the results of comparing sets of shorter precursors to motivate the search for longer ones. For example, if some singleton subsumes a set of pairs, it seems unlikely to be necessary to look at longer combinations beyond the pairs. In effect, Heterogeneity testing becomes a means of controlling heuristic search through the potentially combinatorial space of possible dependencies.

Marker Dependencies.

Some dependencies might function as markers for particular characteristics of environments. For example, severely resource constrained environments might be characterized by resource contention failures repeating over and over, leading to the observed dependency that one resource contention failure leads to another. We would expect this dependency 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. To look for such markers, we will need to describe a hierarchy of failures and actions such that dependency sets for different task environments can be compared.

8.3 References

[Corbato 91] Fernando J. Corbato. On building systems that will fail. *Communications of the ACM*, 34(9):72–81, September 1991.

[Howe 92] Adele E. Howe. Analyzing failure recovery to improve planner design. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 387–393, July 1992.

[Sokal and Rohlf 81] Robert R. Sokal and F. James Rohlf. *Biometry: The Principles and Practice of Statistics in Biological Research*. W.H. Freeman and Co., New York, second edition, 1981.