Event-Based Detection of Concurrency

Jonathan E. Cook

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA

jcook@cs.nmsu.edu

Alexander L. Wolf

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

alw@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-860-98 April 1998

New Mexico State University
Department of Computer Science
Technical Report NMSU-CSTR-9808 April 1998

A version of this report to appear in
The Proceedings of SIGSOFT '98: Sixth International Symposium
on the Foundations of Software Engineering, November 1998

ABSTRACT

Understanding the behavior of a system is crucial in being able to modify, maintain, and improve the system. A particularly difficult aspect of some system behaviors is concurrency. While there are many techniques to specify intended concurrent behavior, there are few, if any, techniques to capture and model actual concurrent behavior. This paper presents a technique to discover patterns of concurrent behavior from traces of system events. The technique is 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. The technique is useful in a wide variety of software engineering tasks, including architecture discovery, reengineering, user interaction modeling, and software process improvement.

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

Understanding a complex system's behavior is a task that is often required in software engineering. Typically, a specification of intended behavior does not exisit or, if it does exist, is woefully out of date. The source code of the system, while relatively accurate, can be too cumbersome and low level to deal with directly, especially for a large system. Fortunately, many systems have the ability to log their execution and thereby generate a trace of events that captures the actual behavior of the system. If one could analyze the trace to discover a model of the behavior, then that model could reliably be used to evaluate, maintain, and modify the system.

In previous work [3, 4], we developed methods for using 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 systems, 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 finite-state-machine model cannot capture the true behavior of a system. Thus, we need new methods to discover concurrent behavior. Uncovering a system's concurrent behavior is useful in such tasks as architecture discovery, reengineering, user interaction modeling, and software process improvement.

In this paper, we develop and demonstrate a technique 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. Our goal is not to infer the precise concurrent behavior of a system, 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 system should perform.

One might think that the goal of discovery for a concurrent process should be to identify the individual threads and their individual behaviors. In a concurrent process, however, the important issue is locating exactly those points where the threads interact. A system might be constructed, for example, having two threads, 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 system, identifying the points of thread interaction and how much actual concurrent behavior is exhibited is not as straightforward.

The next section provides definitions and background discussion to place the technique in context. Section 3 introduces the technique and its component metrics. Section 4 details example uses of the technique and discusses the success of the methods on these examples. Finally, Section 5 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 *process* to mean the whole system, and the term *thread* to mean a sequential execution control path within the process, perhaps running concurrently with other threads. Note that our use of the term process is somewhat non-traditional.

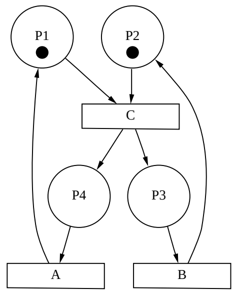

Figure 1: An Example Petri Net.

2.1 Events

Following our previous work [3, 13], we use an event-based model of process actions, where *events* are used to characterize the dynamic behavior of a process in terms of identifiable, instantaneous actions, such as sending a message, beginning a transaction, 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 [10], concurrent-system analysis [1], and distributed debugging [2, 5].

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 method invocation might have a "begin-method" and "end-method" 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". Activities may be composed of subactivities, with the only restriction being that sequential composition must be strictly hierarchical.

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., return value of a method), 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 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, this assumption may be relaxed.

2.2 A View of Concurrency

A concurrent process has simultaneously executing threads of control, each producing events that end up in the resulting event stream. Thus, the event stream represents interleaved event sequences from all of the concurrent threads. 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 events produced.

In this paper, example models of concurrent processes are given in the familiar Petri net formalism [11]. The separate "threads" of execution in the Petri net models are be visible from the connectedness of the places and transitions. Events are produced at transition firings; the sequence of events produced by the process is exactly the sequence of transition firings. Figure 1 shows a small concurrent process in the Petri net formalism.

2.3 Event Dependencies

Discovering a model of a system 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, 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 stream will contain points where parallel threads synchronize—assuming the events in fact represent cooperating threads. Upon entering the synchronization point, the threads become locked together, and 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.

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 any static relationships among events that might reasonably be established.

2.4 Tabulation of Event Sequence Characteristics

Given an event stream, 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 [3]. These tables record the frequencies at which each event and event sequence occur in the event stream. Along with frequencies, we also record the number of occurrences of each event type. Counts of event sequences are derivable from the frequencies and individual event type counts, and thus can be recorded or derived, depending on the need. In this work, we begin with a similar representation, but make use of a different analysis technique.

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

```
CABCBACABCABCBAC
```

If we look at the frequency of an event type following another event type in this event stream, we have the table

	A	B	C
A	0.0	0.6	0.4
B	0.4	0.0	0.6
C	0.5	.33	0.0

where an entry is the frequency of the column event following the row event. For example, if a B occurs in the event stream, 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, can be interpreted as the conditional probability that the second event will occur after the first event. This event stream is shown by its frequency table to be highly structured, since the values differ greatly in each table entry, and there are many 0.0 entries. Note that the probabilities given for the C event do not add up to 1.0 because the last C is followed by an "end-of-stream" event, which is not shown in the table.

The non-zero probabilities directly represent probabilistic dependence relations—that is, the second event type depends on the first occurring, with some probability. Entries of 0.0 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 being the immediate predecessor, but the locality of the frequency table would mask such an effect. We can also account for noise in the data by setting some threshold below which low-valued entries in the frequency table are treated as 0.0.

If we assume that this event stream is exactly correct (i.e., contains no noise) and derives from a sequential process, then every non-zero entry in the table would signify a correct event sequence, and thus a transition sequence in a state machine. But when the process that produces the event stream 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 process in Figure 1 that A and B are produced independently. A large part of the concurrency discovery problem involves deciding which entries in the frequency table are significant and which are not.

2.5 Assumptions

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

The first assumption is that each event type is produced at a single point in the process. In terms of a Petri net, this means that no two transitions are labeled with the same event. In Figure 2, for example, the three transitions are labeled uniquely as A, B, and C.

With this assumption, instead of expending effort on producing a model in a powerful formalism such as Petri nets, we can simply produce a Moore-type state machine, where each event type is produced in a single state and where output events are associated with states, not transitions. On this machine we can interpret the behavior as having multiple threads of execution, and can then mark those states that signify forks, joins, or other synchronization points for the concurrent threads. Transitions in this machine are unlabeled, and represent direct dependence relations of sequential, conditional, or concurrent types. For sequential and conditional transitions, only a single transition is taken out of a state. For concurrent transitions, a set of transitions are taken

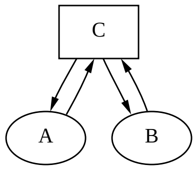

Figure 2: An Equivalent Model to Figure 1 using a Fork/Join Node.

simultaneously from a state, or a set of transitions are taken when entering a state. It should be evident that we are defining a simplified form of statechart [8].

A traditional fork would be a marked state that has a single input transition and multiple output transitions, a traditional join would be a marked state that has multiple input transitions and one output transition, and a synchronization point would have multiple input and output transitions. For example, the process represented by the Petri net in Figure 1 would have a discovered model as that shown in Figure 2. Marking the C node with a box 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.

A second 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 stream 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 [6]. That is, AC in the example above should occur at least five times if we are going to use its frequency in our analysis.

Finally, underlying our approach is the assumption that related events will appear next to each other in a stream with a high enough frequency that inference can be made on this relation. It could certainly happen that a system might produce events that are directly dependent but that never appear contiguous in the event stream. This would occur, for instance, if a slow thread were mixed in with fast threads. Our current technique does not handle this situation, but is the subject of future work.

3 A Concurrency Discovery Technique

With the groundwork laid, 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 two-event loops from concurrent independence.

As a working example, we use the process shown in Figure 3. This process simply produces an

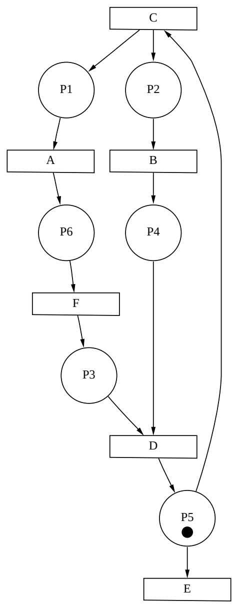

Figure 3: A Simple Concurrent Process Modeled as a Petri Net.

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.

```
CAFBDCBAFDCAFBDCAFBDCAFBDCAFBDCBAFDCBAFDE
```

For analysis purposes, we use 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.

3.1 Entropy

A key calculation that can be derived from the frequency tables is that of *entropy*, which gives a measure of the randomness of, or conversely the amount of information contained in, each event type 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 for AB=1.0, and for all other events E, AE=0.0. This means the behavior after A is perfectly deterministic, and 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.

Entropy is defined by the following formula

$$E(T) = \sum_{i=1}^{N} P(E_i|T) log_N P(E_i|T)$$

with $P(E_i|T)$ being the probability of event E_i occurring given that event T just occurred (i.e., the entry in the frequency table at row T, column E_i) and there being N possible event types.

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 type.

If we assume a fork-style concurrent behavior, there are specific values of entropy that might signal a fork point. Take as an example a two-way fork in a process that produces five different event types. Assuming a balanced production order of the two events in each branch of the fork, the probability values for each will be about 0.5, and for the other three events will be 0.0. Thus, the row in the frequency table will be

	A	B	C	D	E
A	0.0	0.5	0.5	0.0	0.0

assuming A is produced just before the fork, and B and C are produced first on each branch of the fork. The entropy, then, for A is 0.43. This value represents the asymptotic bound on a two-way fork for a process producing five event types. If a fork in this process is balanced, then with enough data the entropy for the fork should approach 0.43, and will never be greater than that in the absence of noise.

For a T-way fork given N token types, this entropy limit is given by the following simple formula.

$$log_N T$$

Note that this same formula and metric can apply to joins as well, by simply viewing the event stream backwards. The reverse frequencies can easily be computed from the forward frequencies using Baye's Law [6], so that extra tables are not needed.

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 as a two-way fork. The following metric, however, can help distinguish between the two cases.

3.2 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. A first indication of which of these might be happening is a count of the events for the event types involved.

If selection is occurring at some point in the process,[1] then the counts of the event types following the selection point should sum to the count of the event 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, 8 As, 5 Bs, and 3 Cs. 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 type at the fork point, and their sum should be a multiple of the count of the event at the fork point. Furthermore, each event type after the fork should have the same count; one event should not have a low count and the other high, even though they sum to a multiple of the event type at the fork point. Thus, in the example above, there would be 8 As, 8 Bs, and 8 Cs.

These relationships do not have to always hold, since the event types involved might be producible through other paths in the model. 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.

[1] Recall that we are assuming that each event type is produced at a single point in the process.

3.3 Periodicity

With th 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 type 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 types 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 type.

Consider the example in Figure 3. 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.

So, to calculate the periodicity of event types, we do the following.

1. Mark the positions in the event stream that an event type occurs.

2. Calculate the differences between successive positions.

3. Calculate the mean and standard deviation of those differences.

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.

3.4 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. Thus, given two events A and B, how can we decide when they are sequentially causally related and when they are not?

If we do not see the sequences AB or BA (perhaps within some threshold of 0.0), then we can say that they are not directly causally related. But if we do have significant probabilities 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. 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. (Remember 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 probabilities that separate these last two cases. If $P(AB)+P(BA) \geq 1.5$, then these two events are causally related in a two-event loop. If $P(AB)+P(BA) < 1.5$, then they are 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.[2] For the first, $P(AB) = 0.5$ and $P(BA) = 1.0$, and for the second, $P(AB) = 1.0$ and $P(BA) = 0.5$. For any longer length sequences, the sum of these two probabilities will only increase, asymptotically approaching 2.0 as the endpoints play less and less of a role.

If A and B are independent, then other event types will have non-zero probabilities from A and B, and even AA and BB may have non-zero probabilities, so that the likelihood of $P(AB) + P(BA) \geq 1.5$ occurring will be small. It is possible that A and B are independent, but that by chance the sequences occur such that the probabilities are at least 1.5, but if this is the case, then more data should eventually show them to be independent.

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

If we assume two concurrent loops, one of As and one of Bs, 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.

3.5 Putting the Metrics Together

The separate metrics above are combined into a single technique for discovering concurrency. The framework we use for discovering 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. In a stream with no noise, we are done when we have used all occurrences to explain all occurrences.

The key in this approach is the order in which dependencies are inferred and explanations are created. Naturally, one should begin by inferring the dependencies that are most likely to be correct. In a probabilistic framework, this means that we first look at those event types that have the most and best information in their values. 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 3.1. 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 that is closest to the actual entropy of a given event type, and use the difference between these two for ranking. 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 it is derived from more sequences than the

[2] Seeing just AB is not enough to determine that a loop is present.

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 N highest probability sequences, where N is the branching factor indicated by the chosen entropy limit. Subtracting this from 1.0 gives us the amount of event sequence frequency not accounted for by the branches allowed with the entropy limit. The final ranking, then, is given in increasing order of the (weighted) sum of the entropy difference and the unaccounted-for frequency.[3]

For example, if $AB = 0.38$ and $AC = 0.62$ occur in a stream of 10 event types, the entropy difference from the ideal two-branch limit would be 0.013, and the unaccounted-for frequency would be $1-(0.38+0.62) = 0$. If in the same stream the sequences $BD = 0.75$, $BE = 0.17$, and $BF = 0.08$ occur, then the entropy for B is still closest to the two-branch entropy, with a difference of 0.011. 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 $1 - (0.75 + 0.17) = 0.08$, 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). These reverse measurements are calculated from the forward measurements using Bayes' Law. This ranking, then, 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 Figure 4 in outline form. Each phrase describing some processing uses the detail of the metrics described above. The algortihm has been prototyped in a discovery tool that reads in files of events and produces a graph representation of the discovered model. This prototype was given as input the 1666-event stream produced from a stochastic simulation of the model in Figure 3. The resulting discovered model is shown in Figure 5. The event type C is found by the entropy metric to be a fork and by the periodicity metric to be a synchronization point. The other event types are successfully separated into their correct dependence relations based on the causality metric, and the event type D is found to be a join based on event count and to be a synchronization point by periodicity.

The next section demonstrates the application of our technique and prototype tool on larger examples.

4 Examples

In this section we give two, somewhat more complex examples of how our technique can discover a concurrency model for a process.

Figure 6 shows a Petri net model of a process with three threads, one having a sub-loop of two events, one having a selection point, and one being strictly sequenced. The concurrency is separated a bit within the process. Transition T4 is a synchronization point—all three threads join at T4, and then two threads are created. One of those threads, at transition T5, splits into two more threads, and then all three continue until they reach T4 again. The process ends when, after T4, the two threads terminate at T9 rather than continuing on. Although small, this process is non-trivial, since it also contains all the sequential constructs (sequence, selection, iteration) that need to be recognized—and distinguished—by our discovery technique.

[3]The sum is weighted because the entropy differences will be small compared to the frequency differences. Our experience to date has led us to use a coefficient of 3 on the entropy value.

Scan event stream and tabulate counts of 1 and 2-event sequences.
Compute frequencies of 1 and 2-event sequences from count tables.
Compute forward and reverse entropies for each event type.
FEL ← *The list of event types and their forward entropies.*
REL ← *The list of event types and their reverse entropies.*
SEL ← *FEL and REL combined and sorted according to ranking criteria (each event type occurs twice in SEL).*
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
        record the dependency E→DE (forward) or DE→E (reverse)
        if ranking is forward
            update count of explained occurrences of DE
            update count of used occurrences of E
        else ranking is reverse
            update count of used occurrences of DE
            update count of explained occurrences of E
        endif
    end
end
Output dependencies in graph form.

Figure 4: The Discovery Algorithm.

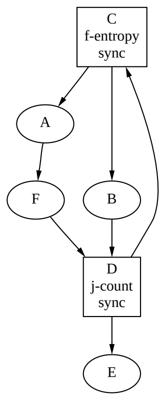

Figure 5: The Discovered Model of the Process in Figure 3.

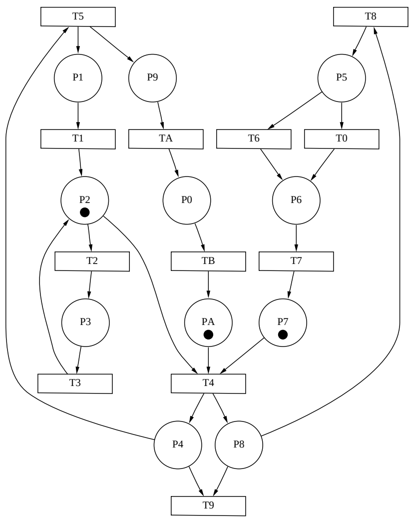

Figure 6: A More Complex Concurrent Process.

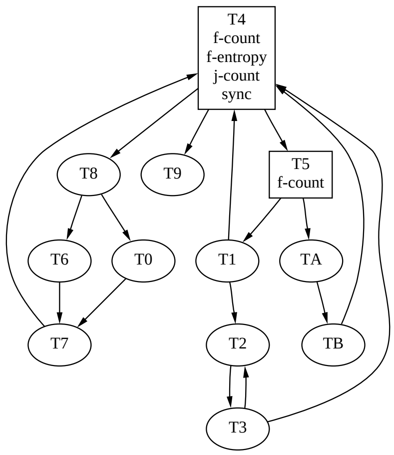

Figure 7: The Discovered Model of the Process in Figure 6.

Figure 7 shows the discovered model, with the analysis run on a 7018-event stream. As shown, all three threads were discovered, and the separate points where they begin. Note that T5 is recognized as a fork only because of the token counts; entropy did not see it, presumably because there was too much randomness due to the other threads. The join place is still altogether, at T4. Even with the sub-loop in one thread, the periodicity signals that T4 is a synchronization point. This illustrates the robustness of the various metrics.

We now turn to a classic concurrent system, that of the dining philosophers. This system contains a rich level of interdependence among the concurrent threads representing the philosophers. We examine both a Petri net model and a C program based on *pthreads* as example implementions of the system.

The Petri net model is shown in Figure 8. This model has a single place representing each

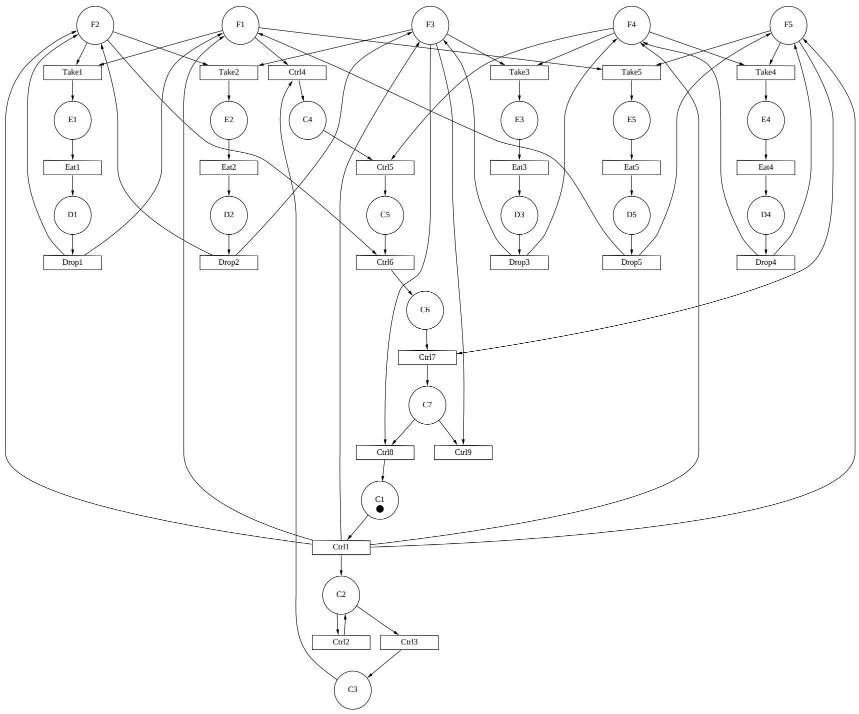

Figure 8: The Dining Philosophers Problem.

fork, available if it is marked with a token. A philosopher has a TakeForks transition that will fire and consume the tokens on that philosopher's adjacent fork places. A philosopher has an Eat transition that fires sometime later, followed by a DropForks transition that again marks the places of the forks. A set of control transitions and places regulate the philosophers. To begin, the control transition Ctrl1 puts forks down by marking all fork places, allowing the philosophers to begin eating. When the control decides that eating should cease (based on the firing of control transition Ctrl3), the control begins a sequence of transitions that pick up the forks one at a time. When all forks are picked up, the beginning control transition can fire again, putting all forks down for another round of eating, or an exit transition can fire, finishing the session. The Petri net is simulated using a stochastic net simulator, and no guarantee of fairness is given.

Figure 9 shows a discovered model from an event stream generated from the Petri net implementation of the dining philosophers. The event stream used for the analysis contains 60,434 events. This diagram shows the successful recognition of the fork that starts all five philosophers. Recognizing the end of the philosopher threads is more difficult because they end one at a time, as the control transitions are able to pick up the forks one by one. Thus, some events are recognized as joins, but it is in general not clear.

Note that two of philosopher 2's events are inferred as synchronization points by the periodicity metric. This led us to look more closely at the event stream and see that philosopher 2 had vastly more take-eat-drop cycles than the other philosophers. We realized that this is because philosopher 2 is the last philosopher that is still able to eat as the forks are picked up, one by one, to finish a dining session. Apparently philosopher 2 wins out over the control transition many more times than it fails. So this result, although seemingly spurious, actually leads us to understand some

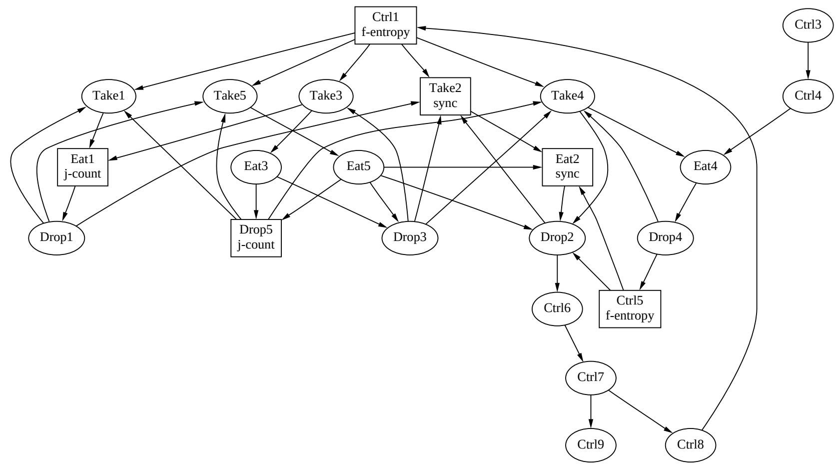

Figure 9: The Discovered Model for the Petri Net Dining Philosophers.

unintended behavior in the system, which, after all, is the point of having a discovery technique. Notice also that the Drop event type of each philosopher has, in general, dependencies to its and its neighbors' Take events. This is correct behavior for the dining philosophers. There are really no "independent" threads in this example, because the independent action of each philosopher—eating—is modeled as a single event.

We turn now to the C implementation of the dining philosophers. This program was taken from a set of example programs using pthreads, so we did not know its internal behavior beforehand. It uses the more conventional solution of having a philosopher pick up the fork to its right, try to pick up the fork to the left and, if unavailable, drop the right fork, wait, and try again. Once both forks are obtained, the philosopher eats and then drops both forks. The events each philosopher generate are: Pn-Sit, Pn-getfork$n$, Pn-dropfork$n$, Pn-getforkn', Pn-Eat, Pn-dropforks, Pn-finished, and Pn-FinishedMEAL, where n is the philosopher's identifying number (0-4) and n' is that number plus 1, modulus the number of philosophers.

The first attempts at discovery produced graphs that had no indicated synchronization points, and were in general very complex. The early graphs did show a high concentration of dependencies on two event types, one of philosopher 4's and one of philosopher 2's. Inspecting the code led us to realize that the amount of time spent eating was hard-coded, and resulted (partially) in some strict orderings. We then modified the code to introduce more randomness, and generated new event traces.

Figure 10 shows a discovered model from the program, using a stream containing 17,962 events. One immediate observation from this graph is that the program runs in a very sequential manner, at least seen through these events. Also, the technique discovers a fork point and a join point, at P3-Start and End-5diners, respectively, although they both only indicate three threads forming and joining. This is not surprising, since the control of the threads does not let them act completely independently. There is no doubt that the threads run in parallel, but the order in which they begin to eat and finish is much more serialized than in the Petri net version. In looking at the program, we determined that the reason for this is that the program attempts to provide fairness to the philosophers in the form of a queue that philosophers enter when they are forced to drop one fork and wait to try again. It is this queue that ends up largely serializing the order in which

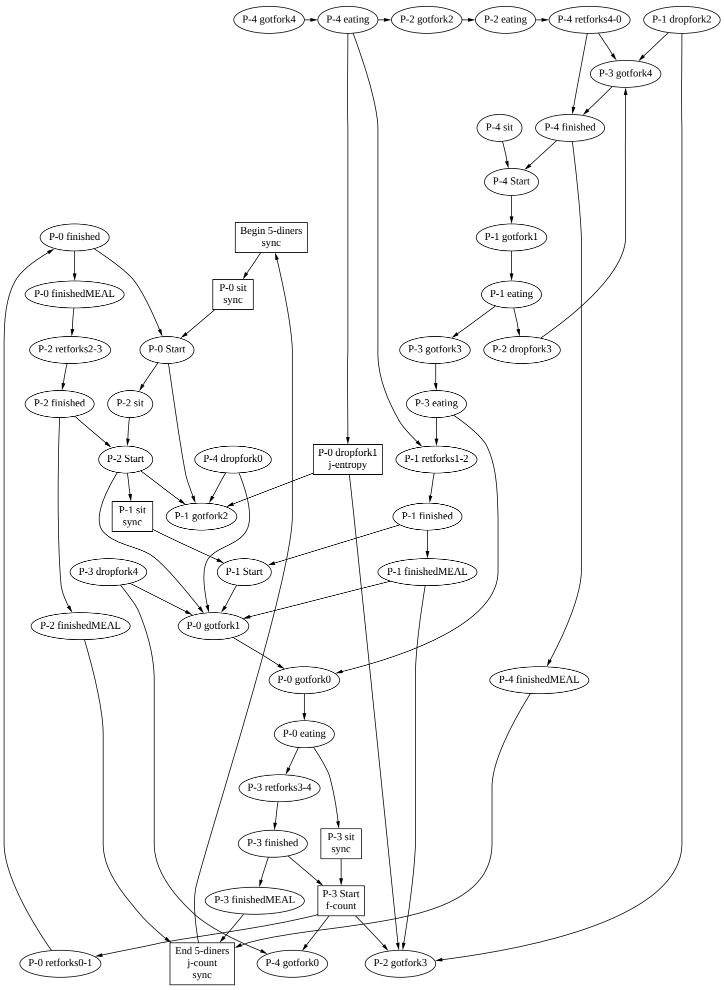

Figure 10: The Discovered Model for the Programmed Dining Philosophers.

the philosophers eat.

This is the type of information that is precisely needed when trying to understand the behavior of a concurrent program, and the discovery technique presented clearly here helped identify those points in the implementations.

5 Conclusion

In this paper we developed and demonstrated a probabilistic technique for discovering concurrent models of system behavior. Providing such a technique to engineers who are maintaining an unfamiliar system will enhance their effectiveness in understanding it and in making the correct changes for sound improvement of the system.

While our work in discovering concurrency from event traces appears to be novel, there certainly has been related work in understanding distributed, concurrent systems.

- Holtzblatt et al. [9] 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 [12] 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. [7] 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.

These works show the utility of using event-based models of behavior for system analysis.

Several further directions need to be explored in this work. The assumption of a single place for events is retrictive. 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.

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 process, or they might know how many total threads are in the process. Domain knowledge may allow some of this information to be gleaned as well. In a software process, for example, knowledge of the number of persons involved in the process might lead to statements about the number of threads inherent in the process.

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 of not the next immediate event, but the event following by a lag of N. We will investigate the suitablity of such a method to the problem of concurrency discovery and modeling.

REFERENCES

[1] 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.

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

[3] 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.

[4] J.E. Cook and A.L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3), July 1998. To appear.

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

[6] J.L. Devore. *Probability and Statistics for Engineering and the Sciences*. Brooks/Cole, Pacific Grove, California, 3rd edition, 1991.

[7] M. Diaz, G. Juanole, and J. Courtiat. Observer—A Concept for Formal On-Line Validation of Distributed Systems. *IEEE Transactions on Software Engineering*, 20(12):900–913, 1994.

[8] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[9] L.J. Holtzblatt, R.L. Piazza, H.B. Reubenstein, S.N. Roberts, and D.R. Harris. Design Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 23(7):461–472, 1997.

[10] 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.

[11] J.L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[12] S. Venkatesan and B. Dathan. Testing and Debugging Distributed Programs Using Gloabl Predicates. *IEEE Transactions on Software Engineering*, 21(2):163–177, 1995.

[13] 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.