Modeling Intelligent System Execution as State Transition Diagrams to Support Debugging*

Adele E. Howe   Gabriel Somlo
Computer Science Department
Colorado State University
Fort Collins, CO 80523
email: {howe,somlo}@cs.colostate.edu
URL: http://www.cs.colostate.edu/˜{howe,somlo}

Abstract

Currently, few tools are available for assisting developers with debugging intelligent systems. Because these systems rely heavily on context dependent knowledge and sometimes stochastic decision making, replicating problematic performance may be difficult. Consequently, we adopt a statistical approach to modeling behavior as the basis for identifying potential causes of failure. This paper describes an algorithm for constructing state transition models of system behavior from execution traces. The algorithm is the latest in a family of statistics based algorithms for modelling system execution called *Dependency Detection*. We present preliminary accuracy results for the algorithm on synthetically generated data and an example of its use in debugging a neural network controller for a race car simulator.

1 Introduction

Traditional approaches to debugging may be inadequate or inappropriate for AI systems. The applications tend to be poorly understood. The base technology (often languages or software architectures built on top of other languages) may be experimental or at least relatively new. The systems themselves rely on heuristics or approximations. The expected performance (e.g., the best answer or the right action to take) may not be known and is likely to be context dependent. The system may comprise many integrated components. Thus, bugs are often intermittent, are manifest over long periods of activity, and are due to detrimental interactions.

At a symposium on Integrated Planning Applications, developers agreed that the primary barriers to successful deployment were user acceptance (i.e., ensuring that the system does what the users need) and acquiring and debugging domain knowledge[8]. Plan debugging has been a topic of some interest as a method for plan construction (e.g., [6, 15]), but planner and system debugging has received less attention. Chien at JPL has developed two methods for analyzing a planner's knowledge base: static and completion analysis [3, 4]; these methods identify certain classes of syntactic errors and gaps in knowledge that prevent complete plans from being generated. Some planning systems include a mechanism for incrementally

*This research was supported in part by by NSF Career Award IRI-9624058 and by DARPA-AFOSR contract F30602-93-C-0100 and F30602-95-0257. We also wish to thank Larry Pyeatt for his contributions on the previous version of the algorithm and for collecting the RARS data.

stepping through plan generation and viewing the plan being developed (e.g.,[18, 5]). Both of these approaches emphasize the plan generation phase, assuming that execution should proceed as planned.

Execution was the focus of Failure Recovery Analysis (FRA) [7, 11]. FRA is a methodology for debugging some canonical bug types in the Phoenix planner by statistically analyzing traces of failure recovery. We focused on statistical analysis because the subtle differences in context, the reliance on occasional stochastic decision making and the time periods in which the system operated (often for hours or days) made it nearly impossible to debug by replicating the failure. Unfortunately, FRA was developed primarily for the Phoenix planner and relied on simple models of subsets of execution traces.

In the past few years, we have been generalizing the statistical analyses underlying FRA to support modeling and debugging of intelligent systems. The resulting family of techniques, called *Dependency Detection*, discover unusually frequently reoccurring patterns of events in execution traces using contingency table analysis. These techniques divide into two types: "snapshots" and "overviews". The snapshot models, called "dependencies", collect frequently co-occurring linear sequences of events [10, 9].

The overview models relate a set of statistically significant patterns as a single model that includes interconnections between events. Our first overview modeling algorithm constructs complete, Semi-Markov models of the execution traces [12]. However, many systems are not Markov in their execution. Additionally, these models include all sequences, including those due to noise, and grow rather large, making them difficult to understand and use for debugging. This paper presents a new overview modeling technique that produces state transition models of system execution that are adjustable in their complexity. We present the modeling algorithm and preliminary results of its accuracy and usage in debugging.

2 Building Transition Models of System Execution

The transition diagram construction algorithm extends snapshot Dependency Detection (DD) by heuristically combining its results. DD is designed to find short significant sequences. By iteratively combining the dependencies detected, we can build a transition diagram that captures the most significant patterns.

2.1 Snapshot Dependency Detection

DD finds sequences that uncommonly frequently precede some event (dependencies). These sequences may indicate causes of a down stream failure, early symptoms of it, related problems or even just noise (random occurrence). Dependencies are found by analyzing execution traces. An execution trace is a sequence of observed events. Some of the events are designated as influences (called *precursors* in DD) and others as the events of interest (called *target* events). One event, T_t, is said to be *dependent* on another, P_a, if T_t is observed to occur more often following P_a than following any other event.

The basic idea is to see whether particular sequences stand out from the whole, based on a statistical analysis. First, the execution traces are scanned to find and count all sequences of a length specified by the user. Then, sequences of influencing events are first tested for whether they are more or less likely to be followed by each of the target events. The output of the process is a list of sequences (e.g., $AB \rightarrow C$

	T_t	$T_{\bar{t}}$
P_a	15	10
$P_{\bar{a}}$	30	50

Table 1: Contingency table for determining sequence significance

which represents that event C depends on the sequence of events AB) and their "strength" as indicated by the probability that the observed predictor appears as often as other predecessors.

The core of DD is contingency table analysis to determine whether a particular target event (e.g., T_t) is more likely after a particular precursor (e.g., P_a). We count four types of sequences in the execution traces: 1) instances of T_t that follow instances of P_a, 2) instances of T_t that follow instances of all precursors other than P_a (abbreviated $P_{\bar{a}}$), 3) target events other than T_t (abbreviated $T_{\bar{t}}$) that follow P_a and 4) target events $T_{\bar{t}}$ that follow $P_{\bar{a}}$. From these frequencies, we construct a two by two contingency table, as in Table 1.

Based on this table, P_a seems to be a good predictor of T_t. A statistical test, the G-test, indicates whether the two events are likely to be dependent [16]; in this case, $G = 5.174, p < .023$, which means that the two events are unlikely to be independent and conclude that T_t depends on P_a. A dependency is added to the list if p falls below a pre-determined threshold. Due to the statistics, dependencies include both uncommonly frequently and uncommonly infrequently co-occurring sequences; we call the latter type *negative dependencies*.

DD has been extended to find more complex patterns[10, 9]. One version collects dependencies up to some length pruning overlapping dependencies based on an analysis of which is the best descriptor. Another detects partial order sequences, which is robust to the introduction of intervening events. At present, we use only the simplest form for constructing transition diagrams.

2.2 State Transition Generation Algorithm

State transition models are built by gradually incorporating significant sequences. The resulting model is not guaranteed to be complete, but because it incorporates only significant dependencies, it is unlikely to include spurious sequences and its complexity (number of nodes) is adjustable by changing the threshold on significance.

At present, the user provides execution traces and designates a dependency length and significance threshold. The basic algorithm, which appears in Figure 1, iteratively connects together the dependencies, merging states where possible. The resulting diagram often has states composed of sequences of events. We display the diagram using a publicly available graphing package, called Dot[13].

The rules for when to link, merge or create new states are based on the information available from dependency detection. This part is the most complex in the algorithm because it requires generating all sequences from the state and its existing predecessors and successors in the state transition diagram.

To show how the algorithm works, we generated data from the diagram in the left side of Figure 2 with a small amount of noise in the traces (0.05). First, we collected dependencies, as shown in Table 2, and clustered them. The cluster with the most data was $CD \rightarrow A \vee B$. On the first pass through the loop, CD was linked to two new states, A and B, with transition probabilities $\frac{87}{87+75} = .537$ and

1. Collect and organize dependencies of the specified length,
    (a) Run DD,
    (b) Prune out negative dependencies,
    (c) Sort the dependencies in descending order of precursor frequency, clustering all the targets (successor states) that may follow it.
2. Create a state from precursor of first dependency and push it on resolution list,
3. Repeat until no states are yet to be resolved and no dependencies left to be included:
    (a) Find successor states from the dependency list,
    (b) Add successors to the model by:
        - Link to existing state if all the sequences that result are consistent with all known dependencies,
        - Merge current state with successor if only one successor and the merge is consistent with dependencies,
        - Otherwise create new state as successor.
    (c) Calculate transition probabilities from counts in the contingency table for the dependencies,
    (d) Add new and merged states to resolution list.

Figure 1: Algorithm for constructing state transition diagrams

$\frac{75}{87+75} = .463$, respectively. On the second pass, B is merged with E by finding the dependency DBE and noting that it matches the existing transition $D \rightarrow B$. On the third pass, state A matches the dependency DAB; we cannot link to state BE because we can find no BBE dependency so state A is extended to AB. By continuing to add in states from the dependencies, the diagram on the right side of Figure 2 resulted. This diagram collapses the four states into two, making it more compact, but also removing one possible transition ($AB \rightarrow CD$).

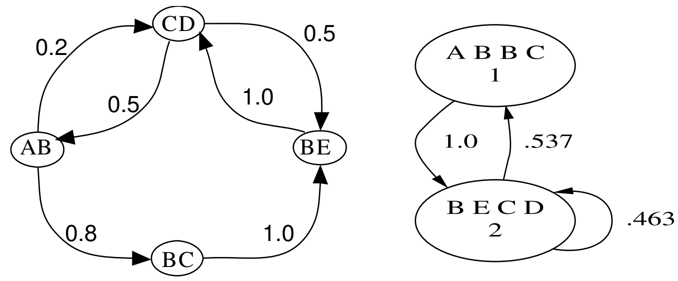

Figure 2: An idealized five token model (on left) and transition diagram constructed from execution data (on right)

(E C D)	(155 1 19 823)	(B E C)	(153 4 95 746)
(D A B)	(86 2 235 675)	(D B E)	(78 1 83 836)
(A B B)	(74 18 247 659)	(B B C)	(74 1 174 749)
(C B E)	(74 1 87 836)	(C D A)	(87 81 7 823)
(C D B)	(75 93 246 584)	(B C B)	(74 14 247 663)

Table 2: Length 3 dependencies (sequence with contingency table) collected for synthetic five token model

3 What We Have Learned So Far

Snapshot dependencies have been used to support debugging failure recovery in the Phoenix planner [11] and to identify search control problems in UCPOP[17]. However, the dependencies were rather limited in their temporal scope and complexity; in some cases, especially with Phoenix, the short sequences led us to make changes that caused other problems, changes that would have been different had we known of other interactions. We expect that overview models should avoid this problem because it helps identify many possible contributors to undesirable state transitions, clarify shared downstream influences and discover cycles in behavior.

We demonstrate the utility of the transition diagram generation method in two ways. First, we tested its accuracy on synthetic data because we can control their underlying complexity. Second, we provide an example of its use on real data.

Accuracy on Synthetic Data Testing accuracy is a four step process. First, we developed three synthetic models in which we varied the number of events (5-13), the number of states (3-6) and the maximum number of events per state (2-7); Figure 2 shows one model with five events, four states and two events per state (abbreviated as M5-4-2). Second, we generated execution traces of length 10,000 by simulating the models and introducing noise (events randomly selected from the set with probability of 0.1). Third, we constructed state transition diagrams from half of the data. Finally, we determined accuracy by comparing the sequences that can be produced by the resulting diagrams to those in the other half of the data.

We measure accuracy as hits, false negatives and false positives. Hits are the number of sequences that both appear in the data and can be produced by the diagram. False negatives are the number of sequences in the data that cannot be produced by the diagram; false negatives equal total sequences minus hits. False positives are the number of distinct sequences that can be produced with the diagram that do not appear in the data. We generated diagrams of three lengths (2-4) and probability thresholds of .01 and .05; we report the results for length four and threshold of .05 as these appeared to give the best trade-off of hits to false positives overall.

Table 3 shows the resulting measures for the four synthetic models summed across sequence lengths of 2-6 (24980 possible sequences from the data). Hit-rate is percentage of hits in the data sequences. The results show that the hit-rate is fairly high and the false positive numbers are low. Thus, the diagrams appear to be capturing most of the dynamics represented by the execution traces. The results also suggest some variability in the performance on different types of models: more states led to less accuracy, but the method appears relatively insenstive to the number of events or maximum number of events per state.

	M5-4-2	M6-6-3	M13-3-7	Total/Avg.
Hits	20,937	21,116	23,774	65,827
Hit-rate	.838	.845	.952	.878
False Positives	27	2,517	604	3,148

Table 3: Accuracy of transition diagram generation on synthetic models

Debugging Example from RARS Reinforcement Learning Controller We have been building a reinforcement learning controller for RARS (Robot Automobile Racing Simulator) [14]. The controller must regulate the acceleration and steering of a race car on simulated tracks, which requires that it negotiate the track, avoid crashing into the walls, pass cars and go in for pit stops. As we develop the system, we are incrementally adding behaviors.

We use the transition diagram generation to describe and debug the behaviors learned by the reinforcement learning system. We would like to avoid failures: when the controller loses control of the car and runs off of the track (`crashed` event).

Figure 3 shows a fragment of the transition diagram[1] generated from execution data from RARS. At this point, the controller has been partially trained to negotiate the track, but is still crashing occasionally. The events describe sensor readings: track position and speed relative to a wall. Each were prefixed with a letter (P and X, respectively) and quantized into discrete intervals (five for P and six for X). For example, `P0` indicates a position next to the inner wall, and `P4` indicates a position next to the outer wall. `X0` is headed fast toward the outer wall, and `X5` is fast towards the inner wall.

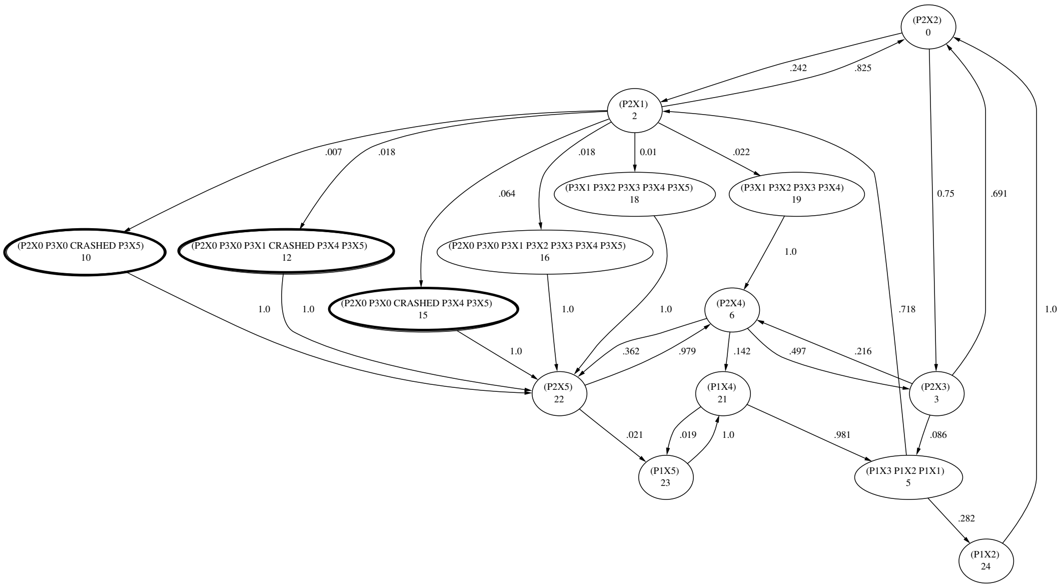

Figure 3: Portion of transition diagram for RARS reinforcement learning controller

From Figure 3, we can identify several problems in the current execution. First, we observe bottlenecks (shared states) immediately before and after all states containing the crashed event (highlighted states 10, 12, and 15, plus some states removed

[1]The full diagram had 24 states and so would not be readable in this format. We removed low probability transitions (< .007) from the diagram to make it fit.

for display purposes). These bottlenecks are critical events indicating cases in which training and reinforcement should be focused. Better control from state 2 (the before state) could prevent crashes. Second, the network may have been overtrained for the straightaway, which constitutes roughly 80% of the track. The transition from the bottleneck state away from states containing "crashed" occurs 82% of the time. Thus, we need to tune the reinforcement schedule to prevent overfitting to the common, straight part of the track.

3.1 Extensions and Conclusions

The algorithm requires two parameters: dependency length and probability threshold. We are currently exploring principled methods of setting these. One option is to use CHAID based analysis to determine dependency length. CHAID constructs an n-step transition matrix, which indicates what earlier point in the execution traces was most predictive of the occurrence of each type of events[2]. The CHAID-based analysis is part of the Chitra family of systems (Chitra 92-96), which were developed to support the modeling and visualization of execution data from parallel and distributed systems [1].

The probability threshold is the primary mechanism for adjusting the ocmplexity of the model. Lower probabilities result in less complex models because fewer transitions are recognized as significant. One option for determining how to set the probability threshold is to model its effect. We have started to extensively evaluate the accuracy and utility of the algorithm with different parameter settings using a larger collection of synthetic and real data.

We have found the dependency based models extremely useful for debugging three intelligent systems: Phoenix, UCPOP and RARS. Currently, few tools are available for assisting developers with debugging intelligent systems. The algorithm described here is a start toward addressing this gap in developing intelligent systems.

References

[1] Marc Abrams, Alan Batongbacal, Randy Ribler, and Devendra Vazirani. CHITRA94: A tool to dynamically characterize ensembles of traces for input data modeling and output analysis. Computer Science Dept. 94-21, Virginia Polytechnical Institute, June 1994.

[2] Horacio T. Cadiz. The development of a CHAID-based model for CHITRA93. Computer Science dept., Virginia Polytechnic Institute, February 1994.

[3] S. Chien, H. Mortensen, C. Ying, and S. Hsiao. Integrated planning for automated image processing. In *Working Notes of the AAAI Spring Symposium on Integrated Planning Applications*, March 1995.

[4] Steve A. Chien. Static and completion analysis for planning knowledge base development and verificiation. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS96)*, pages 53–61, May 1996.

[5] David Christianson and Chung Kwok. *PDB Reference Manual*. University of Washington, 4.0 edition.

[6] Kristian J. Hammond. Explaining and repairing plans that fail. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 109–114, Milan, Italy, 1987.

[7] Adele E. Howe. Improving the reliability of AI planning systems by analyzing their failure recovery. *IEEE Transactions on Knowledge and Data Engineering*, 7(1):14-25, 1995.

[8] Adele E. Howe, editor. *Working Notes of the AAAI Spring Symposium on Integrated Planning Systems*, Palo Alto, CA, March 1995. AAAI.

[9] Adele E. Howe. Detecting imperfect patterns in event streams using local search. In D. Fisher and H. Lenz, editors, *Learning from Data: Artificial Intelligence and Statistics V*. Springer-Verlag, 1996.

[10] Adele E. Howe and Paul R. Cohen. Detecting and explaining dependencies in execution traces. In P. Cheeseman and R.W. Oldford, editors, *Selecting Models from Data; Artificial Intelligence and Statistics IV*, pages 71–78. Springer-Verlag, NY,NY, 1994.

[11] Adele E. Howe and Paul R. Cohen. Understanding planner behavior. *Artificial Intelligence*, 76(1-2):125–166, 1995.

[12] Adele E. Howe and Larry D. Pyeatt. Constructing transition models of AI planner behavior. In *Proceedings of the 11th Knowledge-Based Software Engineering Conference*, September 1996.

[13] Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot.* AT&T Bell Laboratories, Murray Hill, NJ, October 1993.

[14] Larry D. Pyeatt, Adele E. Howe, and Charles W. Anderson. Learning coordinated behaviors for control of a simulated robot. Technical report, Computer Science Dept, Colorado State University, 1996.

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

[16] 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.

[17] Raghavan Srinivasan and Adele E. Howe. Comparison of methods for improving search efficiency in a partial-order planner. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1620–1626, Montreal, CA, August 1995.

[18] David E. Wilkins. SIPE-2: A system for interactive planning and execution. http://www.ai.sri.com/ sipe/, April 1995.