Modeling Discrete Event Sequences as State Transition Diagrams*

Adele E. Howe and Gabriel Somlo

Computer Science Dept, Colorado State University, Fort Collins CO 80523, USA,
email: {howe,somlo}@cs.colostate.edu

Abstract. Discrete event sequences have been modeled with two types of representation: snapshots and overviews. Snapshot models describe the process as a collection of relatively short sequences. Overview models collect key relationships into a single structure, providing an integrated but abstract view. This paper describes a new algorithm for constructing one type of overview model: state transition diagrams. The algorithm, called *State Transition Dependency Detection* (STDD), is the latest in a family of statistics based algorithms for modeling event sequences called *Dependency Detection.* We present accuracy results for the algorithm on synthetic data and data from the execution of two AI systems.

1 Introduction

Many processes can be viewed as long sequences of discrete events over time. Events may capture the state of the process at regular intervals or may encompass changes in the state at irregular intervals. By modeling these sequences, we can predict likely future states and describe what led to the current state. For example, ecologists perform flora/fauna counts regularly within small areas to monitor ecosystems; credit card companies record and track customer transactions to detect fraud and identify opportunities for additional sales [6].

Discrete event sequences have been modeled with two types of representation: snapshots and overviews. Snapshot models describe the process as a collection of relatively short sequences or rules. These rules relate key events to each other. For example, a rule in a grammar indicates that an adjective should be followed by a noun or that the purchase of the first book in a best-selling trilogy is often followed by the purchase of the second and third books [2]. Overview models collect key relationships into a single structure, providing an integrated but abstract view. For example, finite state machines drive some language parsers, and Markov and Bayesian networks structure probabilistic dependencies [14].

Each representation has its purposes. Snapshots are convenient, easily exploited representations for rule-based systems (e.g., planning [13]) and are well-suited for short sequences over a large set of separate processes (the *market*

* 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.

basket problem [3]) and tend to include even extremely rare relationships or events. Overviews provide a more comprehensive view, clarifying longer term relationships and interaction effects.

This paper presents a new overview modeling method, called *State Transition Dependency Detection* (STDD), that automatically generates state transition models from event sequences. The method heuristically combines statistically determined snapshot patterns into a cohesive whole. Because the method relies on a statistical technique for determining the basis patterns, it is fairly robust to the presence of noise in the discrete event sequences. Because the combination is heuristic, the resulting model may include cycles: a desirable characteristic for our applications. Consequently, we evaluated the accuracy of the results of the algorithm on both synthetic data with varying levels of noise and real data: discrete event sequences from two AI systems.

1.1 Applications and Techniques for Event Sequence Modeling

Our primary application for event sequence modeling is debugging. Our method for generating snapshot models, called *dependency detection* (DD), has been used to support debugging failure recovery in the Phoenix planner [9] and to identify search control problems in UCPOP[16]. Snapshot DD methods discover unusually frequently or infrequently co-occurring sequences of events (called *dependencies*) using contingency table analysis [8] (see Section 2.1). Mannila et al. [12] find serial (i.e., a strict ordering of events, perhaps with intervening events) and parallel (i.e., events occurring within some time period of one another, the exact ordering is irrelevant) patterns of events that occur more than some number of times. They applied their methods to a database of telecommunications network faults covering a 50 day period.

While the snapshot models did help identify some previously unknown interactions, we found that bugs that are due to cycles are hard to detect with the snapshots, and that fixes based on the snapshots may cause problems that could not be predicted from the snapshots. Based on this experience, we developed an overview modeling algorithm that constructs complete, Semi-Markov models of the event sequences [10]. Abrams et al. generate Semi-Markov models to debug interactions of distributed computational processes [1]. Their method constructs Semi-Markov models from the most strongly associated pairs of events in execution traces; CHAID analysis determines which are most strongly associated [5]. A variety of other methods have been developed for generating graphical models, especially Bayesian Networks and Markov models, most of which are for acyclic graphs [4].

Business and medical applications dominate the applications of data mining methods to event sequence modeling. The *Quest* system at IBM has been used in business for attached mailing, add-on sales and customer satisfaction as well as medical diagnosis [2]. The algorithm for finding sequential patterns counts subsequences and then generalizes the frequently occurring ones to include sets of possible values in positions; the algorithm is extremely efficient and has been designed for mining massive databases [3].

Intelligent agents may use operators for deciding what action to take to achieve a desired state. Agents can learn these operators by modeling their perceptions of state changes in the environment using *Multi-Stream Dependency Detection* (MSDD) [13]. MSDD performs a simple best first search for dependencies, where the sequences are composed of a set of parallel event streams.

2 Building Transition Models of System Execution

State transition models are built by iteratively combining frequently occurring sequences. Thus, the two core ideas underlying the generation algorithm are: finding statistically significantly occurring sequences (snapshot dependencies) to alleviate the effect of noise and carefully checking proposed transitions for consistency with all of the discovered sequences.

2.1 Snapshot Dependency Detection

DD finds dependencies by analyzing event sequences. The idea is to see whether particular subsequences stand out from the whole, based on a statistical analysis. First, the event sequences are scanned to count all subsequences of a length specified by the user. Then, subsequences of influencing events are tested for whether they are more or less likely to be followed by each of the target events using contingency table analysis on the counts. The output is a list of sequences (e.g., $AB \rightarrow C$ which means 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. Due to the contingency table test, dependencies include both frequently and infrequently co-occurring sequences; we call the types *positive* and *negative* dependencies, respectively.

DD has been extended to find more complex patterns[8, 7]. One version collects dependencies up to some length while 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

To run the algorithm, the user provides discrete event sequences (`dataset`) and designates a sequence length (`seq-length`) and a significance `threshold`. The user can also request either a fully connected model or one that is more complete, but that has states without input from the rest of the diagram (`loose-ends?`).

The basic algorithm, which appears in Table 1, searches for subsequences and then iteratively connects together the dependencies, merging states where possible. The resulting diagram often has states composed of multiple events.

The rules for when to link, merge or create new states are based on the information available from DD. This is the most complex part of the algorithm because it requires generating all sequences that could result from the state and

STDD(dataset, seq-length, threshold, loose-ends?)

1. Collect significant `sequences`:
   (a) Find sequences of length `seq-length` below `threshold` in `dataset` using DD
   (b) Prune out negative dependencies (infrequently occurring sequences)
   (c) Cluster sequences by common beginning subsequences (precursor)
   (d) Sort `clusters` by overall frequency of occurrence
2. Create initial state from precursor of most frequent cluster
3. Push initial state on `resolution-list`
4. Repeat until no states in `resolution-list`:
   (a) Get cluster (with precursor and successor subsequences) for first state in resolution-list
   (b) Add successors to the model by:
      – Link to an existing state if all the subsequences 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 kept with clusters,
   (d) Add new and merged states to resolution list.
5. If `loose-ends?` and not all `clusters` have been included, then:
   (a) Create a state from precursor of most frequent remaining on `clusters`
   (b) Push new state on `resolution-list`
   (c) Restart loop at 4

Table 1. STDD algorithm for constructing state transition diagrams

its existing predecessors and successors in the state transition diagram. Checking these potential sequences against those found in the data reduces the cases in which the diagram produces event sequences that do not actually occur.

The diagram is enhanced until all states have both incoming and outgoing links. However, at this stage, some subsequences may not be present in the transition diagram; these subsequences are not included because some of the events do not appear in both the precursor and successor portions of dependencies. If the user wishes (by setting `loose-ends?` to true), these states can be added in as dangling states: states with outgoing, but not incoming links. We display the diagram using the Dot graphing package[11].

The resulting model is not intended to completely capture the original event sequences; instead, because it incorporates only significant dependencies, it is unlikely to include spurious sequences (e.g., spurious events or noise in perception of events) and its complexity is adjustable by changing the threshold and indicating whether to include loose ends.

To show how the algorithm works, we generated data from the diagram in the left side of Figure 1 with a small amount of noise in the data (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

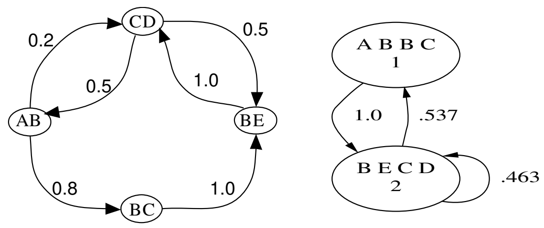


Fig. 1. An idealized five token model (on left) and transition diagram constructed from event data (on right)

linked to two new states, A and B, with transition probabilities $\frac{87}{87+75} = .537$ and $\frac{75}{87+75} = .463$, respectively. On the second pass, B is merged with E by finding the dependency $DB \rightarrow E$ and noting that it matches the existing transition $D \rightarrow B$. On the third pass, state A matches the dependency $DA \rightarrow B$; we cannot link to state BE because we can find no $BB \rightarrow E$ 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 1 resulted. This diagram collapses the four states into two, making it more compact, but also removing one possible transition ($AB \rightarrow CD$), which did not appear in any dependencies.

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

Table 2. Length 3 dependencies (with contingency table) collected for five event model

To illustrate the effect of adding in loose-ends, we ran the same example with more noise (0.1) and a longer seq-length (three). As Figure 2 shows, the diagram on the right includes two new sequences (e.g., [B C B] and [C B E]), but at the cost of many more false positives and a more complicated diagram.

3 Evaluation of State Transition Model Generation

We emphasize three questions for evaluation: how accurately do the resulting models summarize the datasets, what is the effect of the parameters of the algorithm on accuracy and does accuracy scale-up to realistic datasets. To answer the first two questions, we tested STDD's accuracy on synthetic data because we can control its complexity. We varied the complexity of the underlying models, the noise introduced in the data and the number of events to determine the effect on accuracy of varying the datasets; we also varied the parameter settings

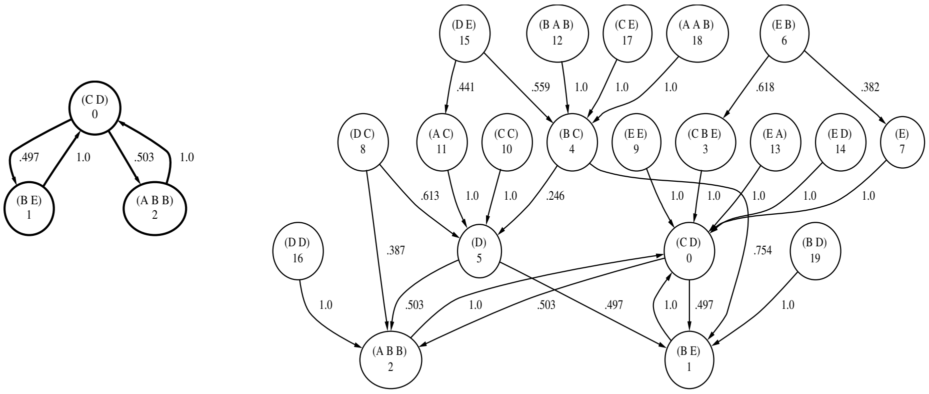


Fig. 2. Diagrams generated with `loose-ends?=nil` (left) and `loose-ends?=T` (right)

of the STDD algorithm. Then, we tested the accuracy on event sequences from real systems to demonstrate that the method scales up to realistic data sets.

We did not specifically test the computational requirements of STDD. Because it is of concern to the data mining community, we report that "typical" transition diagram generation (as exemplified in Table 3) required about a minute on a SPARC Ultra in Allegro Common Lisp for each synthetic example and up to 15 minutes for the actual program traces.

3.1 Accuracy on Synthetic Data

Testing accuracy is a four step process. First, we developed seven synthetic models in which we varied the number of events (5-13), the number of states (3-10) and the maximum number of events per state (2-7); Figure 1 shows a model with five events, four states and two events per state (abbreviated as M5-4-2). Second, for each of the synthetic models, we generated event sequences of varying lengths (2500 to 70,000 events) by simulating the models and introducing noise (events randomly selected from the set with probabilities of 0.05, 0.1, 0.15, 0.20 and 0.25). Third, we constructed state transition diagrams from the data sets using different parameter settings. Finally, we determined the accuracy of each model for each dataset by comparing the subsequences of varying lengths (2-7) that can be produced by the resulting diagrams to data produced without noise.

We measure accuracy as hit-rate and false positives. Hits are the number of subsequences that both appear in the test data and can be produced by the diagram; Hit-Rate is percentage of hits in the datasets. False positives are the number of distinct subsequences that can be produced with the diagram that do not appear in the data.

Effect of Dataset on Accuracy Table 3 shows typical measures for the seven synthetic models; we report the results for seq-length of four, threshold of .05, loose-ends? off, trace length of 10,000 events and noise of 0.1 as the amount of

	5-4-2	5-4-4	5-10-2	6-6-3	8-4-2	13-3-7	13-10-2	Mean
Ideal Hit-Rate	.75	.58	.16	.89	.91	1.0	.80	.73
len 7 False Pos	3288	3746	4993	407	0	306	75	1831
Ideal Hit-Rate	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
len 2 False Pos	567	0	1254	61	2997	1111	440	918

Table 3. Accuracy of typical transition diagram generation on synthetic models; *len* means the length of sequences checked for accuracy.

data and noise level seemed representative of likely data and these parameters appeared to give the best trade-off of hit-rate to false positives overall.

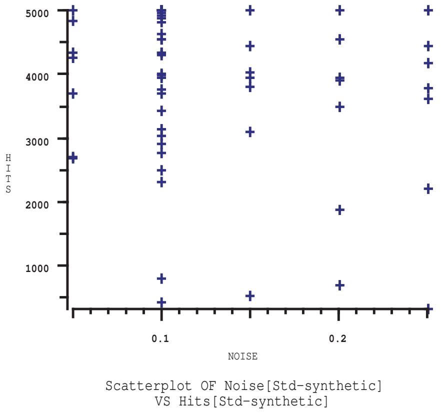
Table 4 summarizes the accuracy across all datasets and parameter settings. The results indicate that most diagrams appear to be capturing most of the dynamics represented by the event sequences. The resulting diagrams have high hit-rates and low false positive rates under most of the tested conditions. A closer examination of the data showed that the measures are best for short sequences (seq-length of 2) with performance degrading as the sequence length increases. Models with fewer types of events tend to be more susceptible to the degradation.

	5-4-2	5-4-4	5-10-2	6-6-3	8-4-2	13-3-7	13-10-2	Median
Hit-Rate								
Best	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Median	.72	.47	.55	.90	1.0	1.0	.86	.86
Worst	0	.02	0	0	.18	.50	.10	.02
False-Pos								
Best	0	0	0	0	0	0	0	0
Median	998	800	3,484	464	4	0	58	464
Worst	31,482	49,989	69,998	49,996	49,997	1,012	43,342	49,989

Table 4. Accuracy of STDD on synthetic models across all trials

The results also suggest variability in accuracy on different types of models. To determine whether the variability was significant, we ran one way ANOVAs with a model characteristic as the dependent variable and an accuracy measure (hit-rate or false positives) as the dependent. We found that accuracy depends on the model characteristics ($P < .002$ in all cases). More events, less states and more events per state tend to result in more accurate diagrams; this result is not surprising because the extra constraints of more events and more events per state should help separate the true model from the noise.

One-way ANOVAs on noise levels for each model showed a significant effect of noise on accuracy ($P < .0003$) across all trials; examination of the data shows that some parameter settings are more sensitive to the noise than others. For example, the typical parameter settings mentioned earlier were shown to be relatively insensitive to noise (see figure 3), while longer seq-lengths were more so. We expected this because fewer examples of longer sequences will be found.



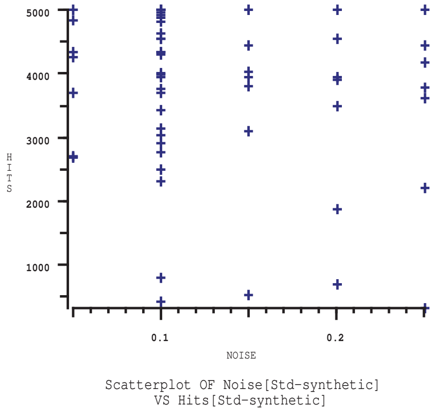


Fig. 3. Noise versus hit-rates and false positives for typical parameter settings

Effect of Algorithm Parameters on Accuracy The algorithm requires the user to make some commitments to parameter settings. We tested the sensitivity of the algorithm to the parameters: `loose-ends?`, `threshold` and `seq-length`. For loose-ends?, we ran t-tests comparing the accuracy of adding the loose ends to not including them for each model for each of the two measures. We found a significant difference ($P < .0001$) for hit-rate on every model; however, for false positives, we found no significant effect for four of the models ($P < .36$) and significant effects ($P < .001$) for models 8-4-2, 6-6-3 and 5-10-2. In general, adding in loose-ends significantly improves the hit-rate while sometimes significantly degrading the false positive rate. As expected, the models are significantly larger when loose ends are included.

Because seq-length and threshold are related through the DD algorithm, we tested the effect of these parameters using two two-way ANOVAs (one for each measure) for each model. In all cases, we found a significant main effect of seq-length ($P < .0001$). In five models, we found a significant main effect of threshold ($P < .01$) on hit-rate; in four models, we found a significant main effect of threshold on false positives ($P < .01$). Only two models exhibited any significant interaction effect (models 5-4-2 and 6-6-3), which suggests that we can set these parameters independently. As expected, the size of the model increases as the threshold becomes more lax.

Both seq-length and threshold affect accuracy. However, the effect is model dependent and is not monotonic. Each model has an ideal parameter setting to minimize false positives and maximize hit-rate. From a scatterplot of the hit-rate versus false positive rate for two models (Figure 4), we see that both models differ significantly on the trade-offs between the two measures and that for the model on the right it is easier to obtain both high hit-rate and low false positives.

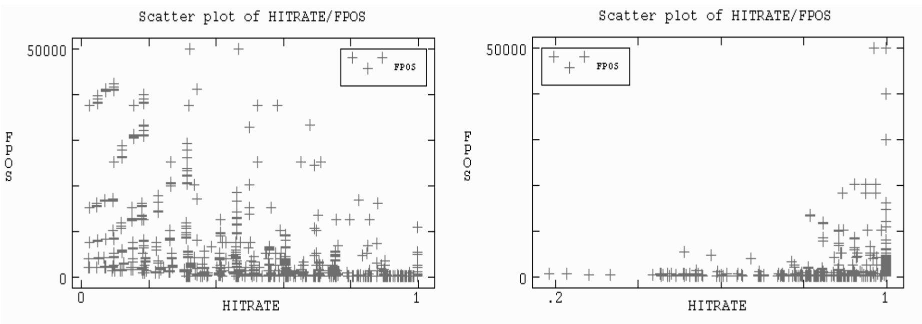


Fig. 4. Scatterplots of false positives and hit-rates for two models: m5-4-4 on left and m8-4-2 on right

3.2 Modeling RARS Controller and Phoenix Planner

We had previously found snapshot dependencies useful for debugging. We have started to build a multi-level, reinforcement learning controller for an agent in a simulated environment (RARS). In this section, we give an example of how we are using transition diagram generation to debug RARS and an assessment of the accuracy of STDD on actual program traces.

Debugging the RARS Controller The RARS (Robot Automobile Racing Simulator) 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 [15]. We would like to the controller to avoid failures: when the controller loses control of the car and runs off of the track (`crashed` event). For example, Figure 5 shows a fragment of a transition diagram[2] generated from event data from RARS. At this point, the controller has been partially trained to negotiate the track, but is still crashing occasionally. The events are discretized sensor readings: track position (P) and speed relative to a wall (X).

From Figure 5, we identified several problems. First, we observe bottlenecks immediately before and after all states containing the crashed event (highlighted states). These bottlenecks are critical events indicating cases in which training

[2] 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.

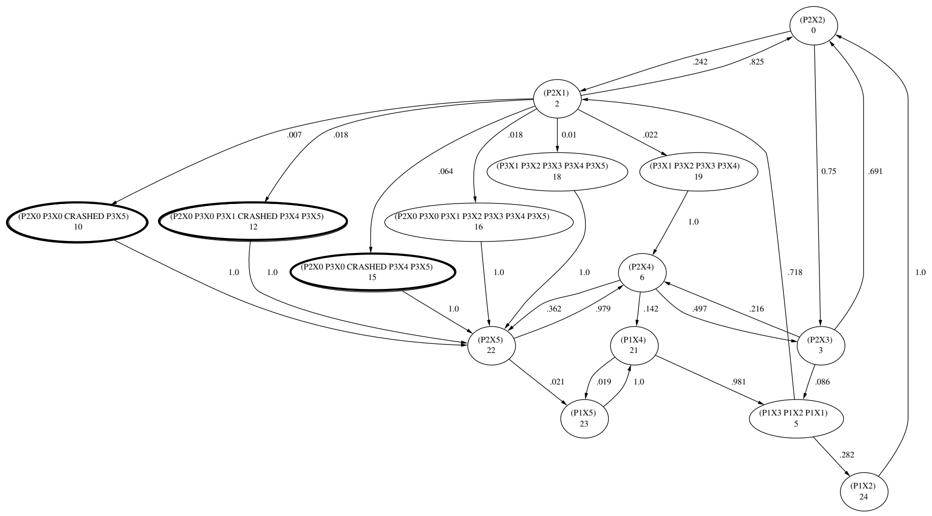


Fig. 5. Portion of transition diagram for RARS reinforcement learning controller

and reinforcement should be focused. 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.

Accuracy on Real Data: RARS and Phoenix We assessed the accuracy of STDD on real data from two systems by comparing the transition diagrams produced from half of the data to the remaining data. Thus, we are testing to see whether a more compact overview model can adequately summarize what was observed.

	Phoenix				RARS			
	3		4		3		4	
	.05	.005	.05	.005	.05	.005	.05	.005
Hit-rate	.81	.73	.95	.83	.98	.97	.98	1.0
len 2 False pos	0	0	0	2	16	0	4	11
Hit-rate	.34	.27	.18	.11	.70	.67	.80	.68
len 7 False pos	1944	37	1459	1430	2653	2887	913	1482

Table 5. Accuracy results for Phoenix and RARS event sequences

We generated transition diagrams using two seq-lengths (3 and 4) and two thresholds (.05 and .005) for the two datasets. Table 5 summarizes the results for subsequences of length two and seven. As with the synthetic, the diagrams accurately capture the short sequences. Performance degrades more slowly on RARS than on Phoenix with longer sequences. In fact, as with the synthetic data, the more robust performance was achieved with more events (30 for RARS as opposed to 17 for Phoenix). Finally, the results showed little difference between the diagrams generated with these different parameter settings.

4 Extensions and Conclusions

Given the current performance of the algorithm, we view two issues as paramount for future work: improving the false positive rates through better filtering and automatically setting the parameters in the algorithm. At present, we have two mechanisms for filtering noise: reducing the threshold and pruning negative dependencies. We have tested the effect of threshold and found it to be model dependent, further supporting the need to better set the algorithm parameters. As our next study, we intend to focus on testing alternative strategies for pruning negative dependencies.

The algorithm requires three parameters: loose-ends?, seq-length and threshold. Loose-ends? is best set by the user who knows what is most critical to the diagram generation (hit-rate or simplicity). As to the other two, we are currently exploring principled methods of setting these. One option is to use CHAID based analysis to determine seq-length. CHAID constructs an n-step transition matrix, which indicates what earlier point in the event sequences was most predictive of the occurrence of each event[5]. Unfortunately, integrating CHAID based analysis requires significant modification to the underlying code to accommodate multiple dependency lengths simultaneously, and so is yet to be completed.

The significance threshold (probability for the contingency table test) is the primary mechanism for adjusting the complexity of the daigram. Lower probabilities tend to result in less complex diagrams because fewer transitions are recognized as significant. One option for determining how to set the probability threshold is to model its effect. We hope that further analysis of the results presented here will uncover a model of how to set the threshold.

We have proposed a new algorithm for automatically generating state transition diagrams from discrete event sequences. The accuracy of the diagrams generated from this algorithm is quite high for both the synthetic and real data that we have tested and shows gradual degradation as datasets become more difficult to characterize (decreasing dataset sizes with increasing noise levels). Our algorithm extends existing representational options because it does not require that the underlying process be Markov or that the resulting model be acyclic. Finally, ultimately, the utility of the algorithm rests in the utility of the models generated. We view these models as concise descriptions of cyclic processes. We have used the models to help debug the RARS reinforcement learning control system and have started using them to characterize protocols of human programmers and as an internal representation of behavior for a multi-layer simulated robot learning system.

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. Department of Computer Science 94-21, Virginia Polytechnical Institute and State University, June 1994.
2. Rakesh Agrawal, Manish Mehta, John Shafer, and Ramakrishnan Srikant. The Quest data mining system. In *Proceedings of the Second International Conferrence on Knowledge Discovery and Data Mining*, Portland, OR, Aug. 1996.
3. Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the Int'l Conference on Data Engineering (ICDE)*, Taipei, Taiwan, March 1995.
4. Wray Buntine. Graphical models for discovering knowledge. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*. AAAI Press, Menlo Park, CA, 1996.
5. Horacio T. Cadiz. The development of a CHAID-based model for CHITRA93. Computer science dept., Virginia Polytechnic Institute, February 1994.
6. Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37–54, Fall 1996.
7. 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.
8. 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*, volume 89 of *Lecture Notes in Statistics*, chapter 8, pages 71–78. Springer-Verlag, NY,NY, 1994.
9. Adele E. Howe and Paul R. Cohen. Understanding planner behavior. *Artificial Intelligence*, 76(1-2):125–166, 1995.
10. 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.
11. Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot.* AT&T Bell Laboratories, Murray Hill, NJ, October 1993.
12. Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovering frequent episodes in sequences. In *Proceedings of the International Conference on Knowledge Discovery in Databases and Data Mining (KDD-95)*, Montreal, CA, Aug. 1995.
13. Tim Oates and Paul R. Cohen. Searching for planning operators with context-dependenct and probabilistic effects. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.
14. Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1988.
15. 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.
16. 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*, Montreal, CA, August 1995.