Automating Process Discovery through Event-Data Analysis

Jonathan E. Cook and Alexander L. Wolf

Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

{jcook,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-741-94 September 1994

A version of this report to appear in
The Proceedings of the 17th International Conference
on the Software Engineering (ICSE17), April 1995

ABSTRACT

Many software process methods and tools presuppose the existence of a formal model of a process. Unfortunately, developing a formal model for an on-going, complex process can be difficult, costly, and error prone. This presents a practical barrier to the adoption of process technologies. The barrier would be lowered by automating the creation of formal models. We are currently exploring techniques that can use basic event data captured from an on-going process to generate a formal model of process behavior. We term this kind of data analysis process discovery. *This paper describes and illustrates three methods with which we have been experimenting: algorithmic grammar inference, Markov models, and neural networks.*

This work was supported in part by the National Science Foundation under grant CCR-93-02739 and by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 Introduction

The issues of managing and improving the process of developing and maintaining software have come to the forefront of software engineering research. In response, new methods and tools for supporting various aspects of the software process have been devised. Many of the technologies, including process automation [4, 12, 25, 27], process analysis [15, 16, 19, 26], process evolution [3, 18], and process validation [9], assume the existence of some sort of formal model of a process in order for those technologies to be applied.

The need to develop a formal model as a prerequisite to using a new technology is a daunting prospect to the managers of large, on-going projects. The irony is that the more a project exhibits problems, the more it can benefit from the process technologies, but also the *less* its managers may be willing or able to invest resources in new methods and tools. Therefore, if we intend to help on-going projects by promoting the use of technologies based on formal models, we must seriously consider how to lower the entry barriers to those technologies.

In that vein, we are exploring methods for automatically deriving a formal model of a process from basic data collected on the process. We term this form of data analysis *process discovery*, because inherent in every project is a process (whether known or unknown, whether good or bad, and whether stable or erratic) and for every process there is some model that can be devised to describe it. We are assuming, of course, that a project is already willing and able to collect data on the current process as part of a process improvement strategy [17]. The challenge in process discovery is to use those data to describe the process in a form suitable for formal-model-based process technologies.

In general, this is a very difficult challenge to meet. To scope the problem somewhat, we have concentrated our efforts on models of the behavioral aspects of a process, rather than on models of the relationships between artifacts produced by the project, or on models of the roles and responsibilities of the agents in the process. Any complete modeling activity would have to address those other aspects of the process as well. To further scope the problem, we have initially restricted ourselves to the discovery of finite state machine models of behavioral patterns.

In this paper we describe three methods for process discovery. They range in approach from the purely algorithmic to the purely statistical. The methods have been implemented as a set of tools operating on process data sets. While the methods are automated, they all require guidance from someone knowledgeable in software processes and at least somewhat familiar with the particular process under study. This guidance comes in the form of tuning parameters built into the tools. The results produced by the tools are initial models of a process that can be refined by a process engineer. Indeed, the initial models may lead to changes in data collection to uncover greater detail about particular aspects of the process. Thus, the discovery methods complement a process engineer's knowledge, providing empirical analysis in support of experience and intuition.

The next section provides background on our choice of methods. The methods themselves are described and illustrated using a simple example in Section 3. Use of the methods on a more complex and significant example is presented in Section 4, along with an evaluation of their relative strengths and weaknesses. Finally, we conclude in Section 5 with a summary of our results and a

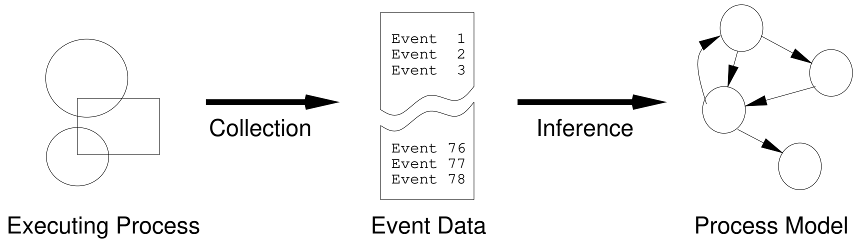

Figure 1: A Framework for Process Discovery.

discussion of related work.

2 Background

The framework in which the process discovery methods presented here were developed is based on a view of processes as a sequence of actions performed by agents, whether human or automaton, possibly working concurrently. This reflects our current focus on behavioral aspects of processes. Following Wolf and Rosenblum [28], we use an event-based model of process actions, where an *event* is used to characterize the dynamic behavior of a process in terms of identifiable, instantaneous actions such as invoking a development tool or deciding upon the next activity to be performed. For purposes of maintaining information about an action, events are typed and can have attributes. One attribute, for example, is the time the event occurred. Another attribute is an indication of the agent associated with the event. Because events are instantaneous, an activity spanning some period of time is represented by the interval between two or more events. For example, a meeting could be represented by a "begin-meeting" event and "end-meeting" event pair. Similarly, a module compilation submitted to a queue could be represented by the three events "enter queue", "begin compilation", and "end compilation". The overlapping activities of a process, then, are represented by a sequence of events, which we refer to as an *event stream*.

Using event data to characterize behavior is widely accepted in other areas of software engineering, such as program visualization [22], concurrent-system analysis [2], and distributed debugging [5, 10]. Our approach in the work described here is to analyze a trace of a process execution—in the form of an event stream—and infer a formal model that can account for the behavior of the process (see Figure 1). We do not envision always being able to infer fully complete and correct process models, but rather our goal is to offer process engineers an approximation of the behavioral patterns that exist in a process. These approximations should be of sufficiently good quality that they can be relied upon as a basis for creating a more complete process model, evolving an existing model, or even adjusting some aspect of the actual process execution.

As mentioned in the introduction, we assume that event data are being collected on the executing process. Methods already exist and are being used to collect such data [7, 28]. Once collected, the data can be viewed as a window onto the execution. In general, this window will not show the whole execution, since there will be some activities for which no event data are collected at all. For example, a chance meeting in a hallway might preclude the occurrence of a scheduled meeting for which events would have been recorded; or a decision might be made to focus the data collection effort on one particular aspect of the process, such as simple duration of work periods [24]. Hence, just as for any other data analysis technique, the results obtained by discovery methods strongly depend upon the content and quality of the data that are collected.

What we seek from the data are recurring patterns of behavior. For our purposes, finite state machines (FSMs) seem to provide a good starting point for expressing those patterns. While FSMs may be somewhat deficient for *prescribing* software processes, they are quite convenient and sufficiently powerful for *describing* historical patterns of actual behavior. Why not choose a more expressive representation than finite state machines, such as push-down automata? One reason is that the more powerful the representation, the more complex the inference problem, and it is not clear that we need that power. In particular, we are initially aiming to infer only the structure of the behavior and not the values controlling that structure. Thus, while our methods could not discover a pattern such as $\mathtt{A}^n\mathtt{B}^n$ (the canonical example that demonstrates the superior power of push-down automata), they would certainly discover the sequenced looping structure of an `A` loop followed by a `B` loop.

To develop our initial set of methods, we have cast the process discovery problem in terms of another, previously investigated FSM discovery problem. That problem is the discovery of a grammar for a regular language given example sentences in that language [1]. If one interprets events as tokens and event streams as sentences, then the problems converge on somewhat similar solutions. There are, however, several important differences, as we discuss in the next section.

3 Inference Methods for Process Discovery

Existing methods for grammar discovery range from algorithmic techniques,[1] wherein a well-understood algorithm is used to compute a grammar from one or more sample sentences, to statistical techniques, wherein probabilities are calculated from observations of sample sentences. Some of the techniques use only samples that are *positive* examples, which are sentences known to be legal in the language. Others also use *negative* examples, which are sentences known to be illegal in the language.

In this section we describe three inference methods that can be used in process discovery. The methods represent three points on the spectrum from algorithmic to statistical techniques. Since we are dealing with data collected from process executions, we have only positive samples with which to work. The methods can be characterized by the manner in which they examine those samples. The KTAIL method is a purely algorithmic approach that looks at the *future* behavior to compute a possible current state. The RNET method is a purely statistical (neural network) approach that looks at the *past* behavior to characterize a state. Finally, the MARKOV method is a

[1] A good survey of algorithmic methods is provided by Angluin and Smith [1].

```
ABCABCBACBACABCBACBACABCBACABCABCBACBA
```

Figure 2: Simple Event Stream Example.

hybrid statistical and algorithmic approach that looks at the *neighboring* past and future behavior to define a state.

To experiment with the methods, we have implemented KTAIL and MARKOV, and for RNET we have extended an implementation by Das [11]. All of the graphical representations presented in this paper have been automatically generated by the *dot* directed-graph drawing tool [21] from the output of the discovery tools.

To explain and illustrate the three methods in this section, we use the simple event stream of Figure 2. The stream is a sentence in a three-token language and is used as sample input for each of the methods. All three methods typically can work on multiple samples to refine their results, but in this simple example we just give them the one stream shown. Of course, there are many different FSMs that could generate this sentence. What is important is that the methods produce an FSM model that reflects the structure inherent in the sample—that is, the `A-B-C` and `B-A-C` loops. In addition, we would like the methods to produce minimal models that sensibly share states among paths.

In Section 4, we demonstrate the methods on a more complex example, exposing the relative performance and possible strengths and weaknesses of each method.

3.1 RNET

The first method we describe comes from the neural network community and is a purely statistical approach to inference. This recently developed method is due to Das and Mozer [11]. We refer to it here as the RNET method.

In a standard feed-forward neural network, neurons are split into layers, with all the outputs of the neurons in one layer feeding forward into all the neurons of the next layer (see Figure 3a). Typically, there is a layer of input neurons, at least one layer of internal or hidden neurons, and a layer of output neurons. For a given input to the input neurons, activation flows forward until the output neurons are activated in some pattern. The network is trained by propagating the difference between actual and desired outputs backwards through the network. We refer to this difference as the *learning error*.

To this basic architecture, Das and Mozer have added a recurrent connection, which feeds the results of the hidden layer directly back to the input layer (see Figure 3b). This lets the network model an FSM by providing the "current" state—in the form of the activation pattern of the hidden layer—as an input to the network. Now the output activity (i.e., the "next" state) is determined, not just by the input, but also by the current hidden neuron activity. This recurrent network is the inference mechanism of RNET.

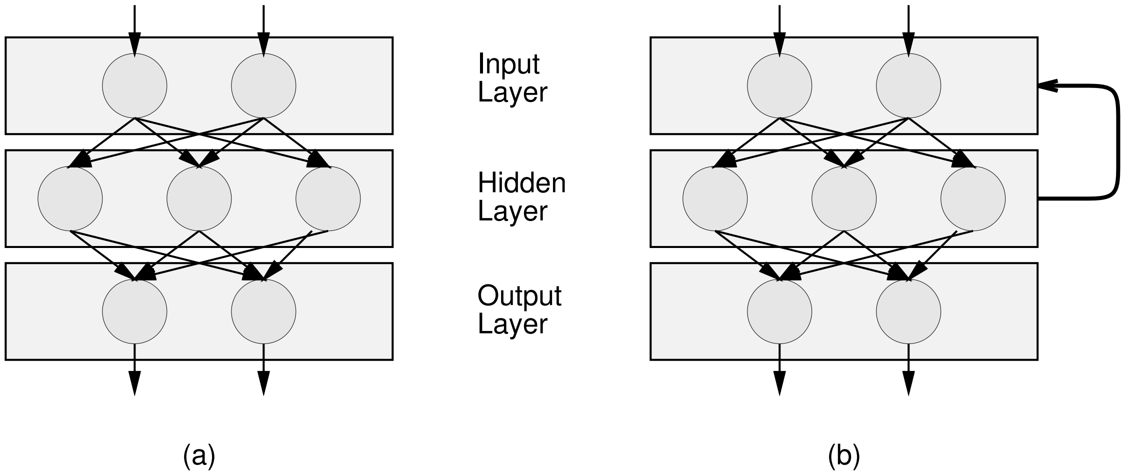

Figure 3: Standard (a) and Recurrent (b) Neural Network Architectures.

Training takes the form of presenting a *window* of a specified length to the network, and having it attempt to predict the next token. Learning error is only back-propagated through the network after the whole window is presented. By sliding the window forward over the sample stream one token at a time, each position in the stream is used in training. RNet therefore takes a historical view of the sample stream, since the window focuses attention on events that precede the current event.

Once the network is trained, the FSM representation is extracted from the network. This is done by presenting the same or different strings to the network, and then observing the activity of its hidden neurons. Activation patterns that are closely related are clustered into the same state. Transitions are recorded by noting the current activation pattern, the input token, and the next activation pattern. This information, collected over all input patterns, then represents the FSM that the network has inferred.

Figure 4 shows the inferred FSM that RNet produces from the example stream of Figure 2. The method successfully produces a deterministic FSM that incorporates both the `A-B-C` and `B-A-C` loops. But it also models behavior that is not present in the stream, such as an `A-A-C` loop and a `B` loop. This shows the inexactness of the neural network approach; even with a known perfect sample input, one cannot direct it to produce a machine just for that stream.

An advantage of the RNet method is that, since it is statistical in nature, it has the potential of being robust with respect to input stream noise (e.g., collection errors or abnormal process events).

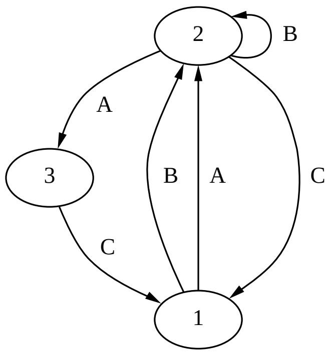

Figure 4: FSM Inferred by the RNet Method for the Example Event Stream of Figure 2.

3.2 KTail

The next method is purely algorithmic and based on work by Biermann and Feldman [6]. Their original presentation was formulated in terms of sample strings and output values for the FSM at the end of the strings. Our formulation of this algorithm does not make use of the output values, and is thus presented as just operating on the sample strings themselves. In addition, we have enhanced their algorithm to reduce the number of states in the resulting FSM.

The notion that is central to KTail is that a state is defined by what future behaviors can occur from it. Thus, for a given history (i.e., token string prefix), the current state reached by that history is determined by the possible futures that can occur from it. Two or more strings can share a common prefix and then diverge from each other, thus giving a single history multiple futures. The "future" is defined as the next k tokens, where k is a parameter to the algorithm. If two different histories have the same future behaviors, then they reside in the same equivalence class; equivalence classes represent states in the FSM.

Formally, KTail is defined as follows. Let S be the set of sample strings and let A be the alphabet of tokens that make up the strings in S. Let P be the set of all prefixes in S, including the full strings in S. Then $p \in P$ is a valid prefix for some subset of the strings in S. Let $p \cdot t$ be the prefix p appended with the token string t. We call t a *tail*. Finally, let T_k be the set of all strings composed from A of length k or less. An equivalence class E is a set of prefixes such that

$$\forall (p, p') \in E,\ \forall t \in T_k,\ \ p \cdot t \in P \longleftrightarrow p' \cdot t \in P$$

This means that all prefixes in E have the same set of tails of length k or less.[2] Thus, all prefixes in P can be assigned to some equivalence class. It is these equivalence classes that are mapped to states in the resulting FSM.

[2] Tails of length less than k, down to zero, occur at the ends of strings.

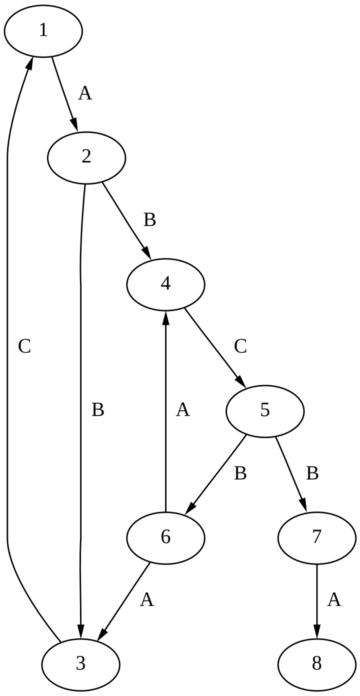

Figure 5: FSM Inferred by the Basic Biermann-Feldman Algorithm for the Example Event Stream of Figure 2.

Transitions among states are defined as follows. For a given state (i.e., equivalence class) E_i and a token $a \in A$, the destination states of the transitions are the set D of equivalence classes

$$D = \bigcup \mathcal{E}[p \cdot a], \ \ \forall p \in E_i$$

where $\mathcal{E}[p \cdot a]$ is the equivalence class of the prefix $p \cdot a$. Intuitively, this says that to define the transitions from E_i, take all $p \in E_i$, append a to them, and calculate the equivalence classes of these new prefixes, which are the destination states of token a from state E_i. If $|D| = 0$, then this transition does not exist; if $|D| = 1$, then this transition is deterministic; and if $|D| > 1$, then this transition is nondeterministic. The transitions, if any, are annotated with the token a in the final FSM.

This is where the algorithm, as Biermann and Feldman define it, stops. While it produces an FSM that is complete and correct, the algorithm has certain tendencies to produce an overly complicated FSM. Figure 5 shows the FSM that is produced by the algorithm when given the sample data of Figure 2. The complexity is that a loop in the data will be *unrolled* to a length of k. The unrolling arises because the algorithm sees a change in the tail of the loop body at the exit point of the loop. This change causes the last iteration of the loop to be placed in a separate

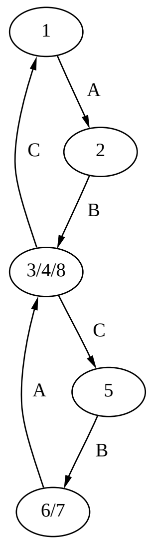

Figure 6: FSM Inferred by the KTAIL Method for the Example Event Stream of Figure 2.

equivalence class. Consider the `A-B-C` loop. In the figure, this loop is represented by the path $< 1, 2, 3, 1 >$ in the FSM. But the last iteration through the loop has a separate representation as the path $< 1, 2, 4, 5 >$. A similar effect occurs with the `B-A-C` loop.

We can improve the basic algorithm by automatically merging states in the following manner. If a state S_1 has transitions to states $S_2 \ldots S_n$ for a token t, and if the set of output transition tokens for the states $S_2 \ldots S_n$ are equivalent, then we merge states $S_2 \ldots S_n$. Intuitively, this procedure assumes that if two (or more) transition paths from a state are the same for a length of more than one transition, then the internal states of those paths should be assumed to be the same and thus merged. In Figure 5, states 6 and 7 can be merged, states 3 and 4 can be merged, and 8 can be merged with the merged states 3/4, once 6 and 7 are merged.

This improvement, implemented in KTAIL, results in a much cleaner FSM than that which results from the basic Biermann-Feldman algorithm. The FSM produced by KTAIL from the example stream of Figure 2 is shown in Figure 6. Clearly, KTAIL does a good job of discovering the underlying loops in the behavior. The `A-B-C` loop is completely embodied in the transition path $< 1, 2, 3/4/8, 1 >$, while the `B-A-C` loop is completely embodied in the path $< 5, 6/7, 3/4/8, 5 >$. Note that there is a point of nondeterminism in state 3 upon the occurrence of a `C`. This is because the algorithm sees two different behaviors (i.e., tails) from the `C` event in the sample data.

KTAIL is controlled in its accuracy by the value of k; the greater that value, the greater the length of the prefix tails that are considered, and thus the more differentiation can occur in states and transitions. Note that as k increases, this algorithm monotonically adds states, and complexity, to the resulting FSM. If k is as long as the longest sample string, then the resulting FSM is guaranteed to be deterministic, but deterministic FSMs can result from much smaller values of k, depending on the structure of the sample strings. It is not always the case that the most interesting and informative FSM will result from k being large enough to generate a deterministic FSM. To the contrary, points of nondeterminism in the resulting FSM might signify important decision points in the process, where the decision will determine the path of execution.

An advantage of KTAIL is that it is parameterized by the simple value k, so the complexity of the resulting FSM can be controlled in a straightforward manner. A disadvantage is that it cannot ignore features in the input stream, and thus is not robust in the presence of input stream noise.

3.3 MARKOV

The third method is one that we invented, and is a hybrid of algorithmic and statistical techniques. This method uses Markov models to find the most probable event sequence productions, and algorithmically converts those probabilities into states and state transitions. Although our method is new, there is previous work that has used similar methods. For example, Miclet and Quinqueton [23] use transition probabilities to create FSM recognizers of protein sequences, and then use the Markov models to predict the center point of new protein sequences.

A discrete, first-order Markov model of a system is a restricted, probabilistic process[3] representation that assumes that:

- there are a finite number of states defined for the process;
- at any point in time, the probability of the process being in some state is only dependent on the previous state that the process was in (the Markov property);
- the state transition probabilities do not change over time; and
- the initial state of the process is defined probabilistically.

In general, the definition of an n^{th}-order Markov model is that the state transition probabilities depend on the last n states that the process was in.

The basic idea behind the MARKOV method is to use the probabilities of event sequences. In particular, it builds event-sequence probability tables by tallying occurrences of like subsequences. The tables are then used to produce an FSM that accepts only the sequences whose probabilities are non-zero or, more generally, that exceed a probability threshold that is a parameter to the method. MARKOV thus proceeds in four steps.

1. The event-sequence probability tables are constructed by traversing the event stream.[4]

[3] The use here of the word "process" does not refer to "software process", but to the generic definition in the terminology of Markov models.

[4] In the current version of our MARKOV tool, only first- and second-order probability tables are constructed. A future version of the tool will accept the maximum order as a parameter.

	A	B	C
A	0.00	0.50	0.50
B	0.54	0.00	0.46
C	0.42	0.58	0.00

	A	B	C
AA	0.00	0.00	0.00
AB	0.00	0.00	1.00
AC	0.50	0.50	0.00
BA	0.00	0.00	1.00
BB	0.00	0.00	0.00
BC	0.33	0.67	0.00
CA	0.00	1.00	0.00
CB	1.00	0.00	0.00
CC	0.00	0.00	0.00

Table 1: First- and Second-order Event-sequence Probability Tables for the Example Event Stream of Figure 2.

Table 1 shows first- and second-order probability tables for the event sequence of Figure 2. For instance, as given by the third row of the second-order table, the event seqeuence `A-C` *is equally likely to be followed by an* `A` *or a* `B`*, but is never followed by a* `C`*.*

2. A directed graph, called the *event graph*, is constructed from the probability tables in the following manner. Each event type is assigned a vertex. Then, for each event sequence that exceeds the threshold probability, a uniquely labeled edge is created from an element in the sequence to the immediately following element in that sequence.

   Consider event sequence `A-C-B`*, whose entry in the second-order table is* 0.50*. For a threshold less than* 0.50*, edges are created from vertex* `A` *to vertex* `C` *and from vertex* `C` *to vertex* `B`*.*

3. The previous step can lead to over-connected vertices that imply event sequences that are otherwise illegal. To correct this, over-connected vertices are split into two or more vertices.

   Consider vertex `B`*. After the previous step, edges exist from* `B` *to* `C` *and from* `C` *to* `B`*. However, sequence* `C-B-C`*, permissible under this connectivity, has a zero probability (see row 8 of the second-order table). Thus, vertex* `B` *is split into two* `B` *vertices, one having an edge to* `C` *and the other having an edge from* `C` *to it. This avoids the illegal sequence* `C-B-C`*.*

   The general definition of this step works by finding disjoint sets of input and output edges for a vertex that have some non-zero sequence probability, and splitting the vertex into as many vertices as there are sets.

4. The event graph G is then converted to its dual G' in the following manner. Each edge in G becomes a vertex in G' marked by the edge's unique label. For each in-edge/out-edge pair of a vertex in G, an edge is created in G' from the vertex in G' corresponding to the in-edge to the vertex in G' corresponding to the out-edge. This edge is labeled by the event type.

   In Figure 7, vertex 5 *and its edges are constructed from an edge labeled "5" in the event graph that connects vertex* `B` *to vertex* `C`*.*

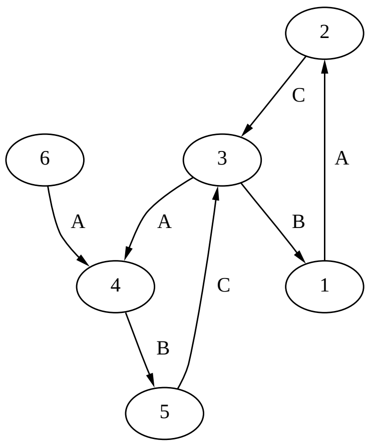

Figure 7: FSM Inferred by the MARKOV Method for the Example Event Stream of Figure 2.

The graph constructed in the last step is an FSM. This FSM can be further reduced using some techniques implemented in the MARKOV tool that merge nondeterministic transitions. For brevity, we do not describe those techniques here.

Figure 7 shows the inferred FSM that MARKOV produces from the example stream of Figure 2. As with KTAIL, MARKOV infers an FSM with exactly two loops: the `A-B-C` loop as the path $< 3, 4, 5, 3 >$ and the `B-A-C` loop as path $< 3, 1, 2, 3 >$. The difference is that MARKOV produces a deterministic FSM. Notice, too, the existence of what can be interpreted as a start state (6) in the FSM produced by MARKOV. This is a result of using the single sample input that begins with the token `A`.

Unlike the KTAIL method, MARKOV is robust in the presence of noise in the event stream. Moreover, the level of this robustness can be easily controlled by the process engineer through the probability threshold parameter.

4 Discovery of an Example Process

In this section we show how the three methods perform on a rather more complex process than the simple example of the previous section. The example in this section is taken from the ISPW 6/7 process problem [20]. We describe the process using an FSM that is based on Kellner's Statemate solution [19]. Our version is shown in Figure 8. The idea is to see how well the methods perform at reproducing this FSM and, thereby, discovering the process.

At a high level, the ISPW 6/7 process proceeds as follows. A change order from the Change Control Board (CCB) triggers the start of the process. The design modification subprocess is scheduled and performed, leading to the scheduling and performing of the code modification sub-

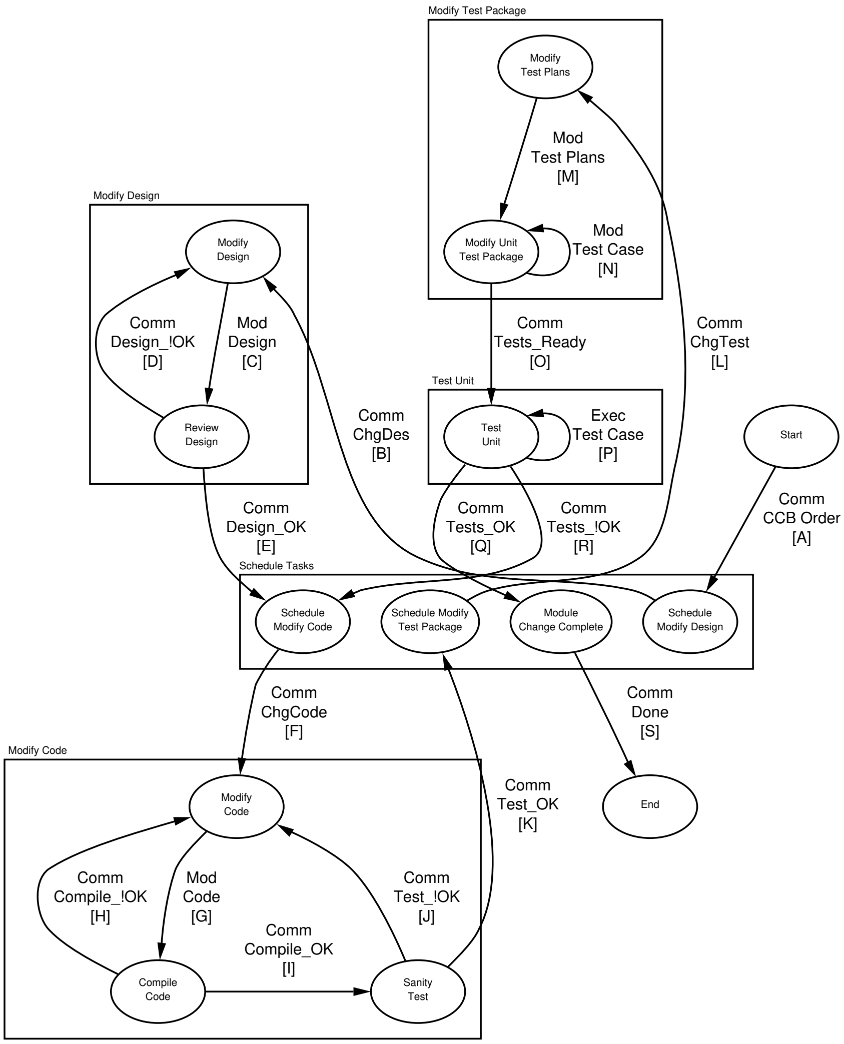

Figure 8: ISPW 6/7 Process.

process. After code modification, the test package modification subprocess is scheduled and takes place, followed directly by the unit testing subprocess. If unit testing fails, the process loops back to the code modification subprocess (since the design modification is assumed to be correct) to redo the previous few process steps. This loop continues until unit testing succeeds and the process completes.

As stated in Section 2, an activity that spans time is represented by the interval between two or more events. Here we simplify the presentation by collapsing an activity into a single event. Transitions in the FSM of Figure 8 are labeled with the events that occur as the process executes. Many of the events are communication events, as indicated by "Comm" in their labels. Others are modification events, indicated by "Mod". Finally, there is one program execution event in the process and it is labeled with "Exec". Along with each label is a letter enclosed in brackets (e.g., `[A]`). The letter denotes the token that is used to identify the corresponding event in the sample streams fed to the methods. In the figures that show the FSMs produced by the methods (i.e., figures 9, 10, and 11), the tokens are used to identify events; the reader can refer to Figure 8 to relate a token to an event.

The boxes in Figure 8 serve to pictorially group individual states into subprocesses. The labels on the boxes correspond to the activity titles given in the ISPW 6/7 problem statement. In the figures for the inferred FSMs, below, we also group states into subprocesses in order to aid in the explanation of the results. Instead of using boxes with solid borders, however, we use boxes with dotted borders to emphasize the fact that we have placed the boxes into the figures by hand; the methods do not automatically group states into subprocesses.

We use the FSM of Figure 8 to generate sample event streams for the methods. Three different event streams were generated and used as input to the KTAIL and MARKOV tools. One of them, for example, is the following.

```
ABCDCEFGHGIJGIKLMNOPRFGIKLMNOPQS
```

RNET was given only one event stream for this example. The reason is that, since RNET is purely statistical in nature, results for a small example such as the ISPW 6/7 process can vary significantly on different sample input streams. To show the best obtainable results for RNET, this one stream artificially contained a balanced proportion of different event sequences.

Figure 9 shows the FSM that was discovered by the RNET neural network method. For this example, we use a network of 20 input neurons, 30 hidden neurons, and 20 output neurons. We also use an event window of size 3.

Looking at the results for RNET, we see that the *Modify Design* subprocess (represented by states 2, 3, and 4) and the *Modify Code* subprocess (represented by states 5 through 9) are evident in the FSM. Overall, however, the results are fairly poor, even given an artificially balanced sample. In particular, RNET has failed to differentiate the state transitions of some important event sequences. For example, state 14 has recurrent edges for three different events, which is not very informative. A similar problem arises with state 13. The root of the problem is that, since RNET looks at the statistical history, it does poorly at differentiating loop exits. That is, upon seeing the exit token, for example the `E` in `C-D-C-D-C-E` (states 3 and 4), it still activates the state in the loop that can

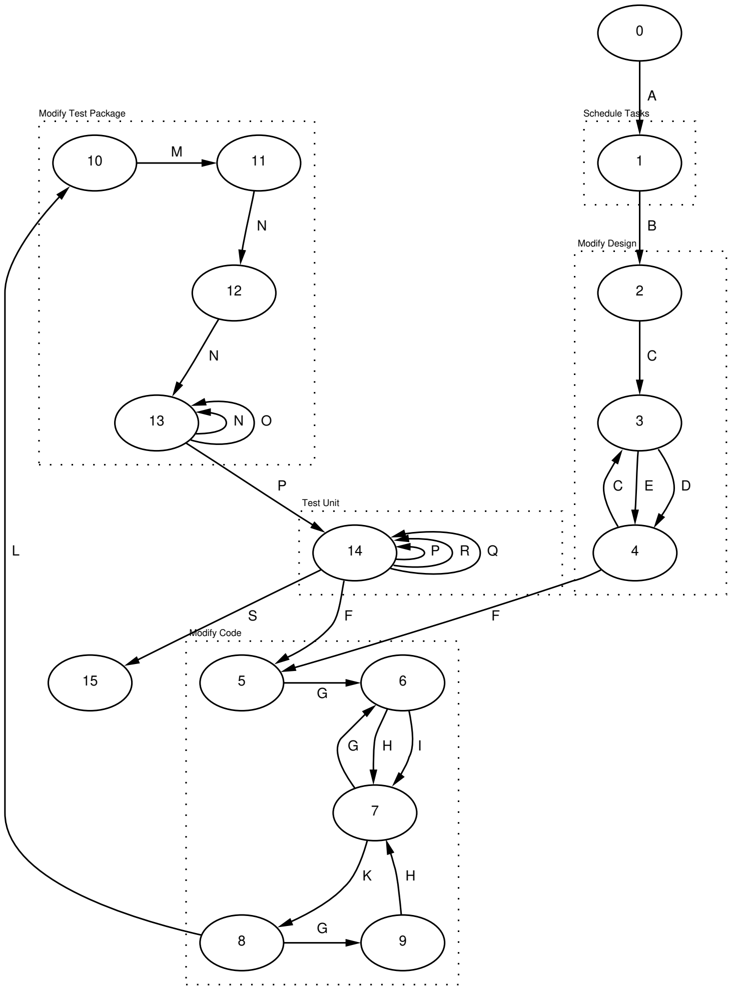

Figure 9: RNet **Discovery of Process in Figure 8.**

accept a `D` (state 4). Thus, the real exit of the loop does not happen until one token (`F`) later. This is true for all loops in this example; `O` continues the loop at 13, `R` and `Q` at 14, and `K` at 8. One effect of this is that a *Schedule Tasks* subprocess is hard to locate in the FSM, since the intermediate scheduling states are not distinct.

Figure 10 shows the FSM that was discovered by the MARKOV method. For this example, since we know that the data are exactly correct (i.e., contain no noise), the probability threshold is set to zero, so that no non-zero transition is ignored.

One can see that this method produces a good model that is very close to the one in Figure 8; it is equivalent in most all of its structure. The subprocesses are easily located in this model, and the loop paths, though having usually one extra state in them, are easily distinguished. The extra state for each loop results from the fact that the algorithm creates an entry state, such as 3, for each loop. However, MARKOV failed to merge the states 6 and 18 in *Schedule Tasks*, both of which provide an entry into the subprocess *Modify Code*. All in all, however, the method provides a clean and understandable representation of the process.

Figure 11 shows the FSM discovered by the KTAIL method. For this example, we use a value of 2 for the tuning parameter k.

The FSM successfully captures all the behavior of the example process. It has the correct number of states in the *Schedule Tasks* subprocess, finds the loops, and implements a correct machine that allows no anomalous behavior. However, as explained in Section 3.2, the algorithm consistently unrolls loops. The $(11, 12)$ and the $(13, 14, 22)$ subgraphs both have the same feature; their loops are unrolled by one state and, in the case of node 13, two exit paths are produced. Notice that the *Modify Code* subprocess shows distinct execution paths for each possible loop and for the exit path in that subgraph. With a straightforward merging of states 7, 18, and 19, and then a consequent merging of 8 and 20, the subgraph becomes identical to the corresponding subgraph in Figure 8.

In contrast to MARKOV, the KTAIL method correctly produces a single state (5) in the *Schedule Tasks* subprocess for the two possible paths into *Modify Code*, thus showing that it can recognize a shared subprocess in its entirety. This fact, along with the fact that most extra loop states can be algorithmically removed, causes us to view KTAIL as producing the best results of the three methods for this example.

4.1 Comparisons and Characterizations of the Inference Methods

Table 2 shows a comparison of the three methods with respect to several characteristics. An in-depth description of those characteristics follows.

Correct Model. Since KTAIL is purely algorithmic, it will always generate a correct model for the data it is given. MARKOV could potentially not generate a complete and correct model for the data given, since a threshold can be set to ignore low-probability event sequences. But if the threshold is set to 0, the method will always generate a complete and correct model. RNET, being purely statistical, cannot be configured to guarantee the generation a correct model.

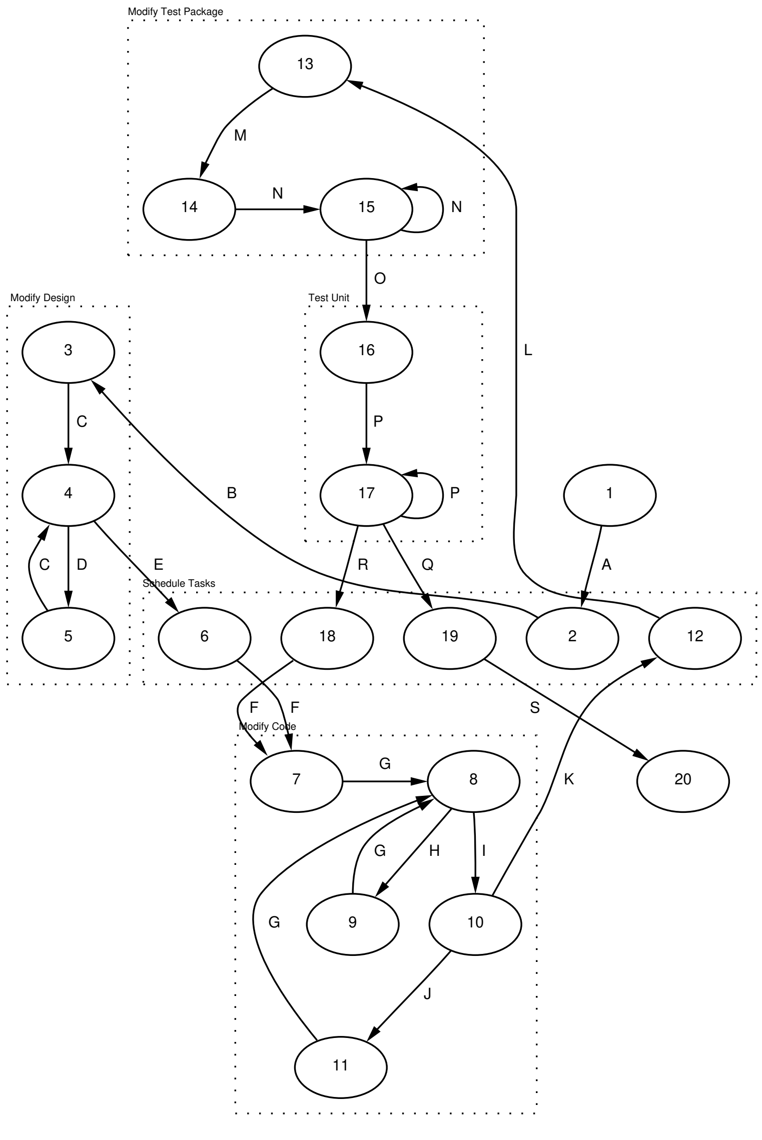

Figure 10: Markov **Discovery of Process in Figure 8.**

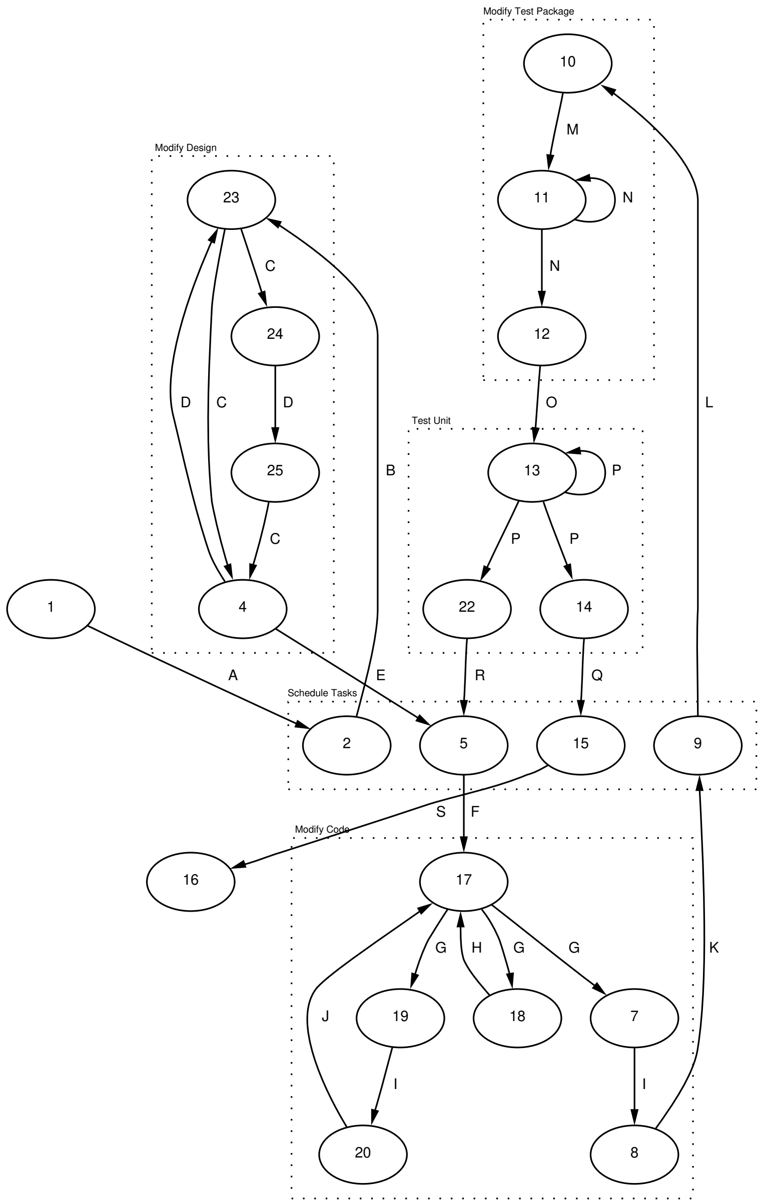

Figure 11: Ktail **Discovery of Process in Figure 8.**

Feature	KTAIL	MARKOV	RNET
Correct Model	Yes	Possibly	Unlikely
Speed	10 sec	10 sec	30 min
Tunability	one param	one param	many params
Usability	easiest	easy	hard
Robustness to Noise	none	good	good

Table 2: Comparison of Discovery Methods.

Speed. The numbers in Table 2 are approximate, intended to show relative speeds of the methods. They were taken from the performance of the methods on the ISPW 6/7 example. KTAIL and MARKOV are comparable, but RNET is considerably slower. This slowness is due mainly to the large amount of required training.

Tunability and Usability. These two characteristics are quite related, because often the more tunable something is, the more complicated is its interface. In this respect, it seems that RNET is overly tunable, or at least it takes too much knowledge of neural networks to establish a reasonable set of initial settings. KTAIL and MARKOV, being only tunable in one dimension, might be overly restrictive, but using them is straightforward.

Robustness to Noise. Both the MARKOV and RNET methods have the potential of being good at inferring models in the presence of collection errors and abnormal process events. KTAIL has no such ability.

5 Conclusions

We have described three methods of process data analysis that can be used to produce formal models corresponding to actual process executions. This analysis supports the process engineer in constructing initial process models. Based on our early experience with these methods, we conclude that the KTAIL and MARKOV methods show the most promise, and that the neural network based RNET method is not sufficiently mature to be used in practical applications. Of course, more experiments are needed to fully explore the strengths and weaknesses of all three methods.

To date, we have not addressed the issue of modeling concurrency in a process. There are several directions that we plan to explore. For example, we might preprocess the event stream to separate unrelated events based on their attributes, we might be able to modify the data collection method to generate separate event streams for different threads, or we might develop inference methods that use formalisms having concurrent behavioral models, such as communicating finite state machines [8].

Process discovery is not restricted to creating new formal process models. Any organization's process will evolve over time, and thus their process models will need to evolve as well. Methods for process discovery may give a process engineer clues as to when and in what direction the process model should evolve, based on data from the currently executing process.

There seems to be little related work in the area of process discovery. The most closely related effort takes a rather different view from that described in this paper. We summarize that work below.

- Garg and Bhansali [13] describe a method that uses explanation-based learning to discover aspects and fragments of the underlying process model from process history data and rules of operations and their effects. This work centers on using a rule base and goals to derive a generalized execution flow from a specific process history. By having enough rules, they show that a complete and correct process fragment could be generated from execution data.

- Garg et al. [14] employ process history analysis (mostly human-centered data validation and analysis) in the context of a meta-process for creating and validating domain-specific processes and software kits. This work is more along the lines of a process post-mortem that analyzes, through participant discussion, the changes that a process should undergo for the next cycle.

We know of no previous work in the field that investigates techniques for semi-automatic generation of formal process models from process execution data.

REFERENCES

[1] D. Angluin and C.H. Smith. Inductive Inference: Theory and Methods. *ACM Computing Surveys*, 15(3):237–269, September 1983.

[2] G.S. Avrunin, U.A. Buy, J.C. Corbett, L.K. Dillon, and J.C. Wileden. Automated Analysis of Concurrent Systems with the Constrained Expression Toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, November 1991.

[3] S. Bandinelli, A. Fuggetta, and C. Ghezzi. Software Process Model Evolution in the SPADE Environement. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, December 1993.

[4] N.S. Barghouti and G.E. Kaiser. Scaling Up Rule-based Development Environments. In *Proceedings of the Third European Software Engineering Conference*, number 550 in Lecture Notes in Computer Science, pages 380–395. Springer-Verlag, October 1991.

[5] P. Bates. Debugging Heterogenous Systems Using Event-Based Models of Behavior. In *Proceedings of a Workshop on Parallel and Distributed Debugging*, pages 11–22. ACM Press, 1989.

[6] A.W. Biermann and J.A. Feldman. On the Synthesis of Finite State Machines from Samples of Their Behavior. *IEEE Transactions on Computers*, 21(6):592–597, June 1972.

[7] M.G. Bradac, D.E. Perry, and L.G. Votta. Prototyping a Process Monitoring Experiment. In *Proceedings of the 15th International Conference on Software Engineering*, pages 155–165. IEEE Computer Society, May 1993.

[8] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 32(2):323–342, 1983.

[9] J.E. Cook and A.L. Wolf. Toward Metrics for Process Validation. In *Proceedings of the Third International Conference on the Software Process*, pages 33–44. IEEE Computer Society, October 1994.

[10] J. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, and D. Stemple. The Adriane Debugger: Scalable Application of Event-Based Abstraction. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–95. ACM Press, 1993.

[11] S. Das and M.C. Mozer. A Unified Gradient-Descent/Clustering Architecture for Finite State Machine Induction. In *Advances in Neural Information Processing Systems*, 1994. To appear.

[12] W. Deiters and V. Gruhn. Managing Software Processes in the Environment MELMAC. In *SIGSOFT '90: Proceedings of the Fourth Symposium on Software Development Environments*, pages 193–205. ACM SIGSOFT, December 1990.

[13] P.K. Garg and S. Bhansali. Process Programming by Hindsight. In *Proceedings of the 14th International Conference on Software Engineering*, pages 280–293. IEEE Computer Society, May 1992.

[14] P.K. Garg, M. Jazayeri, and M.L. Creech. A Meta-Process for Software Reuse, Process Discovery, and Evolution. In *Proceedings of the 6th International Workshop on Software Reuse*, November 1993.

[15] R.M. Greenwood. Using CSP and System Dynamics as Process Engineering Tools. In *Proceedings of the Second European Workshop on Software Process Technology*, number 635 in Lecture Notes in Computer Science, pages 138–145. Springer-Verlag, September 1992.

[16] V. Gruhn and R. Jegelka. An Evaluation of FUNSOFT Nets. In *Proceedings of the Second European Workshop on Software Process Technology*, number 635 in Lecture Notes in Computer Science, pages 196–214. Springer-Verlag, September 1992.

[17] W.S. Humphrey. *Managing the Software Process.* Addison-Wesley, Reading, Massachusetts, 1989.

[18] M.L. Jaccheri and R. Conradi. Techniques for Process Model Evolution in EPOS. *IEEE Transactions on Software Engineering*, 19(12):1145–1156, December 1993.

[19] M.I. Kellner. Software Process Modeling Support for Management Planning and Control. In *Proceedings of the First International Conference on the Software Process*, pages 8–28. IEEE Computer Society, October 1991.

[20] M.I. Kellner, P.H. Feiler, A. Finkelstein, T. Katayama, L.J. Osterweil, M.H. Penedo, and H.D. Rombach. Software Process Modeling Example Problem. In *Proceedings of the 6th International Software Process Workshop*, pages 19–29, October 1990.

[21] E. Koutsofios and S.C. North. Drawing Graphs with Dot. AT&T Bell Laboratories, October 1993.

[22] R.J. LeBlanc and A.D. Robbins. Event-Driven Monitoring of Distributed Programs. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 515–522. IEEE Computer Society, May 1985.

[23] L. Miclet and J. Quinqueton. Learning from Examples in Sequences and Grammatical Inference. In G. Ferrate, T. Pavlidis, A. Sanfeliu, and H. Bunke, editors, *Syntactic and Structural Pattern Recognition*, volume 45 of *NATO ASI Series F: Computer and Systems Sciences*, pages 153–171. Springer-Verlag, New York, 1988.

[24] D.E. Perry, N.A. Staudenmayer, and L.G. Votta. People, Organizations, and Process Improvement. *IEEE Software*, 11(4):36–45, July 1994.

[25] B. Peuschel and W. Schäfer. Concepts and Implementation of a Rule-based Process Engine. In *Proceedings of the 14th International Conference on Software Engineering*, pages 262–279. IEEE Computer Society, May 1992.

[26] M. Saeki, T. Kaneko, and M. Sakamoto. A Method for Software Process Modeling and Description Using LOTOS. In *Proceedings of the First International Conference on the Software Process*, pages 90–104. IEEE Computer Society, October 1991.

[27] S.M. Sutton, Jr., H. Ziv, D. Heimbigner, H.E. Yessayan, M. Maybee, , L.J. Osterweil, and X. Song. Programming a Software Requirements-specification Process. In *Proceedings of the First International Conference on the Software Process*, pages 68–89. IEEE Computer Society, October 1991.

[28] A.L. Wolf and D.S. Rosenblum. A Study in Software Process Data Capture and Analysis. In *Proceedings of the Second International Conference on the Software Process*, pages 115–124. IEEE Computer Society, February 1993.