THESIS

STATE TRANSITION DIAGRAM DEPENDENCY DETECTION

Submitted by

Gabriel L. Somlo

Department of Computer Science

In partial fulfillment of the requirements

for the degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 97

COLORADO STATE UNIVERSITY

September 11, 2004

We hereby recommend that the thesis STATE TRANSITION DIAGRAM DEPENDENCY DETECTION prepared under our supervision by Gabriel L. Somlo be accepted as fulfilling in part requirements for the degree of Master of Science.

Committee on Graduate Work

Committee Member

Committee Member

Adviser

Department Head

ABSTRACT OF THESIS

STATE TRANSITION DIAGRAM DEPENDENCY DETECTION

I present an algorithm that builds state transition diagrams out of event traces, in order to find causal relationships between the various events in these traces. The main application of this algorithm is high-level debugging (for situations where it is difficult or impossible to replicate a specific instance of a failure). But many other uses, such as market prediction, credit card fraud tracking, and data mining, are also possible. The algorithm is the latest in a family of statistics-based techniques for modeling process behavior, called *Dependency Detection*. It collects relatively short, significant sequences (*snapshots*), to generate an integrated, abstract *overview* model of the analyzed process. Also, detailed performance and accuracy evaluations of the algorithm are presented.

Gabriel L. Somlo
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Fall 97

CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Many processes can be viewed as long sequences of discrete events in time. Even continuous processes can be discretized by sampling their variables at regular or irregular intervals of time. A few examples are: monitoring devices in a hospital emergency room, the control of a chemical process, the sequence of purchases recorded at a store, or the sequence of actions, failures, and recovery measures taken by a computer planner. The modeling of such sequences can offer insights into the workings of some process, like predicting the most probable future states, or describing how the process reached its current state.

In this thesis I present an algorithm for modeling processes of the type described above. This algorithm integrates previously existing methods in order to combine their strengths, and to create a more accurate model of the analyzed process. The existing algorithms analyze short pieces of the process execution trace (called subsequences), and use statistics to find significant subsequences which have unusually high or low frequency of occurrence (dependencies). They are part of a family of algorithms called Dependency Detection, or in short, DD (described in [7, 8, 9, 10, 11]). The resulting dependencies are sometimes referred to as *snapshots*.

The algorithm I developed uses the snapshots obtained from members of the DD algorithm family, and integrates them using heuristic techniques. The result is an *overview* model, which in fact is a state transition diagram (STD). This overview or STD model is composed of a set of statistically significant patterns extracted from the process execution trace, together with information about the interconnection between these patterns. I will refer to this algorithm as STDD (short for State Transition diagram Dependency Detection).

Snapshots, while useful and easy to implement, are not powerful enough to capture all aspects of the process being modeled. Using snapshots we can find out a lot about small portions of the analyzed process. However, in order to have an overview of the process as a whole, we need to somehow integrate these snapshots. Snapshots also make it difficult to detect cyclic dependencies. The simplest way to illustrate this latter aspect is the chicken-and-egg problem. If we'd attempt to analyze an execution trace of the form:

... **chicken**, **egg**, **chicken**, **egg**, **chicken** ...

using snapshot DD with a window of length two, we would obtain two apparently contradictory snapshot dependencies: (**chicken** → **egg**), and (**egg** → **chicken**), both with 50% frequency. On the other hand, if we would use an overview algorithm to

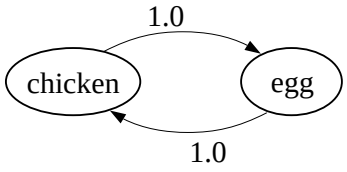

Figure 1.1: Overview STD model of the chicken-and-egg trace

generate an STD of the trace, we would obtain a model like the one depicted in Figure 1.1. The cyclic relationship between the "chicken" and the "egg" becomes evident, without any ambiguity or contradiction.

To illustrate how to use this algorithm, imagine a system with two useful states, **a** and **b**, and an error state, **e**. The system has a set of built-in heuristic rules that makes it switch states every once in a while. We want to keep the system switching between **a** and **b**, and avoid **e** as much as we can. To see if we have achieved the desired behavior, we run STDD on an execution trace of this system, to obtain the graph depicted in Figure 1.2. The system moves from **a** to **b** with probability 1.0, and from **b** to **a** with probability 0.5. If we are not pleased with the fact that from state **b** we end up in **e** 50% of the time, we use domain-specific knowledge of our system to modify the heuristics involved. We can obtain very helpful feedback on our changes by re-running STDD on the updated system. If the chance of ending up in the error state has decreased, while the desirable transitions remained unchanged, we know our modifications were successful. Otherwise, we know we have to backtrack, and change something else in order to improve the system.

Chapter 2 presents some details on the snapshot DD family of algorithms and its underlying statistics, the steps which led to the current form of STDD, and an

Figure 1.2: Overview model of a system with two useful states and an error state

overview of related work. Chapter 3 is a detailed presentation of the algorithm. The various performance tests that I have run on the algorithm, as well as some of the experiment design issues are presented in Chapter 4. The results I have obtained and their interpretation can be found in Chapter 5. Chapter 6 summarizes the conclusions and presents some thoughts about future improvements to the algorithm.

Chapter 2

BACKGROUND

The first part of this chapter is a detailed presentation of the snapshot DD family of algorithms, on which STDD is based. The second part reviews the research that directly led to the development of STDD. The third part describes a series of related applications. The last part presents a series of algorithms that are somewhat related to the DD family.

2.1 Underlying Statistics

This section is intended to give a quick overview of the inner workings of snapshot DD in order to facilitate the understanding of the STDD implementation.

A snapshot DD algorithm finds dependencies by analyzing execution traces (which are sequences of events). Some of the events are considered to be causes (also called *precursors*), and others are referred to as consequences, i.e. the events of interest (called *targets*). A target is said to *depend* on a precursor if it is observed to follow that precursor more often than any other event in the trace.

Snapshot DD finds particular subsequences of the execution trace that stand out from the whole using statistical analysis. First, the event sequences are scanned to count all subsequences of a given length. Then, subsequences of influencing events

(potential precursors) 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 (see subsection 2.1.1). The output is a list of the dependencies (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 precursor appears as often as any other sequence of events in front of the target (i.e. the lower this probability, the stronger the dependency). This probability is usually referred to as the *significance* of the dependency. The user supplies a snapshot DD algorithm with a significance *threshold*, which causes all dependencies that are not significant enough to be discarded.

2.1.1 Contingency Table Analysis

The main component of snapshot DD is contingency table analysis to determine whether a particular target event T is more likely to occur after a particular precursor P. Four types of sequences are counted in the execution traces:

1. Instances of T that follow instances of P;
2. Instances of T that follow precursors other than P;
3. Targets other than T that follow instances of P;
4. Targets other than T that follow precursors other than P.

From these frequencies of occurrence we can build a 2×2 contingency table like the one in Figure 2.1. A statistical test of independence, the G-test (see details in the following paragraph), indicates whether the two events are likely to be dependent. In

	T	$\overline{T}$
P	25	9
$\overline{P}$	30	40

Figure 2.1: Contingency table used for computing the significance of a dependency

this case, $G = 8.92$, with significance $p \leq 0.0028$, which means that the two events are unlikely to be independent and conclude that T depends on P. A dependency is added to the list of results if p falls below a predetermined threshold.

Tests of Independence: Chi-Square vs. G-Homogeneity

Contingency table analysis is based on the concepts of *independence* and *homogeneity*. If the variables in a contingency table are independent, the counts are homogeneously distributed across the cells of the table. A more exact definition of independent variables (according to Ott [20]) is:

> "Two variables that have been categorized in a two-dimensional contingency table are *independent* if the probability that a measurement classified into a given cell of the table is equal to the probability of being classified into that row times the probability of being classified into that column. This must be true for all cells of the table."

In our earlier described case (see Figure 2.1), the two variables are the precursor (with two values, P and $\overline{P}$), and the target (with values T and $\overline{T}$). The *null hypothesis* of any test of independence is that there is no relationship between the variables, i.e. they are independent. We are trying to contradict this, and support the *alternative hypothesis*, that the two variables are dependent, i.e. they form a dependency.

One such test of independence is the *Chi-Square* test:

$$\chi^2 = \sum_{i,j} \left[\frac{(n_{ij} - E_{ij})^2}{E_{ij}} \right]$$

where n_{ij} and E_{ij} are, respectively, the observed and expected number of measurements falling in the cell situated at row i and column j in the contingency table. The *expected number of measurements*, E_{ij} falling in cell (ij) of the table should be

$$E_{ij} = \frac{(\text{total for row } i) \cdot (\text{total for column } j)}{\text{total for the entire table}}$$

The *significance* (or p-value) of such a test measures the probability of obtaining a certain value of χ^2 for a contingency table of a given size, due to random variation in the data. The lower the value of p, the less likely it is that our variables are independent. Significance values are tabulated in statistics books, but can be computed with a much higher accuracy using computer software packages like CLASP [4]. As an example, for the contingency table in Figure 2.1, we obtain $\chi^2 = 7.45$, and $p = 0.0063$.

Another statistic for determining independence is the G-test, recommended by Sokal and Rohlf [23], for 2×2 contingency tables with row and column totals that are not fixed. This type of contingency table is referred to as "Model I". Only the grand total of the cells in the table (the sample size) is fixed. For a test of the null hypothesis one should compute the probability of obtaining the observed or worse differences between actual and expected cell counts out of all possible tables with the same grand total (sample size). In contrast, a "Model II" table would have fixed row or column totals, and independence tests other than the G-test would be appropriate.

The formula for the G-test is:

$$G = 2 \cdot \sum_{i,j} n_{ij} \ln \frac{n_{ij}}{E_{ij}}$$

where n_{ij} and E_{ij} have the same meaning as in the Chi-Square test presented before. The G-test verifies whether the ratio of T to $\overline{T}$ in Figure 2.1 is significantly different from one row of the contingency table (P) to the other ($\overline{P}$). For this reason it is often referred to as a test of *heterogeneity*, i.e. it tests whether the ratios of cell frequencies in the two rows are heterogeneous (the more heterogeneous these rows are, the more significant is our dependency).

The use of the G-test in snapshot DD is also preferred by Howe and Cohen [9] for the *subsumption* snapshot DD algorithm (see subsection 2.1.2), because unlike Chi-Square it is fully *additive* (as described in [23]).

2.1.2 Types of Snapshot Dependencies

The simplest form of snapshot DD is called *absolute order* DD. Subsequences of length n are counted and analyzed. The first $n-1$ events in a sequence are considered to be the precursor, and the last, nth event will be the target.

Subsumption DD (presented in [9]) uses the additivity of the G-test (see 2.1.1) to find the best predictors of any given target. This algorithm is able to distinguish whether a shorter dependency is subsumed by a longer one. This means that the longer dependency extends the precursor of the shorter one, while at the same time being more significant (in other words, a better predictor has been found for a given target).

A more complex snapshot DD algorithm is *heuristic* DD [7, 10], which attempts to heuristically identify good predictors for a given target. The events in the precursor occur in a specified order within a maximum-length window (user specified). It doesn't matter if there are other events intervening between the components of the precursor, or between the precursor and the target. The search for dependencies starts out with all possible combinations of two events. The significant dependencies of length n are extended to length $n + 1$ with all distinct events in the trace, and retested for significance. Significant longer dependencies replace the shorter ones from which they have been extended. Thus, the longest predictors are found for every target.

2.2 Work Leading to the Development of STDD

Howe and Cohen [9] introduced a partially automated methodology for understanding planner behavior over long periods of time. It used statistical dependency detection to identify significant behavior patterns in execution traces. These patterns were interpreted using a model of the planner's interaction with its environment. The main goal was to identify possible causes of plan failures in Phoenix, a planner developed to help fight fires in a simulation of Yellowstone National Park. The use of statistical methods was justified by the subtle context differences and the stochastic decision making process used to select a certain failure recovery strategy, as well as by the long simulation time (hours, days), which made the exact replication of a particular failure practically impossible. This methodology was specially developed to fit the Phoenix planner, and generated simple models for selected parts of the

failure recovery traces. The methodology was found to be most appropriate for those cases where only execution traces, and no complete model of the planner and its environment were available.

The statistical analysis used for Phoenix was later developed and generalized for debugging AI systems [8]. This resulted in the family of algorithms called Dependency Detection (DD). These algorithms are used to search execution traces for patterns with an unusually high or low frequency of occurrence (dependencies). The statistical method employed by the DD algorithms to identify dependencies is contingency table analysis. Throughout the rest of this thesis I will refer to the algorithms of this family of algorithms as *snapshot* DD, to emphasize the fact that they collect short sequences that are statistically significant.

Two new approaches to snapshot DD are introduced by Howe and Fuegi [10] in an attempt to overcome limitations to temporally adjacent events in the execution trace. The first of the two new approaches is *heuristic* search, where short sequences that are found to be significant are extended into longer ones. The second is *local* search, which starts out with a randomly chosen pattern whose component events are then changed one at a time until a significant dependency is obtained.

The idea of local search is further extended by Howe [7]. A new, improved test for matching candidate dependencies with subsequences of the execution traces is introduced, along with local search to prune the number of dependencies to be examined and thus reduce the computational complexity of the algorithm.

Snapshot DD algorithms only have a local view of the program behavior. To correct for this shortcoming, Howe and Pyeatt [11] introduced an *overview* DD method

which integrates the snapshot DD family with analysis based on CHAID (Chi Automatic Interaction Detection, see also [5]). The goal was to produce an abstract model of system behavior in the form of a transition diagram of merged states (a complete, semi-Markov model of the sequence of events). This approach produced promising results. However, the authors observed the need to add better aggregation and pruning criteria when building the transition diagram.

2.3 Related Applications

The first applications of snapshot DD were in planning. Howe and Cohen [9] used one of the first versions to debug the Phoenix planner. Another immediate application of snapshot DD algorithms was identifying search control problems in the UCPOP planner [24]. Using snapshot DD, Srinivasan and Howe determined that UCPOP sometimes searched in circles, repeatedly trying the same variable bindings, and showed that a new construct was needed in the plan language to prevent this problem.

Oates et al. [19] tackled snapshot DD from a different angle, adapting it to handle multiple simultaneous streams of execution traces. The technique, called Multistream Dependency Detection (MSDD) is targeted towards applications such as simultaneously monitoring multiple medical instruments. MSDD was also used by Oates and Cohen [18] as a learning mechanism for planning, in order to find operators that capture an agent's interaction with its changing environment.

Howe and Pyeatt [11] also present two applications of an overview DD algorithm: an improved analysis of Phoenix's failures, and the analysis of an agent steering a

virtual racing car in a simulated environment.

Techniques similar to snapshot DD have been developed by Agrawal et al. [2, 3] at IBM, targeted specifically towards the data mining domain. Their system, known as the "Quest Data Mining System", is used for a wide range of applications: attached mailing, add-on sales, customer satisfaction analysis, medical diagnosis, etc.

2.4 Other Related Work

Agrawal et al. [2] model a process as a collection of rules, in order to emphasize mutual relationships between key events. An example would be that the purchase of the first book in a trilogy is often followed by the purchase of the second and third books. This algorithm, known as "Sequential Patterns" is the principal component of the Quest Data Mining System. Agrawal and Srikant [3] describe an improved algorithm for mining sequential patterns, where the user is allowed to request that each pattern have a minimum support level (similar to the significance level of a snapshot dependency). Time constraints are added to specify a minimum and/or maximum time period (distance) between adjacent elements in a pattern. Efficiency of the algorithm is a priority when mining very large databases.

Research on an overview DD algorithm is presented by Abrams et al. [1]. Chitra94 is a visualization tool that generates semi-Markov models to debug interactions within distributed computational processes. These models are built from the most strongly associated pairs of events in the execution trace. CHAID, introduced by Kass [14], was integrated into Chitra by Cadiz [5], to improve the representation of dependencies between events that are not adjacent in the execution trace.

Pearl and Verma [21] propose a different approach to building overview models. They introduce a minimal-model semantics of causation and demonstrate the possibility of inferring causal influences from spurious covariations by using inductive reasoning. An algorithm that can uncover the direction of causal relationships for a large class of data is also presented. This model differs from the CHAID-Chitra (see [1, 5]) and STDD models by the fact that its underlying mathematics only allow it to build directed *acyclic* graphs (DAGs) of causal relationships. Cycles cannot be represented.

Fayyad et al. [6] present an overview of the field of Data Mining and Knowledge Discovery in Databases (KDD): main areas of application, principal components of a KDD and Data Mining algorithm, and the challenges these algorithms are facing. Among others, *probabilistic graphic dependency models* are presented as a significant new trend in Data Mining methods. These models specify which variables directly depend on each other, and are important mainly because their graphic form easily lends itself to human interpretation.

Chapter 3

THE ALGORITHM AND ITS IMPLEMENTATION

State Transition diagram Dependency Detection (STDD)[1] works by iteratively incorporating significant sequences of events from the list of snapshot dependencies generated with one of the previously described algorithms (see subsection 2.1.2). The result is a model which, although not guaranteed to be complete, gives the user a fairly good idea of the main activity going on in the system that is being analyzed.

Figure 3.1 shows the pseudocode for the basic STDD algorithm. The word "event" will refer to an atomic component of the process execution trace - a token indicating some aspect of the process being modeled. "State" will denote a state of the STD, which can be an entire sequence of events - states of the original process. The main components are:

1. Obtaining and preprocessing snapshot dependencies, described in section 3.1;

2. Generating the "seed" - the starting state of the STD - out of the most frequent precursor from among the dependencies;

3. Building the STD, described in section 3.2.

[1] I developed STDD from an idea and initial code given to me by my advisor, Dr. Adele Howe.

1. Collect and organize snapshot dependencies (see section 3.1):
   (a) Run specified snapshot DD algorithm with specified maximum dependency length, and with optional trace compression (see subsection 3.1.1);
   (b) Eliminate anti-dependencies (see subsection 3.1.2);
2. Cluster dependencies according to precursors, and sort clustered list in decreasing order of precursor frequency;
3. Start with a state created from the most frequent precursor (the *seed*); maintain a resolution list (a list of states that need to be resolved for outgoing links), and repeat until no more states need to be resolved (see section 3.2):
   (a) Find all dependencies that would extend the current state by one event (called an *extending event*);
   (b) For each distinct extending event, attempt to find an existing state (a *legal destination*), starting with the current extending event, such that a link between the current state and the legal destination state would be consistent with all known snapshot dependencies. Resolve current state as follows:
      - if a legal destination exists for the current event:
        - if this is the only extending event, and the legal destination has no incoming links, merge the current state with the legal destination into a new state that replaces both (note: this can only happen if the legal destination is the *seed*);
        - otherwise insert a link from the current state to the legal destination;
      - else (if no legal destination exists for the current event):
        - if this is the only extending event, append it to the end of the current state, and append the result to the resolution list for further processing;
        - otherwise create an entirely new state out of the current extending event, append it to the resolution list, and insert a link from the current state to the newly created one;
   (c) Compute transition probabilities for the newly created links in the STD.

Figure 3.1: Pseudocode for the State Transition diagram DD algorithm (STDD)

3.1 Obtaining Snapshot Dependencies

The first step of STDD - concerned with obtaining snapshot dependencies - requires six parameters (the first two are mandatory, and the rest are optional and have default values):

1. The process execution trace;
2. The maximum length of the snapshot dependencies;
3. Significance threshold for accepting/rejecting a dependency. If no value is supplied for the significance threshold, the default value of 0.05 will be used;
4. Selector for which snapshot DD method to use. This parameter defaults to absolute order snapshot DD;
5. Toggle for trace compression (see subsection 3.1.1 for details). This parameter is turned on by default;
6. Toggle for anti-dependency filtering (described in subsection 3.1.2) - also on by default.

The first three parameters are passed along to the snapshot DD method selected by the fourth parameter. The meaning and effects of the last two parameters will be described in the following subsections.

3.1.1 Trace Compression

Trace compression detects pairs of events that always occur together. These can be viewed as highly significant absolute order snapshots of length two. Trace compression replaces any subsequence of the execution trace that matches these pairs with a single new event that reflects them. This happens before the actual snapshot DD algorithm is called.

This is a preprocessing step, designed to deal with problems that occur while using absolute order snapshot DD due to the fixed, constant length of the snapshots. When the process that is being modeled contains events that are part of both long and short sequences that frequently repeat themselves, STDD might create spurious transitions and fail to correctly reflect the longer sequences. Trace compression forces STDD to see the long sequences as distinct events and thus eliminates this problem.

3.1.2 Filtering Anti-Dependencies

Due to the contingency table test (see subsection 2.1.1), dependencies include sequences that occur both very frequently and very infrequently. The infrequent ones are often called *anti*-dependencies. For the purpose of building an overview algorithm based on snapshot DD, we need to filter out these anti-dependencies.

Filtering out anti-dependencies is an optional post-processing step in obtaining the snapshots for building the overview model. In its current form, the anti-dependency filter keeps only significant sequences whose contingency table satisfies the property:

$$\frac{PT}{P\overline{T}} > \frac{\overline{P}T}{\overline{P}\overline{T}}$$

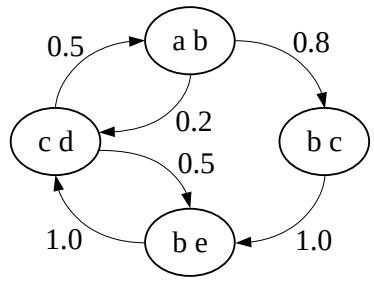

Figure 3.2: Synthetic model used for generating a test execution trace

where P ($\overline{P}$) and T ($\overline{T}$) represent the presence (absence) from a sequence of the precursor and target, respectively (see also Figure 2.1 on page 7). The rationale of this test can be explained easily if we rewrite the above formula as follows:

$$\left.\frac{T}{\overline{T}}\right|_{P} > \left.\frac{T}{\overline{T}}\right|_{\overline{P}}$$

(in other words, the ration $T/\overline{T}$ given the presence of P should be grater than the same ratio when P is not present). This means that we only keep dependencies if their target has better chances of appearing when the precursor is present than when it is absent.

3.2 Generating an STD Model: An Example

Figure 3.2 depicts a synthetic model, which has been executed for $10,000$ steps to generate an execution trace. Running snapshot DD (absolute order, for instance) on the synthetically generated execution trace, we obtain the snapshots depicted in Figure 3.3.

Step 2 of the algorithm finds the most frequent precursor across all snapshot dependencies, in order to use it as the *seed* of the STD to be created. This is done by

Dependency	Contingency Table			
	PT	$P\overline{T}$	$\overline{P}T$	$\overline{PT}$
(b e c)	1612	0	888	7498
(e c d)	1612	0	171	8215
(c d a)	888	894	0	8216
(d a b)	888	0	2328	6782
(a b b)	717	171	2499	6611
(b b c)	717	0	1783	7498
(b c b)	717	171	2499	6611
(c b e)	717	0	894	8387
(c d b)	894	888	2322	5894
(d b e)	894	0	717	8387

Figure 3.3: Snapshot dependencies obtained from synthetic execution trace

clustering the dependencies around common precursors. For instance, (**c d a**) with frequency of occurrence $f_{(\text{c d a})} = 888$ and (**c d b**) with $f_{(\text{c d b})} = 894$ share the same precursor, and therefore the cluster ((**c d**) 1782(=888+894) {**a b**}) will result. The clustered snapshot dependencies, sorted in decreasing order of precursor frequency is presented in Figure 3.4.

Step 3 starts out with an STD consisting of the most frequent precursor from the

Precursor	Freq.	Targets
(c d)	1782	{a b}
(b e)	1612	{c}
(e c)	1612	{d}
(d b)	894	{e}
(d a)	888	{b}
(a b)	717	{b}
(b b)	717	{c}
(b c)	717	{b}
(c b)	717	{e}

Figure 3.4: Snapshot dependencies clustered according to precursors

list in Figure 3.4, i.e. (**c d**). A *resolution list* is maintained with all the states that still need to be processed for outgoing links. In our case, the resolution list will also be initialized to (**c d**). At each iteration of step 3, a state is pulled off the resolution list, resolved for outgoing links, and the new unresolved states that may be generated in the process are added to the end of the resolution list. This is repeated until the resolution list becomes empty, i.e. no more states without outgoing links exist.

Resolving each current state starts out by finding all snapshot dependencies whose precursors are consistent with the state (and the other states providing incoming links to the current state, if necessary). The targets of all these dependencies form a set of *extending events*. To be more specific, at this point, the current state is a "dead end" in the STD, and we are looking for dependencies that would extend this dead end by one event. As an example, if our current state is (**c d**), then we find dependencies (**c d a**) and (**c d b**) which have precursors that are consistent with it, and the set of extending events will be {**a**, **b**}.

For each distinct extending event, the algorithm has to make a decision about how to extend the current state. If a state starting with the current extending event can be found in the STD, such that existing dependencies are consistent with a link to it from the current state (this is called a *legal destination*), two possible actions are possible. If the current extending event is the only one in the set of extending events, and if the legal destination does not currently have any incoming links, then the current state and the legal destination can be merged. Otherwise, a link will be added from the current state to the legal destination. In neither case will any new states be created and added to the resolution list.

If there is no legal destination for the current event, we again have two choices. If the current extending event is the only extending event, it will be added to the end of the current state, which then will be re-inserted at the end of the resolution list. Otherwise a completely new state will be created from the extending event and added to the resolution list.

The last part of the iteration of step 3 of the algorithm will be to compute transition probabilities for any newly added link in the STD. This is done using the frequencies of the dependencies used to find the extending events.

To illustrate the algorithm, let's continue the example we have started earlier. No states starting with events **a** and **b** currently exist in the STD. There are two extending events in the set, so therefore a new state will be created for each of them (see Figure 3.5). The transition probabilities are computed using the frequencies of occurrence of the dependencies (**c d a**) and (**c d b**) from Figure 3.3 on page 20. We have $f_{(\mathrm{c\ d\ a})} = 888$ and $f_{(\mathrm{c\ d\ b})} = 894$. Thus we calculate

$$p_{(\mathrm{c\ d})\rightarrow(\mathrm{a})} = \frac{f_{(\mathrm{c\ d\ a})}}{f_{(\mathrm{c\ d\ a})} + f_{(\mathrm{c\ d\ b})}} = \frac{888}{888 + 894} \approx 0.49$$

and

$$p_{(\mathrm{c\ d})\rightarrow(\mathrm{b})} = \frac{f_{(\mathrm{c\ d\ b})}}{f_{(\mathrm{c\ d\ a})} + f_{(\mathrm{c\ d\ b})}} = \frac{894}{888 + 894} \approx 0.51$$

After this first iteration of step 3 of the algorithm, state (**c d**) is resolved, and the resolution list now contains states (**a**) and (**b**). The intermediate STD is depicted in Figure 3.5.

Next, state (**a**) is pulled off the resolution list. The dependency that is consistent with it and its incoming link is (**d a b**), and thus the only extending event for this

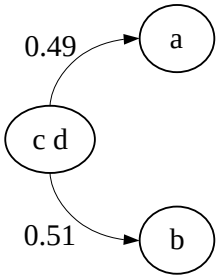

Figure 3.5: Adding new states to the STD

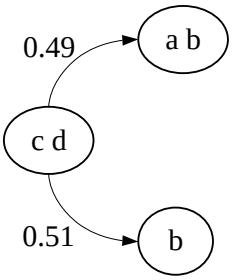

Figure 3.6: Appending events to existing states of the STD

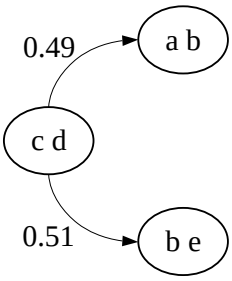

Figure 3.7: Appending events to existing states of the STD (continued)

state is **b**. A state starting with this event already exists in the STD, but a link to it would not be consistent with (supported by) the existing dependencies. In other words, such a link would require that all dependencies with precursor (**a b**) and all dependencies with precursor (**d b**) have identical sets of targets. As we can see in Figure 3.4 on page 20, this is not true, since (**d b e**) conflicts with (**a b b**). Not having a legal destination in the STD, and having only one extending event, we append it at the end of the current state, and re-insert the result into the resolution list (see Figure 3.6). Similarly, we append **e** at the end of (**b**) (see Figure 3.7) and another **b** at the end of (**a b**), ending up with the intermediate STD depicted in Figure 3.8. The resolution list now contains states (**b e**) and (**a b b**).

State (**b e**) has one single extending event, **c**. A link to state (**c d**) is consistent with the set of snapshot dependencies. Because state (**c d**) has no incoming links (which is no surprise, since this state was the seed of the STD), we can merge the two states, obtaining the STD shown in Figure 3.9. The link from (**c d**) to (**b e**) is preserved in the form of a reference from the new state (**b e c d**) back to itself. The only state left in the resolution list is now (**a b b**).

Next, state (**a b b**) is extended into (**a b b c**). The snapshot dependency (**b c b**) directs us towards state (**b e c d**), which is a legal destination for (**a b b c**). Because (**b e c d**) already has an incoming link (from itself), we can only add a link between our current state and the legal destination. The resulting STD is shown in Figure 3.10. The resolution list becomes empty, and therefore the algorithm terminates here.

The STD depicted in Figure 3.10 is an abstraction, a simplified model of the initial diagram from Figure 3.2 on page 19. A weak transition (strength 0.2) between

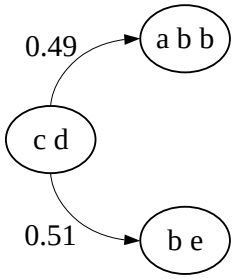

Figure 3.8: STD after appending events to existing states

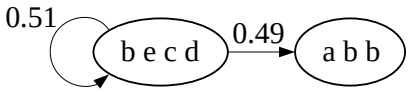

Figure 3.9: Merging two existing states of an STD

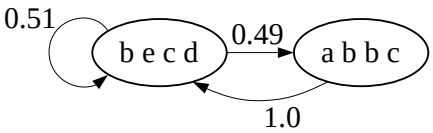

Figure 3.10: STD resulting from the analysis of the synthetic trace

states (**a b**) and (**c d**) is now gone, and some states have been merged, but essentially the two STDs are very similar. The purpose of this algorithm, to give a high-level, overview picture of what is going on in the analyzed system, has been fulfilled.

Chapter 4

EXPERIMENT DESIGN

Several factors can influence the performance of STDD. These factors are reviewed in section 4.1. The testing procedure is described in detail in section 4.2. My hypotheses and questions about STDD performance are summarized in section 4.3.

4.1 Factors Influencing STDD Performance

The quality of a model generated with STDD depends on three main groups of factors:

1. Complexity of the process being modeled. Two questions arise here:
   (a) What makes a process complex?
   (b) How does STDD behave when process complexity varies?
2. The amount and quality of available process data (i.e. the execution trace): more data and less noise should result in a better model of the process.
3. The set of parameters used when calling STDD: how to fine-tune the algorithm for different processes?

The experiments have been designed to account for variation in each of these groups of factors, in order to answer and verify the above questions and hypotheses.

Model Name	Total Events	Total States	Max Evts. per State	Total Transitions	Max Trans. per State	See Figure
m13-3-7	13	3	7	5	2	4.2
m5-4-2	5	4	2	6	2	4.3
m6-2-4	6	2	4	3	2	4.4
m5-4-4	5	4	4	7	3	4.5
m8-4-2	8	4	2	7	3	4.6
m13-10-2	13	10	2	16	6	4.7
m5-10-2	5	10	2	16	6	4.8

Figure 4.1: Synthetic models used for testing the accuracy of STDD

4.1.1 Process Complexity

In order to determine the characteristics that make a process complex, and to characterize how the performance of STDD degrades with increasing process "complexity", I have built a set of synthetic models, with a varying number of: states (2-10), total events (5-13), maximum number of events per state (2-7), total transitions (3-16), and maximum numbers of outgoing transitions per state (2-6). The characteristics of these synthetic models are summarized in Figure 4.1. The models themselves are depicted in Figures 4.7 through 4.6. In these figures, letters represent individual events. Therefore, states containing two letters are composed of sequences of two events; states containing three letters are sequences of three events, etc.

Several of these models have equal outgoing transition probabilities from most of their states. My expectation is that equal outgoing transition probabilities are a harder problem than, for instance, a pair of high-probability and low-probability transitions. In the latter case, the low-probability transition would most likely be eliminated by the snapshot DD process, making the task of STDD even simpler.

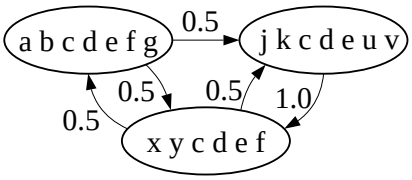

Figure 4.2: Synthetic model m13-3-7

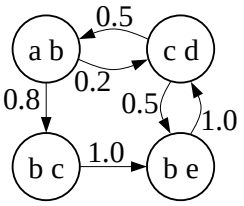

Figure 4.3: Synthetic model m5-4-2

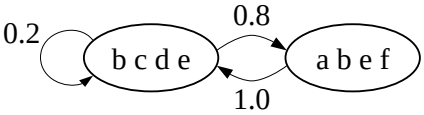

Figure 4.4: Synthetic model m6-2-4

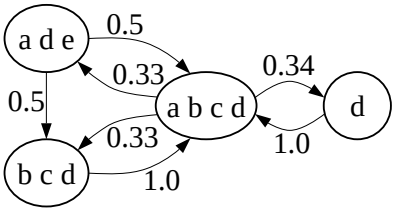

Figure 4.5: Synthetic model m5-4-4

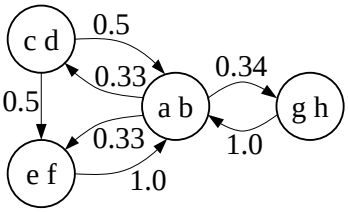

Figure 4.6: Synthetic model m8-4-2

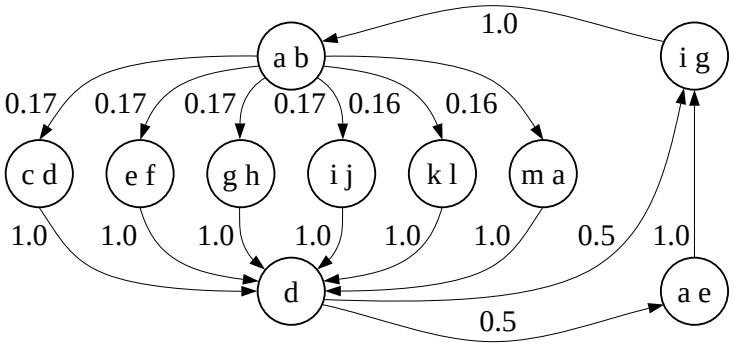

Figure 4.7: Synthetic model m13-10-2

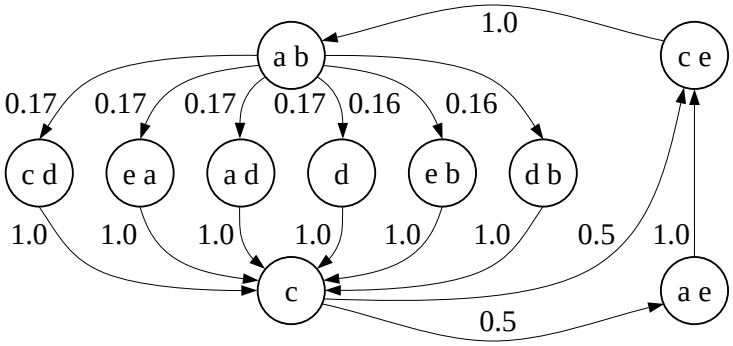

Figure 4.8: Synthetic model m5-10-2

The similarities in the topologies of some of the models have been inserted in order to test the effects of varying the number of total events and state size on the performance of STDD.

Execution traces of these synthetic models will be analyzed using STDD, and the resulting STDs will be compared against the originals to assess accuracy.

4.1.2 Characteristics of Available Process Data

Another set of factors expected to directly influence the performance of STDD comes from the way the process execution traces were obtained. Two elements need to be considered:

- Quantity: the amount of available information about the process - the length of the execution trace;
- Quality: the amount of noise introduced into the execution trace before STDD analysis.

In order to measure the influence of execution trace length and noisiness on the quality of STDD-generated models, a set of traces of various lengths (2,500-20,000 events) and noise levels (0.01-0.2) have been generated out of the synthetic models presented before (see subsection 4.1.1). The noise level is the probability of appearance of a spurious event between states of the original model while recording the execution trace (see how execution traces are recorded below).

How Execution Traces Are Generated

Execution traces are generated by executing the STD associated with a given synthetic model. An initial state is picked randomly, and subsequent states are picked according to their associated transition probabilities from the current state. Every time the generator passes through a state, all the events from that state will be appended to the execution trace. Before making the transition to the selected next state, the generator decides to insert a random event with a given noise probability. For the purpose of these experiments, I have decided to vary only the noise probability. The list of noise events will be constant for a given synthetic model, and equal to the model's set of events (i.e. any valid event of the model can be "inadvertently" added to the execution trace as noise).

4.1.3 Parameters Specific to STDD

There are several parameters intended to allow for the "fine-tuning" of STDD. The parameters and their possible values during the experiments are:

- Snapshot length - from 2 to 7;
- Significance threshold - from 0.005 to 0.1;
- Snapshot DD method - *absolute order* and *subsumption*;
- Trace compression - *on* or *off*;
- Filtering of anti-dependencies - *on* or *off*.

I am looking for the "best" combination of these parameters for each model. All combinations will be used on every synthetic model. The goal is to indicate some relationship between process complexity and the way of choosing the optimal set of parameters such that the highest accuracy can be obtained.

4.2 Testing Issues

The accuracy of an STDD-generated model was determined by comparing it against the original synthetic model it corresponds to. Accuracy is measured in terms of how many sequences that can be produced from the original synthetic model are actually captured in the model generated with STDD. An indirect comparison method was used: execution traces of length 20,000 without noise were generated out of both the original and the STDD-generated models. Subsequences of a given length were then counted from each execution trace, resulting in "subsequence maps" - lists of subsequences that occur in a trace with their associated counts.

When two subsequence maps are compared, three measures are recorded:

- *Hits*: these are the subsequences that appear in both subsequence maps;
- *False Positives*: subsequences that occur in the STDD-generated model, but not in the original. These are subsequences that were inadvertently allowed by the STDD algorithm;
- *False Negatives*: subsequences occurring in the original model, but not in the STDD-generated one. They were inadvertently left out by STDD.

These three measures only account for the presence or absence of a subsequence in an execution trace. For instance, a subsequence might appear one thousand times in the trace obtained from the original synthetic model, but only twice in the trace obtained from the STDD-generated model. This would indicate that STDD failed to accurately capture the transition probabilities between different states of the model.

In order to measure how well STDD accounts for transition probabilities, I also recorded hits, false positives and false negatives in terms of *counts*, rather than just presence or absence. If a subsequence is a false positive (or false negative), its count will be added to the *false positive* (or *false negative*) *count*. If a hit subsequence appears more often in one trace than the other, only the minimum of its two respective counts will be added to the *hit count*. The difference will be added to either the false positive or the false negative count, depending in which trace the subsequence occurred more often.

Due to the random nature of execution trace generation, one might argue that this latter test would be unfair, and false positives or negatives might appear even between "clean" traces (i.e. generated without noise) obtained from the *same* STD. This is true for smaller trace lengths, but becomes less and less important when the trace length is large. The larger the trace, the closer the subsequence counts will reflect the actual transition probabilities of the STD. For instance, comparing two clean traces of length 20,000 of model m8-4-2 (see Figure 4.6 on page 29), the rate of false positives and false negatives is less than one percent of the total number of subsequences, for test subsequence lengths from 2 to 7.

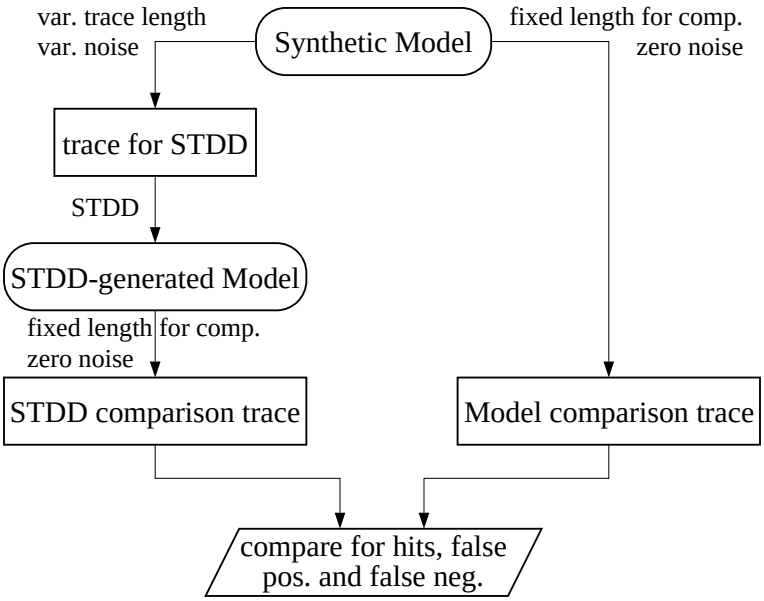

Figure 4.9: Comparison method between synthetic and STDD-generated models

There is an apparent contradiction between the requirement to have long traces for good accuracy tests and the need to test for the effects of different trace lengths on the quality of STDD-generated models. This problem has been solved by generating two execution traces from each synthetic model: a clean one of constant length 20,000 (a "comparison" trace), and a noisy one of variable length (between 2,500 and 20,000) on which to run STDD. Another clean trace of length 20,000 is generated from the resulting STDD model, and tested for hits, false positives and false negatives against the comparison trace. This process is depicted in Figure 4.9.

Along with the accuracy measures described above, data about the complexity of the STDD-generated model is also recorded. This includes the following:

- Number of states in the STD;
- Number of events in the STD;

- Maximum number of events per state;
- Number of transitions in the STD;
- Maximum number of outgoing transitions per state.

All the parameters that vary during the experiments are summarized in Figure 4.10. The pseudocode for the testing procedure is presented in Figure 4.11.

4.3 Hypotheses and Questions about STDD Performance

The following is a summary of questions and hypotheses to verify with the tests described in this chapter:

- Related to process complexity:
  - What makes a process difficult to model?
  - How does STDD handle the complexity of a process?
- Related to the available process data:
  - Performance should increase with both the quality of data (low noise) and with its quantity (trace length);
  - Trace length should be less important to performance than quality;
- Related to STDD itself (and its parameters):
  - How does snapshot length affect STDD Performance?
  - How does significance threshold affect STDD Performance?

Variable	Testing for influence of:	Set of possible values
model	process complexity	All models in Figure 4.1 on page 28
pnoise	quality of available data	$\{0.01, 0.05, 0.1, 0.15, 0.2\}$
tracelen	quantity of available data	$\{2500, 5000, 10000, 20000\}$
deplen	snapshot length	$\{2, 3, 4, 5, 6, 7\}$
pval	snapshot signif. threshold	$\{0.005, 0.01, 0.05, 0.1\}$
method	snapshot DD method used	$\{absolute\_order, subsumption\}$
compress	trace compression	$\{true, false\}$
filter	anti-dependency filtering	$\{true, false\}$

Figure 4.10: Variables used during the testing of STDD

for all possible combinations of the variables in Figure 4.10 do:

- record complexity information for *model*;
- from *model*, generate *comparison_trace* of length 20,000 with zero noise;
- from *model*, generate *trace_to_analyze* of length *tracelen* with noise *pnoise*;
- run STDD on *trace_to_analyze*, with parameters *deplen*, *pval*, *method*, *compress*, and *filter*;
  - record processor time required for the operation;
  - obtain *stdd_model*;
- record complexity information for *stdd_model*;
- from *stdd_model*, generate *stdd_comparison_trace* of length 20,000 with zero noise;
- for subsequences of length 2 to 7 do:
  - compare *comparison_trace* with *stdd_comparison_trace* and record hits, false positives and false negatives both in terms of presence/absence and in terms of their counts;

Figure 4.11: Pseudocode for the STDD testing procedure

- How much does trace compression improve on the performance of absolute order snapshot DD?
- What is the effect of anti-dependency filtering on STDD performance?
- What are the best results obtained for each model? Can we make a connection between them?
- Subsumption snapshot DD will do better than absolute order snapshot DD because of its adaptive snapshot length (i.e. because it captures significant snapshot dependencies with their appropriate length);

Finally, I also measured the CPU time required by STDD for the different models and parameter combinations.

Chapter 5

RESULTS AND INTERPRETATION

This chapter introduces the *hit rate* - a measure of STDD performance, and proceeds to answer the questions and hypotheses presented in section 4.3.

5.1 Hit Rate: A Measure of STDD Performance

For each STDD experiment, three values are recorded: hits, false positives, and false negatives (see section 4.2). It can be demonstrated that the counts of false positives and false negatives will always be equal. Given two execution traces of length $tracelen$, and a specified test subsequence length $slen$, we define the following sets:

- $SUBS1$ and $SUBS2$ are the sets of subsequences of length $slen$ obtained from the two execution traces, respectively. A particular subsequence can occur multiple times in one of these sets - as many times as it occurs in the corresponding execution trace;

- $HITS = SUBS1 \cap SUBS2$ is the set of hit subsequences, i.e. those that appear in both traces;

- $FPOS = SUBS2 - SUBS1$ is the set of false positives, i.e. subsequences that "inadvertently" appear in the second trace only;

- $FNEG = SUBS1 - SUBS2$ is the set of false negatives, i.e. subsequences that were omitted in the second trace.

Using the above sets, we define the following counts:

- $hit\_count = |HITS|$ is the count of hit subsequences (the cardinality of the $HITS$ set);

- $fpos\_count = |FPOS|$ is the count of false positives;

- $fneg\_count = |FNEG|$ is the count of false negatives;

Note that a subsequence appearing in both execution traces, but not equally often in each of them, will be counted as both hits and as false positives or negatives.

From the definition of the above sets we have the following property:

$$hit\_count + fpos\_count = hit\_count + fneg\_count = tracelen - slen + 1$$

where $tracelen + slen + 1$ is the number of total subsequences of length $slen$ that can be found in an execution trace of length $tracelen$. Therefore, given my measures, it follows that $fpos\_count = fneg\_count$.

This conclusion enables me to use hit_count as the single measure of performance for any given STDD experiment. However, for the sake of clarity, I will divide this number by $tracelen$ (in my case 20,000), to obtain the *hit rate* of the experiment. The hit rate is a measure of STDD performance on a scale from 0 (poor) to 1 (excellent).

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F	P
slen	5	1097.43	219.49	4196.46	0.0000
error	80,634	4217.35	0.05		
total	80,639	5314.78			

slen	2	3	4	5	6	7
mean hitrate	0.8427	0.7597	0.6885	0.6188	0.5567	0.5014

Figure 5.1: ANOVA: Influence of *slen* on the hit rate

A quick overview of the results shows that the hit rate appears to consistently degrade with the increase of *slen*, the subsequence length for which it is measured. This dependence was confirmed by running a one-way ANOVA test with the hit rate as the dependent variable and *slen* as the independent variable (see Figure 5.1).

The conclusion of this test is that generally, STDD has a much easier time recovering dependencies if they are short. This finding lead me to measure the performance of STDD in terms of hit rates at both extremes of the range of *slen* (2 and 7). All the following results will include data for both these values, together with an explanation of the differences or similarities.

5.2 Effects of Process Complexity

To answer the question "What makes a process difficult to model?", I ran one-way ANOVA tests with the hit rate as the dependent variable and the set of different synthetic models as the independent variable (see Figure 5.2). It can be observed that the decreasing order of performance of the different models is roughly the same at both $slen = 2$ and $slen = 7$. However, the performance degrades much more rapidly

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F	P	*slen*
process	6	44.50	7.42	605.05	0.0000	
error	13,443	164.65	0.01			2
total	13,439	209.15				
process	6	448.37	74.73	1334.47	0.0000	
error	13,443	752.24	0.06			7
total	13,439	1200.61				

process	m13-3-7	m8-4-2	m6-2-4	m13-10-2	m5-4-4	m5-4-2	m5-10-2	*slen*
mean	0.9562	0.8563	0.8519	0.8405	0.8305	0.8170	0.7467	2
hitrate	0.8038	0.6954	0.4913	0.5165	0.3459	0.4291	0.2281	7

Figure 5.2: ANOVA: Influence of the process on the hit rate

at $slen = 7$. The conclusion is that, for STDD, an "easy" process to model would be one like **m13-3-7** (see Figure 4.2 on page 29), where sequences of higher length can be picked up more easily. A "hard" process would be something like **m5-10-2** (see Figure 4.8 on page 30), where it seems very difficult to pick up long sequences.

By looking at the STDs of these two processes, we see that **m13-3-7** has the highest number of events per state (7), highest number of total events (13), and a relatively small number of states (3). In contrast, **m5-10-2** has a small number of events per state (2), a small number of total events, and one of the largest numbers of distinct states.

In conclusion, processes described by STDs with a high number of total events, many events per state, and few states should be easier to model using STDD. This also results from running one-way ANOVAs to test for the effects of the process parameters on the hit rate (see Figure 5.3). These tests, although somewhat biased, show that STDD performance degrades mainly with the decrease in the number of

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F	P	*slen*
no. of states	3	34.62	11.54	888.49	0.0000	
error	13,436	174.52	0.01			2
total	13,439	209.14				
no. of states	3	240.49	80.16	1121.82	0.0000	
error	13,436	960.12	0.07			7
total	13,439	1200.61				
no. of events	3	23.83	7.94	576.37	0.0000	
error	13,436	185.23	0.01			2
total	13,439	209.07				
no. of events	3	329.98	109.99	1697.46	0.0000	
error	13,436	870.64	0.06			7
total	13,439	1200.63				
max. events/state	2	30.51	15.25	1148.16	0.0000	
error	13,437	178.54	0.01			2
total	13,439	209.05				
max. events/state	2	210.81	105.40	1431.01	0.0000	
error	13,437	989.74	0.07			7
total	13,439	1200.55				

no. of states	2	3	4	10	*slen*
mean	0.8519	0.9562	0.8347	0.7937	2
hitrate	0.4913	0.8038	0.4902	0.3723	7

no. of events	5	6	8	13	*slen*
mean	0.7891	0.8519	0.8563	0.8984	2
hitrate	0.3344	0.4913	0.6955	0.6602	7

max. events/state	2	4	7	*slen*
mean	0.8152	0.8412	0.9562	2
hitrate	0.4673	0.4186	0.8038	7

Figure 5.3: ANOVA: Influence of the process parameters on the hit rate

total events, which makes it easy for spurious effects to be introduced during the analysis.

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F	P	*slen*
noise	4	5.78	1.44	96.51	0.0000	
trace len.	3	0.97	0.32	21.66	0.0000	
interaction	12	1.18	0.09	6.59	0.0000	2
error	13,420	201.12	0.01			
total	13,439	209.07				
noise	4	37.51	9.37	108.55	0.0000	
trace len.	3	1.06	0.35	4.10	0.0064	
interaction	12	2.52	0.21	2.43	0.0072	7
error	13,420	1159.47	0.08			
total	13,439	1200.58				

mean hitrate ($slen = 2$)		noise 0.01	0.05	0.1	0.15	0.2
trace len.	2500	0.8773	0.8497	0.8311	0.8060	0.7765
	5000	0.8814	0.8588	0.8435	0.8308	0.8171
	10000	0.8716	0.8549	0.8492	0.8337	0.8283
	20000	0.8624	0.8620	0.8465	0.8389	0.8357

mean hitrate ($slen = 7$)		noise 0.01	0.05	0.1	0.15	0.2
trace len.	2500	0.6000	0.5269	0.4771	0.4394	0.3999
	5000	0.6101	0.5418	0.4993	0.4729	0.4403
	10000	0.5893	0.5177	0.4974	0.4693	0.4535
	20000	0.5522	0.5309	0.4870	0.4662	0.4584

Figure 5.4: ANOVA: Influence of the quantity and quality of available data

5.3 Effects of Data Quantity and Quality

In order to test for the effects of the quantity and quality of available data on STDD performance, I ran a two-way ANOVA test with the noise level and trace length as the independent variables, and the hit rate as the dependent (see Figure 5.4). It can be observed that noise has a much larger impact than trace length, and that as noise increases, the usefulness of large quantities of data is less and less important.

Also, the noise has a much larger impact on long sequences ($slen = 7$) than on short ones ($slen = 2$). This happens because long sequences are much more likely to be disrupted by noise than short ones.

From the table of mean hit rates, we see that performance constantly degrades with an increase in noise, while the trace length seems to have little effect.

5.4 Effects of STDD Parameters

The hit rate is affected by various STDD parameters: snapshot length and significance threshold, snapshot DD algorithm, and trace compression and anti-dependency filtering. A hypothesis about what constitutes a "good" set of parameters and considerations about the time complexity of STDD are also presented.

5.4.1 Effects of Snapshot Length

By running two-way ANOVA with the process and snapshot length as independent variables and the hit rate as the dependent (see Figure 5.5), we observe that there is a strong interaction effect between the two independent variables. This means that the optimal snapshot length depends on which process it is applied to.

Another observation is that, for most processes, long sequences are best detected when the snapshot length is large (e.g., $slen = 7$), and short sequences are best detected when the snapshots are short (e.g., $slen = 2$). Exceptions to this are **m8-4-2** (see Figure 4.6 on page 29), which seems to consistently do better for short snapshots (length 2 and 3), and **m5-4-2** (see Figure 4.3 on page 29), which consistently does better at snapshot length 3. These two processes have two common properties: two

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F	P	*slen*
snapshot len.	5	13.53	2.70	296.14	0.0000	
process	6	44.49	7.41	811.29	0.0000	
interaction	30	28.56	0.95	104.16	0.0000	2
error	13,398	122.47	0.01			
total	13,439	209.07				
snapshot len.	5	83.82	16.76	431.79	0.0000	
process	6	448.37	74.72	1924.58	0.0000	
interaction	30	148.15	4.93	127.19	0.0000	7
error	13,398	520.22	0.03			
total	13,439	1200.58				

mean hitrate ($slen = 2$)		process						
		m13-3-7	m8-4-2	m6-2-4	m13-10-2	m5-4-4	m5-4-2	m5-10-2
snapshot len.	2	0.9723	0.9496	0.9619	0.8960	0.9086	0.8894	0.7011
	3	0.9726	0.9454	0.8262	0.8391	0.8334	0.9000	0.8202
	4	0.9675	0.8955	0.7444	0.8361	0.8036	0.8058	0.7118
	5	0.9227	0.8841	0.8091	0.8486	0.8083	0.7729	0.6981
	6	0.9530	0.7398	0.8340	0.8277	0.8365	0.7656	0.7850
	7	0.9491	0.7237	0.9360	0.7960	0.7929	0.7686	0.7642

mean hitrate ($slen = 7$)		process						
		m13-3-7	m8-4-2	m6-2-4	m13-10-2	m5-4-4	m5-4-2	m5-10-2
snapshot len.	2	0.6982	0.8103	0.1772	0.2156	0.2335	0.2539	0.0295
	3	0.7730	0.8231	0.5460	0.5976	0.3034	0.6279	0.3107
	4	0.7930	0.7463	0.3132	0.6243	0.2876	0.4140	0.1544
	5	0.7911	0.7306	0.5171	0.6170	0.3338	0.3443	0.1320
	6	0.8908	0.5406	0.5469	0.5472	0.4269	0.4731	0.3837
	7	0.8766	0.5220	0.8475	0.4975	0.4902	0.4620	0.3584

Figure 5.5: ANOVA: Influence of the snapshot length on STDD performance

events in each state, and high connectivity. Using short snapshots will best capture all the transitions, and thus will help STDD in generating accurate models of the processes.

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F	P	*slen*
threshold	3	0.005	0.002	0.125	0.9449	
error	13,436	209.074	0.015			2
total	13,439	209.080				
threshold	3	0.042	0.140	0.156	0.9254	
error	13,436	1200.564	0.089			7
total	13,439	1200.606				

Figure 5.6: ANOVA: Influence of snapshot DD significance threshold

5.4.2 Effects of Significance Threshold

Somewhat to my surprise, the significance threshold when running snapshot DD does not significantly impact the hit rate of STDD. The one-way ANOVA test with the significance threshold as an independent variable and the hit rate as the dependent is presented in Figure 5.6.

Although unexpected, this result can be easily explained. The "fine-tuning" effect of the significance threshold is overshadowed by the more "brute-force" method of filtering anti-dependencies (see subsection 3.1.2), combined with the fact that STDD starts out with a seed consisting of the most frequently occurring precursor among the snapshots (see step 2 of the algorithm depicted in Figure 3.1 on page 16).

5.4.3 Why Not Subsumption DD?

Contrary to my expectation, the subsumption version of snapshot DD did not perform well in combination with STDD. Quite often, the termination criterion of STDD is never met, causing it to keep duplicating existing states as new ones, in an infinite cycle. The problem occurs when STDD attempts to link to an existing state

Subsumption					Absolute Order				
Dependency	Contingency Table				Dependency	Contingency Table			
(a b)	585	0	2	1921	(f a b)	290	0	297	1920
					(h a b)	199	0	388	1920
					(d a b)	96	0	491	1920
(e f)	290	0	1	2217	(b e f)	209	0	82	2216
					(d e f)	80	0	211	2216
(e f a)	289	1	296	1921	(e f a)	289	1	296	1921
(a b e)	209	375	81	1842	(a b e)	209	375	81	1842

Figure 5.7: Comparison of snapshot lists: subsumption vs. absolute order

of the STD being built, and fails (step 3.b of the algorithm in Figure 3.1 on page 16).

A partial comparison between the list of snapshots returned by subsumption DD versus the one returned by absolute order DD is presented in Figure 5.7. As described in subsection 2.1.2, subsumption DD finds the best predictors for a given target. It builds the snapshots backwards, starting with the target, and prepending events to the precursor until the significance of the snapshot starts to decrease. This is the very quality that renders it unsuitable for STDD, as illustrated in the example presented in Figure 5.8. In this example, STDD fails to make a connection to an existing state of the STD (represented by the dotted line in Figure 5.8), because it can't confirm it with the snapshot (**f a b**), which does not exist in the list returned by subsumption

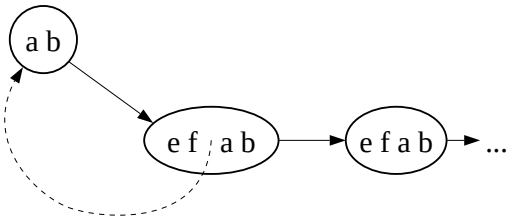

Figure 5.8: An example of subsumption DD failure

T	P	Standard Error	Degrees of Freedom	mean hitrate ON	mean hitrate OFF	*slen*
0.8501	0.3953	0.0022	13,438	0.8419	0.8437	2
3.7803	0.0002	0.0052	13,438	0.5112	0.4917	7

Figure 5.9: T-test: Influence of trace compression on the hit rate

DD (see Figure 5.7). More and more (**e f a b**) states get created and added to the STD, and the algorithm never terminates.

In all experiments presented throughout this thesis, I have only used absolute order snapshot DD, which does not create this type of problem.

5.4.4 Effects of Trace Compression

The influence of trace compression (see subsection 3.1.1) has been tested by partitioning the data on whether it had been used or not, and by running a two-sample, two-tailed T-test on the two resulting partitions. The results are presented in Figure 5.9.

Trace compression has no significant influence on the modeling of short sequences. However, it seems to improve the performance of STDD on longer sequences. This comes as no surprise, since trace compression has been designed to treat longer sequences of events that almost always occur together as a single event, thus preventing their disruption by spurious events.

5.4.5 Effects of Anti-Dependency Filtering

The influence of anti-dependency filtering (see subsection 3.1.2) has been tested by partitioning the data on whether it had been used or not, and by running a two-

T	P	Standard Error	Degrees of Freedom	mean hitrate ON	mean hitrate OFF	*slen*
4.0128	0.0022	0.0022	13,438	0.8471	0.8385	2
12.5182	0.0000	0.0051	13,438	0.5336	0.4694	7

Figure 5.10: T-test: Influence of anti-dependency filtering on the hit rate

sample, two-tailed T-test on the two resulting partitions. The results are presented in Figure 5.10.

Anti-dependency filtering significantly improves the performance of STDD on both short and long sequences. The effect is stronger when the sequences are long. Again, this happens because long sequences are more prone to be disrupted by noise.

5.4.6 Best Results

The best results obtained for each model are presented in Figure 5.11. They mostly share anti-dependency filtering, and longer snapshot length. Overall, these characteristics are in agreement with the conclusions of the tests performed in the previous subsections.

5.4.7 Time Complexity

For each experiment, the CPU time has been recorded for the entire STDD procedure (snapshot DD phase, and model construction). The experiments have been run on Sun Ultra-Sparc workstations.

Process parameters have a significant effect on the run time of STDD. The slowest process to model was **m13-10-2**, with an average run time of 259,684 milliseconds. The fastest was **m6-2-4**, with an average run time of 7,071 milliseconds. The con-

data		process	snap. len.	compr. trace	filter	time (ms)	hit rate $slen = 2$	hit rate $slen = 7$
trace length 20,000	noise level 0.01	m13-3-7	6	no	yes	5230	0.999050	0.998099
		m6-2-4	6	yes	no	40910	0.998849	0.997599
		m8-4-2	3	no	no	5630	0.993249	0.977742
		m5-4-4	5	yes	no	51150	0.982747	0.895758
		m5-4-2	3	no	yes	3870	0.977847	0.877457
		m13-10-2	5	no	yes	14260	0.990549	0.974040
		m5-10-2	6	no	yes	13870	0.899139	0.670184
trace length 2,500	noise level 0.2	m13-3-7	7	no	yes	3400	0.989901	0.969095
		m6-2-4	3	no	yes	400	0.931743	0.846046
		m8-4-2	2	no	yes	690	0.963146	0.939878
		m5-4-4	5	no	yes	1640	0.857650	0.628007
		m5-4-2	3	no	yes	780	0.964446	0.815385
		m13-10-2	4	yes	yes	67890	0.884182	0.668317
		m5-10-2	3	no	yes	1220	0.827982	0.458810

Figure 5.11: Best results obtained with STDD

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F	P
compression	1	2.4478e13	2.4478e13	235.6587	0.0000
trace len.	3	6.0144e13	2.0048e13	193.0047	0.0000
interaction	3	3.6449e13	1.2150e13	116.9687	0.0000
error	13,432	1.3952e15	1.0387e11		
total	13,439	1.5163e15			

mean runtime	(ms)	trace length 2500	5000	10000	20000
compress	no	4479	10058	28060	44522
	yes	9520	26050	85954	307012

Figure 5.12: ANOVA: Influence of trace length and compression on the run time

clusion is that many states and transitions need more iterations of STDD and thus take a longer time to execute.

The trace length and trace compression have both strong individual effects, and a strong interaction effect on the run time (see Figure 5.12). The run time grows exponentially with the increase in trace length when trace compression is turned on.

Chapter 6

CONCLUSIONS AND FUTURE WORK

In this thesis I have presented STDD, an algorithm that analyzes process execution traces by integrating lists of snapshot dependencies. Performance tests have been run on seven synthetic models, while varying the quality and quantity of available data, and the parameters of STDD.

Two snapshot DD algorithms, absolute order and subsumption, have been tested with STDD. Contrary to the initial hypothesis, subsumption did not perform well in conjunction with STDD.

Longer snapshot dependencies have been found to result in increased accuracy. Trace compression and anti-dependency filtering have been found to be much better mechanisms for controlling the complexity of the resulting models than the snapshot dependency significance threshold. Turning on anti-dependency filtering significantly helps eliminate spurious effects, and thus results in increased accuracy for longer sequences.

It is my hypothesis that, for further improvements of STDD, a new snapshot DD algorithm should be implemented. It should be similar to subsumption, but it should develop the snapshot dependencies by appending, rather than prepending events to

them, until the highest significance is reached. Also, finding and handling exceptions to the cases from step 3 of the algorithm (see Figure 3.1 on page 16) should also result in a higher accuracy of STDD. For instance, before merging with an existing state of the STD that has no current incoming links, one should make sure that no "potential" incoming links are possible.

This algorithm has already been used successfully on real data in two instances: the debugging of an AI agent steering a race car on a simulated track [22], and in an analysis on how people behave while trying to understand large-scale software [17]. With relatively few adjustable parameters, the algorithm can achieve very high hit rates, even when processing limited data obtained under noisy conditions.

REFERENCES

[1] Marc Abrams, Alan Batongbacal, Randy Ribler, 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, Ramakrishnan Srikant. "The Quest Data Mining System", *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, Portland, OR, August 1996.

[3] Rakesh Agrawal, Ramakrishnan Srikant. "Mining Sequential Patterns", *Proceedings of the International Conference on Data Engineering (ICDE)*, Taipei, Taiwan, March 1995.

[4] Scott D. Anderson, David L. Westbrook, Matthew Schmill, Adam Carlson, David M. Hart, Paul R. Cohen. "Common Lisp Analytical Statistics Package: User Manual", Experimental Knowledge Systems Lab, Computer Science Department, University of Massachusetts, January 1995.

[5] Horacio T. Cadiz. "The Development of a CHAID-based Model for CHITRA93", Computer Science Department, Virginia Polytechnic Institute, February 1994.

[6] Usama Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth. "From Data Mining to Knowledge Discovery in Databases", *AI Magazine* 17(3), pp. 37-54, Fall 1996.

[7] Adele E. Howe. "Detecting Imperfect Patterns in Event Streams Using Local Search", *Learning From Data: Artificial Intelligence and Statistics*, pp. 155-164, V.D. Fisher, H. Lenz (eds.), Springer Verlag, 1996.

[8] Adele E. Howe, Paul R. Cohen. "Detecting and Explaining Dependencies in Execution Traces", *Lecture Notes in Statistics*, ch. 8. pp. 71-78, Springer-Verlag, NY, 1994.

[9] Adele E. Howe, Paul R. Cohen. "Understanding Planner Behavior", *Artificial Intelligence*, vol. 76(1-2), pp. 125-166, 1995.

[10] Adele E. Howe, Aaron D. Fuegi. "Methods for Finding Influences on Program Failure", *Proceedings of 6th International Conference on Tools with Artificial Intelligence*, November 1994.

[11] Adele E. Howe, Larry D. Pyeatt. "Constructing Transition Models of AI Planner Behavior", *Proceedings of the 11th Knowledge-Based Software Engineering Conference*, September 1996.

[12] Adele E. Howe, Gabriel L. Somlo. "Modeling Intelligent System Execution as State Transition Diagrams to Support Debugging", to appear in *Proceedings of the Second International Workshop on Automated Debugging*, May 1997.

[13] Adele E. Howe, Gabriel L. Somlo. "Modeling Discrete Event Sequences as State Transition Diagrams", to appear in *Proceedings of the Second Conference on Intelligent Data Analysis*, 1997.

[14] G. V. Kass. "An Exploratory Technique for Investigating Large Quantities of Categorical Data", *Applied Statistics*, vol. 29, no. 2, pp. 119-127, 1980.

[15] Eleftherios Koutsofios, Stephen C. North. "Drawing Graphs with Dot", AT&T Bell Laboratories, Murray Hill, NJ, October 1993.

[16] John C. Mallery. "A Common LISP Hypermedia Server", *Proceedings of The First International Conference on The World-Wide Web*, Geneva:CERN, May 25, 1994.

[17] Anneliese von Mayrhauser, A. Marie Vans, Adele E. Howe. "Program Understanding Behavior During Enhancement of Large-Scale Software", to appear in *Journal of Sofware Maintenance*, 1997.

[18] Tim Oates, Paul R. Cohen. "Searching for Planning Operators with Context-Dependent and Probabilistic Effects", *Proceedings of the 13th National Conference on Artificial Intelligence*, pp. 863-868, AAAI Press, 1996.

[19] Tim Oates, Matthew D. Schmill, Dawn E. Gregory, Paul R. Cohen. "Detecting Complex Dependencies in Categorical Data", *Learning From Data: Artificial Intelligence and Statistics*, pp. 353-362, V.D. Fisher, H. Lenz (eds.), Springer Verlag, 1996.

[20] R. Lyman Ott. "An Introduction to Statistical Methods and Data Analysis", 4th edition, Wadsworth Publishing Company, 1992.

[21] Judea Pearl, Thomas S. Verma. "A Theory of Inferred Causation", *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, pp. 441-452, Morgan Kaufmann, 1991.

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

[23] Robert R. Sokal, F. James Rohlf. "Biometry - The Principles and Practice of Statistics in Biological Research", 2nd edition, W.H. Freeman and Company, New York, 1981.

[24] Raghavan Srinivasan, Adele E. Howe. "Comparison of Methods for Improving Search Efficiency in a Partial-Order Planner", *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, Canada, August 1995.