Methods for Finding Influences on Program Failure [*]

Adele E. Howe Aaron D. Fuegi
Computer Science Department
Colorado State University
Fort Collins, CO 80524
howe@cs.colostate.edu

Abstract

This paper describes two approaches for detecting patterns of detrimental program behavior, called dependencies, over long periods of time; these dependencies indicate cases where previous events influence the occurrence of later failure. This research extends a previous approach that was limited to temporally adjacent events. The two approaches, heuristic search and local search, are demonstrated on several data sets from an AI planner and are compared on their efficiency and the dependencies they detect.

1 The Problem

You carefully design and implement your program. When you test it and it fails, you need to determine what caused the failure. The problem is that the cause of the failure may occur long before its manifestation. In this paper, we will describe two statistical methods for finding detrimental patterns (combinations of possible causes and failure) in execution traces. These methods extend previous work on finding discrete causal influences over short time horizons [5] and have been applied primarily to identifying causes of failure in a planning system.

1.1 Previous Approaches to Solution

Generally speaking, the problem is modeling causal influences on failure. We wish to know whether one event influences the occurrence of another. Previous research has addressed two aspects: modeling failure and modeling causal influences. This section will describe some approaches to each of these.

[*]This research was supported by a Colorado State University Diversity Career Enhancement grant and ARPA-AFOSR contract F30602-93-C-0100. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

Modeling Failures in Software In Software Engineering, the emphasis of modeling failure is on fault tolerance and debugging. Approaches to fault tolerance decide how to avoid or repair failures through a complete model, built by an expert, of failure and its causes; common techniques are time Petri net models, real-time logic and fault tree analysis [6]. Two other approaches to modeling software bugs are a formal language for describing program failures [1] and a belief network of canonical bugs [2]. These models are constructed all or in large part by a programmer.

Statistically Modeling Causal Influences Recent research has emphasized causal induction, inferring causal structure from observational data. One well-known approach [7] infers causal relationships from covariance data by testing conditional independence of variables. Another approach by Cohen et. al [3] extends statistical path analysis to construct the structure of the causal model (i.e., the relationships between variables and the directionality of the relationships) as well as to estimate the strength of those relationships. As in the Pearl's approach, their algorithm, FBD, uses covariance information to build the model, but directs the incremental construction of a multi-layered causal diagram heuristically.

These algorithms have been shown to efficiently infer the causal structure of extremely complex systems. However, they do not allow cycles in the models and require covariance information, which means that the underlying variables should be numeric.

2 Dependency Detection

An approach to modeling short-term categorical causal influences on failure is *Dependency Detection* [5]. Dependency Detection applies statistical techniques to build a set of simple models of failure. Like the above approaches to modeling failures, it relates alternative paths to a particular failure; unlike these approaches, it does not do so a priori and a human is not involved in the process. Like causal influence techniques, it applies statistics to detect trends and co-occurrences, but unlike them, it is insensitive to repetitions in the same trace and focuses on events (i.e., categorical rather than numerical data).

Dependency Detection was developed originally to search the Phoenix planner's failure recovery execution traces for immediately preceding causes of failure. The execution traces were sequences of alternating failures and the recovery actions that repaired them, e.g., $F_a \rightarrow R_{aa} \rightarrow F_b \rightarrow R_{aa} \rightarrow F_a \rightarrow R_{zz}$. Because this version tested all combinations of failures and recovery actions, search for long patterns was computationally impractical. DD was generalized by removing references to Phoenix and recovery actions and extending its search to temporally separated causes and effects. The generalized DD algorithm is:

1. Gather execution traces of the program.
2. Generate an initial set of patterns to be tested.
3. For each pattern[1]:
   (a) Slide a window incrementally over the execution traces,
   (b) At each window position, increment one of 4 counts: the pattern precursor is followed by the pattern failure, the pattern precursor is followed by another failure, some other precursor precedes the pattern failure and some other precursor precedes some other failure.
   (c) The four counts are arranged in a 2x2 contingency table, and significance is tested with the G-test[2].
4. Prune or add to the the set of patterns and repeat from 3 until no test patterns left.

First, execution traces are collected; execution traces are sequences of salient events (i.e., occurrences expected to be of interest to the programmer such as program failures, subroutine startups, changes to key variable values, etc.) observed during execution of the program. Such traces are searched for patterns of some sequence of events that typically precede a particular failure.

Second, we test the significance of patterns. A pattern has two parts: a combination of salient events for the precursor and a target failure type. The target failure must appear as the last element in the window. For a given pattern, incidence counts are gathered by sliding a window over the execution traces. The window defines a area of length N which is checked for the pattern. The window is re-positioned by moving its last position (the target failure) to the next failure. Figure 1 shows four positionings of the sliding window for a pattern of R_{sp} (recovery action of type sp) for the precursor and F_{ip} (failure of type ip) for the failure.

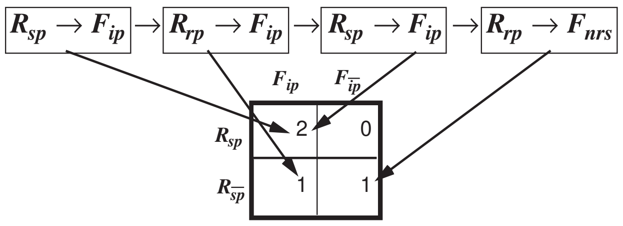
$R_{sp} \rightarrow F_{ip}$
$R_{rp} \rightarrow F_{ip}$
$R_{sp} \rightarrow F_{ip}$
$R_{rp} \rightarrow F_{nrs}$
F_{ip}
$F_{\overline{ip}}$
R_{sp}
$R_{\overline{sp}}$
2
0
1
1

Figure 1: Constructing a contingency table for a [R_{sp}, F_{ip}] pattern by sliding a window across execution traces

Four counts are tallied for the contingency table. Each window positioning contributes to one and only one of the four cells in the contingency table for each pattern tested. Assuming the final element (e.g., failure F_{ip} in the first position of Figure 1) in the window matches the final element (e.g., F_{ip}) in the target pattern, the window contributes to the first column; otherwise, it contributes to the second column. Similarly, the remaining elements of the window and the pattern precursor are compared. After sliding the window through the execution traces, the resulting contingency table is analyzed to see if the target failure was statistically dependent on the precursor.

Finally, the set of patterns is either increased or decreased. New patterns with potential can be added to the set or insignificant or uninteresting (e.g., due to lack of data or overlap with other patterns) patterns can be removed from the test set. How the set of patterns is initially set and then changed depends on the search method. The remainder of the paper describes two search methods for controlling dependency detection, heuristic and local search, and presents results of finding dependencies in execution data from an AI planner (the Phoenix system).

2.1 Heuristic Search Methods

The search starts by seeding the pattern set with a core set of patterns. Each pattern is tested; if the pattern passes some evaluation criteria, then its ex-

[1] Formally, a pattern is a sequence of events comprising a precursor and subsequent failure; a dependency is a pattern found to be statistically significant (i.e., $\alpha = 0.05$, which indicates a $> .05$ probability that the observed dependency was due to chance).

[2] A statistical test similar to the Chi-square that roughly tests whether two observed ratios are significantly different.

tensions are added to the pattern set. This results in heuristic, depth first search.

The patterns may consist of particular types of events from the traces and wildcards for positions within the pattern than may be any event. In the case of Phoenix data, the core patterns are of the form E#F where E is any event (i.e., action or failure) and # is any number of wildcards from zero to seven. Patterns are added to this core set when the "parent" pattern still has wildcards and it passes certain criteria for continuing evaluation.

The evaluation criteria is an additive combination of a value based on pattern significance and the number of instances of the pattern type. For each pattern, significance is ranked from one to three based on the p for the contingency table (i.e., $p < .05$ is worth three, $p < .25$ is worth two, else one for just found in data). Number of instances is rated from one to three based on the number of wildcards in the pattern and the amount of data to support it (i.e., number of instances of pattern should be greater than number of wildcards times some threshold for three levels of threshold) . If the two measures add to four or more, then simple extensions to this pattern (filling in wildcards with specific events) are added to the test set.

A summary of some of the results is presented in Table 1. The percentage of significant patterns does not seem to be at all dependent on the length of the patterns being analyzed, which suggests that the heuristic control is not missing too many dependencies. On the other hand, the quantity of patterns both found in the data and analyzed as significant roughly monotonically increases as the length increases. This is not surprising because longer patterns allow for far more possible combinations. This explanation should be partly offset, though, because the availability of data decreases with length.

2.2 Local Search Methods

Another approach is to apply local search techniques to find relative order patterns. We shifted to relative order for two reasons: first, it accounts for irrelevant events appearing in the midst of an otherwise causal sequence; second, because matching is more flexible, more patterns are likely to be found and local search is more likely to be effective.

The algorithm is roughly the same as in the general form with three differences. First, the window size must be larger than the pattern; for a pattern of size N, we use a window of size $2N$. Second, a pattern matches a given window position when the target failure is at the last position and when all the remaining elements in the pattern appear in the window in the same *order* they do in the pattern. Although a particular pattern could occur multiple times in a given window, it contributes only one to a single cell in the contingency table.

		Pattern Length				
Data Set		3	5	7	8	9
A	Found	71	279	595	929	1623
	Significant	29	100	212	253	511
	Ratio	41%	36%	36%	27%	31%
B	Found	109	487	1328	784	2313
	Significant	33	175	480	274	818
	Ratio	30%	36%	36%	35%	35%
C	Found	141	615	1153	1385	3188
	Significant	62	212	387	536	1261
	Ratio	44%	34%	34%	39%	40%

Table 1: Results of heuristic search on three representative Phoenix data sets

Third and most important, patterns are added to the core set using local search. For each of some number of trials, the initial pattern is a pattern of length N with the elements chosen randomly. Several local search operators were considered that reorder and replace pattern elements, but the best operator, in terms of efficiency and efficacy, 1OPT optimizes one element in the pattern at a time. Progressing through the pattern elements in order, this operator considers all available elements as replacements for the pattern element being considered; the operator replaces the element originally in the position with the best element for that position after testing all possibilities.

The results of applying the heuristic method suggested that dependencies with many elements in common tend to both be significant. This characteristic is exploited by local search. We tested the efficacy of local search by searching for several length relative order dependencies. For each target failure, we started with 100 randomly generated relative order strings and searched for the highest rated neighbor that could be reached from it. Table 2 shows the top ranking dependencies found by local search for each target failure for a relative order of length four as well as the ratio of the trials that found a significant dependency. The searches find a significant dependency in about one-third of the trials, and the dependencies found are highly significant (the higher the G value, the lower the probability that it is due to chance). Searching for length five patterns exhibited similar, if

Dependency	G	P	Trials
[NER RP NER VIT]	18.72	.0001	63/100
[NER BDU RM CFP]	8.44	.0037	29/100
[CCP RM CCP CCP]	75.62	.0001	35/100
[PTR RM RM PTR]	29.16	.0001	31/100
[CCP RM CCP NER]	19.04	.0001	41/100
[RP NER RV BDU]	7.20	.0073	21/100
[FNE CCV RM CCV]	6.90	.0086	27/100
[PRJ PRJ PRJ PRJ]	15.68	.0001	49/100

Table 2: Most significant dependencies found by local search with 100 trials on data set 1

slightly less successful, results; highly significant dependencies are found in, on average, 18 out of 200 trials. Local search appears to be an efficient method of finding highly significant relative order dependencies.

3 Summary

Evaluation largely depends on whether models are found to be useful for characterizing behavior. Dependency detection was incorporated into a procedure for debugging Phoenix called *Failure Recovery Analysis* (FRA) [4]. Dependency detection identifies problematic sequences of events, which FRA relates to portions of the knowledge base and then generates explanations of how the knowledge base may have contributed to the observed dependencies. Patterns discovered through heuristic search have been used for debugging in Failure Recovery Analysis; local search, relative order patterns are being incorporated into the process currently. Additionally, the code for the two methods is available and has been incorporated into the Common Lisp Analytical Statistics Package.

One problem with these searches is that a good method finds too many different patterns. We would like to cluster related dependencies to enrich their meaning (e.g., find commonalities) and to reduce the burden of looking over a long list. The search neighborhood in the local search method provides a natural clustering: the significant basins of attraction that most locally optimal relative order patterns appear to have. With local search, the algorithm can use the optimization procedures to find locally optimal patterns and then back out of the basin at individual positions to find other highly significant related patterns.

To summarize the results, exhaustive search of arbitrary length patterns is impractical; however, both the heuristic and local search methods are effective at detecting long dependencies: they find a large number in a manageable amount of computation time. The most computationally intensive method (200 trials of local search for patterns of length five with a particular target) required an hour; the least, a more constrained form of the heuristic search described here, required 1-10 minutes. Heuristic methods are goal directed and find a high proportion of significant dependencies for what is tested. The local search method is data directed, finds highly significant dependencies and provides a convenient method of clustering. Related work compensates for the exclusive use of categorical data, discrete events, by integrating dependency models with path analysis models. Future work will demonstrate how these models can be used to debug different knowledge based systems.

References

[1] Peter C. Bates and Jack C. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. COINS Dept. TR 83-29, Univ. of Mass., March 1983.

[2] Lisa J. Burnell and Scott E. Talbot. Incorporating probabilistic reasoning in a reactive program debugging system. In *Proceedings of the Ninth Conference on Artificial Intelligence for Applications*, pages 321 –327, Orlando, FL, March 1-5 1993.

[3] Paul R. Cohen, Lisa A. Ballesteros, Dawn E. Gregory, and Robert St. Amant. Regression can build predictive causal models. TR 94-15, Computer Science, Univ. of Mass., 1994.

[4] 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.

[5] Adele E. Howe and Paul R. Cohen. Isolating dependencies on failure by analyzing execution traces. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, College Park, MD, 1992.

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

[7] Judea Pearl and T. S. Verma. A theory of inferred causation. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, April 1991. Morgan Kaufmann.