Available online at www.sciencedirect.com

SCIENCE DIRECT®

Computers in Industry 53 (2004) 297–319

COMPUTERS IN INDUSTRY

www.elsevier.com/locate/compind

Discovering models of behavior for concurrent workflows

Jonathan E. Cook[a,*], Zhidian Du[a], Chongbing Liu[a], Alexander L. Wolf[b]

[a]*Department of Computer Science, New Mexico State University, Las Cruces, NM 88003 USA*
[b]*Department of Computer Science, University of Colorado, Boulder, CO 80309 USA*

Abstract

Understanding the dynamic behavior of a workflow is crucial for being able to modify, maintain, and improve it. A particularly difficult aspect of some behavior is concurrency. Automated techniques which seek to mine workflow data logs to discover information about the workflows must be able to handle the concurrency that manifests itself in the workflow executions. This paper presents techniques to discover patterns of concurrent behavior from traces of workflow events. The techniques are based on a probabilistic analysis of the event traces. Using metrics for the number, frequency, and regularity of event occurrences, a determination is made of the likely concurrent behavior being manifested by the system. Discovering this behavior can help a workflow designer better understand and improve the work processes they are managing.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Dynamic analysis; Workflow data analysis; Behavior inference; Concurrency analysis; Workflow model discovery

1. Introduction

Most workflow management systems can and do log the activities that occur in the executed workflows, and even non-workflow-specific computer-based systems generally can log activities that users perform. Thus within and without workflow systems, there is often a rich source of data that can be mined to learn something about the work processes that are occurring. Our research is centered around discovering behavioral models of the processes that capture the structured orderings of activities.

Even within a model-driven workflow system, the actual activities that occur might contain many exceptions to the model, or might occupy a subset of the model. Thus discovering a model can still offer a method for learning new aspects of how the work is actually being performed.

Our methods view the activity logs as a trace of events being produced by a black-box system. In previous work [1], we developed methods that use event traces to automatically discover a sequential model of behavior. For that purpose, an event trace is viewed as a sentence in some unknown language, and the discovery methods produce a grammar, in the form of a finite state machine, as a model of the language. Using the domain of finite state machines allows us to capture the basic structure of sequential processes: sequence, selection, and iteration.

Many workflows, however, exhibit concurrent behavior, where more than one thread of control is producing the events that comprise a single event trace. In such cases, the sequential state machine model cannot capture the true behavior of a system and a sequential discovery algorithm will discover overly complex models, as it tries to impose some

[*] Corresponding author. Tel.: +1-505-646-6243;
fax: +1-505-646-1002.
E-mail address: jcook@cs.nmsu.edu (J.E. Cook).

sequencing on events that can be interleaved in many different ways. Thus, we are motivated to discover concurrent behavior.

In this paper, we develop and demonstrate methods that can detect concurrent behavior in an event trace and infer a model that describes that concurrent behavior. The technique uses statistical and probabilistic analyses to determine when concurrent behavior is occurring, and what dependence relationships may exist among events. With this approach, we rely on an assumption of randomness in the event orderings resulting from the concurrent behavior. Our goal is not necessarily to reconstruct a model similar to what an engineer who understood the system would write down, but rather to identify gross patterns of behavior that can be useful in understanding the system. Indeed, our technique may be most valuable when revealing that the actual behavior does not match a preconceived notion of how the workflow should perform.

One might consider that the goal for discovery of a concurrent process should be to identify the individual threads and their individual behaviors. The first step of our work does just that. In a concurrent process, however, it is important to also locate those points where the threads interact. A workflow might be constructed having two threads, for example, but those threads may execute in lock step and actually not exhibit any concurrency at all. Thus, while an engineer might fairly quickly see from a specification the intended concurrency in a workflow, identifying the points of thread interaction and how much actual concurrent behavior is exhibited is not as straightforward.

The contributions of this paper are techniques for probabilistically identifying concurrent patterns of behavior in event traces. This paper extends and formalizes a preliminary presentation found in [2], and contributes a small extension to previous sequential (or individual thread) discovery techniques [1].

The next section provides definitions and background discussion to place the technique in context. Section 3 discusses techniques to discover individual thread behaviors where thread-specific events can already be identified. Section 4 discusses techniques to discover threads where no thread information is available to pre-associate events to threads. Section 5 details an example use of the technique and discusses the success of the methods on these examples. Finally, Section 7 concludes with some observations and some related work.

2. Background

In this section, we detail our view of events, concurrency, and dependencies among events that constrain concurrency. We also discuss several assumptions that underlie our work. Throughout, we use the term *system* to mean the whole workflow system, and the term *thread* to mean a sequential execution control path within the workflow, running concurrently with other threads.

2.1. Events

Following our previous work [1,3], we use an event-based model of system actions, where *events* are used to characterize the dynamic behavior of a system in terms of identifiable, instantaneous actions, such as sending a message, beginning an activity, or invoking a development tool. The use of events to characterize behavior is already widely accepted in diverse areas of software engineering, such as program visualization [4], concurrent-system analysis [5], and distributed debugging [6,7].

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 document review might have a "begin-review" and "end-review" event pair. Similarly, a work order submitted to a queue could be represented by the three events "enter-queue", "begin-job", and "end-job". In this work, we take an event-level view of behavior, and do not attempt to reason about higher level activities.

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, resources, and data associated with an event, the tangible results of the action (e.g., the decision on a loan application), 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 logical time, i.e., the ordering of events.

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

In general, we interested in discovering the patterns of behavior involving the event types. That is, for example, we would like to discover what pattern(s) all events of the type A are involved in. Thus, we will use ‘‘event A’’ as a shorthand to refer to ‘‘events of the type A’’, but sometimes the use of ‘‘event’’ will refer to a specific occurrence of the event type. The context will make the usage clear.

For a more detailed presentation of event data, systems that can collect event data, and using event data for analysis purposes, please see [8]. In addition, [9] demonstrates that event data can be available in an industrial workflow setting (software maintenance), even when they are not specifically collected as such.

2.2. A view of concurrency

A concurrent workflow has simultaneously executing threads of control, each producing events that end up in the resulting event trace. Thus, the event trace represents interleaved event sequences from all of the concurrent threads (assuming the threads cannot easily be captured as separate traces). These threads, since they are presumably cooperating to achieve a goal, are not totally independent. In addition to being created and destroyed, they will synchronize at certain points, and this will be reflected in the sequences of events produced.

In this paper, example models of concurrent systems are given in the familiar Petri net formalism [10]. The separate ‘‘threads’’ of execution in Petri net models are visible from the connectedness of the places and transitions. Events are produced at transition firings; the sequence of events produced by the system is exactly the sequence of transition firings. Fig. 1 shows a small concurrent system in the Petri net formalism where the ‘‘threads’’ are the single events A and B. Note that this and all other figures in this paper

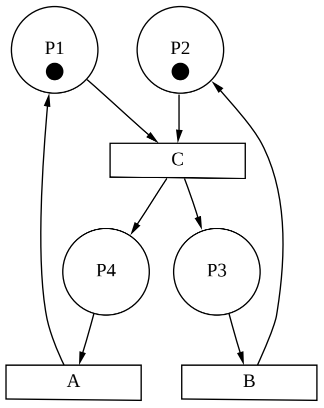

Fig. 1. An example Petri net.

were automatically drawn from textual model specifications using the graph layout tool *dot* [13].

2.3. Event classifications

The types of events that make up a trace of workflow execution can be quite varied, and will depend on the trace collection instrumentation that is available. We do not assume a certain class of events, but rather work with whatever is available. Most workflow systems can log events and activity, and even non-workflow computer applications can often generate a history of actions.

Clearly, we can only reverse engineer that part of the system behavior from which we can collect information, and thus the discovered abstractions will only reflect a view of the system as provided from the collected event trace. Nevertheless, being able to uncover interesting aspects of concurrent behavior from those views may be helpful to the engineers needing to understand the system.

2.4. Event dependencies

Discovering a workflow model from event data basically involves determining the logical dependencies among events. *Direct dependence* is defined as the occurrence of one event type directly depending (with some probability) on another event type. We define three types of direct dependence. *Sequential dependence* captures the sequencing of events, where one event directly follows another. *Conditional dependence* captures selection, or a choice of one event from a set of events potentially following a given event. *Concurrent dependence* captures concurrency in terms of "fork" and "join"; in a fork, all of a set of events follow a given event (not necessarily simultaneously, however), and in a join, a specific event follows a given set of events. A synchronization point, where two or more threads meet to coordinate, can be thought of as a join and a fork combined into the same instant.

While we use the terms fork and join, we do not mean to imply a particular concurrency construct. Regardless of the model of concurrency, the event trace will contain points where parallel threads synchronize—assuming the events in fact represent cooperating threads. Upon entering the synchronization point, the threads execute in lock-step, and are thus joined. Upon exit, they again freely execute in parallel, and are thus forked. If the synchronization point involves several sequential events, then the fork and join points bound those sequential events.

It is these direct dependencies that must be inferred in order to discover a model. For example, the iteration construct in the sequential case is built up of direct and conditional dependencies connected together in a cycle. Indirect dependencies arise from a transitive closure over direct dependencies, and so do not need to be discovered as separate dependencies. However, systems may have a higher order complexity than immediate event-to-event dependence. The occurrence of some event might depend on the previous two events rather than just the previous one. For example, a D may occur only when preceded by AB, and not by CB. Thus, we need to be able to infer these higher order (but still direct) dependencies.

If a time-spanning activity is represented by two or more instantaneous events, such as a begin-event/end-event pair, we know that those events represent activity boundaries. What this gives us is a dependency between the two events that is predefined and, therefore, does not require discovery. Nevertheless, the relationship between the events is not necessarily that of a direct dependence. An activity might be composed of several subactivities in sequence, each of which must complete before the activity is completed. At the event level, the end event of the activity is directly dependent on the end event of the last subactivity, since that ordering is always maintained. It is not directly dependent on the begin event of the whole activity, but rather it is indirectly dependent. In addition, an activity might fork several other threads, or simply have a synchronization point within it. In such cases, the end event will still not be directly dependent on the begin event. What this points out is that the dynamic relationships among events can be more complicated than the static relationships among events might imply.

With this in mind, we do not assume the presence of information denoting to which thread an event belongs. Such information may in fact be readily available, since many collection mechanisms tag each event with the thread to which it belongs. In deciding direct dependence, as explained above, this knowledge is not always useful, and may even be misleading at times. In this paper, we will demonstrate a technique to use this information, but we move beyond that and develop techniques that work without thread-identifying information.

2.5. Tabulation of event sequence characteristics

Given an event trace, some numerical representation of its characteristic sequencing behavior is needed upon which analysis can be performed. One of our successful sequential techniques, Markov, is based on a notion of frequency tables [1]. These tables record the frequencies at which each event and event sequence occur in the event trace. Along with frequencies, we also record the number of occurrences of each event type and sequence, up to a maximum sequence length. In this work, we begin with a similar representation, but add different analysis techniques.

As a working example, consider the system shown in Fig. 1. This system simply produces an event C, followed concurrently by events A and B, and repeats. A sample event trace from this system is

```
CABCBACABCABCBAC
```

We use S to represent a sequence that is a subsequence of the event trace and for which data is being tabulated. We define the sequence constructors

$$\text{Prefix}(S) = \text{sequence } S \text{ with the last event removed} \quad (1)$$

$$S : e = \text{sequence } S \text{ concatenated with event } e \quad (2)$$

which construct a sequence one shorter and one longer than S, respectively. These are related such that $\text{Prefix}(S:e) = S$. If the length of S is 1 (i.e., a single event), then $\text{Prefix}(S) = \text{null}$. As the basis for data tabulation, we start with a simple counting of sequences, and define

$$\text{Occur}(S) = \text{number of occurrences of sequence } S \tag{3}$$

For example, $\text{Occur}(\text{AC}) = 2$ in the above example. We define Occur(null) to be the total number of events in the event trace.

From this, we define the conditional probability of occurrence of a sequence as

$$\text{CondProb}(S) = \frac{\text{Occur}(S)}{\text{Occur}(\text{Prefix}(S))} \tag{4}$$

This is essentially the frequency of occurrence of the last event of S following the $\text{Prefix}(S)$. For example, if we look at the frequency of two-event sequences in the above example, we have the table

	A	B	C
A	0.0	0.6	0.4
B	0.4	0.0	0.6
C	0.5	0.33	0.0

where an entry is the frequency of the row event followed by the column event.

For example, $\text{Occur}(\text{BA}) = 2$ and $\text{Occur}(\text{B}) = 5$, and thus $\text{CondProb}(\text{BA}) = 0.4$. In other words, if a B occurs in the event trace, 40% of the time the next event is an A (60% of the time it is a C, and 0% of the time it is another B). These frequencies, then, are interpreted as the conditional probability that the second event will occur after the first event. This event trace is shown by its frequency table to be highly structured, since the values differ greatly in each table entry, and there are many 0 entries. Note that the frequencies given for the C event do not add up to 1 because the last C is followed by an ‘‘end-of-trace’’ event, which is not shown in the table.

For sake of conciseness, in the text we will just use $P(S)$ to indicate $\text{CondProb}(S)$; however, we will use $\text{CondProb}(S)$ in all formulas.

The non-zero frequencies directly represent probabilistic dependence relations—that is, the second event type depends on the first occurring, with some probability. Entries of 0 in the frequency table are interpreted as immediately signifying independence, though one can imagine scenarios where this might not be true. For instance, an event type might, for some reason, depend on the event preceding it by two rather than on the immediate predecessor, but the locality of the frequency table would mask such an effect. We can also partially account for noise in the data by setting some threshold below which low-valued entries in the frequency table are treated as 0.

If we assume that this event trace is exactly correct (i.e., contains no noise) and derives from a sequential system, then every non-zero entry in the table would signify a correct event sequence, and thus a transition sequence in a discovered state machine. But when the system that produces the event trace is concurrent, some of the table entries indicate spurious or false dependencies. In the example, the AB and BA entries are not significant, since we can see from the system in Fig. 1 that A and B are produced independently. A large part of the concurrency discovery problem involves deciding which entries in the frequency tables are significant and which are not.

The example above shows a table for two-event sequences, but we extend this representation to longer sequences, e.g., for the three-event sequences beginning with A, we have the following table:

	A	B	C
AA	0.0	0.0	0.0
AB	0.0	0.0	1.0
AC	0.0	1.0	0.0

We again interpret these numbers as a conditional probability, this time defining it as the conditional probability of the last event following the preceding two events. In general, we view the frequency tables of N-length sequences as providing the (observed) conditional probability that the last event follows the preceding $N-1$ events. This table shows that increasing the sequence length to 3, at least for those sequences beginning with A, gives frequency values that offer exact predictability. Thus, using these higher-order tables can be important.

2.6. Assumptions

The work described in this paper represents an initial investigation of the problem that makes use of several simplifying assumptions.

First, underlying our approach is the assumption that related events will appear next to each other with a high enough frequency that inference can be made on this relation. This is similar to the *Markov* principle, which states that the current state (or, for a higher order Markov model, the last N states) is what determines the next step in behavior, along with the input. Essentially, we assume that behavior is, for the most part, locally determined.

It could certainly happen that a system might produce events that are directly dependent but that never appear contiguous in the event trace. This would occur, for instance, if a slow thread were mixed in with fast threads. Our current technique does not handle this situation, but this issue points up an area of future work.

With this locality assumption, we focus on creating descriptive models that are simpler than more powerful notations such as Petri Nets, but are concise visualizations of the concurrent relationships found in the event trace. Specifically, our tools output Moore (state-labeled) or Mealy (transition-labeled) state machines, but allowing a vector of active states rather than just a single state so that the state machine model is viewed as having multiple, concurrent threads. To create and destroy threads, some states are specified as special fork and join states. For a fork state, all transitions leaving the state are taken together. For a join state, all transitions entering the state are taken together.

For example, the system represented by the Petri net in Fig. 1 would have a discovered model as that shown in Fig. 2. Marking the C node with a box and the "(f,j)" annotation signifies that its input transitions join the threads into one (for the instant that it executes), and that its output transitions execute concurrently. Thus, it represents a synchronization point.

Although this representation may not be powerful enough to *prescribe* the system, it is useful for *describing* the system. The advantage of this visually simple representation is that it cleanly separates the behavior in terms of event sequencing, and clearly depicts where concurrency exists. This is very important when offering to the engineer a model that in fact may be wrong or incomplete in places—which any probabilistic automated method might do. We intend to give an engineer a descriptive model so that they can understand more deeply the actual behavior. The advantage of this simple representation becomes clear in the presentation of examples. This descriptive model can easily be transitioned into the foundation of a prescriptive model in a more powerful notation, such as Petri nets, UML activity charts, or statecharts [11].

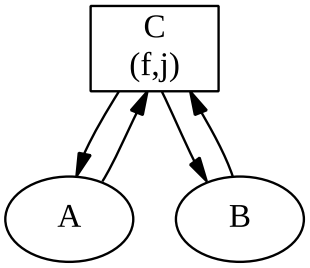

Fig. 2. An equivalent model to Fig. 1 using a fork/join node.

A second assumption is that the observed sequences of events will display randomness because of the fact that they are happening concurrently. This is the essential outcome of true, independent concurrency. Some concurrent systems, however, may not display much randomness, at least at the level of collectible events. Our technique is not targeted towards those kinds of systems.

A third assumption is that we have repeated presentations of system behavior, either in the form of a multiply executed loop or of multiple traces. Because we are looking at event sequence frequency, we need multiple occurrences of those events and event sequences to reliably interpret the frequency. For example, if both sequences AB and AC occur once, but the presence of AC in the trace is due to noise, the sequences will have the same frequency, but not the same validity. Thus, we need enough data to make the analysis meaningful.

How much data does our technique require? The statistical rule of thumb for using probabilities is that the number of observations of an occurrence should be at least five if the probability is to be used in some inference [12], although this minimal level is quite weak. That is, AB in the example above should occur at least five times if we are going to use its frequency in our analysis. If one assumed that only half of all possible two-event sequences occurred in a space of N event types, and that those all occurred equally, then $(5/2)N^2$ events should be collected before analysis is done. Of course, some event sequences will occur many times more than others, so this minimal lower bound is not realistic. It does, however, show that even the lower bound grows with the square of the number of event types when considering two-event sequences, and will grow in higher powers with longer sequences (although the occurring sequences tend to become sparser).

3. Techniques to discover individual threads

If our event data does contain information about the threads to which individual events belong, then we can extract out each individual thread's event trace (e.g., using *grep* or *awk*), and use that in a stand-alone fashion to infer that thread's behavior. This is certainly a useful start when such information exists, although it does not offer a complete picture of the concurrent system, because there may be intentional and also unintentional sequencing dependencies between the threads. In this section, we briefly present a method for inferring sequential behavior, most of which is detailed in [1].

The method starts with the frequency tables as discussed in Section 2.5. If we only process a single thread's event trace, then the nonzero frequencies directly represent actual correct event sequencing (assuming there was no noise in the data collection).

We first create a Moore type state machine with one state per event type, and then instantiate a transition between each two event states that have a direct dependence. At this point we have a model that captures the lowest possible behavior, and has only one state per event type. We need to refine the model to account for higher-order dependencies.

To do this we use the higher order frequency tables to infer which event states to *split*—that is, if the contexts of an event do not share similar subsequent behavior, then we need to split that event state into multiple states that each are consistent with those contexts. For example, if ABC and DBE occur in the event trace but ABE and DBC never occur, then the state for B needs split in two, one for an ABC path, and one for a DBE path.

Our previous work already did this refinement for three-event sequences. At this level, the model is refined (i.e., split) on the middle event, as the example above shows. We have since extended it to work with longer event sequences. We do this by first assuming that all shorter sequences have already been processed and model refinement on those has already taken place. If this is the case, then the next longer sequence only needs to refine the model on the next-to-last event in the longer sequences. In this way, the processing of the middle event in the three-event sequences is just a special case of always processing the next-to-last event in higher order event sequences. This generalized algorithm is shown in Fig. 3.

Our current tools allow the user to select the highest order of processing. In practice, the three-event order had already worked well, and it is rare that systems are complex enough to require orders higher than 5.

In summary, we have advanced techniques that can take sequential, single-thread event traces and infer a model of the behavior represented in the event trace. If thread identification is available on a collected event trace from a parallel system, this is a valuable first step towards understanding the behavior of the whole system.

4. Discovery of concurrent behavior

It will not always be possible to separate events from different threads, and even if it is possible, it may still be useful to try to find relationships between events from different threads.

Thus we now introduce a technique for discovering concurrent behavior. This section proceeds in a bottom-up fashion. That is, we first present four specific metrics that contribute key information to the task of discovering concurrency, and then present the framework in which these metrics are combined to discover complete models of the concurrent behavior shown in event streams. The metrics include: *entropy*, a measure of the amount of information a specific event type contains; *event type counts*, which are important in distinguishing sequential and concurrent behavior; *periodicity*, a measure of the regularity of occurrence of each event type; and a *causality* metric that distinguishes sequential dependence from concurrent inde-

```
Input: Moore-type FSM needing refined and the node for event E,
        Length N to do the splitting on
Output: Refined FSM with node E possibly split into multiple nodes
Description: divides all incoming paths to E into sets that produce
    equivalent output events, then creates new nodes for each of these
    sets and creates the necessary edges.
Auxiliary Functions:
        Occurs(Seq) is true if Seq occurs in the event trace,
                false otherwise;
        LastNode(Seq) is the node of the last event in sequence Seq.

foreach P in the (N-2)-length sequences to E
    foreach state T such that edge E→T exists
        if Occurs(P:E:T) in the event trace
            add T to OutSet(P)
        endif
    endfor
    if OutSet(P) does not exist in OutSets
        add OutSet(P) to OutSets
    endif
    add LastNode(P) to InPaths(OutSet(P))
endfor
foreach OS in OutSets
    create node V for event E
    foreach node T in OS
        add out edge V→T
    endfor
    foreach node T in InPaths(OS)
        add in edge T→V
    endfor
endfor
remove original node E
```

Fig. 3. The event-state splitting algorithm. This algorithm is executed for each node in the FSM, and for each processing length above 2.

pendence. An early formulation of these ideas was presented in [2].

In this section, we do not do higher-order processing similar to what was detailed in the previous section. Rather we assume a first-order view of the system, with each unique event type being a unique aspect of system behavior that is modeled in a single place in a discovered model. Thus, we do not attempt to ‘‘split’’ event types based on higher-order contexts. This is necessary because of the metrics we use to discover concurrency, which the previous section did not deal with.

As a working example, we use the process shown in Fig. 4. This process simply produces an event C, followed by concurrent events A, B, and F (after A), then a D. The process then repeats until, at the end, an event E is produced. A sample event stream from this process is the following.

```
CAFBDCBAFDCAFBDCAFBDCAFBDCAFB
  DCBAFDCBAFDE
```

For analysis purposes in this section, we used a longer event stream from an execution of this process, in particular one that is 1666 events long, generated from a stochastic simulation of the model.

4.1. Entropy

A key calculation that can be derived from the frequency tables is that of *entropy*, which gives a

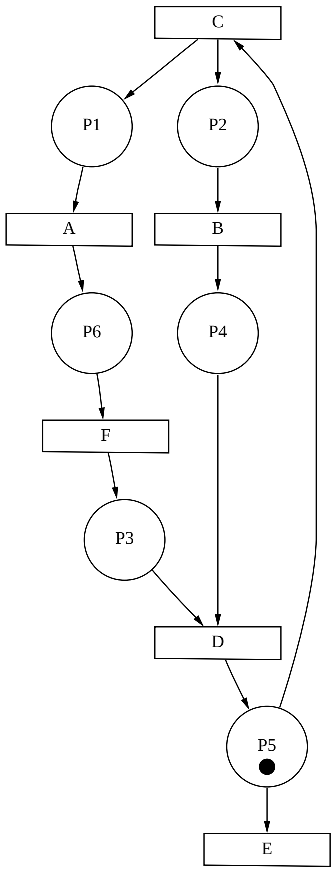

Fig. 4. A simple concurrent process modeled as a Petri net.

measure of the randomness of, or conversely the amount of information contained in, each event sequence and its occurrences. In essence, the entropy calculation tells us how the frequencies are distributed. If an event B always follows an event A, then the frequency $P(\mathrm{AB}) = 1.0$, and for all other events e, $P(\mathrm{A}e) = 0.0$. This means the behavior after A is perfectly deterministic, and thus the entropy is 0.0. As more event types occur after an A and the frequencies become more distributed, entropy increases until, if all event types are equally likely, the entropy is 1.0. An entropy of 0.0 means that we have complete information: when an A is seen, we know that the next event must be a B. On the other hand, an entropy of 1.0 means that we have no information: seeing an A gives no insight into the next event type.

The entropy of a sequence S is defined by the following standard formula:

$$\mathrm{Entropy}(S) = -\sum_{e\in E}\mathrm{CondProb}(S:e)\times\log_N(\mathrm{CondProb}(S:e)) \quad (5)$$

where E is the set of all event types, and $N = |E|$ (used in the log base).

One would think that concurrency could not be discovered, because it is represented by apparent randomness, or noise, in the event stream. But it is precisely this randomness that we can look for. We can measure the entropy for each event sequence.

If we assume a fork-style concurrent behavior, there are specific values of entropy that might signal a fork point. Take as an example the two-way fork shown in Fig. 4. If we had a perfectly balanced production order of the events beginning each branch of the fork (A and B), the frequency values for each will be about 0.5, and for the other four events will be 0.0. Thus, the row in the two-event frequency table will be

	A	B	C	D	E	F
C	0.5	0.5	0.0	0.0	0.0	0.0

since C is produced just before the fork, and A and B are produced first on each branch of the fork. The entropy, then, for C is 0.39. This value represents the asymptotic bound on a two-way fork for a process producing six event types. If a fork in this process is balanced, then with enough data the entropy for the fork should approach 0.39, and will never be greater than that in the absence of noise.

For a T-way fork given N event types, this entropy limit is given by the following simple formula (from a straightforward algebraic reduction using Eq. (5)).

$$\mathrm{EntLim}(T) = \log_N(T) \quad (6)$$

The closer an entropy value is to one of these limits, the more likely it is to be signifying a T-way behavior. Note that this same formula and metric can apply to joins as well, by simply viewing the event stream backwards. Although the reverse frequencies can easily be computed from the forward frequencies using Bayes' rule [12], so that extra tables are not needed, for efficiency reasons our implementation does indeed explicitly record the reverse metrics as well as the forward metrics.

Unfortunately, this metric alone is insufficient, because the reasoning above applies to sequential branching behavior as well. A two-way branch that is balanced in its production of events will have the same frequency values (and entropy) as a two-way fork. The following metric, however, can help distinguish between the two cases.

4.2. Event type counts

Given an event type that has several event types following it with various frequencies, a decision has to be made as to whether the behavior at this point is a sequential selection or a concurrent fork. An indication of which of these might be happening is a count of the events for the event types involved.

It will always be true that

$$\mathrm{Occur}(S) = \sum_{e\in E} \mathrm{Occur}(S:e) \tag{7}$$

but in this case we will look at the individual counts of e as well. Recall that we are assuming that for the most part, each event type is produced at a single point in the process. Thus, the individual event counts can be used to characterize that point.

If selection is occurring at some point in the process, then the counts of the event types following the selection point should sum to the count of the sequence at the selection point itself. For example, if A is followed by either a B or a C through a selection behavior, then the event counts might be, say, eight A's, five B's, and three C's. That is,

$$\mathrm{Occur}(S) = \sum_{e\in E, \mathrm{Occur}(S:e)>0} \mathrm{Occur}(e) \tag{8}$$

If, on the other hand, concurrency is occurring at that point, then the counts of the event types following the fork point should each be equal to the count of the event sequence at the fork point,

$$\forall(e \in E, \mathrm{Occur}(S:e) > 0),\quad \mathrm{Occur}(e) = \mathrm{Occur}(S) \tag{9}$$

and their sum should be a multiple of the count of the event at the fork point.

$$T \times \mathrm{Occur}(S) = \sum_{e\in E, \mathrm{Occur}(S:e)>0} \mathrm{Occur}(e) \tag{10}$$

for a T-way fork. Thus, in the example above, there would be eight A's, eight B's, and eight C's.

This reasoning assumes that forks are symmetric and synchronous, in the sense of all participating threads starting at a single point (though the overall model could have multiple and even nested fork points). In a system where, for example, worker threads are created asynchronously, this count differentiation would probably not help, and we would have to rely on some other mechanism to distinguish threads.

Even within a well-behaved system, these relationships do not have to always or exactly hold, since the event types involved might be producible through other paths in the model or there may be some noise in the event trace. Thus, their numbers will not always conform neatly to this scenario. However, the counts can indicate if the selection or fork is more likely to hold, since there would be a large difference in the expected event counts.

4.3. Deciding causality

The previous metrics were directed at discovering the synchronization points in a concurrent process. But with multiple threads, any two events produced concurrently may have spurious frequencies because by chance they happen to be produced near each other. Thus, given two events A and B, how can we decide when they are sequentially causally related and when they are not? Remember that in the non-concurrent case, two events might be sequentially dependent or conditionally dependent. This section deals with both identically, as sequential dependence is a special case of conditional dependence, where there is only one choice. In this discussion, we are assuming A and B have already been eliminated as events signaling forks, joins, or synchronization points.

If we do not see the sequences AB or BA (i.e., the probabilities are 0, or perhaps within some threshold of 0), then we can say that they are not directly dependent. But if we do have significant frequencies of these sequences occurring, then a decision needs to be made if they are related or not.

If we only see one of AB and BA occurring, then we can decide that there is a causal order from the first event to the second. However, if we see both sequences, then there are two possibilities: that the two events iterate in a simple two-event loop, or that they are independent and not causally related to each other, but are occurring in either order by chance. Recall that we are assuming only one event site per event type in the model, so these are the only two possibilities. There is a distinguishing relation in the frequencies $P(\mathrm{A})$ and $P(\mathrm{B})$ that separate these last two cases. If the sum $P(\mathrm{AB}) + P(\mathrm{BA})$ is greater than or equal to 1.5, then these two events are likely causally related in a two-event loop. If the sum $P(\mathrm{AB}) + P(\mathrm{BA})$ is less than 1.5, then they are likely independent.

The reasoning behind this is as follows. If A and B are part of a two-event loop, then the minimal sequences to see for recognizing the loop is *XABAY* or *XABABY*, where *X* and *Y* are other events, and assuming that the loop might exit from either end (seeing just AB is not enough to determine that a loop is present). For the first, the frequencies are $P(\mathrm{AB}) = 0.5$ and $P(\mathrm{BA}) = 1.0$, and for the second, $P(\mathrm{AB}) = 1.0$ and $P(\mathrm{BA}) = 0.5$. For any longer length sequences, the sum of these two frequencies will only increase, asymptotically approaching 2.0 as the endpoints play less and less of a role.

Of course, the possibility of concurrent production of other events from other threads during an AB loop and the possibility of other "noise" would act to reduce the sum $P(\mathrm{AB}) + P(\mathrm{BA})$, so that the sum could be less than 1.5 even if there truly is a loop. However, the 1.5 threshold is already minimal—we expect to have much more data than just a three or four event example of the loop—so a user-definable threshold parameter near 1.5 still serves to help deal with this situation, and we have seen good results with this threshold set between 1.3 and 1.5 in test cases.

If A and B are independent, then other event types will have non-zero frequencies from A and B, and even AA and BB may have non-zero frequencies, so that the likelihood of $P(\mathrm{AB}) + P(\mathrm{BA}) \geq 1.5$ occurring will be small. It is possible that A and B are independent, but that by chance the sequences occur such that the frequencies are at least 1.5, but if this is the case, then more data should eventually show them to be independent, or there might be some hidden constraint in the the system that is truly causing them to exhibit such behavior.

As an example, assume a model such as that in Fig. 1, where C is followed by concurrent (independent) A and B. In this model it is always true that $P(\mathrm{AB}) + P(\mathrm{BA}) = 1$, since they each occur once between every C. Thus, they cannot be mistaken for being dependent.

If we assume two concurrent single-event loops, one of A's and one of B's, this is the only mechanism that could accidently produce a mistaken dependence result. For this to occur, the timing of the two loops would have to be in a high degree of synchronization, and perhaps discovering a causal relation between them may actually reflect some implementation factor that is important to understand, such as an unforeseen resource constraint or timing effect.

4.4. Periodicity

With the periodicity metric we consider the repetitive behavior of a process and its event stream. Our probabilistic analysis, as mentioned above, depends on repeated presentations of the behavior of a system. In this repetition, an event sequence will have some *period* of occurring. Because of the other threads around it, this period may be very irregular or very regular. By looking at which event sequences have regular periods, we can identify the points in the process that are potential synchronization points, because these will be the most regular. Periodicity is a measurement, then, of the regularity of the period of occurrence for each event sequence.

Consider the example in Fig. 4. The events A, B, and F produced inside each of the threads will be a bit jumbled; their periods will not be regular. But the event C marking the fork and D marking the join will always have a period of 5, because all of the events in the threads will always occur.

If the threads have selection branches that produce differing numbers of events, or internal loops, then the synchronization points will not have an exactly regular period. But even so, their period should be the most regular—that is, the other events internal to the concurrent processes will also suffer from these differences, on top of the irregularity they already exhibit.

We first define the position of the *i*th occurrence of a sequence as

$$\mathrm{Position}(S, i) = \text{position of the last event, of the } i\text{th occurrence of } S \tag{11}$$

where each event in the event trace is numbered sequentially, beginning at 1. With this, then, the average period of a sequence is

$$\mathrm{PeriodMean}(S) = \frac{\sum_{i=2}^{\mathrm{Occur}(S)} \mathrm{Position}(S,i) - \mathrm{Position}(S,i-1)}{\mathrm{Occur}(S) - 1} \tag{12}$$

and

$$\mathrm{Dsq}(S,i) = (\mathrm{Position}(S,i) - \mathrm{Position}(S,i-1))^2$$

$$\mathrm{PeriodStdDev}(S) = \sqrt{\frac{\sum_{i=2}^{\mathrm{Occur}(S)} [\mathrm{Dsq}(S,i) - \mathrm{PeriodMean}(S)^2]}{\mathrm{Occur}(S) - 2}} \tag{13}$$

The standard deviations capture the regularity of the periods of the event types, and thus are the periodicity measurements. The means capture how long (in events produced) is the process within that period, and can be used as supplemental measurements. Those event types with the lowest standard deviations should be the event types that mark the synchronization points in the process.

4.5. *Putting the metrics together*

The separate metrics described in Sections 4.1–4.4 are combined into a single technique for discovering concurrency. The framework we use for combining the techniques to discover the true dependencies is one of *explaining* why occurrences of event types appear in the event stream. The goal is to find dependencies that explain as much of the event stream as possible, even all of it if we know that there is no noise present in the event stream.

Specifically, we keep track of two numbers: the number of occurrences that have already been explained by some inferred dependency, and the number of occurrences that have been used to explain some dependency. By keeping track of the number already explained, we know when to stop trying to explain some event type occurrence, and by keeping track of the number used to explain others, we know when to stop using an event type to explain others.

When a fork (or join in reverse) is inferred, the event f denoting the fork is marked as fully used, as $\mathrm{Used}(f) = \mathrm{Occur}(f)$, and each event e with a dependency from f is explained by the same amount, $\mathrm{Explained}(e) += \mathrm{Occur}(f)$. As noted before, we assume that an event marking a fork is not used anywhere else in the system. If one of the events e has many more occurrences from some other path in the system, those occurrences can still potentially be explained later in the discovery process.

When a sequential dependency from event d to event e is inferred, we only mark the used and explained based on the observed occurrence of that sequence: $\mathrm{Used}(d) += \mathrm{Occur}(d:e)$, $\mathrm{Explained}(e) += \mathrm{Occur}(d:e)$. In this way, if there are several conditional dependencies from an event, each one will be explained according to its proportion of occurrence.

In an event trace from a concurrent system, however, there will be many spurious event sequences due to the interleaving of the threads. In order to account for this, when an occurring sequence $d:e$ is *not* used to infer a dependency (i.e., it does not satisfy the metrics that have been detailed), we still perform the update $\mathrm{Used}(d) += \mathrm{Occur}(d:e)$. In this manner, we are acknowledging that those occurrences of d that are followed by e are being ignored.

The key in this approach is the order in which dependencies are inferred and explanations are created. A haphazard processing order would not infer the correct dependencies, but if we first infer the dependencies that are most likely to be correct, the rest will then hopefully "fall into place". In a probabilistic framework, this means that we first process those event types that have the most and best information in their values. In other words, we need to infer a ranking of the quality of information before we infer dependencies over that information.

Entropy is a direct measure of the "information" in a particular event type's sequences. But just looking at entropy and using it directly to rank the event types ignores the entropy limits for ideal branching factors, as discussed in Section 4.1. For example, an event A followed equally by events C and B would have a higher entropy than if C occurred 3/4 of the time and B only 1/4. But the first actually gives us better information because it exactly shows a balanced branch (or fork) of two, whereas in the second we cannot be sure if the branch is two but slightly unbalanced, or if the B occurrences are just spurious—due, for example, to another thread.

Thus, we find the branch entropy limit (Eq. (6)) that is closest to the actual entropy of a given event sequence (Eq. (5)), and use the difference between these two for ranking. This is defined as

$$\mathrm{EntDiff}(S,i) = |\mathrm{EntLim}(i) - \mathrm{Entropy}(S)| \tag{14}$$

$$\mathrm{BrFactor}(S) = k \quad \text{s.t. } \forall i,\ i \neq k,\ \mathrm{EntDiff}(S,k) \leq \mathrm{EntDiff}(S,i) \tag{15}$$

$$\mathrm{EntropyRank}(S) = N^{\mathrm{EntDiff}(S,\mathrm{BrFactor}(S))} \tag{16}$$

where EntDiff$(\cdot)$ is the entropy difference and BrFactor$(\cdot)$ is the branch entropy limit that is gives the minimal entropy difference. Sequences can then be ranked based on this value, as shown in the EntropyRank$(\cdot)$ definition; however, we calculate the rank as N (the number of unique event types) raised to its power, since the entropy calculation is a $\log_N$ calculation. This linearizes the ranking value.

This difference alone, however, is still not sufficient to rank the event types for processing. An entropy value might be very close to some branching factor entropy limit, but in actuality is derived from more sequences than the branching factor allows. For example, three event types might follow event A in such a way that the entropy measured is very close to the limit for a branching factor of two.

To distinguish these cases, we also calculate the total frequency of the k highest probability sequences, where k is the branching factor indicated by the chosen entropy limit. This is defined over a sorted list of the sequences:

$$\begin{aligned}\mathrm{SortedSeq}(S,i) = S:e \quad &\text{s.t. } e \in E \text{ and } \forall j, j < i,\\ &\mathrm{CondProb}(\mathrm{SortedSeq}(S,j)) \geq \mathrm{CondProb}(S:e)\\ &\text{and } \forall j, j > i,\ \mathrm{CondProb}(\mathrm{SortedSeq}(S,j))\\ &\leq \mathrm{CondProb}(S:e)\end{aligned} \tag{17}$$

Subtracting the k highest sequences from 1.0 then gives us the amount of event sequence frequency not accounted for by the branches allowed with the entropy limit:

$$\mathrm{ProbRank}(S) = 1 - \sum_{i=1}^{\mathrm{BrFactor}(S)} \mathrm{CondProb}(\mathrm{SortedSeq}(S,i)) \tag{18}$$

We also include the periodicity in the sequence ranking, by the following formula:

$$\mathrm{PeriodRank}(S) = \frac{\mathrm{PeriodStdDev}(S)}{\mathrm{MaxPeriodStdDev}(\mathrm{length}(S))} \tag{19}$$

which is simply the standard deviation of the period of S normalized by the maximum period standard deviation over all sequences of the same length.

The final ranking, then, is given as a combination of the entropy, unaccounted-for frequency, and periodicity rankings:

$$\mathrm{Rank}(S) = W_E \times \mathrm{EntropyRank}(S) + W_F \times \mathrm{ProbRank}(S) + W_P \times \mathrm{PeriodRank}(S) \tag{20}$$

In this ranking, sequences are processed from lowest rank value (best information) to highest rank value (worst information). The sum is weighted because the three components vary differently. The entropy differences are generally large compared to the frequency differences, and our experience is to put less preference on the periodicity. Also, since periodicity is largely unrelated to the other metrics and is useful by itself, we also print out the periodicity values separately for the user to inspect. Our experience to date has led us to use $W_E = 1$, $W_F = 1.5$, and $W_P = 0.25$ as coefficient weights, but more experimentation is needed to determine whether this should be a user-definable parameter, or if some other relationship would perform better over a different variety of data.

For an example of this ranking, assume events A through J occur in a trace of just these 10 event types, and that $P(\mathrm{AB}) = 0.38$, $P(\mathrm{AC}) = 0.62$, $P(\mathrm{BD}) = 0.75$, $P(\mathrm{BE}) = 0.17$, and $P(\mathrm{BF}) = 0.08$, and all other two-event sequences beginning with A or B do not occur. Furthermore, assume that PeriodStdDev(A) $= 1.2$, PeriodStdDev(B) $= 1.2$, and MaxPeriodStdDev(1) $=$ 4.5.

The entropy difference for the ideal two-branch limit would be $\log_{10}(2) = 0.693$. Event A's entropy is 0.664, for a difference from the ideal two-branch limit of 0.029, and an EntropyRank of 1.07. The unaccounted-for frequency (i.e., ProbRank) would be $1 - (0.38 + 0.62) = 0$, and the PeriodRank would be $1.2/4.5 = 0.267$. Thus, after applying the weights given above, the overall Rank(A) is 1.14.

Given the example frequencies for B, the entropy for B (0.719) is still closest to the two-branch entropy, with a difference of 0.026. Thus, B is even closer to the two-branch limit than the event type A. But its unaccounted-for frequency with a branch of two is $1 - (0.75 + 0.17) = 0.08$, and with the same periodicity

```
Input: Conditional Probabilities, Entropies, and Periods of
        N-length sequences.
Output: Moore-style FSM of inferred model.
Description: Rank all events (once for forward and once for reverse) and
        then proceed to use and explain each event until all events
        occurrences are explained and used.


SEL ← list of forward and reverse event in order of increasing
        Rank (each event type occurs twice).
foreach E in SEL
    if E shows signs of a fork or join, mark it as such.
        (this uses the entropy, count, and periodicity
        metrics)
    Build list of inferred dependencies to or from E
        (depending on the direction of this ranking).
    foreach DE in E's inferred dependencies
        if (E can be used and DE needs to be explained (forward) or
            DE can be used and E needs to be explained (reverse))
            if ranking is forward
                record the dependency E→DE
                update count of explained occurrences of DE
                update count of used occurrences of E
            else (ranking is reverse)
                record the dependency DE→E
                update count of used occurrences of DE
                update count of explained occurrences of E
            endif
        else (no dependency is inferred)
            update count of used occurrences of E
        endif
    end
end
Output dependencies in graph form.
```

Fig. 5. The discovery algorithm.

as A, its overall Rank(B) is 1.25, so B would be ranked lower than A in terms of the quality of the information it gives. The ranking is done simultaneously for both the forward entropies and frequencies (the direct sequence occurrences in the stream) and for the reverse entropies and frequencies (the reverse sequences in the event stream). This ranking thus intermixes processing both the forward and reverse indications of dependence; in this manner we can use the best indications of dependence in either direction before we process the weaker indications of dependence.

The whole algorithm is shown in Fig. 5 in outline form. Each phrase describing some processing uses the detail of the metrics described above. The algorithm has been instantiated in a discovery tool that reads in files of events and produces a textual graph representation of the discovered model, which is visualized using the *dot* graph layout tool [13].

The discovery tool was given as input the 1666-event stream produced from a stochastic simulation of the model in Fig. 4. Table 1 shows the forward frequencies of the two-event sequences, along with the periodicity metric and the forward and reverse rankings for each event, which indicates the processing order of each event's information.

Table 1
Forward frequencies, periodicity deviation, and processing order (ranking) from a simulation of Fig. 4

	A	B	C	D	E	F	Pd	For	Rev
A	0.00	0.25	0.00	0.00	0.00	0.75	0.69	10	4
B	0.44	0.00	0.00	0.31	0.00	0.25	1.17	7	6
C	0.56	0.44	0.00	0.00	0.00	0.00	0.00	3	2
D	0.00	0.00	0.99	0.00	0.01	0.00	0.00	11	5
E	0.00	0.00	0.00	0.00	0.00	0.00	0.00	12	1
F	0.00	0.31	0.00	0.69	0.00	0.00	0.63	8	9

Event E is processed first using its reverse information. This is because all of its occurrences (just one) are exactly preceded by D, giving no entropy and perfect information. Note that E's forward information is processed last, because it essentially has no information since nothing succeeds it in the event stream. Event C is processed next, first in reverse and then in forward. C is almost always preceded by D (except for its first occurrence), and so the reverse ranking is very high. C's successors (A and B) are very closely balanced in their occurrences (0.56 and 0.44), so this results in an entropy ranking very close to the two-way branch limit, and thus C's forward ranking is also very high, and the two-way fork from C is inferred nicely. All other events begin to be a bit jumbled as we step down the ranking; note, for example, that B and F are both processed relatively late; this is because they are the two events that display the most nondeterminism in their orderings. The resulting discovered model is shown in Fig. 6, with the thicker arrows being stronger dependencies (they are inferred first). The event types C and D are found by the entropy metric to be fork/join events and by the periodicity metric to be synchronization points. The other event types are successfully separated into their correct dependence relations based on their rankings, and the causality metric.

The next section demonstrates the application of our technique on a larger workflow example.

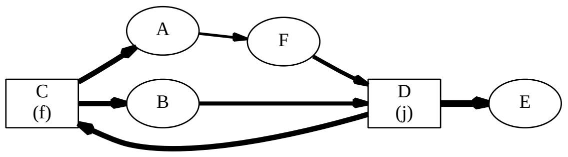

Fig. 6. The discovered model of the process in Fig. 4.

5. Example

Consider as an example the workflow process of reviewing a conference paper, with a Petri net model such as that shown in Fig. 7. In this process, an author submits a paper along with their contact information, a program chair then assigns the paper to three reviewers. The reviewers then each obtain the paper and submit their reviews. In our version, the reviewers can remotely collaborate rather than attend a PC meeting, so they read each others' reviews and submit comments, until they are satisfied and are done. The program chair then reads all the reviews and makes a decision as to whether the paper should be accepted or rejected.

With a Petri net simulator we generated event traces of this workflow process, such that we had 77,831 events representing 2591 executions of this workflow. We then applied our concurrency discovery tool to this data, and produced the discovered model shown in Fig. 8.

The discovered model closely matches the model used to generate the trace, thus validating the discovery method's ability to find structure in an event trace. It did mark as forks and joins the correct events, and cleanly separated the three concurrent sub-processes. It weakly inferred an erroneous dependency between reviewer 3's "review finished" and reviewer 2's "read reviews", and did not infer a dependency between "add comments" and "review finished" in reviewer 3's sub-process, but the overall model that is discovered is highly accurate, most noticable with the correct fork and join points.

However, the original model is missing some important control in the three reviewer sub-processes. The three reveiwers should not be able to read other reviews until the other reviewers submit them. Thus, perhaps this example was "too easy" in the sense that the three threads could display completely independent behavior, and thus the concurrent interleavings would be easy to detect.

Thus, we augmented the original workflow model with synchronization controls between the threads, with the new model shown in Fig. 9. In this Petri net model, each reviewer's submit action marks a synchronization place with two tokens, so that the other reviewers' threads will wait until the synchronization places are marked, and then proceed with reading the other reviews.

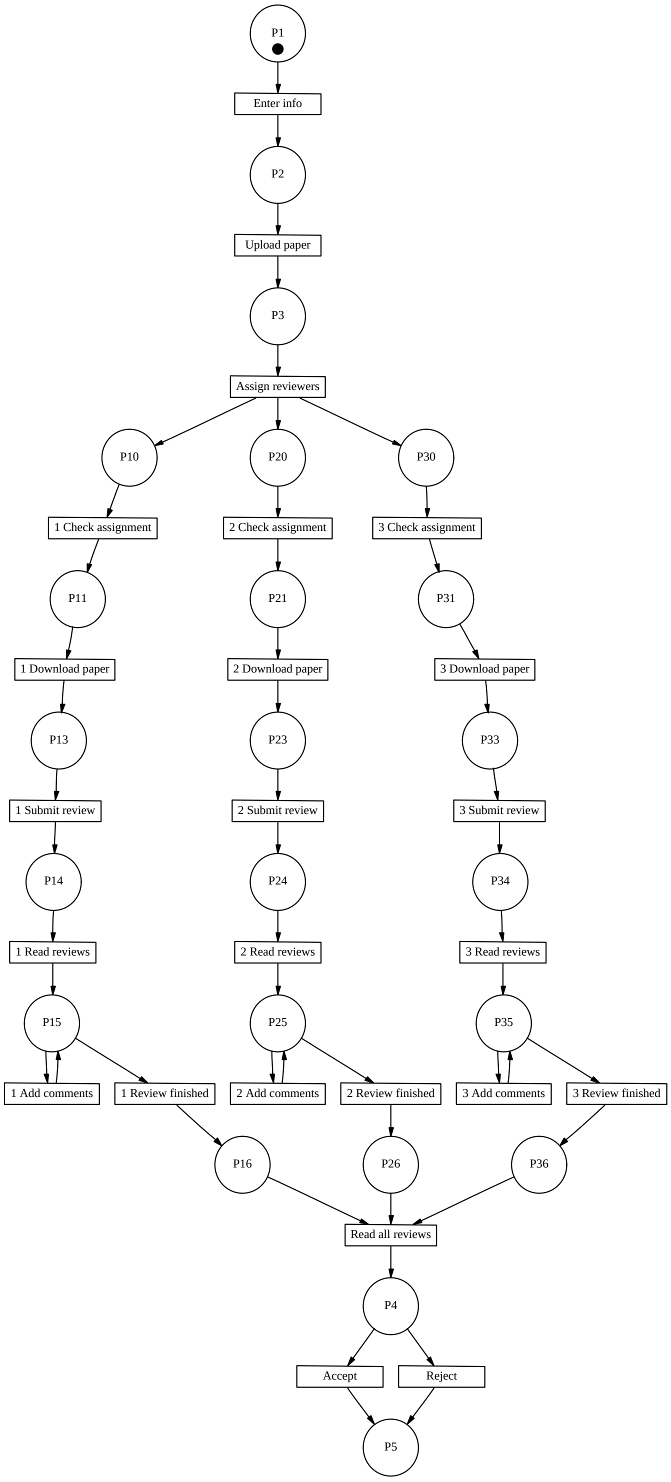

Fig. 7. Petri net model of a review process workflow.

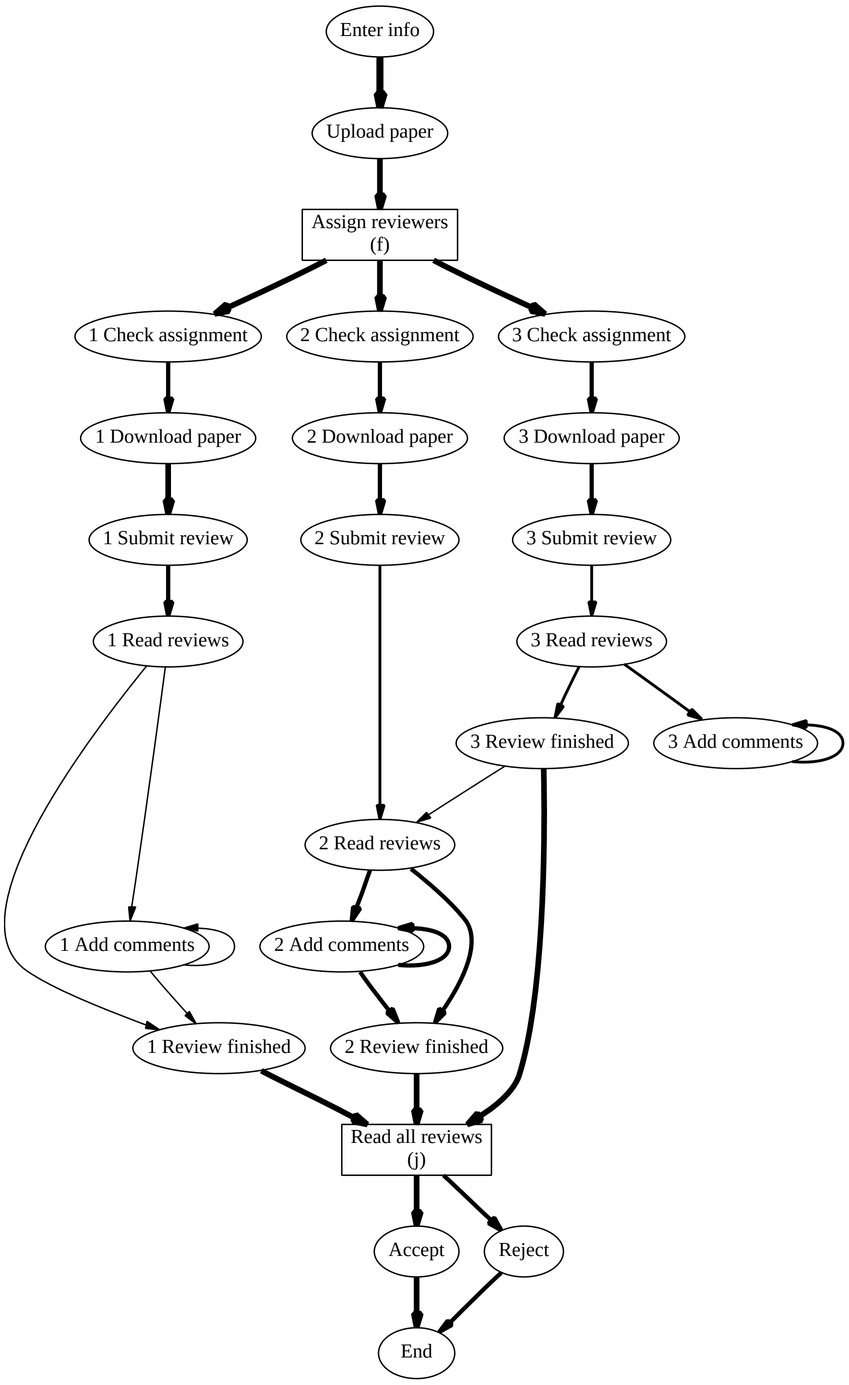

Fig. 8. Discovered model from event traces of Fig. 7.

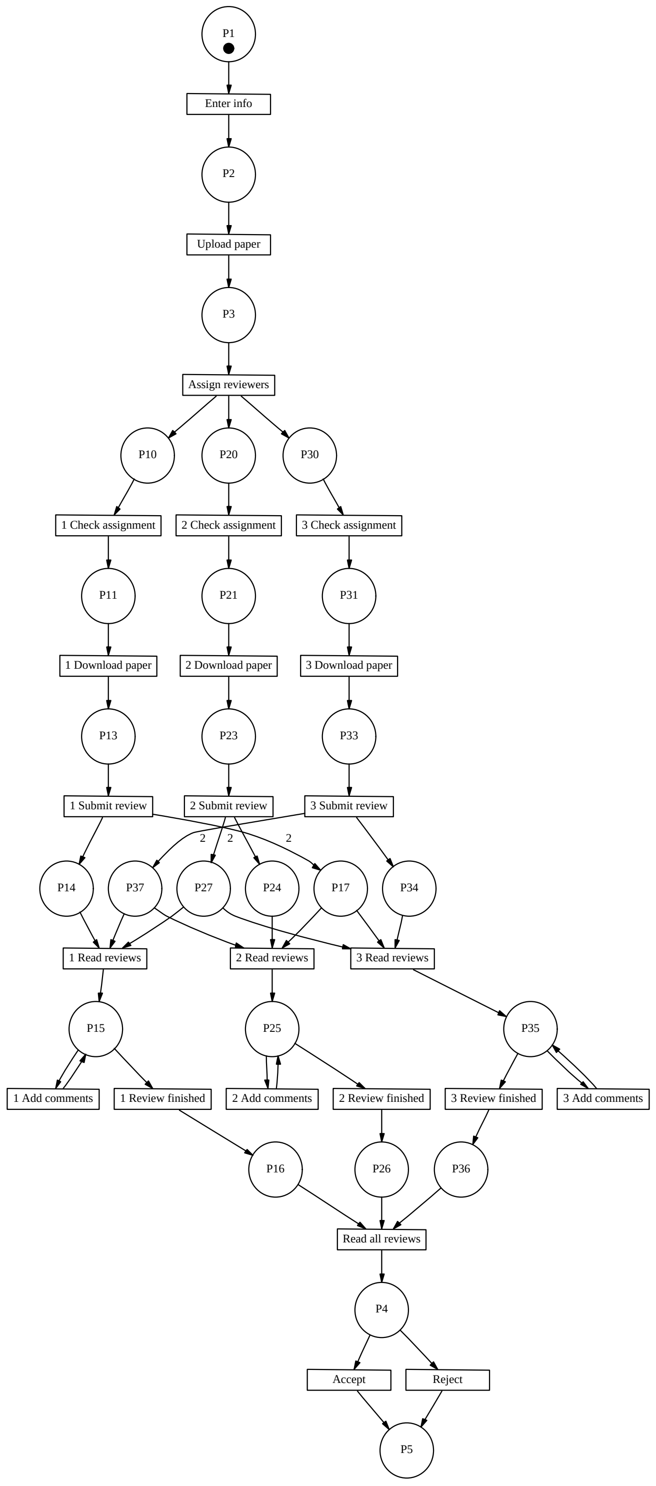

Fig. 9. Petri net model of augmented review process workflow.

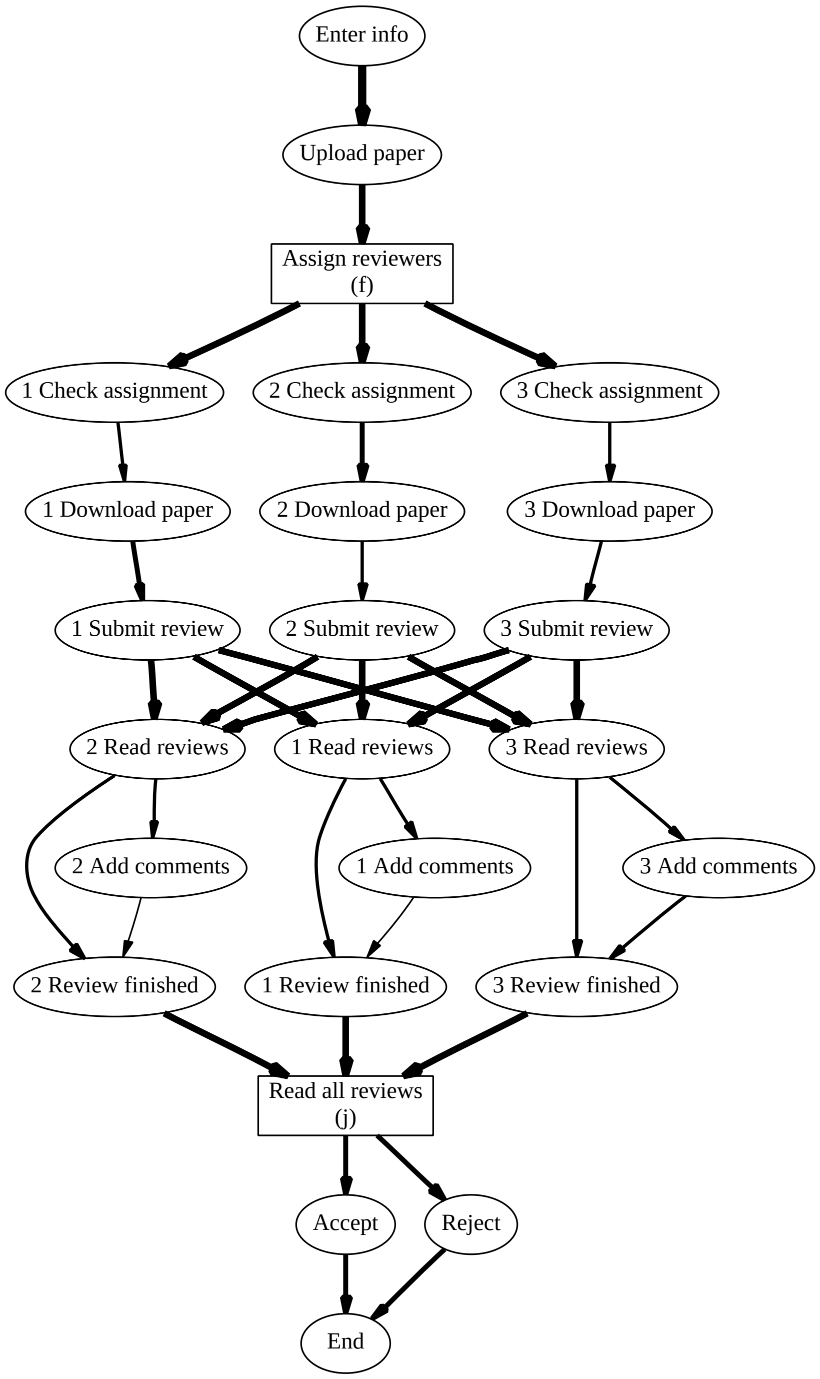

Fig. 10. Discovered model from event traces of Fig. 9.

We again simulated this workflow model, resulting in 76,957 events over 2560% executions of the workflow. Note that the set of event types is exactly the same. We applied our discovery tool to this event trace, and discovered the model shown in Fig. 10.

This discovered model is quite remarkable, in that it still correctly identifies the fork/join points, the three reviewer threads, and it also identifies the synchronization dependencies between the threads. Furthermore, these are the only dependencies that it discovers between the threads. It also is void of the mistake in Fig. 8 where reviewer threads 2 and 3 have an erroneous dependency between them. The synchronization in the middle of the reviewer threads acts as an anchor point and breaks up the possible concurrency into the beginning of the threads and the end of the threads, and this actually makes it easier to separate the real concurrency, if the synchronizations are discovered correctly. This model does not, however, have the loops on the ‘‘add comments’’ events, but rather only infers an optional single event path. In both examples, there was on average only 2.3 comment events per thread, so these single-event loops were not heavily exemplified in the event trace, and it is not unreasonable for a discovery algorithm to miss them, especially in the face of concurrency.

6. Related work

There is a long history of theoretical work concerned with inferring grammars for languages given example sentences in the language [14–19]. Other efforts have also used statistical methods [20,21]. None of these early efforts looked at the problem of concurrency in the trace.

Several other research efforts specifically in the workflow area have been aimed at inferring both sequential and concurrent models from event or activity logs.

- Herbst [22–24] investigates the discovery of both sequential and concurrent workflow models from logged executions. In this work, the data used is a log of partially ordered, time-spanning activities, not totally ordered instantaneous events. This allows the algorithms to view each activity ordering as a correct and useful datum from which to induce a model. Our techniques are geared towards recovering a model in the face of the ‘‘noise’’ of randomly ordered instantaneous events.
- Weijters and van der Aalst [25,26] explore the area of workflow process mining. Their view of the data as instantaneous events, their use of frequency counts, and their heuristic rules for when to infer a dependency appear to be similar to our work. However, their causality metric cleverly takes into account the possibility of causal events not appearing directly contiguous in an event trace from a concurrent process, and could be useful in our framework as well. They also translate the dependency graphs into their own desired workflow modeling notation, which is based on Petri nets.
- Agrawal et al. [27] investigate producing activity dependency graphs from event-based workflow logs. The logs already identify the partial ordering of concurrent, time-spanning activities, and they are concerned with producing correct and minimal graphs. There is no notion of identifying synchronization points within the activities.

Other related work can also be found in the area of debugging and understanding distributed, concurrent software systems.

- Due to the popularity of message sequence charts (MSC) as a scenario-based requirement and design specification method, several recent efforts have centered around combining individual MSCs into a single behavioral model [28–30]. These efforts are quite different than ours in that the ‘‘traces’’ are small, human-designed sequences that each represent a unique aspect of behavior for the system. Thus, every sequence is sure to represent some important aspect, and there is no need to probabilistically reason about which observed sequences are useful and which are not. Some dynamic analysis efforts do take large amounts of trace information and condense them first into MSCs, and then aggregate those into state diagrams [31]. This work is based on purely sequential traces.
- Holtzblatt et al. [32] explore methods of design recovery for distributed systems, where they look at recovering the design architecture of the task flow

from the source code. They do not look at dynamic behavior, and indeed point to this as a limitation in their approach.

- Venkatesan and Dathan [33] use an event-based framework for testing and debugging distributed programs. They provide distributed algorithms for evaluating global predicates that specify the correct behavior that the system should be exhibiting.
- Diaz et al. [34] use an event-based framework for on-line validation of distributed systems. Their mechanism employs an active observer that can listen to events (or messages) and compare the actual behavior to a formal specification of the correct behavior.

7. Conclusion

In this paper, we developed and demonstrated probabilistic techniques for inferring concurrent models of system behavior from event traces. If the events have some attribute that identifies the thread they belong to, then sequential model techniques can be applied to infer a single-thread model. This paper summarized previous work in this area [35], and presented a minor extension to this work.

If events do not have thread-identifying information, new techniques are needed to find the correct dependencies within the random orderings of events produced from multiple, concurrent threads. This paper mathematically formalized and extended techniques first sketched in [2], and presented an example workflow demonstrating the capabilities of these techniques.

Providing these techniques to engineers who are maintaining or creating workflow models will enhance their effectiveness in understanding the actual work processes, and can help in making changes for beneficial improvement of the workflow.

Several further directions need to be explored in this work. The assumption of a single place for events is restrictive. Extending the technique to allow for a model that produces an event type at multiple points would be a significant improvement. Our previous sequential discovery method, Markov, did just that, but the concurrent case is more complex because of the computations of entropy and periodic behavior. These computations would need to be separated for the set of production points of each event type. The work in [24] can be built on for this extension.

Incorporating domain or existing knowledge about the system would enhance the validity of the metrics. An engineer might, for example, know a priori that a certain event type will signal a synchronization point for the threads in the workflow, or they might know how many total threads there are. As mentioned previously, some collection methods may even be able to associate events with specific threads, though this in itself does not remove the difficult points of detecting actual concurrency. Domain knowledge may allow some of this information to be gleaned as well. For example, knowledge of the number of persons involved in a workflow might lead to statements about the number of threads inherent in it. Investigating the thresholding behavior of the technique, and providing better threshold parameters and guidance in using them, is also an important direction to make the technique widely usable.

The technique presented here is essentially a greedy algorithm that never retracts decisions about which dependencies to instantiate. Extending this to allow multiple possible dependencies to be explored might significantly enhance the quality of the results achieved. The problems with such techniques is that they usually increase the running times of the algorithms prohibitively.

Finally, the limitation mentioned previously about slower threads never producing events near each other and thus not being recognized would be worth studying and pursuing. Other domains have looked at *lagged* frequencies, where one calculates the frequency not of the next immediate event, but of the event following by a lag of N. We will investigate the suitability of such a method to the problem of concurrency discovery and modeling. The causality metric in [26] demonstrates positive results along these lines.

Our methods use purely relative time, but if we have a timed event trace, using absolute time would provide different, and alternatively interesting results. A long period of inactivity would look short in relative time, but the states that the system is in may be purposely synchronized for that idle period. Other researchers [36] are also heading in this direction.

Acknowledgements

This work was supported in part by the National Science Foundation under grants CCR-9302739, CCR-9804067, EIA-9810732, and EIA-0220590; and by the Air Force Materiel 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.

References

[1] J. Cook, A. Wolf, Discovering models of software processes from event-based data, ACM Transactions on Software Engineering and Methodology 7 (3) (1998) 215–249.

[2] J. Cook, A. Wolf, Event-based detection of concurrency, in: Proceedings of the Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering, Association for Computer Machinery, 1998, pp. 35–45.

[3] A. Wolf, D. Rosenblum, A study in software process data capture and analysis, in: Proceedings of the Second International Conference on the Software Process, IEEE Computer Society, 1993, pp. 115–124.

[4] R. LeBlanc, A. Robbins, Event-driven monitoring of distributed programs, in: Proceedings of the Fifth International Conference on Distributed Computing Systems, IEEE Computer Society, 1985, pp. 515–522.

[5] G. Avrunin, U. Buy, J. Corbett, L. Dillon, J. Wileden, Automated analysis of concurrent systems with the constrained expression toolset, IEEE Transactions on Software Engineering 17 (11) (1991) 1204–1222.

[6] P. Bates, Debugging heterogenous systems using event-based models of behavior, in: Proceedings of a Workshop on Parallel and Distributed Debugging, ACM Press, 1989, pp. 11–22.

[7] J. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, D. Stemple, The Adriane Debugger: scalable application of event-based abstraction, in: Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, ACM Press, 1993, pp. 85–95.

[8] J. Cook, Process discovery and validation through event-data analysis, Technical Report CU-CS-817-96, University of Colorado, Boulder, CO, November 1996.

[9] J. Cook, L. Votta, A. Wolf, Cost-effective analysis of in-place software processes, IEEE Transactions on Software Engineering 24 (8) (1998) 650–663.

[10] J. Peterson, Petri Nets, ACM Computing Surveys 9 (3) (1977) 223–252.

[11] D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (1987) 231–274.

[12] J. Devore, Probability and Statistics for Engineering and the Sciences, third ed., Brooks/Cole, Pacific Grove, CA, 1991.

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

[14] D. Angluin, C. Smith, Inductive inference: theory and methods, ACM Computing Surveys 15 (3) (1983) 237–269.

[15] L. Pitt, Inductive inference, DFAs, and computational complexity, in: Analogical and Inductive Inference, vol. 397 of Lecture Notes in Artificial Intelligence (subseries of LNCS), Springer-Verlag, New York, 1989, pp. 18–44.

[16] E. Gold, Language identification in the limit, Information and Control 10 (1967) 447–474.

[17] L. Valiant, A theory of the learnable, Communications of the ACM 27 (1984) 1134–1142.

[18] S. Jain, A. Sharma, On monotonic strategies for learning R.E. languages, in: Algorithmic Learning Theory, vol. 872 of Lecture Notes in Artificial Intelligence (subseries of LNCS), Springer-Verlag, New York, 1994, pp. 349–364.

[19] S. Lange, J. Nessel, R. Wiehagen, Language learning from good examples, in: Algorithmic Learning Theory, vol. 872 of Lecture Notes in Artificial Intelligence (subseries of LNCS), Springer-Verlag, New York, 1994, pp. 423–437.

[20] R. Carrasco, J. Oncina, Learning stochastic regular grammars by means of a state merging method, in: Grammatical Inference and Applications, vol. 862 of Lecture Notes in Artificial Intelligence (subseries of LNCS), Springer-Verlag, New York, 1994, pp. 139–152.

[21] L. Miclet, Syntactic and Structural Pattern Recognition: Theory and Applications, vol. 7 of Series in Computer Science: Grammatical Inference, World Scientific, New Jersey, 1990, Chapter 9, pp. 237–290.

[22] J. Herbst, Inducing workflow models from workflow instances, in: Proceedings of the SCS Concurrent Engineering Europe Conference, Society for Computer Simulation, 1999.

[23] J. Herbst, A machine learning approach to workflow management, in: Proceedings of the 11th European Conference on Machine Learning, vol. 1810, Springer-Verlag, Berlin, 2000, pp. 183–194.

[24] J. Herbst, Dealing with concurrency in workflow induction, in: Proceedings of the 7th European Concurrent Engineering Conference (ECEC'2000), 2000.

[25] T. Weijters, W. van der Aalst, Process mining: discovering workflow models from event-based data, in: Proceedings of the 13th Belgium–The Netherlands Conference on Artificial Intelligence (BNAIC 2001), 2001, pp. 283–290.

[26] T. Weijters, W. van der Aalst, Rediscovering workflow models from event-based data, in: Proceedings of the 11th Dutch–Belgian Conference on Machine Learning (Benelearn 2001), 2001, pp. 93–100.

[27] R. Agrawal, D. Gunopulos, F. Leymann, Mining process models from workflow logs, Lecture Notes in Computer Science 1377 (1998) 469–483.

[28] I. Kruger, R. Grosu, G. Scholz, M. Broy, From MSCs to statecharts, in: Proceedings of the 1998 Distributed and Parallel Embedded Systems, Kluwer Academic Publishers, Dordrecht, 1999, pp. 61–71.

[29] S. Some, R. Dssouli, J. Vaucher, From scenarios to timed automata: building specifications from users requirements, in: Proceedings of the 1995 Asia Pacific Software Engineering Conference (IEEE'1998), 1998, pp. 48–57.
[30] R. Alur, K. Etessami, M. Yannakakis, Inference of message sequence charts, in: Proceedings of the 22nd International Conference on Software Engineering, 2000, pp. 304–313.
[31] T. Systa, Understanding the behavior of Java programs, in: Proceedings of the 2000 Working Conference on Reverse Engineering, 2000, pp. 214–223.
[32] L. Holtzblatt, R. Piazza, H. Reubenstein, S. Roberts, D. Harris, Design recovery for distributed systems, IEEE Transactions on Software Engineering 23 (7) (1997) 461–472.
[33] S. Venkatesan, B. Dathan, Testing and debugging distributed programs using global predicates, IEEE Transactions on Software Engineering 21 (2) (1995) 163–177.
[34] M. Diaz, G. Juanole, J. Courtiat, Observer—a concept for formal on-line validation of distributed systems, IEEE Transactions on Software Engineering 20 (12) (1994) 900–913.
[35] J. Cook, A. Wolf, Software process validation: quantitatively measuring the correspondence of a process to a model, ACM Transactions on Software Engineering and Methodology 8 (2) (1999) 147–176.
[36] W. van der Aalst, B. van Dongen, Discovering workflow performance models from timed logs, in: Proceedings of the International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002), LNCS vol. 2480, 2002, pp. 45–63.

Jonathan E. Cook is a faculty member in the Computer Science Department at New Mexico State University. His research interests are in the areas of software process data analysis, dynamic analysis of software, reliable component-based systems, and large software system maintenance. Dr. Cook is a member of the ACM and the IEEE Computer Society.

Zhidian Du received his MS degree in computer science in 2001 from New Mexico State University.

Chongbing Liu received his BS and MS in geophysics from China University of Geosciences in 1991 and 1994, respectively. He is currently a PhD student in Department of Computer Science at New Mexico State University. His current research interests include parallel and distributed systems for logic programs, and event-based software model discovery.

Alexander L. Wolf is a faculty member in the Department of Computer Science, University of Colorado at Boulder. Previously he was at AT&T Bell Laboratories. Dr. Wolf's research interests are in the discovery of principles and development of technologies to support the engineering of large, complex software systems. He has published papers in the areas of software engineering environments and tools, software process, software architecture, configuration management, distributed systems, persistent object systems, networking, and security. Dr. Wolf is currently serving as Chair of the ACM Special Interest Group in Software Engineering and is on the editorial boards of the ACM journal Transactions on Software Engineering and Methodology and the Wiley journal Software Process Improvement and Practice.