Discovering Models of Software Processes from Event-Based Data

Jonathan E. Cook and Alexander L. Wolf

Software Engineering Research Laboratory
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-819-96 November 1996

A version of this report to appear in
ACM Transactions on Software Engineering and Methodology

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, which would be lowered by automated assistance in creating formal models.

To this end, we have developed a data analysis technique that we term process discovery*. Under this technique, data describing process events are first captured from an on-going process and then used to generate a formal model of the behavior of that process. In this paper we describe a Markov method that we developed specifically for process discovery, as well as describe two additional methods that we adopted from other domains and augmented for our purposes. The three methods range from the purely algorithmic to the purely statistical. We compare the methods and discuss their application in an industrial case study.*

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 Defense 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

A *software process* is a set of activities applied to artifacts, leading to the design, development, or maintenance of a software system. Examples of software processes include design methods, change-control procedures, and testing strategies. The general intent of a software process is to coordinate individual activities so that they achieve a common goal.

The challenge of managing and improving the software process followed by an organization has come to the forefront of software engineering research. In response, new methods and tools have been devised to support various aspects of the software process. Many of the technologies, including process automation [8, 9, 28, 48, 53], process analysis [36, 37, 40, 50], and process evolution [7, 38], assume the existence of a 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 have explored methods for automatically deriving a formal model of a process from basic event data collected on the process [25]. 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. The challenge in process discovery is to use those data to describe the process in a form suitable for formal-model-based process technologies, and in a form that allows an engineer to understand the model, and thus the process.

In general, this is a very difficult challenge to meet. To scope the problem, we have concentrated our efforts on models of the behavioral aspects of a process, rather than, for example, 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 restricted ourselves to the discovery of finite state machine models of behavioral patterns.

In this paper we present three methods for process discovery, ranging from the purely algorithmic to the purely statistical. The approach underlying the methods is to view the process discovery problem as one of *grammar inference*. In other words, the data describing the behavior of a process are viewed as sentences in some language; the grammar of that language is then the formal model of the process. The methods have been implemented as a tool operating on process data sets [22] and have been successfully employed in an industrial case study [23].

Although the methods are automated, they still require guidance from a process engineer who is at least somewhat familiar with the particular process under study. This guidance comes in the form of tuning parameters built into the methods, and from the selection and application of the event data. The results produced by the methods are initial models of a process that can be refined by the process engineer. Indeed, the initial models may lead to changes in data collection to uncover greater detail about particular aspects of the process. The discovery methods therefore complement a process engineer's knowledge, providing empirical analysis in support of experience and intuition.

We note that although our presentation of the discovery methods is done in the context of software processes, the methods are applicable to other kinds of processes—and, more generally, other kinds of behaviors—that can be characterized by event data. For instance, we have successfully applied the techniques to the discovery of operating-system-level interprocess communication protocols. Our experience with the methods to date, however, has been mainly with software processes.

The next section of the paper discusses the framework in which we define and analyze event data. Section 3 gives a more complete statement of the discovery problem and outlines our grammar inference approach. Section 4 provides needed background on grammar inference. The discovery methods themselves are then described in Section 5. Section 6 presents a comparative evaluation of the methods. Section 7 describes DaGama, the tool implementing the discovery methods. The application of the methods in an industrial case study is reviewed in Section 8. Finally, we conclude in Section 9 with a summary of our results, an overview of related work, and a discussion of future work.

2 An Event-Based Framework for Process Discovery

Our work in process discovery rests on a behavioral view of software processes as a sequence of actions performed by agents, either human or automaton, possibly working concurrently. Following Wolf and Rosenblum [56], 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. The use of event data to characterize behavior is already widely accepted in other areas of software engineering, such as program visualization [45], concurrent-system analysis [6], and distributed debugging [10, 26].

The "instant" of an event is relative to the time granularity that is needed or desired; thus, certain activities that are of short duration relative to the time granularity are represented as a single event. An activity spanning some significant 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 batch queue could be represented by the three events "enter queue", "begin compilation", and "end compilation".

For purposes of maintaining information about an action, events are typed and can have attributes; one attribute is the time the event occurred. Generally, the other event attributes would be items such as the agents and artifacts associated with an event, the tangible results of the action (e.g., pass/fail from a design review; errors/no-errors from a compilation), and any other information that gives character to the specific occurrence of that type of event. In the work described here, we do not make use of attributes other than time. This framework, however, is the basis for more extensive analyses (see Section 9).

The overlapping activities of a process, then, are represented by a sequence of events, which we refer to as an *event stream*. For simplicity, we assume that a single event stream represents one execution of one process, although depending on the data collection method and the organization, this assumption may be relaxed.

One can view a process execution as completely represented by an event stream containing events from all of the possible event types that might be generated during that process. However, the set of event types that are actually collected as data is usually smaller than this complete set. There are several reasons why this might occur, but two obvious ones are that data about a particular event type might be considered inconsequential or the data might be considered too expensive to collect. For instance, events that occur off the computer, such as most staff meetings, are likely to be more expensive to collect than events that occur on the computer, simply because off-computer events would require manual, as opposed to automated, collection techniques.

For this reason, event data collected from an executing process can be viewed as a window onto

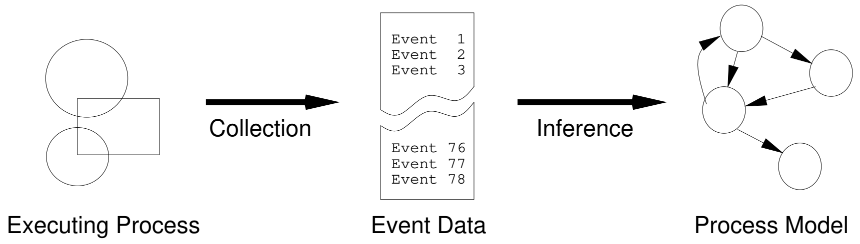


Figure 1: Process Model Discovery.

the execution. A discovered model can only account for those portions of the process for which events are collected. 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. This issue is currently being investigated in the domain of software process [13, 23, 55, 56].

3 Problem Statement and Approach

Our goal in this work is to use event data, in the form of an event stream, collected from a software process execution to infer a formal model of the behavior of the process. This framework is depicted in Figure 1. We do not envision being able to infer fully complete and correct process models, since this is in general an intractable problem. Rather, we aim to provide a process engineer with some formal description of the patterns of behavior that exist in the process, as seen through the collected execution event stream. An engineer can then use this information as a basis for creating a more complete process model, for evolving an existing (possibly informal) model, or for correcting some aspects of the execution.

What we seek to infer from the data are recurring patterns of behavior, specifically those involving sequencing, selection, and iteration. For our purposes, finite state machines (FSMs) provide a good starting point for expressing those patterns. By an FSM we mean a nondeterministic, transition-labeled state machine [19]. The ability to discover the start and stop states of FSMs is not of particular importance in this work, although in some cases it would be straightforward (i.e., the states that have no incoming/outgoing transitions, respectively). We assume that the process engineer can provide this additional context in the other cases, if such an identification is of interest.

One could consider choosing a more powerful representation than finite state machines, such as push-down automata or even Petri nets. The primary argument against this is that the more powerful the representation, the more complex the discovery problem. Furthermore, it is not clear the additional power is required in order to provide a useful model. More powerful representations than FSMs are arguably better suited for *prescribing* a software process. On the other hand, our experience has shown that FSMs are quite convenient and sufficiently powerful for *describing* historical patterns of actual behavior [23]. The drawback to FSMs is that they have no inherent ability to model concurrency. We address this issue in Section 5.5.

To develop our techniques, 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 [5]. This area of research historically is referred to as *grammar inference*. If one interprets events as tokens and event streams as sentences in the language of the process, then a natural analog becomes evident. There are, however, several important differences, as we discuss in Section 4.

Beside grammar inference, there are other areas of computer science that could provide a possible foundation for process discovery. Two prominent ones are *data mining* and *reverse engineering*. In the remainder of this section we briefly summarize how our problem and approach relate to these two areas.

3.1 Data Mining

Data mining is the name given to the task of discovering information in data. Different methods have emerged, targeting different kinds of data, such as relational databases, images, numerical time series, and sequence data. The specialized methods take advantage of the nature of the specific data to which they are applied.

Our kind of data is sequence data. For this kind of data, the long-established field of grammar inference has provided the primary foundation for data mining techniques. We therefore present our work using the terminology of this particular data mining method.

Recently, data mining has been placed in the larger framework of *knowledge discovery* [31]. In this framework, the various data mining methods provide just a single step in the whole process of information discovery [30].

3.2 Reverse Engineering

In reverse engineering, the goal is to recover the design of a system from its implementation. Reverse engineering methods generally examine a low-level specification (i.e., the implementation) and distill it into a high-level specification (i.e., a design) that captures information of help to an engineer in understanding the system as a whole [21].

There is a superficial relationship between reverse engineering and process discovery in that we are also trying to create a high-level specification, the process model. But rather than starting from a low-level specification, we start from behavioral trace data.

4 Grammar Inference

The grammar inference problem can be informally stated as follows.

> *Given some sample sentences in a language, and perhaps some sentences specifically not in the language, infer a grammar that represents the language.*

In this section we present a formal definition of the problem of grammar inference, its theoretical complexities, and a detailed look at work in this field that is applicable to the process discovery problem.

4.1 A Formal Definition

We define the grammar inference problem using the following terms.[1] Let Σ be the alphabet of tokens, and Σ^* be all possible sequences (sentences) over Σ. Then, we define a language L as

[1]This presentation is consistent with others in the field [5, 49, 52].

$L \subset \Sigma^*$, that is, L is a subset of sentences from Σ^*.[2] A grammar G is a formalism of some class (e.g., regular, context-free) that describes L; L is said to be in that class of languages.

We define Θ_L as the infinite set of all pairs $\langle s, l \rangle$ from $\Sigma^* \times \{0, 1\}$, where $l = 1$ if $s \in L$ and $l = 0$ otherwise. Thus, Θ_L is just Σ^* tagged with whether each sequence belongs to L or not. We call the sequence s in $\langle s, l \rangle$ positive if $l = 1$ (i.e., it is in L) and negative otherwise. Π_L is defined as the possibly infinite set of all positive sequences, and Γ_L is defined as the possibly infinite set of all negative sequences.

For learning purposes, an algorithm needs some presentation of data. A complete presentation is just a presentation of Θ_L. A positive presentation is just a presentation of Π_L. A negative presentation is just a presentation of Γ_L. Note that any of these sets are or can be infinite, so that an algorithm cannot use the whole presentation; yet, we cannot beforehand eliminate any single sequence from the presentation. Because some of the sequences can be infinite themselves, any theoretical analysis assumes only a presentation of finite sequences. A partial presentation is a finite subset of a complete, positive, or negative presentation.

The grammar inference problem can now be stated formally as follows.

Given some presentation of all or part of Θ_L, infer a grammar G describing L.

4.2 Previous Complexity Results

Gold's "identification in the limit" was the first framework for analyzing the grammar inference problem [34]. Identifying in the limit frames the problem in terms of looking at the complete presentation of Θ_L. If, after some finite presentation of pairs from Θ_L, the algorithm guesses G correctly, and does not change its guess as the presentation continues, then the algorithm is said to identify G in the limit. Gold [35] showed that finding a deterministic finite state automata (DFA) having a minimum number of states, given a presentation of Θ_L, is NP-hard and, moreover, that it is impossible given just a positive presentation using Π_L. In essence, the problem with using only positive samples is that the algorithm has no way to determine when it is overgeneralizing.

Thus, this early result shows that learning the best grammar of L, even for relatively simple classes (i.e., regular languages), is very difficult, and that learning from only positive examples is strictly weaker than learning from both positive and negative examples.

The PAC model for learning concepts from examples was proposed by Valiant [54]. PAC stands for "Probably Approximately Correct". Probably is defined as within some probability $1 - \delta$, and approximately is defined as within some $1-\epsilon$ of the correct solution. If a polynomial-time algorithm can be constructed for some learning problem that, within some likelihood (δ) produces a solution that is close (within ϵ) to the correct one, then that domain is PAC-learnable. Classes of boolean formulas and decision trees, and some geometric and algebraic concepts have been shown to be PAC-learnable. Results for grammars, including DFAs, are more negative, but less definite. As Pitt reports:

> *If DFAs are polynomially approximately predictable, then there is a probabilistic polynomial time algorithm for inverting the RSA encryption function, for factoring Blum integers, and for deciding quadratic residues [49].*

Angluin [4] phrases the learning problem in terms of an oracle. An algorithm can make a fixed set of queries to the oracle, the basic two being "Is this string accepted by the correct grammar?" and "Is this grammar equivalent to the correct grammar?". This active model of learning is different from

[2] $L = \Sigma^*$ is not a very interesting language.

the passive presentations above; the algorithm has a certain amount of control over the information it receives and uses, rather than just being given a set of data. Specifically, the algorithm can ask questions to elucidate conflicts in the grammar. With this framework, Angluin gave a polynomial-time algorithm for learning DFAs, and others have extended this to other classes of languages.

There is much work describing the computational complexities of various learning paradigms and classes of languages or formulas. The survey by Angluin and Smith [5] is a broad look at inductive inference learning. Pitt's survey [49] is a good starting point for understanding the theoretical complexities involved in learning regular grammars. Further theoretical work on language learning is also available [2, 39, 43, 44, 51].

4.3 Practical Inference Techniques

There has been some work that has devised or used practical algorithms for grammar inference in the analysis of real-world problem domains. One class of techniques represents a common paradigm in learning from positive examples. The basic idea behind this class is to build a prefix tree machine from the input data, where each sentence has a linear state-transition sequence that produces it. Each linear sequence starts at the same start state, and shares states until that sequence diverges from every other example sentence. Thus, the machine is a tree, the root being the start state, and each state that has more than one out transition being a point where example sentences no longer share a prefix.

For a set of data that covers a language (has examples of every behavior), any grammar for the language can be shown to be contained in the prefix tree machine. This machine is then iteratively merged according to some rules until a final state machine is output as the learned language. Examples of this are the following.

- Ahonen et al. [1] describe a prefix tree method where states are merged based on previous contexts. That is, if two states are entered from the same k-length contexts, then they are merged.

- Angluin [3] merges states of a prefix tree based on a notion of k-reversibility, which restricts the class of languages the algorithm can infer. While 0-reversible languages can be inferred in near-linear time, higher k values cause the algorithm to be cubic in the length of the input.

- Bierman and Feldman [11] describe a prefix tree method where states are merged based on k-length future behaviors. Although their method is not directly described as prefix tree merging, it does fall into this class.

- Carrasco and Oncina [18] describe a statistical method for learning stochastic regular languages, based on state merging from a prefix tree, where states are merged based on statistical likelihood calculations. Running time is maximally n^3; however they claim actual times are near-linear.

- Castellanos et al. [20] apply a method of this class to learning natural language translators.

Another class of techniques is directed more towards iteratively building up a regular expression from the sequences, and then translating that into an FSM (if necessary). In this sense, these algorithms make a pass through a sequence, find repetitive patterns, replace all occurrences with the regular expression, then start all over until it is decided that they are done. Two examples are the following.

- Brāzma et al. [15, 16, 17] have been able to construct fast algorithms that learn restricted regular expressions. The restrictions they place on the inferred regular expressions are that both the selection operator and the nesting level of iterations are limited and fixed. Although the algorithm is in worst case cubic, in practice they report near-linear times. Moreover, they have extended this work to handle noisy strings by treating the noise as an edit distance problem, although currently they only allow for a single-token edit, not multi-token gaps. Their work is geared towards analyzing biosequence (i.e., DNA/RNA and protein) data.
- Miclet [46] describes a method called the uv^kw algorithm, where each pass looks for repetitions of substrings (the v^k), chooses the best candidate, and replaces it with an iteration expression. The next pass is then started on the reduced string. This method is well-adapted for loops, but has problems with the selection operator as well.

Miclet's survey [46] describes several other techniques for inferring (N)DFAs from positive samples, and is, in general, a good reference for practical inference techniques.

4.4 Inference Using Neural Nets

The neural network community has also looked at the problem of grammar inference from positive samples [27, 57]. The various methods all consist of defining a recurrent network architecture, and then analyzing the hidden neuron activity to discover the states and transitions for the resulting grammar. The difference among these methods is how they inspect the hidden neurons to infer state information. The most advanced techniques apply clustering to the neuron activations during training, to help induce better state-machine behavior.

These techniques are somewhat controversial, because many neural network researchers believe one should not look inside the net to try to learn anything, but should only be concerned with the input/output performance of the net. Nevertheless, this is a thread of research that is applicable to our investigations.

4.5 Limitations of Grammar Inference

The grammar inference methods presented here look only at the tokens that make up the language. For our purposes, this means that they are limited to operating on event types (being mapped to tokens), and ignore the potential information in event attributes. Thus, causal relationships between event attributes cannot be deduced using grammar inference techniques.

Another limitation is that the methods generally do not support seeding the algorithm with pre-known information about a model (or grammar). Often an engineer will know something about the process under study, and may be able to formulate a skeleton of a model. It would be useful to have methods that could be seeded with such partial information and then automatically infer the details.

A final restriction is that the methods assume a single state machine. In the typical process, activities generally occur concurrently, which produce an event stream that may have nondeterministic orderings of events. In fact, an event stream could have interleaved executions of the same process model, and separations of these executions might only be done by relating the event attributes. We discuss the problem of discovering concurrent behavior in Section 5.5.

`RCERCECRECRERCECRECRERCECRERCERCECRECR`

Figure 2: Simple Event Stream Example.

5 Methods for Process Discovery

In this section we describe three grammar inference methods. Two of the methods predate our work in process discovery, but required adaptation and, in some areas, significant extension for process discovery. The third method is a new one that we have devised. After describing the methods, we consider what it means to discover a "good" model and then consider the issue of concurrency in the process.

Because 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 way in which they examine those samples.

1. The RNet method is a purely statistical (neural network) approach that looks at the *past* behavior to characterize a state. For this method we have extended an implementation by Das and Mozer [27]. Our extensions allow this method to handle more event types (theirs was restricted to two), and enable the easier extraction of the discovered model from the net.

2. The Ktail method is a purely algorithmic approach that looks at the *future* behavior to compute a possible current state. We modified and then implemented the theoretical description given by Biermann and Feldman [11] to make it less dependent on multiple sequences and to extend it to handle noisy data. In addition, we added post-analysis steps that remove some overly complex constructs that the basic algorithm tends to place into a discovered model.

3. The Markov method is a hybrid statistical and algorithmic approach that looks at the *neighboring* past and future behavior to define a state. This method is a new inference method.

To illustrate the three methods in this section, we use a simple process involving three types of events—`Edit`, `Review`, and `Checkin`—that characterize a common set of activities associated with a software artifact such as a design document. We use the sample process event stream shown in Figure 2. In the figure, the three event types are abbreviated `E`, `R`, and `C`. All three of our methods typically are given multiple samples to refine their results, but in this simple example we just give them the one stream shown.

Two sequences of events are exhibited in the data, `Edit-Review-Checkin` and `Edit-Checkin-Review`, reflecting two ways that the activities are carried out. This is a perfect sample, in the sense that only those two sequences are present, with no extraneous "noise" in the data. 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, namely the two sequences within the iterative behavior of the process. While such a simple example may seem contrived, it involves the fundamental problem of identifying two different patterns over the same event types.

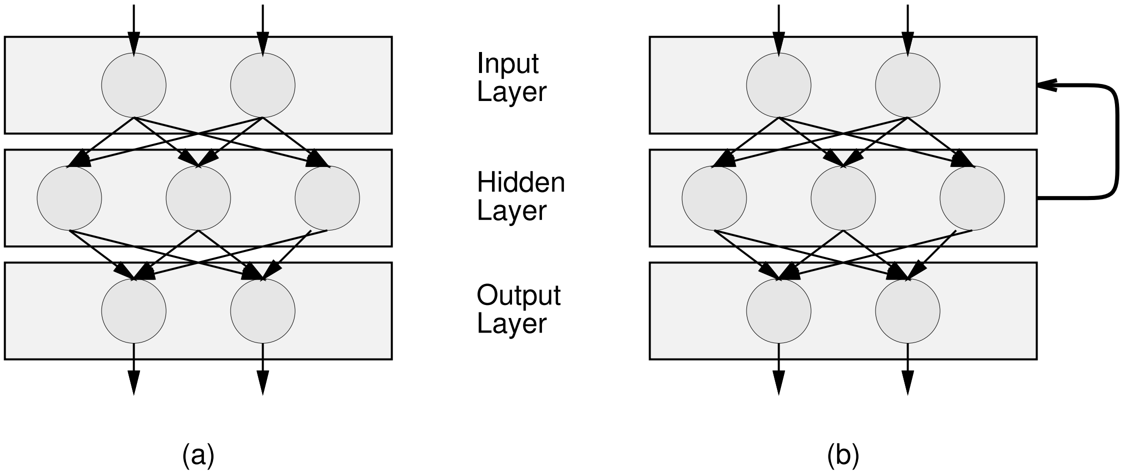


Figure 3: Standard (a) and Recurrent (b) Neural Network Architectures.

5.1 RNet

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. This difference is referred to 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.

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 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.

The parameters to RNet are numerous and varied, and the realistic application of RNet requires some knowledge of how neural networks operate and what can be expected from them.

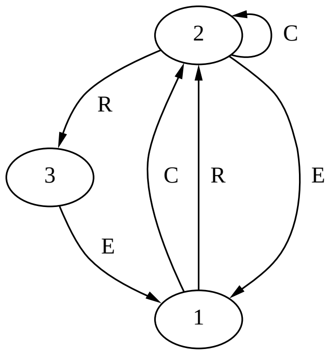


Figure 4: FSM Inferred by the RNet Method for the Example Event Stream of Figure 2.

The parameters are the following.

- *Window size presented:* controls the size of the locality that is used in learning.
- *Number of hidden neurons:* controls the level of detail in information (and thus an FSM) the network can learn.
- *Weight of the recurrent connection with respect to the forward connections:* controls the importance of the history of the network.
- *Learning rate and momentum:* control how fast the algorithm learns and the importance given to the previous direction in learning.
- *Weight of the clustering method:* controls how strongly the network tries to cluster similar states into one.
- *Number of training iterations:* controls the error threshold at which to stop.

It is beyond the scope of our work to fully evaluate all the dimensions. (In fact, many neural network practitioners still refer to parameter setting as an "art".) Our experience is that as one becomes more familiar with RNet, its behavior with respect to certain parameters eventually can be learned.

The Das and Mozer implementation is restricted to two-token languages, and is thus quite limited. We extended the implementation to allow for an arbitrary number of token types, and also enhanced the output of the network to more easily extract the complex models that result from having multiple token types.

Figure 4 shows the FSM that RNet infers from the example stream of Figure 2. The method successfully produces a deterministic FSM that incorporates both the `Edit-Review-Checkin` and `Edit-Checkin-Review` loops. But it also models behavior that is not present in the stream, such as an `Edit-Review-Review` loop and a `Checkin` 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).

Its disadvantages are that it is in general very slow, because of the training time required, and the size of the net grows rapidly with the number of token types and the expected size of the resulting model. The plethora of tuning parameters and the lack of guidelines in setting them are drawbacks as well.

5.2 Ktail

The next method is purely algorithmic and based on work by Biermann and Feldman [11]. Their formulation is given in terms of sample strings and output values for the FSM. 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, our method is less dependent on having multiple sample strings, reduces the number of states in the resulting FSM, and provides a threshold parameter for dealing with noisy data.

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 its possible futures. 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.

The equivalence class is called the *Nerode* relation for the prefixes in the class, and the extension that Ktail uses is to limit the Nerode relation to a fixed k-length future. Additionally, the definition by Biermann and Feldman only examines the last k-length portion of the string (presumably because they were interested in a large set of short strings) and the output function of the state machine. We instead examine a k-length future from all points in an input string.

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.[3] 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.

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.

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

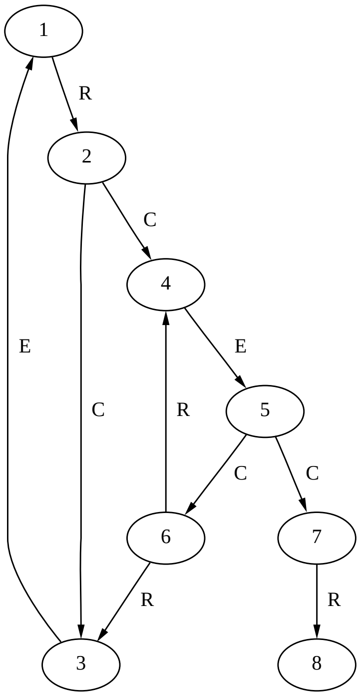


Figure 5: FSM Inferred by the Basic Biermann-Feldman Algorithm ($k = 2$) for the Example Event Stream of Figure 2.

5.2.1 Our Enhancements to the Basic Algorithm

While the Biermann and Feldman algorithm produces an FSM that is complete and correct, it has certain tendencies to produce an overly complicated FSM. Figure 5 shows the FSM that is produced by the algorithm, with $k = 2$, 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 equivalence class. Consider the `Edit-Review-Checkin` loop. In the figure, this loop is represented by the path $< 3, 1, 2, 3 >$ in the FSM. But the last iteration through the loop has a separate representation as the path $< 2, 4, 5 >$. A similar effect occurs with the `Edit-Checkin-Review` loop.

We 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 sets of output transition tokens for the states $S_2 \ldots S_n$ are equivalent or strict subsets, 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 3 and 4 can be merged, states 6 and 7 can be merged, and state 8 can be merged with the merged state 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 `Edit-Review-Checkin` loop is completely embodied in the

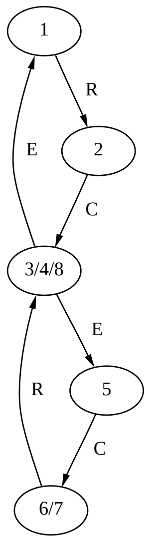


Figure 6: FSM Inferred by the KTAIL Method ($k = 2$) for the Example Event Stream of Figure 2.

transition path $< 3/4/8, 1, 2, 3/4/8 >$, while the `Edit-Checkin-Review` loop is completely embodied in the path $< 3/4/8, 5, 6/7, 3/4/8 >$. Note that there is a point of nondeterminism in state 3/4/8 upon the occurrence of an `Edit`. This is because the algorithm sees two different behaviors (i.e., tails) from the `Edit` event in the sample data.

In addition to reducing the complexity of the FSM, we add the ability to ignore noise in the data. Assuming that noise is low-frequency event sequences, then an equivalence class with very few members may be interpreted as a place where the k-length futures are erroneous. Thus, we could decide that this equivalence class does not represent proper behavior and throw it away, not allowing it to appear as a state in the final FSM. This formulation suggests setting a threshold on the size of the equivalence class (in terms of the number of members), and then pruning all classes less than that size. Pruned equivalence classes will not appear in the resulting state machine, and potential transitions to them will also not appear. This pruning mechanism is implemented in the KTAIL method.

5.2.2 Evaluation

If the noise threshold is greater than zero, then we cannot guarantee the correctness of the resulting FSM with respect to the input data. On the other hand, with the noise threshold set to zero (i.e., we assume that there is no noise in the data), KTAIL necessarily produces an FSM that can recognize all sample input data. This is because every position in the set of input strings belongs to some equivalence class, and is thus mapped to some state in the state machine. Since equivalence classes share k-length subsequent behaviors, the transitions from each state are guaranteed to satisfy each position in the set of input strings that is in that equivalence class. Moreover, the sequence of equivalence classes for the positions in an input string is the production path in the FSM for that string.

Since we only merge states that have similar output behavior when reducing the FSM, this transformation has no effect on the correctness of the resulting FSM. It only (possibly) reduces the number of spurious states that the FSM contains.

KTAIL is controlled in its accuracy by the value of k; the greater the value, the greater the length of the prefix tails that are considered, and thus the more differentiation that 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 its simple approach to noise thresholding may not be tunable enough to be robust in the presence of significant input stream noise.

5.3 MARKOV

The third method, a hybrid of algorithmic and statistical approaches, is one that we devised. The method uses the concept of 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 [47] use sequence 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[4] representation that assumes:

- 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 sequences of events. 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 probability and number of occurrences are above user-specified cutoff thresholds. MARKOV thus proceeds in four steps.

1. The event-sequence probability tables are constructed by traversing the event stream. (In the current version of our implementation of the MARKOV method, only first- and second-order probability tables are constructed.) In addition, counts of event sequence occurrences are tallied.

[4]Here the term "process" does not refer to "software process", but rather to the standard meaning given in the terminology of Markov models.

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 sequence `Review-Edit` *is equally likely to be followed by a* `Review` *or a* `Checkin`*, but is never followed by an* `Edit`*.*

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

   Consider event sequence `Review-Edit-Checkin`*, which occurs three times in the event stream and whose entry in the second-order table is 0.50. For a count threshold less than 3 and a probability threshold less than 0.50, edges are created from vertex* `R` *to vertex* `E` *and from vertex* `E` *to vertex* `C`*.*

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. This is done 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.

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

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* `C` *to vertex* `E`*.*

The graph G' constructed in the last step is an FSM. This FSM can be further reduced by merging nondeterministic transitions. Specifically, Markov can produce states where more than one transition from the state is annotated with the same event, but where this nondeterminism is not useful because the next states' behaviors are non-conflicting; nondeterministic transitions only make sense if they resolve a conflict in some way. Thus, for the nondeterministic transitions from a state, if their destination states' output transitions are non-conflicting (that is, they accept a disjoint set of events), then these states are merged and the set of nondeterministic transitions becomes one transition.

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 `Edit-Review-Checkin` loop as the path $< 5, 3, 4, 5 >$ and the `Edit-Checkin-Review` loop as path $< 2, 3, 1, 2 >$. 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 `Review`.

5.3.1 A Bayesian Extension to the Markov Method

Markov uses the frequencies of event sequences directly in its tables as forward probabilities of these sequences occurring. But there is some argument that Bayesian probabilities, which reverse

	R	C	E
R	0.00	0.50	0.50
C	0.54	0.00	0.46
E	0.42	0.58	0.00

	R	C	E
RR	0.00	0.00	0.00
RC	0.00	0.00	1.00
RE	0.50	0.50	0.00
CR	0.00	0.00	1.00
CC	0.00	0.00	0.00
CE	0.33	0.67	0.00
ER	0.00	1.00	0.00
EC	1.00	0.00	0.00
EE	0.00	0.00	0.00

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

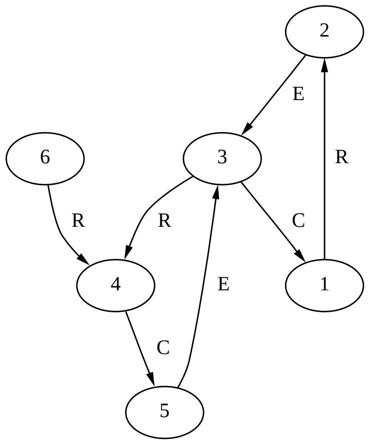


Figure 7: FSM Inferred by the MARKOV Method for the Example Event Stream of Figure 2.

the probabilities, would be better. Using the frequencies directly as probabilities can be affected by the occurrences of the events. For example, if the second event of a two-event sequence occurs very often in an event stream, then the frequency of that sequence might be high simply by chance. A Bayesian calculation can account for this.

We have implemented a Bayesian transformation in MARKOV that is selectable as an option. This transformation takes the forward probability tables built in the standard MARKOV method, and converts them in place using Baye's Law:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Note that all components of the right-hand side are in the forward probability tables. The Bayesian probabilities replace the forward probabilities, so that when MARKOV looks up the probability of the sequence `RC`, for example, it is now using the Bayesian probability instead.

To date, we have not intensively explored the Bayesian extension in the presence of thresholding, but have not yet seen radically different results from the regular MARKOV methods. It is available, however, for those who believe that Bayesian probabilities are better.

5.3.2 Evaluation

As with KTAIL, if the noise threshold used with MARKOV is greater than zero, then we cannot guarantee the correctness of the resulting FSM with respect to the input data. But with the threshold set to zero, MARKOV will create an FSM that accepts all the sample input data. For input stream S, consider the subsequences S_i^N of length N at position i, and S_{i+1}^N of length N at position $i+1$, where N is the Markov order of the MARKOV algorithm producing M. The algorithm specifically includes production paths for all sequences of length N or less, so S_i^N and S_{i+1}^N are necessarily produced by M. Thus, S_i^{N+1}, by the union of the two above, is also produced by M. Induction proves that all of S is then producible by M.

MARKOV is more robust in the presence of noise in the event stream than the KTAIL method. Moreover, the level of this robustness can be easily controlled by the process engineer through the probability threshold parameters. These parameters can also be used to control the complexity of the discovered model from large amounts of data, ignoring low-probability sequences, even if they are not the result of noise.

5.4 "Goodness" of a Model

In general, for a given set of sample inputs, there are an infinite number of possible models that can be constructed. Our goal, of course, is to construct a good model. But how do we do this? To answer the question, we must first understand the meaning of "good".

Theoreticians consider an FSM to be good if it is both *accurate* and *minimal*. By accurate they mean that the FSM accepts all legal sentences in the language and rejects all illegal sentences. For our purposes, accuracy implies that we are modeling all and only the behaviors of the process. By minimal, theoreticians mean that the FSM contains the fewest number of states necessary.

Accuracy and minimality are, in a sense, in conflict. The simplest way to achieve accuracy is to create an independent path through the FSM for each sample input, which typically means a high degree of redundancy. The simplest way to achieve minimality, on the other hand, is to create a single state with self-transitions for each possible input token. As mentioned in Section 4, Gold [35] showed that both positive and negative samples are required to construct an accurate *and* minimal model. Moreover, the samples must be complete in that they fully cover the possible inputs.

In the domain of process discovery, where typically we have only positive samples and those samples represent only a proper subset of all possible behaviors, this definition of "good" is not particularly useful. Thus, for our purposes, a "good" result from a discovery method is one that:

- fully accounts for the sample behaviors it is given, subject to constraints on noise in the data;
- successfully identifies patterns made up of sequencing, selection, and iteration; and
- does not needlessly complicate the patterns identified with extra states and transitions.

In the discussion of the methods above we pointed out places where these criteria are satisfied and where they are violated.

Our approach to compensating for deficiencies in the basic methods is to provide tuning parameters to the process engineer. In addition, the tool that implements process discovery, described in

Section 7, not only makes it possible to apply the methods and control them through parameter settings, but also makes it possible for the process engineer to refine the model through operations that merge, split, create, and delete states and transitions. In our experience with the process discovery tool, one instance of which is described in Section 8, we have found that the combination of parameters and manual refinement are a straightforward means to develop reasonable process models.

A final point concerning the goodness of a model has to do with the significance of the discovered states. A state arises simply from the commonality of events causing the entering and exiting of the state. It is left to the process engineer to interpret the meaning of the state within that context. In doing this, it would be useful for the process engineer to try to relate the states in the discovered model to, say, the states of various artifacts as they are manipulated in the process. Such a correlation may provide further insight into the nature of the process.

5.5 Concurrency

Software processes typically involve concurrent behavior, arising from one or more agents performing multiple simultaneous activities. Our representation of the behavior as event streams imposes an artificial ordering of the events stemming from independent activities. Moreover, our methods infer an FSM model, which is inherently sequential. How, then, can we deal with concurrency in software processes?

The critical issue comes down to finding a way to separate out the causally unrelated events in the event stream. Once that is done, we can account for the concurrency by extending the model to, for example, communicating finite state machines [14].

One approach is to make use of event attributes, such as the identifier of the agent and/or artifact associated with an event. We could preprocess an event stream to separate it into substreams involving particular agents or artifacts and then apply the methods to those individual substreams.

Another approach is to view concurrency as randomly interleaved sets of event streams. The concurrency then manifests itself as "noise", in the sense that the sequencing of the events will contain some randomness. At a boundary point between the concurrent streams, the concurrency may result in low-probability sequences. Given proper settings of thresholds in the algorithms, these sequences will be ignored. The resulting FSM may even be disjoint, actually consisting of separate state machines for each concurrent agent. If one interprets each disjoint machine as having its own thread of control, then the algorithm has, in fact, discovered concurrent processes from the data.

Neither of these approaches has been fully investigated or implemented. These and other approaches are subjects of future work. Yet despite this shortcoming in our current technique, the discovery tool proved useful when applied in the industrial case study described in Section 8.

6 Evaluation of the Discovery Methods

In this section we briefly compare and evaluate the three discovery methods with respect to several important characteristics, described below. Our evaluation is based on experiments that were conducted on a benchmark developed by the software process community referred to as the ISPW 6/7 process problem [41]. The process involves a software modification and testing activity, and is illustrated in Figure 8. Details of the experiments are described elsewhere [22, 25]. Here, we summarize the results (see Table 2).

We compared and evaluated the methods against the following characteristics.

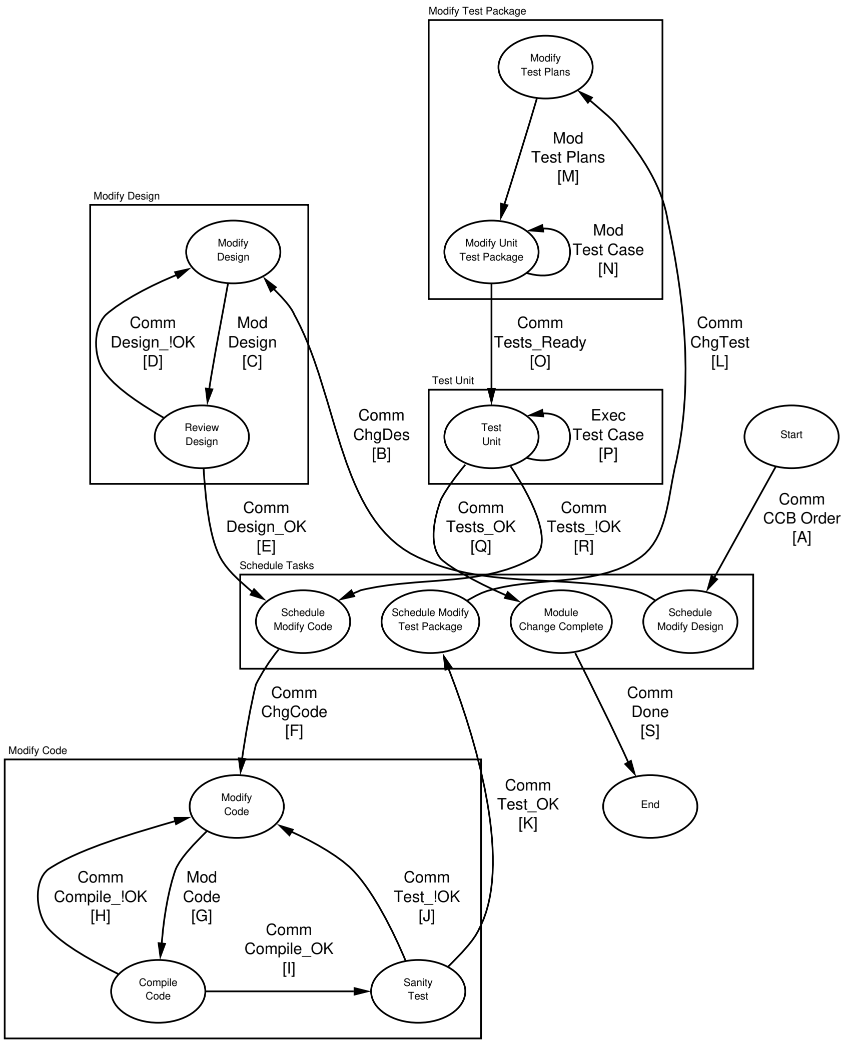


Figure 8: ISPW 6/7 Benchmark Process.

Accuracy. Both Ktail and Markov are guaranteed to generate an accurate model for the data they are given, if their noise thresholds are set to zero. If their noise thresholds are set above zero, then this guarantee does not hold. RNet, being purely statistical, cannot in any case be configured to satisfy this guarantee.

Speed. The numbers in Table 2 are approximate, intended to show relative speeds of the methods. 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 reasonable initial settings. Ktail and Markov, being only tunable in a few dimensions, might be overly restrictive, but using them is straightforward.

Feature	KTAIL	MARKOV	RNET
Accuracy	possible	possible	unlikely
Speed	10^1 sec	10^1 sec	10^3 sec
Tunability	two params	few params	many params
Usability	easiest	easy	hard
Robustness to Noise	moderate	good	good

Table 2: Comparison of Discovery Methods.

Length of Event Stream	MARKOV		KTAIL	
	Time (s)	Size (KB)	Time (s)	Size (KB)
312	4.61	670	1.07	386
625	4.74	701	2.10	432
1250	5.07	780	5.26	533
2501	5.62	917	16.30	736

Table 3: Time and Space Requirements Versus Event Stream Lengths.

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 some ability, in its equivalence class size threshold, but we regard it as less configurable to handle noisy data streams.

Based on this comparison, MARKOV and KTAIL are seen to be superior to RNET. We next take a closer look at the performance characteristics of MARKOV and KTAIL.

Table 3 shows the performance of MARKOV and KTAIL over different event stream lengths, but with the same number of event types. Both show a linear time relationship to the stream length, but MARKOV's coefficient is so low that the increase in running time is almost negligible, whereas KTAIL shows significant increase in running time. Still, both of the methods show good performance. As for space usage, both increase slightly for longer stream lengths, yet neither shows extreme space requirements.

Table 4 shows the performance of MARKOV and KTAIL over different numbers of event types, keeping the stream length constant at 625 events. MARKOV displays non-linear time and space relationships to the number of event types processed, while KTAIL performs much better.

It is evident from tables 3 and 4 that KTAIL is affected by event stream length, while MARKOV

Number of Event Types	MARKOV		KTAIL	
	Time (s)	Size (KB)	Time (s)	Size (KB)
20	1.03	432	1.51	426
40	4.74	701	2.10	432
60	13.90	1304	2.24	433

Table 4: Time and Space Requirements Versus Number of Event Types.

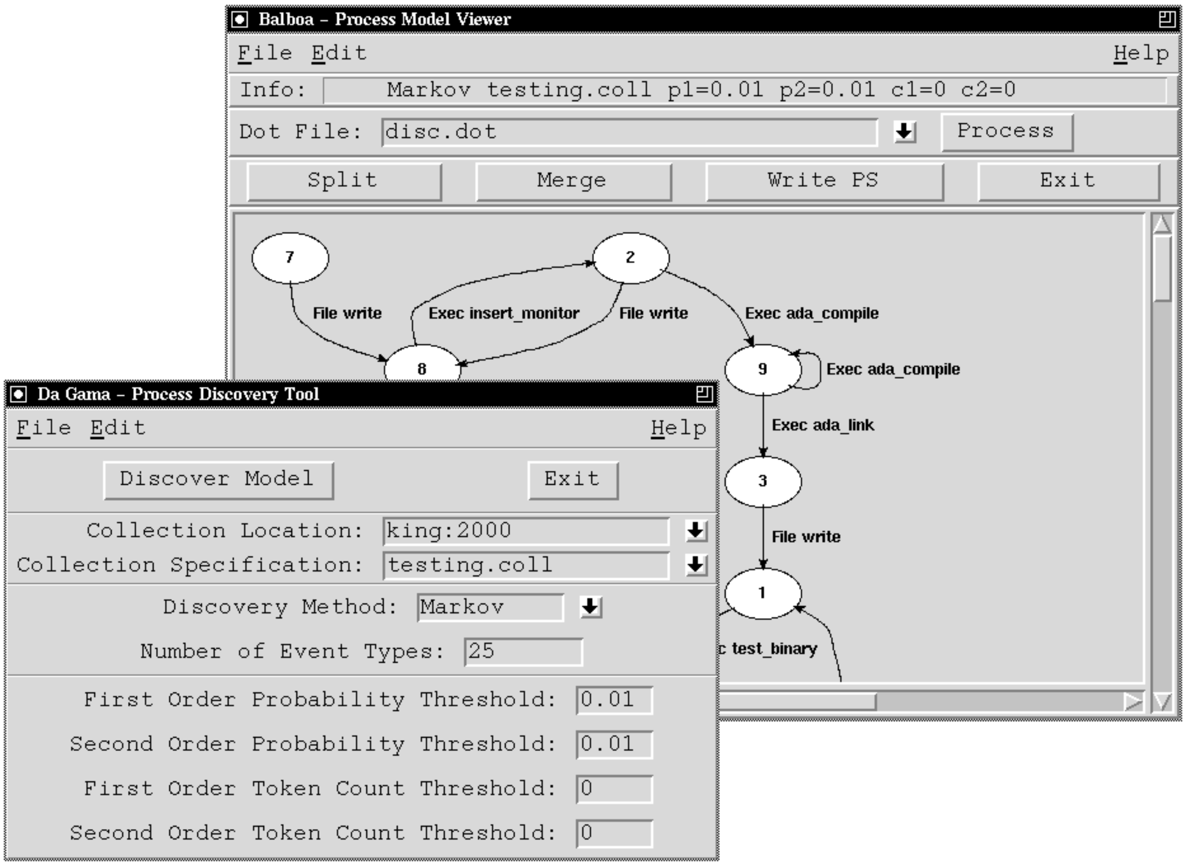


Figure 9: User Interface of DaGama.

is affected by the number of event types. This is what we would expect, since Ktail processes each event position in the stream as being in some equivalence class, and Markov process tables of probabilities of event type sequences.

7 DaGama: The Discovery Tool

The methods described in this paper have been implemented in a tool called DaGama. DaGama is fit into the Balboa data analysis framework [22], which provides standard data and user interface management capabilities. RNet was adapted from an implementation by Das and Mozer [27], while Markov and Ktail were implemented especially for DaGama. All three methods produce FSMs in a format compatible with the *dot* graph layout system [42]. Thus, DaGama provides an end-to-end solution, from event data input to a visual display of the discovered process model.

The user interface of DaGama is shown in Figure 9. It allows one to select event streams, choose a discovery method, specify the method's parameters, and run the methods to produce discovered process models. The discovered model is displayed in a Balboa process model viewer. The model can then be edited using operations that include merging nodes, creating/deleting nodes, creating/deleting edges, naming/renaming nodes, and splitting nodes.

The discovered models can also be saved to a file for use by other tools. One such tool that can read BALBOA process model files is Endeavors, which is a process model design and execution tool developed by a research group from the University of California at Irvine [12].

8 Employing the Discovery Methods in Practice

The DAGAMA discovery tool was used in an industrial process case study conducted at AT&T Bell Laboratories (now Lucent Technologies). This section reviews the use of the discovery methods in that case study, which provides real-world evidence for their utility. The full details of the study, its results, and its significance are presented elsewhere [23].

8.1 The Overall Study

The study focused on a change request process for a large telecommunications software system. In the process, a customer reports a problem (resulting in the creation of a so-called CAROD ticket to track the problem), the problem is assigned to a developer (resulting in a modification request tracked by a so-called MR), the developer performs and tests a fix, and the fix is sent out to the customer. Although the prescribed process was documented by the organization, it was not strictly enforced. Thus, the actual process executions were highly variable, even though there was some notion of an abstract model to be followed. It was, fundamentally, a *real*, industrial, human-based process.

An execution of the process that resulted in the fix being accepted by the customer was considered a successful execution of the process, while an execution resulting in the fix being rejected by the customer was considered an unsuccessful execution. The goal of the study was to statistically identify specific process behaviors that correlated with the acceptance or rejection of the fix and, thus, the success or failure of the process. By gathering event data on the process from historical data sources (tool logs, version control logs, databases, and the like), we were able to reconstruct event streams for 159 executions of the process, 18 of which resulted in a rejected fix. Each event stream consisted of approximately 32 events. Product and non-event-based process data were also collected, and were measured for correlation with the outcome of the fix.

The correlation of behaviors was performed using a *process validation* technique [24]. The technique compares event data collected from a process execution to a formal model of the process in an effort to measure the deviation of actual process executions from the intended execution.

The correlation measurements were first performed using a process model derived from the documentation of the prescribed process. It was shown that this model did not capture even half of the behavior that was being exhibited in the process executions. Thus, we applied DAGAMA in an effort to construct a model that better represented the true behavior of the process. In essence, DAGAMA was used to elicit a "canonical" behavior from the data, against which deviations of particular executions could be measured. That this was just one step in the study reflects the nature of the knowledge discovery process—discovery of a model (i.e., data mining) is a single step in the larger process of understanding the real world through data analysis [31].

8.2 Discovering a Process Model

Although each individual process event stream was relatively small, the number of streams and their variability made it practically impossible to construct a model by hand that would have represented the behavior seen in the data. Furthermore, a model that describes all of the behavior seen in the executions is not practical either, since it then is not an abstraction of the process, and

Measure	P-value (2-tailed)	Sig Test (W)	Accept Pop.(N=141) Mean	Accept Pop.(N=141) Std Dev	Reject Pop.(N=18) Mean	Reject Pop.(N=18) Std Dev
SSD	0.41	0.82	0.56	0.22	0.61	0.23
NSD	0.00	3.65	24.49	83.90	59.96	95.82
Matched Events	0.79	0.27	21.15	16.25	21.39	8.55
Missed Events	0.10	1.63	7.18	2.90	8.28	3.16
Extra Events	0.27	1.11	7.71	5.74	10.17	7.77
Blocks of Missed Events	0.49	-0.68	3.48	1.51	3.22	1.52
Blocks of Extra Events	0.74	-0.34	4.20	1.91	4.22	2.34

Table 5: Correspondence Metrics for the Discovered Process Model.

would be so large as to be incomprehensible. Thus, the task for DaGama in this study was to find the general patterns of behavior entrenched within the data as a sound starting point for a process engineer to construct an accurate and useful model.

Figure 10 shows the model and model fragments that Markov produced using two different threshold settings. The low threshold setting, with probability cutoffs at 0.2 and sequence count cutoffs at 3, gives more detail than is desirable, as shown in Figure 10a. Conversely, the high threshold setting, with probability cutoffs at 0.4 and sequence count cutoffs at 5, results in just the major recurring patterns in the data, as shown in Figure 10b. Notice that in the high threshold result, the patterns are disjoint—the patterns that Markov found do not have any direct event occurrence to relate them. This is not an error, but simply how the process behaves.

From these model fragments, a complete model was produced by merging, deleting, and modifying some of the states. This was done by hand (using DaGama), and underscores the point that the discovery techniques alone will not automatically create a completed process model, but provide a good starting point for the critical role played by the process engineer. Figure 11 shows the final model that was produced using the initial model provided by DaGama as a basis.

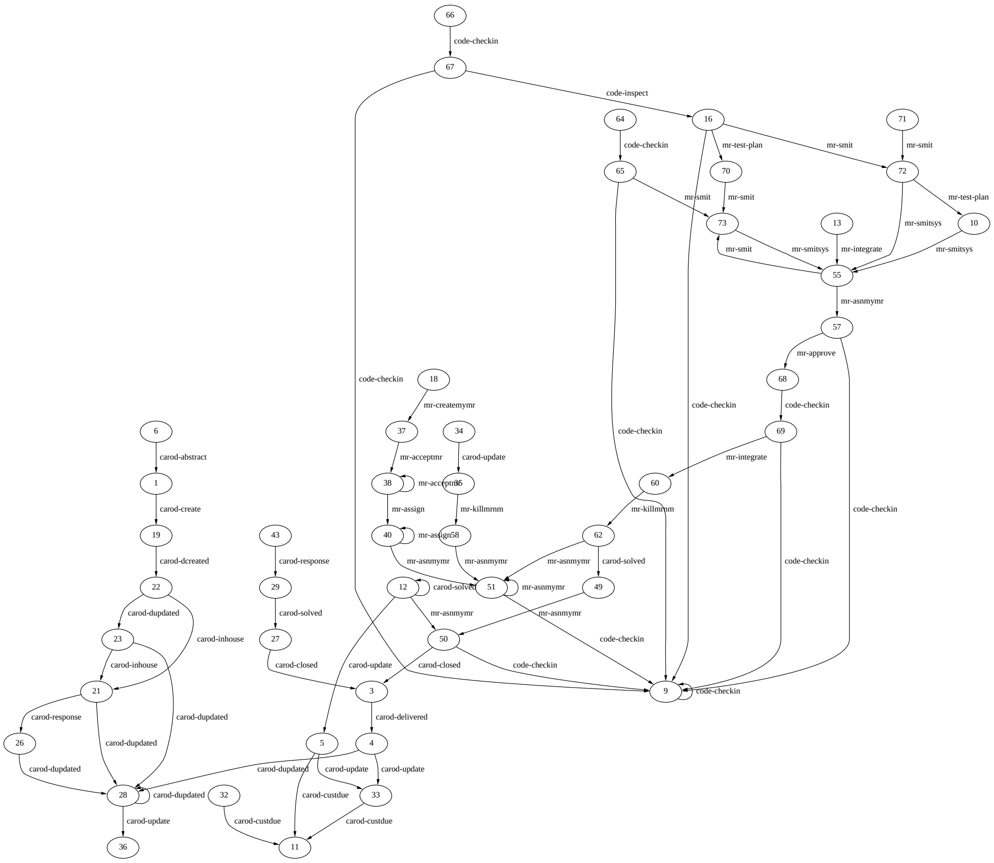
The final model has 27 states, where those in Figure 10 have 47 and 41, respectively. Thus, some of the behavior was merged together and simplified. For example, state 6 in Figure 10b has a transition to itself labeled with the code-checkin event, indicating that more than one code-checkin can occur in a row. Because of the variability of the data, Markov left some states separate from this one, but still having code-checkin events with a transition to state 6. Thus, these states can be merged together without changing the behavior of the model, in terms of accepting one or more code-checkin events in a row.

Note that depending on how the process engineer decides to compose the final model, the model may violate patterns found in the data. It is the process engineer's job to create a model that abstracts away insignificant dependencies, and that reflects the important behaviors of the process.

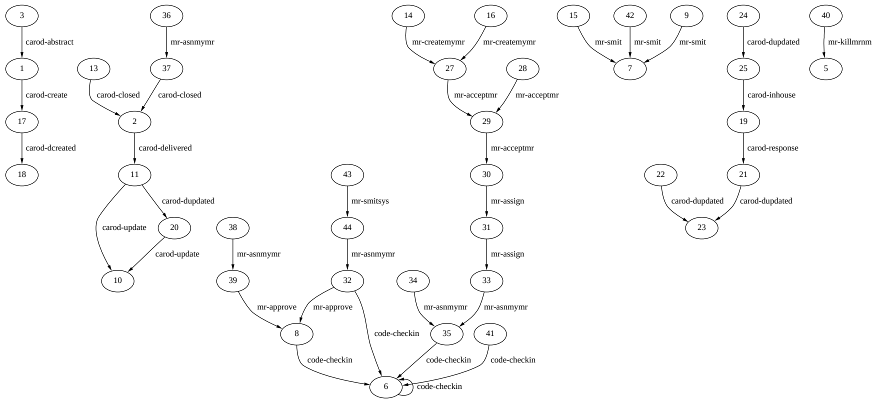
8.3 Measurements Using the Discovered Model

This final model was then used to perform the process validation [24], and to correlate process behavior with the outcome of the process, namely an accepted or rejected fix. Table 5 shows the measurements and statistical tests for the discovered model.[5] One aggregate validation metric,

[5]In the table, SSD and NSD are the names of two aggregate validation metrics. Matches are where events in the process exactly matched the model. Missed events and extra events are where the execution and model deviate. Blocks indicate runs of missed or extra events, and are measured because they may indicate a serious period of



(a)



(b)

Figure 10: Process Models Discovered using Low Thresholds (a) and High Thresholds (b) in the MARKOV Method.

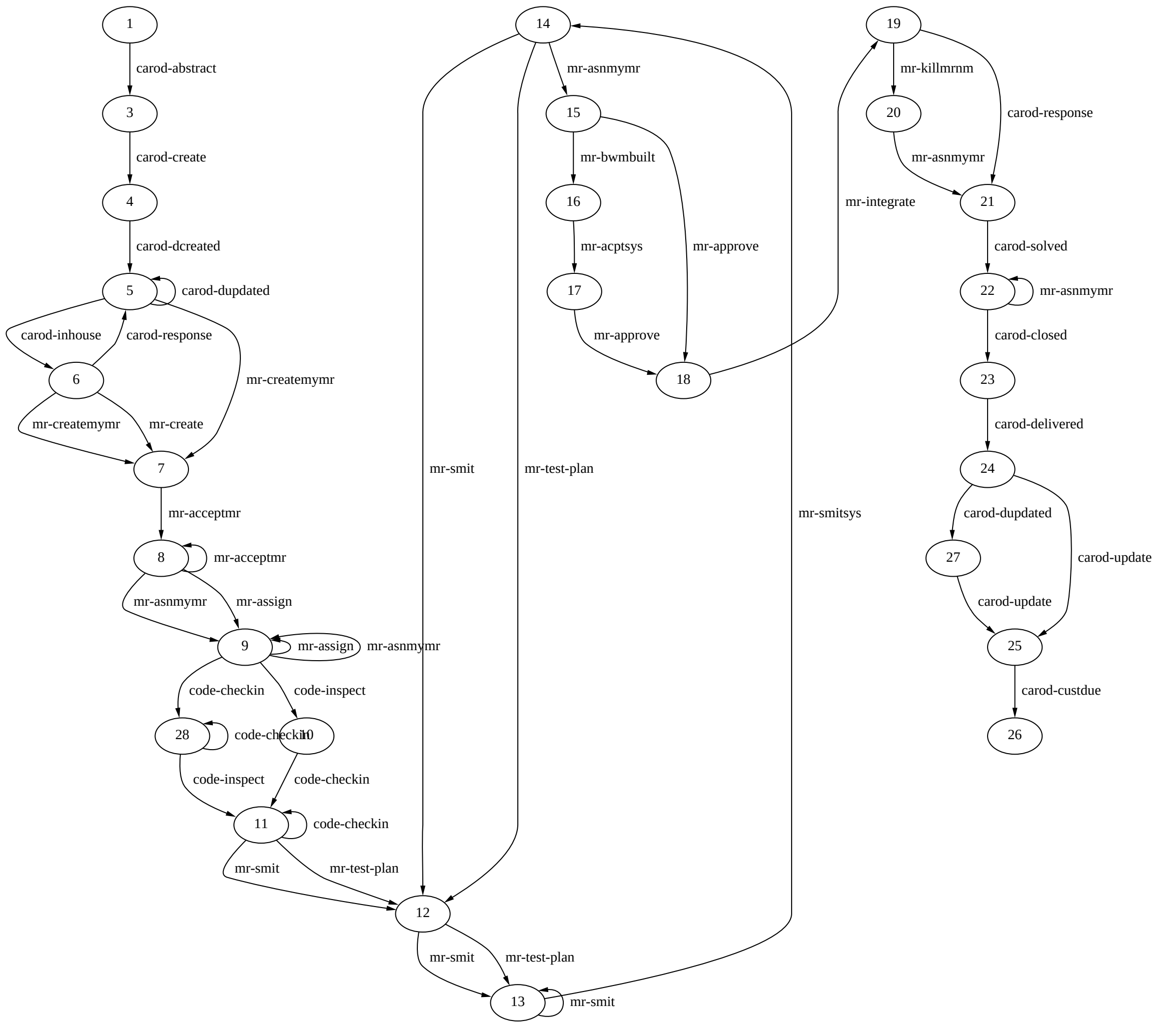


Figure 11: Final Process Model after Refinement.

NSD, measures a very significant difference between the level of correspondence between the discovered model and the accepted-fix and rejected-fix populations. Moreover, it indicates that the accepted-fix population matches the model more closely, leading one to suspect that the deviations were inappropriate.

It was also found that the discovered model reflected a much greater amount of the process behavior than the prescribed process model documented by the organization—about 65%, compared to 40% for the prescribed model. More importantly, the discovered model was useful in showing the areas of behavior that were statistically different between the successful and unsuccessful process executions. This seems to indicate that the model captured something important about the process.

Overall, DaGama and its discovery techniques proved their applicability to real-world process data by contributing to the formulation of a process model that usefully abstracted the actual process executions and captured important properties of the process behavior.

deviation.

9 Conclusion

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.

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.

While we have focused here on the use of these techniques for generating formal models, we also believe that they are useful in visualizing the data collected on a process. An engineer simply may be interested in a way to better understand the current process, as captured by the event data. Discovering patterns of behavior, such as those represented by a finite state machine, may be of help.

We are unaware of any work, other than our own, aimed specifically at investigating techniques for discovering process models from process execution data. Nevertheless, there are three efforts, besides the general data mining work mentioned in Section 3.1, that are at least somewhat related.

1. Garg and Bhansali [32] describe a method using 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 showed that a complete and correct process fragment could be generated from execution data.

2. Garg et al. [33] employ process history analysis, mostly human-centered data validation and analysis, in the context of a meta-process for creating and validating domain specific process and software kits. This work is more along the lines of a process post-mortem, to analyze by discussion the changes that a process should undergo for the next cycle.

3. Madhavji et al. [29] describe a method for eliciting process models from currently executing processes. The method is basically a plan for how a person would enter a development organization, understand their process, and describe the process in a formal way. There is no notion of automation or tool support.

There are several directions that future work in process discovery might take, including the following.

- The discovery methods could be seeded with partial information about a model derived from documentation, rather than data. This would take advantage of a process engineer's existing knowledge of a process. Both KTAIL and MARKOV would seem to naturally allow this, just by initializing them with equivalence classes and probabilities, respectively. A related issue to seeding is making the methods interactive, allowing the user to dynamically control the result.

- The techniques for accommodating concurrency in event streams, discussed in Section 5.5, could be implemented and evaluated.

- Extension of the discovery methods to infer models other than FSMs could be explored. Work in the area of grammar inference has involved the learning of more powerful grammars, such as those for context-free languages. But such models do not naturally fit software process. Petri nets or rule bases are a more appropriate direction for the software process domain.
- Small improvements to the visualizations of the process data created by the discovery tool could significantly help the process engineer better understand a process. For example, one could draw transitions with a thickness relative to their probability in the MARKOV-discovered model, or draw states with a shading relative to the size of the equivalence class in KTAIL.
- The parameters to the methods could be more fully studied to provide the process engineer with assistance in selecting parameter values. This would reduce the trial-and-error iteration that the process engineer must go through to apply the discovery methods. We have implemented a graphical viewer of the probabilities in the MARKOV method to help guide parameter selection, and a similar tool would be useful for KTAIL.

We intend to explore these and other areas, and to continue improving the practicality and usefulness of these process discovery methods.

Acknowledgments We appreciate the many helpful comments on this work provided by Clarence (Skip) Ellis, Dennis Heimbigner, David Rosenblum, Lawrence Votta, and Benjamin Zorn, as well as the anonymous reviewers of this paper.

REFERENCES

[1] H. Ahonen, K. Mannila, and E. Nikunen. *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 153–167. Springer-Verlag, New York, 1994.

[2] D. Angluin. Inductive Inference of Formal Languages from Positive Data. *Information and Control*, 45:117–135, 1980.

[3] D. Angluin. Inference of Reversible Languages. *Journal of the ACM*, 29(3):741–765, July 1982.

[4] D. Angluin. Learning Regular Sets from Queries and Counter-Examples. *Information and Computation*, 75:87–106, 1987.

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

[6] 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.

[7] 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.

[8] S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza. SPADE: An Environment for Software Process Analysis, Design, and Enactment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modeling and Technology*, pages 223–248. Wiley, 1994.

[9] 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.

[10] 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, January 1989.

[11] 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.

[12] G.A. Bolcer and R.N. Taylor. Endeavors: A Process System Integration Infrastructure. In *Proceedings of the Fourth International Conference on the Software Process*, pages 76–85. IEEE Computer Society, December 1996.

[13] M.G. Bradac, D.E. Perry, and L.G. Votta. Prototyping a Process Monitoring Experiment. *IEEE Transactions on Software Engineering*, 20(10):774–784, October 1994.

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

[15] A. Brāzma. *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 260–271. Springer-Verlag, New York, 1994.

[16] A. Brāzma and K. Čerāns. *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 76–90. Springer-Verlag, New York, 1994.

[17] A. Brāzma, I. Jonassen, I. Eidhammer, and D. Gilbert. Approaches to the Automatic Discovery of Patterns in Biosequences. Technical Report TCU/CS/1995/18, City University (London), December 1995.

[18] R.C. Carrasco and J. Oncina. *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 139–152. Springer-Verlag, New York, 1994.

[19] J. Carrol and D. Long. *Theory of Finite Automata*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[20] A. Castellanos, I. Galiano, and E. Vidal. *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 93–105. Springer-Verlag, New York, 1994.

[21] E.J. Chikofsky and J.H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[22] J.E. Cook. Process Discovery and Validation through Event-Data Analysis. Technical Report CU-CS-817-96, University of Colorado, University of Colorado, Boulder, Colorado, November 1996.

[23] J.E. Cook, L.G. Votta, and A.L. Wolf. A Methodology for Cost-Effective Analysis of In-Place Software Processes. Technical Report CU-CS-825-97, University of Colorado, University of Colorado, Boulder, Colorado, January 1997.

[24] 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.

[25] J.E. Cook and A.L. Wolf. Automating Process Discovery through Event-Data Analysis. In *Proceedings of the 17th International Conference on Software Engineering*, pages 73–82. Association for Computer Machinery, April 1995.

[26] 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, May 1993.

[27] S. Das and M.C. Mozer. A Unified Gradient-Descent/Clustering Architecture for Finite State Machine Induction. In *Proceedings of the 1993 Conference*, number 6 in Advances in Neural Information Processing Systems, pages 19–26. Morgan Kaufmann, 1994.

[28] 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.

[29] K. El Emam, N. Moukheiber, and N.H. Madhavji. An Evaluation of the G/Q/M Method. Technical Report MCGILL/SE-94-11, McGill University, 1994.

[30] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD Process for Extracting Useful Knowledge from Volumes of Data. *Communications of the ACM*, 39(11):27–34, November 1996.

[31] U. Fayyad and R. Uthurusamy. Data Mining and Knowledge Discovery in Databases. *Communications of the ACM*, 39(11):24–36, November 1996.

[32] 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 Press, May 1992.

[33] 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*, page [need pages], November 1993.

[34] E.M. Gold. Language Identification in the Limit. *Information and Control*, 10:447–474, 1967.

[35] E.M. Gold. Complexity of Automatic Identification from Given Data. *Information and Control*, 37:302–320, 1978.

[36] 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.

[37] 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.

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

[39] S. Jain and A. Sharma. *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 349–364. Springer-Verlag, New York, 1994.

[40] 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.

[41] 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.

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

[43] S. Lange, J. Nessel, and R. Wiehagen. *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 423–437. Springer-Verlag, New York, 1994.

[44] S. Lange and P. Watson. *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 438–452. Springer-Verlag, New York, 1994.

[45] 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.

[46] L. Miclet. *Syntactic and Structural Pattern Recognition: Theory and Applications*, volume 7 of *Series in Computer Science*, chapter 9: Grammatical Inference, pages 237–290. World Scientific, New Jersey, 1990.

[47] L. Miclet and J. Quinqueton. *Syntactic and Structural Pattern Recognition*, volume 45 of *NATO ASI Series F: Computer and Systems Sciences*, pages 153–171. Springer-Verlag, New York, 1988.

[48] 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.

[49] L. Pitt. *Analogical and Inductive Inference*, volume 397 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 18–44. Springer-Verlag, New York, 1989.

[50] 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.

[51] Y. Sakakibara. Efficient Learning of Context-Free Grammars from Positive Structural Examples. *Information and Computation*, 97:23–60, 1992.

[52] Y. Sakakibara. Grammatical Inference: An Old and New Paradigm. Technical Report ISIS-RR-95-9E, Institute for Social Information Science, September 1995.

[53] 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.

[54] L.G. Valiant. A Theory of the Learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[55] L.G. Votta and M. Zajac. Design Process Improvement Case Study Using Process Waiver Data. In *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, number 989 in Lecture Notes in Computer Science, pages 44–58. Springer-Verlag, September 1995.

[56] 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.

[57] Z. Zeng, R.M. Goodman, and P. Smyth. Learning Finite State Machines with Self-Clustering Recurrent Networks. *Nueral Computation*, 5:976–990, 1993.