Mining Process Models from Workflow Logs

Rakesh Agrawal *Dimitrios Gunopulos*
IBM Almaden Research Center
San Jose, CA 95120, USA

Frank Leymann
IBM German Software Development Lab
D-71034 Böblingen, Germany

January 22, 1998

Abstract

Modern enterprises increasingly use the workflow paradigm to prescribe how business processes should be performed. Processes are typically modeled as annotated activity graphs. We present an approach for a system that constructs process models from logs of past, unstructured executions of the given process. The graph so produced conforms to the dependencies and past executions present in the log. By providing models that capture the previous executions of the process, this technique allows easier introduction of a workflow system and evaluation and evolution of existing process models. We also present results from applying the algorithm to synthetic data sets as well as process logs obtained from an IBM Flowmark installation.

1 Introduction

Organizations typically prescribe how business processes have to be performed, particularly when activities are complex and involve many people. A *business process* specifies the way in which the resources of an enterprise are used. The performance of an enterprise depends on the quality and the accuracy of the business process. Thus techniques to manage and support business processes are an active research area. [RW92] [DS93] [GHS95] [LA92] [MAGK95].

In particular, a significant amount of research has been done in the area of modeling and supporting the execution of business processes. The model generally used is the *workflow* model [Hol94]. Workflow systems assume that a process can be divided in small, unitary actions, called *activities*. To perform the process, one must perform the set (or perhaps a subset) of the activities that comprise it. In addition, there may be *dependencies* between different activities.

The main approach used in workflow systems is to model the process as a *directed graph*. The graph vertices represent individual activities and the edges represent dependencies between them. In other words, if activity A has to be executed before activity B, an edge appears in the graph from A to B. In practice, certain executions of the process may include a given activity and others may not. Each edge $A \rightarrow B$ is, therefore, annotated with a Boolean function that determines whether the control flows from A to B.

Current workflow systems assume that a model of the process is available and the main task of the system is to insure that all the activities are performed in the right order and the process terminates successfully [GR97] [LA92]. The user is required to provide the process model. Constructing the desired process model from an unstructured model of process execution is quite difficult, expensive and in most cases require the use of an expert [CCPP96] [Sch93].

Contribution We present a new approach to address the problem of model construction. We describe an algorithm that, given a log of unstructured executions of a process, generates a graph model of the process. The resulting graph represents the control flow of the business process and satisfies the following desiderata:

- *Completeness*: The graph should preserve all the dependencies between activities that are present in the log. It should also permit all the executions of the process present in the log.
- *Irredundancy*: The graph should not introduce spurious dependencies between activities.
- *Minimality*: To clarify the presentation, the graph should have the minimal number of edges.

The work we present has been done in the context of the IBM workflow product, Flowmark [LA92]. However, the process model we consider is quite general and the algorithms we propose are applicable to other workflow systems. The new capability we are proposing can be applied in several ways. A technique that takes logs of existing process executions and finds a model that captures the process can ease the introduction of a workflow management system. In an enterprise with an installed workflow system, it can help in the evaluation of the workflow system by comparing the synthesized process graphs with purported graphs. It can also allow the evolution of the current process model into future versions of the model by incorporating feedback from successful process executions.

The following schema is being adopted in Flowmark for capturing the logs of existing processes in an enterprise that does not yet have an workflow system in place. First, all the activities in a process are identified. But since the control flow is not yet known, all possible activities are presented to the user for consideration through a graphical interface. The user selects the activities that, according to user's informal model of the business process, have to be executed next. Thus the successful executions of the process are recorded.

Related research The specification of dependencies between events has received much attention [Kle91] [ASE$^+$96]. Our dependency model is a simplification of that proposed in [Kle91], and is consistent with the directed graph process model.

In previous work in process discovery [CW95] [CW96], the finite state machine model has been used to represent the process. Our process model is different from the finite state machine model. Consider a simple process graph: $(\{S, A, B, E\}, \{S \to A, A \to E, S \to B, B \to E\})$, in which two activities A and B can proceed in parallel starting from an initiating activity S and followed by a terminating activity E. This process graph can generate $SABE$ and $SBAE$ as valid executions. The automaton that accepts these two strings is a quite different structure. In an automaton, the activities (input tokens) are represented by the edges (transitions between states), while in a process graph the edges only represent control conditions and vertices represent activities. An activity appears only once in a process graph as a vertex label, whereas the same token (activity) may appear multiple times in an automaton.

The problem considered in this paper generalizes the problem of mining sequential patterns [AS95] [MTV95], but it is applicable in a more restricted setting. Sequential patterns allow only a total ordering of fully parallel subsets, whereas process graphs are richer structures: they can be used to model any partial ordering of the activities and admit cycles in the general setting. On the other hand, we assume that the activities form only one graph structure, whereas in the sequential patterns problem the goal is to discover all patterns that occur frequently.

Organization of the paper The rest of the paper is organized as follows. In Section 2 we describe the process model used in the paper. In Section 3 we present an algorithm to find a process graph, assuming that the graph is acyclic and that each activity appears exactly once in each execution. The algorithm finds the minimal such graph in one pass over the log. In Section 4 we extend this algorithm to handle the case where some activities may not appear in each execution. In Section 5 we consider the case of general directed graphs admitting cycles. In these sections, we make the assumption that the log contains correct executions of the business process. However, this may not be the case in practice, and we outline a strategy to deal with this problem in Section 6. Section 7 presents implementation results using both synthetic datasets and logs from a Flowmark installation. We conclude with a summary in Section 8.

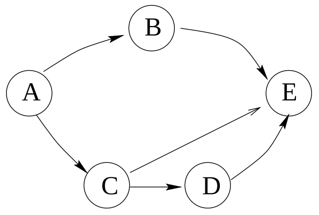

Figure 1: Example 1

2 Process model

Business processes consist of separate activities. An activity is an action that is a semantical unit at some level. In addition, each activity can be thought of as a function that modifies the state of the process. Business processes are modeled as graphs with individual activities as nodes.

The edges on the graph represent the potential flow of control from one activity to another[1]. Each edge is associated with a Boolean function (on the state of the process), which determines whether the edge will be followed or not. If a vertex (activity) has more than one outgoing edge, the respective Boolean functions are independent from each other.

Definition 1 (Business process) A business process P is defined as a set of activities $V_P = V_1, \ldots, V_n$, a directed graph $G_P = (V_P, E_P)$, an output function $o_P : V_P \rightarrow \mathcal{N}^k$, and $\forall (u,v) \in E_P$ a Boolean function $f_{(u,v)} : \mathcal{N}^k \rightarrow \{0,1\}$.

We will assume that G_P has a single source and a single sink. These are the process' activating and terminating activities. If there are no such activities, one can add an activating node with edges to the first executed activities in the graph, and a terminating node with edges from the terminating activities of the process. The execution of the business process follows the activity graph: for each activity u that terminates, the output $o(u)$ is computed. Then the functions on the outgoing edges are evaluated on the output. If $f_{(u,v)}(o(u))$ is true, then we test if v can be executed. This test in general is a logical expression involving the activities that point to v in G. When v is ready, the outputs of incoming activities are passed as input to v, and it is inserted into a queue to be executed by the next available agent.

Example 1 Figure 1 gives the graph G_P of a process P. The process consists of five activities $V_P = \{A, B, C, D, E\}$. A is the starting activity and E is the terminating activity. The edges of the graph G_P ($E_P = \{(A,B)$, (A,C), (B,E), (C,D), (C,E), $(D,E)\}$ represent the flow of execution, so that D always follows C, but B and C can happen in parallel. Not shown in Figure 1 are o_P and the Boolean conditions on the edges. Each activity has a set of output parameters that are passed along the edges, $o(A), \ldots, o(E) \in \mathcal{N}^2$. The

[1]For the purposes of this paper, we will not differentiate between control flow and data flow, a distinction made in some systems [GR97] [LA92].

output parameters are represented as a vector $(o(A)[1], o(A)[2])$. Each edge has a Boolean function on the parameters, such as: $f_{(C,D)} = (o(C)[1] > 0) \wedge (o(C)[2] < o(C)[1]))$. For example an execution of this process will include activity D if $f_{(A,C)}$ and $f_{(C,D)}$ are true.

Each execution of a process is a list of events that record when each activity was started and when it terminated. We can therefore consider the log as a set of separate executions of an unknown underlying process graph.

Definition 2 (Execution log) The log of one execution of a process (or simply execution) is a list of event records (P, A, E, T, O) where P is the name of the process execution, A is the name of the activity, $E \in \{\text{START, END}\}$ is the type of the event, T is the time the event occured, and $O = o(A)$ is the output of the activity if $E = \text{END}$ and a null vector otherwise.

For notational simplicity, we will not write the process execution name and output in the event records. We assume that the activities are instantaneous and no two activities start at the same time. With this simplification, we can represent an execution as a list of activities. This simplification is justified because if there are two activities in the log that overlap in time, then they must be independent activities. As we will see, the main challenge in inducing a process graph from a log of past executions lies in identifying dependency relationship between activities.

Example 2 Sample executions of the graph in Figure 1 are $ABCE$, $ACDBE$, $ACDE$.

If there exists a dependency between two activities in the real process, then these two activities will appear in the same order in each execution. However only the executions that are recorded in the log are known, and so we define a dependency between two activities with respect to the log. In the model graph, each dependency is represented either as a direct edge or as a path of edges from an activity to another.

Definition 3 (Following) Given a log of executions of the same process, activity B follows activity A if either activity B starts after A terminates in each execution they both appear, or there exists an activity C such that C follows A and B follows C.

Definition 4 (Dependence between activities) Given a log of executions of the same process, if activity B follows A but A does not follow B, then B depends on A. If A follows B and B follows A, or A does not follow B and B does not follow A, then A and B are independent.

Example 3 Consider the following log of executions of some process: $\{ABCE$, $ACDE$, $ADBE\}$. The activity B follows A (because B starts after A in the two executions both of them appear) but A does not follow B, therefore B depends on A. On the other hand, B follows D (because it is recorded after D in the only execution that both are present) and D follows B (because it follows C, which follows B), therefore B and D are independent.

Let us add $ADCE$ to the above log. Now, B and D are no longer independent; rather, B depends on D. It is because B follows D as before, but C and D are now independent, so we do not have D following B via C.

Given a log of executions, we can define the concept of a dependency graph, that is, a graph that represents all the dependencies found in the log.

Definition 5 (Dependency graph) Given a set of activities V and a log of executions L of the same process, a directed graph G_{VL} is a dependency graph if there exists a path from activity u to activity v in G_{VL} if and only if v depends on u.

In general, for a given log, the dependency graph is not unique. In particular, two graphs with the same transitive closure represent the same dependencies.

Every execution of the process recorded in the log may not include all the activities of the process graph. This can happen when not all edges outgoing from an activity are taken (e.g. the execution ACE in Figure 2). An execution R *induces* a subgraph G' of the process graph $G = (V, E)$ in a natural way: $G' = (V', E')$, where $V' = \{v \in V \mid v \text{ appears in } R\}$ and $E' = \{(v, u) \in E \mid v \text{ terminates before } u \text{ starts in } R\}$.

Definition 6 (Consistency of an execution) Given a process model graph $G = (V, E)$ of a process P and an execution R, R is consistent with G if the activities in R are a subset V' of the activities in G, and the induced subgraph $G' = (V', \{(u, v) \in E \mid u, v \in V'\})$ is connected, the first and last activities in R are process' initiating and terminating activities respectively, all nodes in V' can be reached from the initiating activity, and no dependency in the graph is violated by the ordering of the activities in R.

This definition of consistency is equivalent to the following one: R can be a successful execution of P for suitably chosen activity outputs and Boolean edge functions.

Example 4 The execution $ACBE$ is consistent with the graph in Figure 1, but $ADBE$ is not.

Given a log of executions, we want to find a process model graph that preserves all the dependencies present in the log. At the same time, we do not want the graph to introduce spurious dependencies. The graph must also be consistent with all executions in the log. A graph that satisfies these conditions is called a *conformal* graph.

Definition 7 (Conformal graph) A process model graph G is conformal with a log L of executions if all of the following hold:

- *Dependency completeness*: For each dependency in L, there exists a path in G.
- *Irredundancy of dependencies*: There is no path in G between independent activities in L.

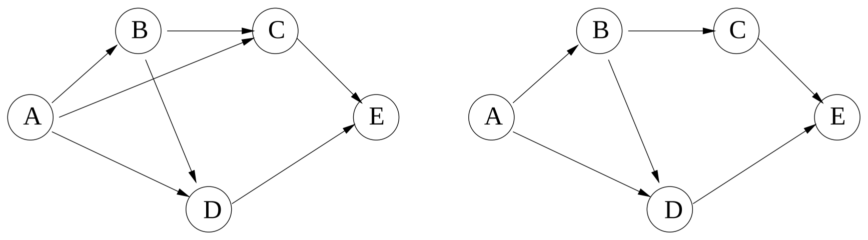

Figure 2: Example 5

- *Execution completeness*: G is consistent with every execution in L.

Example 5 Consider the log $\{ADCE, \ ABCDE\}$. Both the graphs in Figure 2 are dependency graphs. The first graph is conformal, but the second is not because it does not allow the execution $ADCE$.

Problem statement. We define the following two problems:

Problem 1: Graph mining. Given a log of executions of the same process, find a conformal process graph.

Problem 2: Conditions mining. Given a log of executions of the same process and a corresponding conformal process graph $G = (V, E)$, find the Boolean functions $f_{(u,v)}, \ (u, v) \in E$.

Having thus divided the process model mining problem into two parts, we will consider Problem 1 in Sections 3–6. We will address Problem 2 in Section 7. Assume throughout that the process graph has $|V| = n$ vertices, and the log contains m separate executions of the process. Generally, $m \gg n$.

In Sections 3 and 4, we will assume that the process graph is acyclic. This assumption is reasonable in many cases and, in fact, it is also frequently the case in practice [LA92]. We will relax this assumption in Section 5 and allow for cycles in the process graph.

3 Finding directed acyclic graphs

We first consider the special case of finding model graphs for acyclic processes whose executions contain exactly one instance of every activity. For this special case, we can obtain a faster algorithm and prove the following minimality result:

Given a log of executions of the same process, such that each activity appears exactly once in each execution, there exists a unique process model graph that is conformal and minimizes the number of edges.

Lemma 1 *Given a log of executions of the same process, such that each activity appears in each execution exactly once, if B depends on A then B starts after A terminates in every execution in the log.*

Proof: Assume that this is not the case. Then there exists an execution such that B starts before A terminates. From the definition of dependency, there must be a path of followings from A to B. But since all activities are present in each execution, there must be at least one following which does not hold for the execution where B starts before A, a contradiction. □

Lemma 2 *Let G and G' be graphs with the same transitive closure. Then both graphs are consistent with the same set of executions if each activity appears exactly once in each execution.*

Proof: Since every activity appears in each execution, the induced subgraph for any execution is the original graph. The two graphs have the same transitive closure, so if there is a path between two activities in one, then there is a path between the same activities in the other. It follows that if a dependency is violated in one graph, then it must be violated in the other one. □

Lemma 3 *Given a log of executions of the same process, where all activities appear in each execution once, and a dependency graph G for this log, G is conformal.*

Proof: By definition, the dependency graph preserves all the dependencies present in the log, and none other. For a given execution in the log, the induced subgraph is again the graph G because all activities are present. Further, no dependency is violated because if one was, it would not be in the dependency graph. It follows that G is conformal. □

We can now give an algorithm that finds the minimal conformal graph.

Algorithm 1 (Special DAG) *Given a log L of m executions of a process, find the minimal conformal graph G, assuming there are no cycles in the graph and each activity appears in each execution of the process.*

1. *Start with the graph $G = (V, E)$, with V being the set of activities of the process and $E = \emptyset$. (V is instantiated as the log is scanned in the next step.)*

2. *For each process execution in L, and for each pair of activities u, v such that u terminates before v starts, add the edge (u, v) to E.*

3. *Remove from E the edges that appear in both directions.*

4. *Compute the transitive reduction[2] of G.*

[2]The transitive reduction of a directed graph G is the smallest subgraph of G that has the same closure as G [AGU72]. A DAG has a unique transitive reduction.

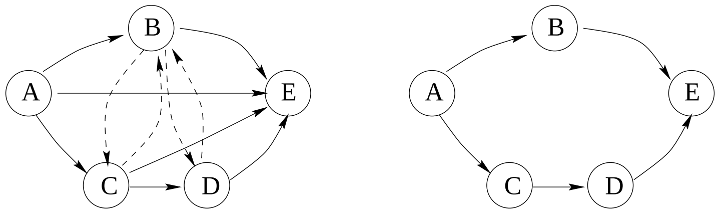

Figure 3: Example 6

5. *Return (V, E).*

Theorem 4 *Given a log of m executions of a given process having n activities, Algorithm 1 computes the minimal conformal graph in $O(n^2m)$ time.*

Proof: First we show that after step 3, G is a dependency graph. From Lemma 1 we know that the graph after step 2 at least contains an edge corresponding to every dependency. Since the edges we remove in step 3 form cycles of length 2, where there are activities u and v such that u follows v and v follows u, such edges cannot be dependencies.

After step 4, G is the minimal graph with the same transitive closure and, using Lemma 2, the minimal dependency graph.

Lemma 3 shows that this graph is also conformal and, since a conformal graph has to be a dependency graph, it is the minimal conformal graph.

Since $m \gg n$, the second step clearly dominates the running time. The running time of step 4 is $O(|V||E|) = O(n^3)$ [AGU72]. A simpler algorithm to compute the transitive reduction is given in the Appendix. □

Example 6 Consider the log $\{ABCDE, ACDBE, ACBDE\}$. After step 3 of the algorithm, we obtain the first graph of Figure 3 (the dashed edges are the edges that are removed at step 3), from which the next underlying process model graph is obtained with the transitive reduction (step 4).

4 The complete algorithm

We now consider the general case where every execution of an acyclic process does not necessarily include all the activities. The problem is that all dependency graphs are no longer conformal graphs: it is possible to have a dependency graph that does not allow some execution present in the log (Example 5).

The algorithm we give that solves this problem is a modification of Algorithm 1. It makes two passes over the log and uses a heuristic to minimize the number of the edges.

First it computes a dependency graph. As before, we identify those activities which ought to be treated as independent because they appear in reverse order in two separate executions. In addition, to guard against spurious dependencies, we also identify those activity pairs A,B that have a path of followings from A to B as well as from B to A, and hence are independent. To find such independent activities we find the strongly connected components in the graph of followings. For two activities in the same strongly connected component, there exist paths of followings from the one to the other; consequently, edges between activities in the same strongly connected component are removed.

We must also ensure that the dependency graph is such that it allows all executions present in the log. Having formed a dependency graph as above, we remove all edges that are not required for the execution of the activities in the log. An edge can be removed only if all the executions are consistent with the remaining graph. To derive a fast algorithm, we use the following alternative: for each execution, we identify a minimal set of edges that are required to keep the graph consistent with the execution, and include them in the final graph. Note that we can no longer guarantee that we have obtained a minimal conformal graph. We can now state our algorithm.

Algorithm 2 (General DAG) *Given a log L of m executions of a process, find the dependency graph G, assuming there are no cycles in the process graph.*

1. *Start with the graph $G = (V, E)$, with V being the set of activities of the process and $E = \emptyset$. (V is instantiated as the log is scanned in the next step.)*
2. *For each process execution in L, and for each pair of activities u, v such that u terminates before v starts, add the edge (u, v) to E.*
3. *Remove from E the edges that appear in both directions.*
4. *For each strongly connected component of G, remove from E all edges between vertices in the same strongly connected component.*
5. *For each process execution in L:*
   - *(a) Find the induced subgraph of G.*
   - *(b) Compute the transitive reduction of the subgraph.*
   - *(c) Mark those edges in E that are present in the transitive reduction.*
6. *Remove the unmarked edges in E.*
7. *Return (V, E).*

Theorem 5 *Given a log of m executions of a given process having n activities, Algorithm 2 computes a conformal graph in $O(mn^3)$ time.*

Proof: After step 3, we are left with a directed graph where each edge path represents a following in L. Step 4 finds cycles in this graph and the set of vertices in each cycle represent independent activities by definition. We thus have a dependency graph for L after step 4. This graph maintains execution completeness as step 2 created a graph that

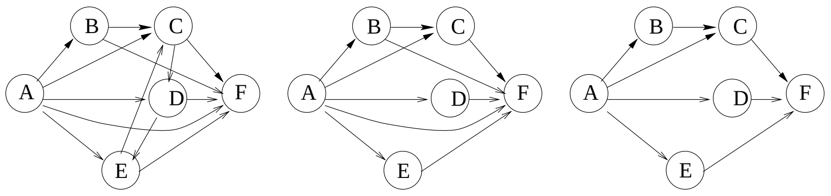

Figure 4: Example 7

at least allows every execution in L and steps 3-4 do not exclude any of them. Steps 5-6 retain only those edges from this graph that are necessary for at least one execution in L.

The running time is dominated by step 5 ($m \gg n$), whose asymptotic time complexity is $O(mn^3)$. $\square$

Example 7 Consider the log $\{ABCF, ACDF, ADEF, AECF\}$. After step 2 of Algorithm 2, the graph G is the first graph in Figure 4. Step 3 does not find any cycle of length 2. There is one strongly connected component, consisting of vertices C, D, E. After step 4, G is the second graph in Figure 4. Some of the edges are removed in step 6, resulting in the last graph in Figure 4.

An open problem In absence of the requirement that each execution of a process contain all of its activities, there may be more than one graph that is conformal with a given log. Consider the log $\{ACF$, $ADCF$, $ABCF$, $ADECF\}$. Both the graphs in Figure 6 are conformal with this log and they have the same number of edges.

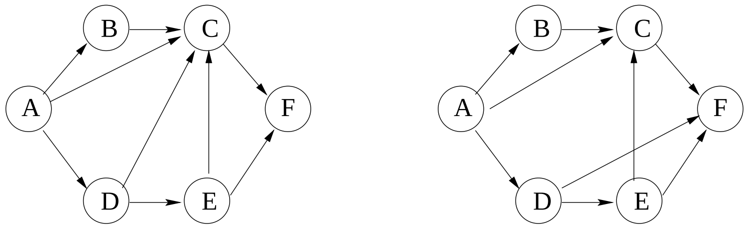

Figure 5: Two conformal graphs for the same log

The difference in the two graphs is that they allow a different set of extraneous executions (executions other than those present in the log). In general, one cannot construct a graph that allows only those executions that are present in a log. A valid goal for a process graph discovery algorithm could be to find a conformal graph that also minimizes extraneous executions. Properly defining the semantics of an extraneous execution and developing a polynomial algorithm for this task is an open, intriguing problem. However, as we will see in Section 8, we did not find this problem to be a major handicap in our experiments.

5 Finding general directed graphs

If the process model graph can have cycles, the previous algorithms break down. The main problem is that we are going to remove legitimate cycles along with cycles created because two activities are independent and have appeared in different order in different executions. An additional problem is that in the case of a directed graph with cycles the transitive reduction operation does not have a unique solution.

A modification of our original approach works however. The main idea is to treat different appearances of the same activity in an execution as two distinct activities.

A cycle in the graph will result in multiple appearances of the same activity in a single process execution. We use labeling to artificially differentiate the different appearances: for example the first appearance of activity A is labeled A_1, the second A_2, and so on. Then Algorithm 2 is used on the new execution log.

The graph so computed contains, for each activity, an equivalent set of vertices that correspond to this activity. In fact, the size of the set is equal to the maximum number that the given activity is present in an execution log.

The final step is to merge the vertices of each equivalent set into one vertex. In doing so, we put an edge in the new graph if there exists an edge between two vertices of different equivalent sets of the original graph.

Algorithm 3 (Cyclic Graphs) *Given a log L of executions of a process, find the dependency graph G.*

1. *Start with the graph $G = (V, E)$, with V being the set of activities of the process and $E = \emptyset$.*

2. *Go through each execution in the log and uniquely identify each activity recorded in the log, thus create a new set of vertices V' and graph $G = (V', E')$.*

3. *For each process execution in L, and for each pair of activities u, v such that u terminates before v starts, add the edge (u, v) to E'. (In practice, steps 1-3 are executed together in one pass over the log.)*

4. *Remove from E' the edges that appear in both directions.*

5. *For each strongly connected component of G', remove from E' all edges between vertices in the same strongly connected component.*

6. *For each process execution present in the log:*

   (a) *Find the induced subgraph of G'.*

   (b) *Compute the transitive reduction of the subgraph.*

   (c) *Mark those edges in E' that are present in the transitive reduction.*

7. *Remove the unmarked edges in E'.*

8. *In the graph so obtained, merge the vertices that correspond to different instances of the same activity in the graph, thus reverting to the original set of vertices V.*

9. *Return the resulting graph.*

Theorem 6 *Given a log of m executions of a process having n activities with each activity repeated at most k times, Algorithm 3 finds a conformal graph in $O(m(kn)^3)$ time.*

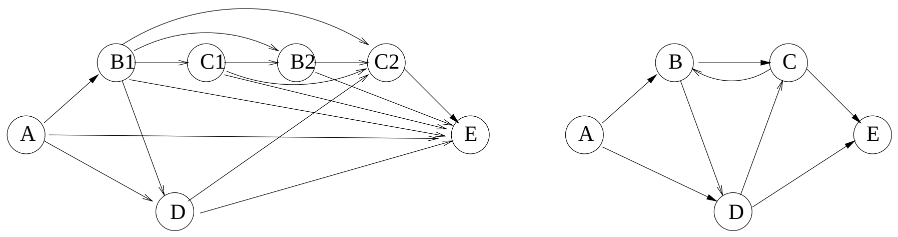

Figure 6: Example 8

Example 8 Consider the log $\{ABDCE, ABDCBCE, ABCBDCE, ADE\}$. The first graph in Figure 6 is the graph computed after step 4 of Algorithm 3. There are no edges between D and $C1$ because D there is an execution where D appears before $C1$, and an execution where it appears after $C1$. Similarly there are no edges between D and $B2$. The edges (A, E), $(B1, B2)$, $(B1, C2)$, $(B1, E)$, $(C1, C2)$ and $(B2, E)$ are removed in step 7. The result after the merging (step 8) is the second graph in Figure 6. This graph shows the cycle consisting of the activities B and C.

6 Noise

A problem we have to consider is noise in the log. This problem can arise because erroneous activities were inserted in the log, or some activities that were executed were not logged, or some activities were reported in out of order time sequence.

We make a slight modification of Algorithm 2 to deal with these kinds of noise. The main change is in step 2 where we add a counter for each edge in E to register how many times this edge appears. Then, we remove all edges with a count below a given threshold T. The rationale is that errors in the logging of activities will happen infrequently. On the other hand, if two activities are independent, then their order of execution is unlikely to be the same in all executions.

One problem here is determining a good value for T. A few extra erroneous executions may change the graph substantially, as the following example illustrates.

Example 9 Assume that the process graph is a chain with vertices A, B, C, D, E. Then there is only one correct execution, namely $ABCDE$. Assume that the log contains $m-k$ correct executions, and k incorrect executions of the form $ADCBE$. If the value of T is set lower than k, then Algorithm 2 will conclude that activities B, C, and D are independent.

Let us assume that activities that must happen in sequence are reported out of sequence with an error rate of ϵ. We assume that $\epsilon < 1/2$. Then, given m executions, the expected number of out of order sequences for a given pair of activities is ϵm. Clearly T must be larger than ϵm. The probability that there are at least T errors, assuming they happen at random, is [CLR90]:

$$P[\text{ more than } T \text{ errors in } m \text{ executions}] = \sum_{i=1}^{T} \left(\begin{array}{c} m \\ i \end{array} \right) \epsilon^i (1-\epsilon)^{n-i} \leq \left(\begin{array}{c} m \\ T \end{array} \right) \epsilon^T$$

The use of T implies that if two independent activities have been executed in a given order at least $m - T$ times, a dependency between them will be added. We assume that activities that are independent in the process graph are executed in random order. Then the probability that they were executed in the same order in at least $m - T$ executions is

$$P[\text{ more than } m-T \text{ executions in same order}] \leq \left(\begin{array}{c} m \\ m-T \end{array} \right) (1/2)^{(m-T)}$$

Then with probability $\delta \geq 1 - \max \left(\left(\begin{array}{c} m \\ T \end{array} \right) \epsilon^T, \left(\begin{array}{c} m \\ m-T \end{array} \right) (1/2)^{(m-T)} \right)$, Algorithm 2 finds the correct dependency.

Note that, if T increases, the probability of wrongly reporting an edge decreases, but the probability of adding an edge increases. If ϵ is approximately known, then we can set $\left(\begin{array}{c} m \\ T \end{array} \right) \epsilon^T = \left(\begin{array}{c} m \\ m-T \end{array} \right) (1/2)^{(m-T)}$, and from there we get $\epsilon^T = (1/2)^{(m-T)}$, and we can obtain the value of T that minimizes the probability that an error occurs.

7 Learning the conditions

The control conditions can be arbitrary Boolean functions of some global process state. To obtain useful information about these functions, additional information about the changes in the global state of the process must be present in the log.

We can however make the simplifying assumption that the control conditions are simple Boolean functions of the output of the activity [LA92].

In this case, the set of output parameters $o(u)$ of a given activity u define the state of the activity. From each execution of an activity u we obtain an example for all functions $f_{(u,v)}$, $(u, v) \in E$. If activity v is also executed in the same process execution, the example is a positive one, otherwise it is a negative.

Formally the training set for $f_{(u,v)}$ is defined as follows. For each execution of the process that u and v appear, the point $(o(u), 1) \in \mathcal{N}^k \times \{0, 1\}$ is inserted. For each execution of the process that u but not v appears, the point $(o(u), 0) \in \mathcal{N}^k \times \{0, 1\}$ is inserted.

We can now use a classifier [WK91] to learn the Boolean fuctions $f_{(v,u)}$. In particular, the use of a decision tree classifier will give a set of simple rules that classify when a given activity is taken or not.

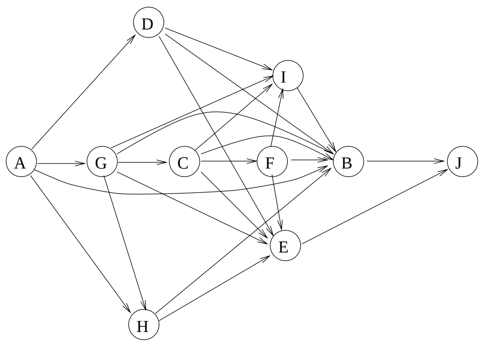

Figure 7: A synthetic process model graph (Graph10) with 10 activities. Typical executions are $ADBEJ$, $AGHEJ$, $ADGHBEJ$, $AGCFIBEJ$.

8 Implementation results

In this section, we present results of applying our algorithm to synthetic datasets as well as logs obtained from a Flowmark installation. Both the synthetic data and the Flowmark logs are lists of event records consisting of the process name, the activity name, the event type, and the timestamp. The experiments were run on a RS/6000 250 workstation.

8.1 Synthetic datasets

To generate a synthetic dataset, we start with a random directed acyclic graph, and using this as a process model graph, log a set of process executions. The order of the activity executions follows the graph dependencies. The START activity is executed first and then all the activities that can be reached directly with one edge are inserted in a list. The next activity to be executed is selected from this list in random order. Once an activity A is logged, it is removed from the list, along with any activity B in the list such that there exists a (B, A) dependency. At the same time A's descendents are added to the list. When the END activity is selected, the process terminates. In this way, not all activities are present in all executions.

Figure 7 gives an example of a random graph of 10 activities (referred to as Graph10) that was used in the experiments. The same graph was generated by Algorithm 2, with 100 random executions consistent with Graph10.

Table 1 summarizes the execution times of the algorithm for graphs of varying number of vertices and with logs having varying number of executions. The physical size of the log was roughly proportional to the number of recorded executions (all executions are not of equal length). For 10,000 executions, the size of the log was 46MB, 62MB, 85MB and 107MB for graphs with 10, 25, 50 and 100 vertices respectively.

For practical graph sizes, the number of executions in the input is the dominant factor in determining the running time of the algorithm. Table 1 shows that the algorithm is fast and scales linearly with the size of the input for a given graph size. It also scales well

Number of executions	Number of vertices			
	10	25	50	100
100	4.6	6.5	9.9	15.9
1000	46.6	64.6	100.4	153.2
10000	393.3	570.6	879.7	1385.1

Table 1: Execution times in seconds (synthetic datasets)

with the size of the graph in the range size that we ran experiments.

Number of vertices		10	25	50	100
Edges Present		24	224	1058	4569
Edges found with executions	100	24	172	791	1638
	1000	24	224	1053	3712
	10000	24	224	1076	4301

Table 2: Number of edges in synthesized and original graphs (synthetic datasets)

Table 2 presents the size of the graphs that our algorithm discovered for each of the experiment reported in Table 1. The graphs our algorithm derived in these experiments were good approximations of the original graphs (checked by programmatically comparing the edge-set of the two graphs). When a graph has a large number of vertices, the log must correspondingly contain a large number of executions to capture the structure of the graph. Therefore, the largest graph was not fully found even with a log of 10000 executions. When the number of vertices was small, the original graphs were recovered even with a small number of executions. In the case of 50 vertices, the algorithm eventually found a supergraph of the original graph. As we noted earlier, in the case when every execution of a process does not contain all the activities, the conformal graph for a given log is not unique. We use heuristics to minimize the number of edges in the graph we find.

8.2 Flowmark datasets

For a sanity check, we also experimented with a set of logs from a Flowmark installation. Currently, Flowmark does not log the input and output parameters to the activities. Hence, we could not learn conditions on the edges. The correctness of the the process model graphs mined was verified with the user. In every case, our algorithm was able to recover the underlying process.

Table 3 sumarizes the characteristics of the datasets and the execution times.

Process Name	Number of vertices	Number of edges	Number of executions	Size of the log	Execution time (seconds)
Upload_and_Notify	7	7	134	792KB	11.5
StressSleep	14	23	160	3685KB	111.7
Pend_Block	6	7	121	505KB	6.3
Local_Swap	12	11	24	463KB	5.7
UWI_Pilot	7	7	134	779KB	11.8

Table 3: Experiments with Flowmark datasets

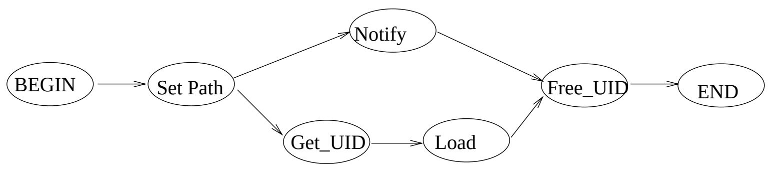

Figure 8: Process model graph for process Upload_and_Notify

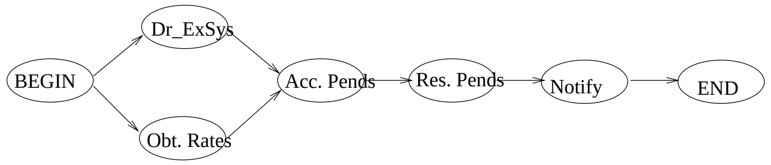

Figure 9: Process model graph for process UWI_Pilot

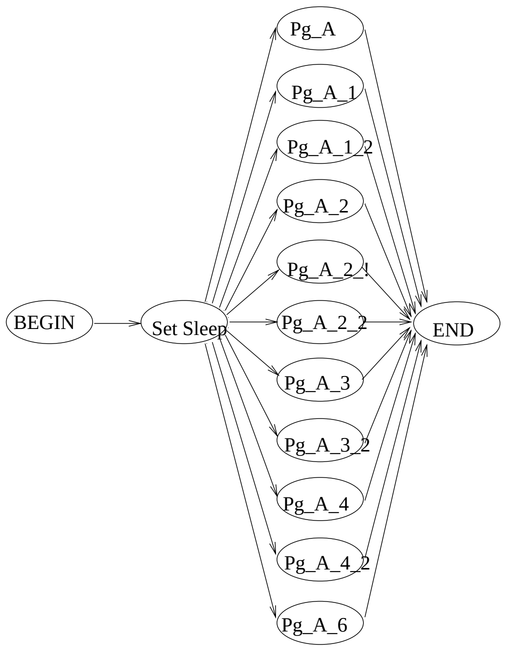

Figure 10: Process model graph for process StressSleep

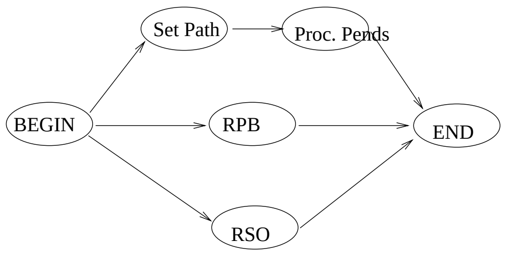

Figure 11: Process model graph for process Pend_Block

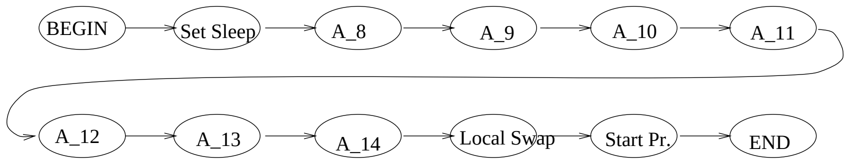

Figure 12: Process model graph for process Local_Swap

9 Summary

We presented a novel aproach to expand the utility of current workflow systems. The technique allows the user to use existing execution logs to model a given business process as a graph. Since this modeling technique is compatible with workflow systems, the algorithm's use can facilitate the introduction of such systems.

In modeling the process as a graph, we generalize the problem of mining sequential patterns [AS95] [MTV95]. The algorithm is still practical, however, because it computes a single graph that conforms with all process executions.

The algorithm has been implemented and tested with both real and synthetic data. The implementation uses Flowmark's model and log conventions [LA92]. The results obtained from these experiments validated the scalability and usability of the proposed algorithm.

References

[AGU72] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal of Computing*, 1(2), 1972.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining Sequential Patterns. In *Proc. of the 11th Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[ASE$^+$96] P. Attie, M. Singh, E.A. Emerson, A. Sheth, and M. Rusinkiewicz. Scheduling workflows by enforcing intertask dependencies. *Distributed Systems Engineering Journal*, 3(4):222–238, December 1996.

[CCPP96] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. In *Proceedings of ER '96*, Springer Verlag, Cottbus, Germany, October 1996.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[CW95] Jonathan E. Cook and Alexander L. Wolf. Automating process discovery through event-data analysis. In *Proc. 17th ICSE*, Seattle, Washington, USA, April 1995.

[CW96] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. Research Report Technical Report CU-CS-819-96, Computer Science Dept., Univ. of Colorado, 1996.

[DS93] U. Dayal and M.-C. Shan. Issues in operation flow management for long-running acivities. *Data Engineering Bulletin*, 16(2):41–44, 1993.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2), 1995.

[GR97] D. Georgakopoulos and Marek Rusinkiewicz. Workflow management — from business process automation to inter-organizational collaboration. In *VLDB-97 Tutorial*, Athens, Greece, August 1997.

[Hol94] D. Hollinsworth. Workflow reference model. Technical report, Workflow Management Coalition, TC00-1003, December 1994.

[Kle91] J. Klein. Advanced rule driven transaction management. In *IEEE COMPCON*, 1991.

[LA92] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, (2), 1992.

[MAGK95] C. Mohan, G. Alonso, R. Gunthor, and M. Kanath. Exotica: A research perspective on workflow management systems. *Data Engineering*, 18(1), March 1995.

[MTV95] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovering frequent episodes in sequences. In *Proc. of the 1st Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Montreal, Canada, August 1995.

[RW92] B. Reinwald and H. Wedekind. Automation of control and data flow in distributed application systems. In *DEXA*, pages 475–481, 1992.

[Sch93] A. L. Scherr. A new approach to business processes. *IBM Systems Journal*, 32(1), 1993.

[WK91] Sholom M. Weiss and Casimir A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems.* Morgan Kaufman, 1991.

A Computing the transitive reduction

We outline an algorithm to compute the transitive reduction of a directed acyclic graph. The fact that the graph we are considering is known to be acyclic allows for a simpler algorithm than the one given in [AGU72].

Lemma 7 ([AGU72]) *Let $G = (V, E)$ be a directed acyclic graph, and let $G' = (V, E')$ be the transitive reduction of G. Then $\forall (u, v) \in E, (u, v) \in G'$ iff there exists no other path from u to v in G.*

Our algorithm is based on this lemma. The algorithm first finds the topological ordering of the directed acyclic graph. We keep two arrays of size $|V|$ for each vertex v. One keeps the descendants of v, and the other the successors of v (that is, the nodes u such that $(v, u) \in E$). Then each vertex is visited in reverse topological order.

Algorithm 4 (TR) *Given a directed acyclic graph $G = (V, E)$, find its transitive reduction.*

```
1. Find a topological ordering of $G$.
2. For each $v$, let the successors of $v$ be $succ(v) = \{u | (v,u) \in E\}$.
3. For each vertex $v$, in reverse topological order:
   (a) Set the descendants of $v$ equal to the union of the descendants of its successors.
   (b) If a successor of $v$ is also a descendant of $v$, remove it from the successors of $v$.
   (c) Add the remaining successors of $v$ to its descendants.
4. Return the graph $(V, \{(v,u) | u \in succ(v)\})$.
```

Theorem 8 *Given a directed acyclic graph $G = (V, E)$, Algorithm 2 computes its transitive reduction in $O(|V||E|)$ time.*

Proof The correctness of the algorithm is straightforward from the discussion above. The running time is $O(|V||E|)$ because we perform two $O(|V|)$ operations for each edge in the graph. □