ELSEVIER

Decision Support Systems 34 (2002) 41–57

Decision Support Systems

www.elsevier.com/locate/dsw

On the discovery of process models from their instances ☆

San-Yih Hwang*, Wan-Shiou Yang

Department of Information Management, National Sun Yat-Sen University, Kaohsiung, 80424 Taiwan

Accepted 30 November 2001

Abstract

A thorough understanding of the way in which existing business processes currently practice is essential from the perspectives of both process reengineering and workflow management. In this paper, we present a framework and algorithms that derive the underlying process model from past executions. The process model employs a directed graph for representing the control dependencies among activities and associates a Boolean function on each edge to indicate the condition under which the edge is to be enabled. By modeling the execution of an activity as an interval, we have developed an algorithm that derives the directed graph in a faster, more accurate manner. This algorithm is further enhanced with a noise handling mechanism to tolerate noise, which frequently occur in the real world. Experimental results show that the proposed algorithm outperforms the existing ones in terms of efficiency and quality. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Process discovery; Business process reengineering; Workflow management; Data mining

1. Introduction

The unprecedented growth of computer and networking technologies has changed the way in which enterprises operate. Organizations that seek to stay competitive in a rapidly changing environment are compelled to incorporate information technologies into many aspects of their business operations, which often call for the radical redesign of current business processes. Such a revolutionary change of business processes is termed *business process reengineering*, abbreviated as BPR. Reports on the successful implementations of BPR effort that achieve major improvement on organizational objectives such as high service quality and low cost can be widely seen in the literature [12,20]. To perform BPR, several sets of guidelines have been proposed, including the five-step approach by Davenport [12], the six-step approach by Furey [13], and AT&T's seven-step approach [18]. Regardless of differences in their subtle details, these guidelines suggest that analysis of existing critical business processes as well as redesign of these processes are two essential BPR tasks. To facilitate these two tasks, a thorough understanding of the way in which the existing business processes currently practice is instrumental. Although organizations typically prescribe how business processes have to be performed, such prescription may not completely reflect the reality due to the following reasons:

(1) Business processes are usually described in a loose manner such that many aspects are left

☆ This work is supported in part by the National Science Council in Taiwan under grant number NSC89-2213-E-110-047.

* Corresponding author.

E-mail addresses: syhwang@mis.nsysu.edu.tw (S.-Y. Hwang), ryang@mis.nsysu.edu.tw (W.-S. Yang).

0167-9236/02/$ - see front matter
PII: S0167-9236(02)00008-8

unspecified. This is especially common in human-oriented processes.
(2) Some parts of the business processes are seldom executed and should be considered as exceptions thereafter.
(3) The real processes are deviated from the preplanned business processes because of environmental change.

One promising approach in improving process efficiency and customers' satisfaction, as advocated by many vendors and BPR experts, is to adopt a workflow management system (WFMS) that automates process executions [23]. A WFMS coordinates the execution of constituent activities as planned, enabling the tracking of ongoing process instances and reporting the statistical figures of processes being executed. However, current WFMSs assume that a precise model of all processes is available, whereas it has been widely recognized that defining a workflow type which totally represents all properties of the underlying business process is a difficult job [12]. Current practices for identifying a process model are usually performed in ad hoc manners, involving numerous meetings and discussions with authorized and knowledgeable persons.

Our primary objective in this paper is to propose a framework and develop algorithms for modeling the existing processes automatically. Specifically, we assume the existence of unstructured executions of a process, called instances. Taking the process instance data as the input, our algorithms will derive the control flow and the associated conditions of the underlying process. Instance data of a process may be collected in various ways. On one hand, in a traditional human-coordinated, document-driven process, instance data can be found in a collection of documents, each of which describes the execution information of a process instance, such as the completion time and the identity of the responsible person for each step involved. In this case, the discovered process model may help ease the introduction of a workflow management system. On the other hand, in an environment where a workflow system has been employed for coordinating process executions, detailed workflow logs are already available electronically (e.g., see Ref. [16] for a list of commercial WFMSs and the log information they provide). In this case, the discovered process model serves as a feedback from the practical process executions and will help the evolution of the current process. In addition, commercial project management tools are also capable of recording some historical information about process executions. Some research prototypes have also been developed to monitor processes in a specific domain, such as software development [5].

1.1. Related work

The research on process discovery traces its origin to grammar discovery in the early 1970s [4]. The goal was to identify the underlying grammar from a finite number of sample strings. Grammars are commonly represented as finite state machines (FSMs). After a correct FSM is identified, it can then be used to tell the correctness of a given input string. More recently, researchers have started to adopt the existing grammar discovery algorithms to the problem of process discovery [9,11]. The idea was to treat an execution of a process as a string of events, each of which represents an execution outcome of an involving activity. With several executions of the same process as the input, these algorithms will be able to synthesize a process definition that best satisfies these historical data. Process definitions were described in the form of FSMs. For example, consider Fig. 1(a) for an example FSM of the program development process [9], which involves three sequential steps: code modification, compilation, and testing. After the code is modified (G), the subsequent compilation is performed and produces the result of either *OK* (I) or *not OK* (H). If the compilation is not okay, the code has to be modified again and the procedure has to be repeated; otherwise, a testing activity is performed. A successful testing (K) ends this process, and a failed testing (J) calls for the repetition of the entire procedure. Fig. 1(b) and (c) show the FSMs discovered from two different algorithms, KTAIL and Markov [9], respectively.

The original FSMs like the one shown in Fig. 1(a) are very easy to perceive and can be converted to an activity-based process model without much difficulty. As a matter of fact, in Fig. 1(a), each state corresponds to the execution of exactly one real-world activity, and its outgoing transitions represent the possible execution results. However, in a derived FSM, such as that in Fig. 1(b) or (c), a state may not have its clear semantic meaning, and an execution outcome may appear

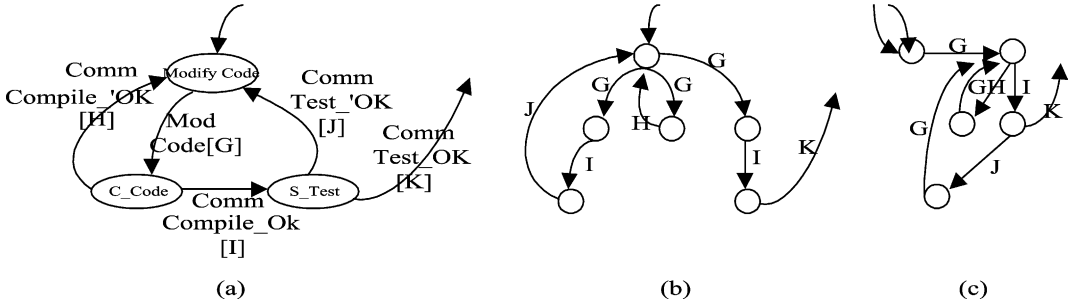

Fig. 1. (a) The underlying process definition, (b) an FSM discovered by KTAIL, (c) an FSM discovered by Markov.

in the transitions of multiple states. While a derived FSM can be used to verify the validity of a given execution, it would be difficult to infer the underlying process model from this FSM.

Realizing the inadequacy of FSM as a process model, Agrawal et al. [2] took a different approach, which produced a process definition in the form of a directed graph, with vertices being the set of constituent activities and edges being the set of control dependencies among them. As the directed graph is the main process model adopted by most workflow systems [27], the output of their algorithm can be readily employed in a commercial workflow system. The idea of their algorithm is quite simple. Each activity in a process execution is represented as an instantaneous event, say the start or end of it. With this simplification, an execution of a process can be represented as a sequence of activities. Their algorithm then tries to locate all possible dependencies from each execution instance. For example, the execution {ABDCE} implies the following dependencies: $\{A \rightarrow B,\ A \rightarrow D,\ A \rightarrow C,\ A \rightarrow E,\ B \rightarrow D,\ B \rightarrow C,\ B \rightarrow E,\ D \rightarrow C,\ D \rightarrow E,\ C \rightarrow E\}$. Dependencies derived from all process executions are first unionized, but those that appear in both directions are then removed. That is, for any pair of activities A and B, if both dependencies $A \rightarrow B$ and $B \rightarrow A$ are found from some (but different) executions, then the two activities are pronounced independent. Transitive reduction is finally performed for each process instance to induce a minimal graph. This final step dominates the whole process in terms of running time. The total time complexity is $O(N \times n^3)$, where N is number of instances and n is number of activities of a process.

Their algorithm uses a simple mechanism for handling cycles. Appearances of the same activity in an execution are first treated as distinct activities. After the algorithm described above is performed for identifying all transitions, vertices of the same activity are finally merged as a single one. However, this simple method may unnecessarily remove some dependencies in some cases. Consider the example process shown in Fig. 2, in which a cycle exists between activities B and E, and an or-split branch follows activity B. In one process instance, suppose B and C are executed 10 times and 4 times, respectively. In another instance, B and C are performed 10 times and 3 times, respectively. Let B_{10} denote the tenth occurrence of activity B and C_3 the third occurrence of activity C in a process instance. From the first process instance, $C_3 \rightarrow B_{10}$ is inferred, while $B_{10} \rightarrow C_3$ is induced from the second one. As a result, B_{10} and C_3 are considered independent. In the final transitive reduction step, spurious transitions (e.g., $B \rightarrow E$) would remain due to the incorrect conclusion on the relationship between B and C.

1.2. Contribution

We present a novel approach to solve the problem of process discovery. Just as Agrawal et al.'s algorithm [2] uses directed graphs as the process model for easier interpretation, so does our approach. Our ap-

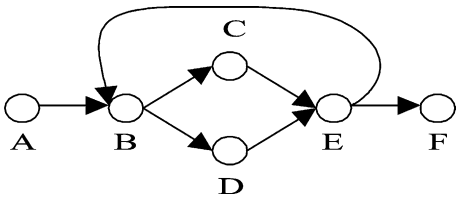

Fig. 2. An example process.

proach not only handles cycles correctly but also produces more information in a faster manner. Specifically, the directed graph derived by our approach embodies control dependencies between activities as well as the conditions pertaining to them. These two major components together make our work a complete framework for process discovery.

Unlike other work on process discovery that makes use of only an instantaneous event for each activity instance, our work assumes an interval for the execution of an activity instance. For those processes whose executions are either automated (via WFMSs) or monitored (through some monitoring tools), the execution intervals of their constituent activities are already available. However, manual processes may provide merely partial information such as the completion times of their activities. In this case, the starting time of an activity instance can still be inferred by subtracting the anticipated execution duration from its completion time. Consequently, the derived intervals may contain noise, which must be screened out by some noise handling mechanism, as will be discussed in Section 5. With execution intervals of activities as the input, we shall be able to develop an algorithm that produces a process model closer to the real one in shorter time.

This paper is structured as follows. Section 2 describes the process model and the kind of data we assume to be available. Section 3 presents two algorithms that derive the control dependencies and the associated conditions, respectively. Section 4 compares the performance of the proposed algorithm with that of the algorithm proposed by Agrawal et al. [2] by applying synthetic datasets to both algorithms. In Section 5, we consider a practical environment in which noisy data is present. Strategies for dealing with noise are discussed. Finally, Section 6 summarizes the paper and provides potential directions for future research.

2. Process model

Various models have been proposed in the literature to capture the various requirements about processes. Some models provide solid theoretical foundations and can be used for correctness verification and analysis. Petri-Net [1], State Diagram [29], and temporal logic-based process models [3] fall into this category. These models, though theoretically sound, are not easy to understand from the viewpoint of enterprises' users. Some focus on the interaction between customer and producer and try to captures their behavior. A typical one is the Action Model [21]. However, process models of this kind are loose in the sense that interactions are defined at a conceptual level and physical interactions are not precisely captured. The most popular process models, as adopted by most commercial WFMSs, are based on directed graphs. Though different vendors may provide slightly different process models, a reference model that covers features provided by most vendors is available in [15]. This model, defined by the Workflow Management Coalition (WfMC), intends to serve as the interchange means between process data produced by different WFMSs. The process model assumed in this paper, as will be briefly described in the following, is based on the reference model proposed by WfMC.

Enterprises typically have many business processes. A business process comprises a set of activities and the interdependencies between them. Each activity is a logical unit of work, performed by either a human or a computer program. Each activation of a business process generates a process instance, which will achieve a prescribed goal when it terminates successfully. A process can be modeled as a structure similar to a directed graph, with vertices being the constituent activities and the edges being the potential control flow between activities. Moreover, each edge is associated with a Boolean function, which determines whether this control transition will be enabled during a process execution.

We distinguish three types of transition structures.

(1) Sequential transitions: activities are executed sequentially. In other words, an activity is followed by exactly one other activity, and the Boolean function associated to the transition is simply "true."

(2) Parallel split: following an activity, a number of activities are executed possibly concurrently. The respective Boolean function pertaining to each edge is independently evaluated.

(3) Loops: a loop specifies a set of activities to be repeatedly executed. The ending activity of a loop has two outgoing edges. One connects to the starting activity of the loop and the other leaves. Boolean functions pertaining to these two outgoing edges must be exclusive and determine whether the loop is exe-

Table 1

ActInsNo	Start_time	End_time	Var_1	$Value_1$	...	Var_n	$Value_n$

cuted again or not. Furthermore, activities of the loop body may be connected by sequential or parallel transitions, or even form a nested loop.

Note that in WfMC's process mode, another type of transition structure, called exclusive-or (XOR) split is defined [17]. An XOR split is similar to a parallel split, except that one more parameter is needed: a list of outgoing transitions that defines the order in which conditions on transitions are evaluated. When a condition of a transition is evaluated to be true, the following transitions in the sequence specified by the list are ignored to guarantee an exclusive-or transition. Nevertheless, an XOR split can be emulated by embedding conditions on the transitions of a parallel split. Therefore, to simplify the discussion in the remainder of the paper, we shall consider only the three transition structures described above.

An activity execution spans a temporally extended period and results in the occurrence of several events. An event is an instantaneous action, and no two events occur at the same time. Within the context of this work, we consider three types of events: (1) start_event, which denotes the starting of an activity instance, (2) end_event, which indicates the ending of an activity instance, and (3) write_event, which signifies the data writing action performed by an activity instance.

Both start_event and end_event are characterized by a 3-tuple (Action, ActInsNo, TS), where Action distinguishes the start or end event, ActInsNo uniquely identifies an activity instance, and TS is the timestamp at which this event occurs. Similarly, a write event can be represented by a 5-tuple (Write, ActInsNo, Var, Value, TS), where Var is the name of a data variable and Value records the value written to it. By combining all the event information about the same activity instance, we can summarize an execution of an activity as a record shown in Table 1. In the following section, we will develop algorithms to extract the underlying process model from the information such as that shown in Table 1.

3. The algorithms

In this section, we will present two algorithms that derive the process model digraph and the conditions pertaining to edges, respectively.

3.1. Deriving the process model digraph

Activities involved in a process instance can be visualized as a set of intervals. For example, suppose a process constitutes five activities *A*, *B*, *C*, *D*, and *E*, and their control dependencies are shown in Fig. 3(a). Using Time as the horizontal axis, Fig. 3(b) shows the interval diagram of a possible execution.

For any pair of activities of the same process instance, the relationship between their temporal durations can be classified into two kinds: *disjoint* and *overlapped*. Two activity instances are said to be disjoint if either one starts after the end of the other. They are said to be overlapped if they are *not* disjoint. Obviously, if there is a path from an activity *X* to another *Y* in the underlying process model digraph, *X* and *Y* must be disjoint in any execution in which they both appear, but the opposite is not necessarily true. Our goal is to find all the disjoint activity pairs (*X*, *Y*) such that *Y* immediately follows *X* in some process instances because these pairs have the potential of being edges in the underlying process model digraph.

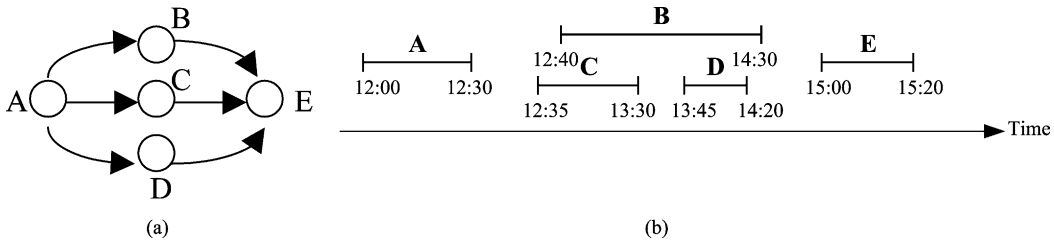

Fig. 3. (a) An example process, (b) a possible process instance.

Definition 1. A process instance of a process model digraph (V, E) is a set of 3-tuples (V_i, st, et), where $V_i \in V$, and st and et are timestamps representing the starting time and ending time of V_i, respectively.

Definition 2. An activity V_i is said to be followed by V_j in a process instance I, denoted as *Followed*(V_i, V_j, I), if there exist two tuples (V_i, st_i, et_i) and (V_j, st_j, et_j) in I such that $et_i < st_j$.

Definition 3. The activity set of a process instance I, denoted as *ActSet*(I), is the set of activities that appear in I.

Definition 4. An activity V_i is said to be directly followed by V_j in a process instance I if *Followed*(V_i, V_j, I) and $\neg\exists V_k \in ActSet(I)\text{-}\{V_i, V_j\}$ such that *Followed*(V_i, V_k, I) and *Followed*(V_k, V_j, I) both hold. *DirFollowedSet*(I) denotes the set of ordered pairs (V_i, V_j) such that V_i is directly followed by V_j in I.

Obviously, *DirFollowedSet*(I) for an instance I represents (part of) the potential edges in the process model digraph. However, *DirFollowedSet*(I) may contain spurious transitions. For example, from the process instance shown in Fig. 3(b), it is determined that C is directly followed by D, while we can see from Fig. 3(a) that C and D are actually independent. Besides, some transitions may be missing from *DirFollowedSet*(I), even though the pertaining activities both appear in I. Still, consider the process shown in Fig. 3(a). While there exists an edge $C \rightarrow E$, C is not directly followed by E in the instance shown in Fig. 3(b). To find the missing transitions, as well to eliminate the spurious ones, we need to take into account all the available process instances.

Definition 5. The overlapped set of a process instance I, denoted as *OverlappedSet*(I), is the set of activity pairs $\{V_i, V_j\}$ such that both V_i and V_j appear in I, and V_i and V_j incur overlapped execution durations.

Property 1. *Consider two instances I_i and I_j of the same process. For any activity pair V_i and V_j, if $(V_i, V_j) \in DirFollowedSet(I_i)$ and $\{V_i, V_j\} \in OverlappedSet(I_j)$, then $(V_i, V_j) \notin E$, where E is the set of edges in the process model digraph.*[1]

[1] This argument is valid only when every instance is correctly recorded. We make this assumption at this moment to simplify the discussion. This assumption will be dropped later, and approaches to handling noise will be discussed in Section 5.

In other words, we can use the overlapped set of one instance to prune the spurious transitions appeared in the directly followed set of another. This observation leads to the following definition.

Definition 6. The directly followed set of a set of process instances S, denoted as *DirFollowedSet*(S), is the set of ordered activity pairs (V_i, V_j) such that $\exists I \in S$, $(V_i, V_j) \in DirFollowedSet(I)$ and $\forall I \in S$, $(V_i, V_j) \notin OverlappedSet(I)$. That is, $DirFollowedSet(S) = \cup_{I \in S} DirFollowedSet(I) - \cup_{I \in S} OverlappedSet(I)$.

Note that we overload the notation *DirFollowedSet*() to represent the directly followed set of both a single instance and a set of instances. With the sufficient number of process instances, *DirFollowedSet*(S) should be close to the set of real transitions.

We are still left with the problem of how to efficiently compute *DirFollowedSet*(S) for a given set of process instances S. To do so, we need to first compute the directly followed set for each instance I in S. The algorithm, named *CompDirFollowedSetI*(), serves this purpose and is listed below.

```
CompDirFollowedSetI(process-instance: I): set of activity pairs
{
 DirFollowedSetI = ∅;
 For (each activity a in I listed in descending order on their ending time) {
   direct-follow = a;
   do {
          direct-follow= next activity of direct-follow in I;
   } while direct-follow.et < a.st;
   end-limit = direct-follow.st;
   do {
          DirFollowedSetI = DirFollowedSetI ∪ {(direct-follow, a)};
          if (direct-follow.st > end-limit) end-limit=direct-follow.st;
          direct-follow= next activity of direct-follow in I;
   } while [(direct-follow.et < end-limit) or (direct-follow = ∅)]
 } /* end of For */
 return DirFollowedSetI;
} /* end of CompDirFollowedSetI */
```

We use an example to illustrate how *CompDirFollowedSetI*(I) works. Again consider the process instance shown in Fig. 3(b). Activities are examined in the reverse order of their ending time, i.e., E, B, D, C, and A. We first examine E and found B to be the first that is directly followed (*direct-follow*) by E. Thus, a pair (B, E) is inserted into *DirFollowedSetI*. Following that, we can see that D is also directly followed by E but C does not, because D starts before C ends. Therefore, *DirFollowedSetI* becomes {(B, E), (D, E)} at this point. Next, we examine B and find A to be the only one that is directly followed. Similarly,

when we subsequently check D and C, C and A, respectively, are found to be the ones that are directly followed. Thus, *DirFollowedSetI* finally becomes $\{(B, E), (D, E), (A, B), (C, D), (A, C)\}$. Computing the overlapped set of an instance I is easier. The algorithm, named *CompOverlappedSetI*(), is listed below.

```
CompOverlappedSetI(process-instance: I): set of activity pairs
{
 OverlappedSetI = ∅;
 For (each activity a in I listed in descending order on their ending time) {
    overlap = a;
    do {
       overlap= next activity of overlap in I;
       if (overlap.et > a.st) OverlappedSetI = OverlappedSetI∪{(overlap,a), (a,overlap)};
    } while overlap.et < a.st;
 } /* end of For */
 return OverlappedSetI;
} /* end of CompOverlappedSetI */
```

Again, consider the same example shown in Fig. 3(b), *OverlappedSetI* finally becomes $\{(B, D), (D, B), (B, C), (C, B)\}$.

Property 2. *The running time of CompDirFollowedSetI(I) (CompOverlappedSetI(I)) is O(m), where m is the number of activity pairs in DirFollowedSet(I). Note that m is equal to O(n) in the best case and $O(n^2)$ in the worst one, where n is the number of activity instances in I.*

Now computing *DirFollowedSet*(S) for a set of process instances, S becomes straightforward. The pseudo code is listed below.

```
CompDirFollowedSet(set of process-instance: S)
{
    DirFollowedSet = OverlappedSet = ∅;
    for (each process instance I in S) {
        DirFollowedSet = DirFollowedSet ∪ CompDirFollowedSetI(I);
        OverlappedSet= OverlappedSet ∪CompOverlappedSetI(I);
    }
    return DirFollowedSet–OverlappedSet;
}
```

Property 3. *The running time of CompDirFollowedSet(S) is $O(\max(N \times m, n^2))$, where N is the number of process instances in S, m is the maximum number of activity pairs in DirFollowedSet(I), I∈S, and n is the total number of activities. Note that $O(N \times m)$ is the total time for computing DirFollowedSet and OverlappedSet in the algorithm, and $O(n^2)$ is that for combining the two for the final result.*

For a process model digraph that contains cycles, our algorithm can be directly applied. At this time, the same activity may appear multiple times in a single process execution. Unlike Agrawal et al.'s algorithm [2], in computing the directly followed set for a given instance, we treat various occurrences of the same activity as a single one. Consider the process model digraph shown in Fig. 2. Fig. 4(a) shows three example process instances, and Fig. 4(b) shows their corresponding directly followed sets (*DirFollowedSet*(I)). By unionizing them and then pruning the spurious transitions using the overlapped sets, our algorithm will return the result exactly the same as that shown in Fig. 2.

The algorithm we have described so far only makes use of the starting time and ending time of each activity instance. The data written by activity instance can be used for deriving conditions associated to the edges discovered by the algorithm presented here. The following subsection is devoted to the discussion on how this can be achieved.

3.2. Deriving control conditions

The objective of this subsection is to find the control conditions on the transitions obtained from the algorithm described in the previous subsection. To simplify the discussion, for now, we assume that the condition on each outgoing edge of an activity is a function of only the variable values written by the activity. Later, we will discuss how to modify our proposed algorithm if this assumption has to be dropped.

Suppose an activity A has more than one outgoing edge, forming a transition structure of a parallel split or a loop. To find the condition associated to an outgoing edge $A \rightarrow B$, we first identify two sets of process instances $S1$ and $S2$, such that each instance in $S1$ involves both A and B and each one in $S2$ involves only A. From $S1$ and $S2$, we form two lists, $L1$ and $L2$ of variable values written by A. While each entry in $L1$ represents a set of data values that enables B after A completes, each entry in $L2$ indicates the case when B is *not* followed after A completes. We can then apply some classification technique to derive the condition associated to the edge $A \rightarrow B$.

We will use an example to illustrate this approach. Suppose there are three outgoing edges of A: $A \rightarrow B$, $A \rightarrow C$, and $A \rightarrow D$, and A writes only to the variable $V1$. Table 2 lists 10 process instances that involve A. It can be seen that every instance in $\{I_1 \ldots I_4\}$ enables B, but none of the instances $\{I_5 \ldots I_{10}\}$ does. The condition pertaining to $A \rightarrow B$ can then be derived by

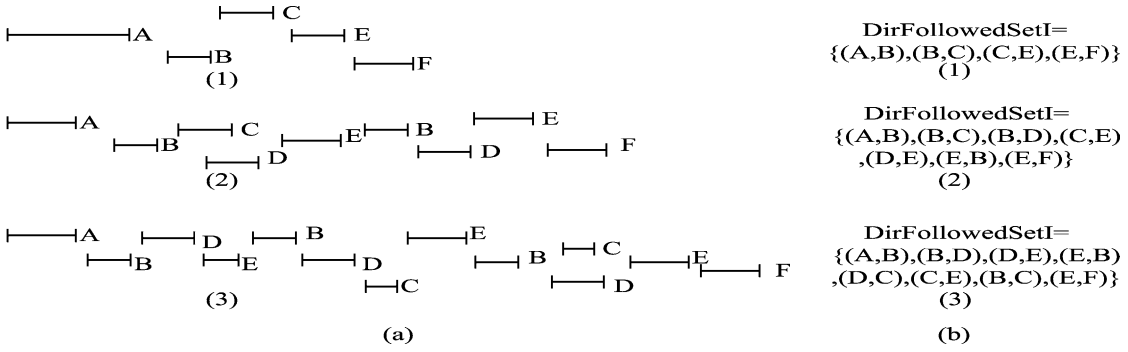

Fig. 4. (a) Three process instances, (b) the directly followed sets.

applying some classification technique on the output values of $V1$.

Many classification techniques are available in the literature and even in software packages. We can roughly classify them into four categories according to the formats of derived classification models, namely decision tree (e.g., ID3 and its descendants [24]), decision rule (e.g., AQ family [22], and CN2 [7]), discrimination analysis [19] and neural network approach (e.g., backpropagation neural network [25]). The decision tree approach induces a decision tree that describes the classification model between input attributes and decision outcomes, and the decision rule approach discovers a set of decision rules, ordered or unordered, as its classification model. The discrimination analysis approach derives linear combination functions of input attributes under normal distribution and equal dispersion assumptions. The last approach, neural network, produces an appropriate set of weighted links according to a predetermined network topology that differentiates decision outcomes based on input attribute values.

Table 2
An example set of process instances

Instances	$V1$	Executed activities
I_1	95	B
I_2	85	B
I_3	75	B
I_4	65	B, C
I_5	55	C
I_6	45	C
I_7	35	C, D
I_8	25	D
I_9	15	D
I_{10}	5	D

The neural network approach is known for its noise-tolerance and fault-resistance. However, being a holistic approach, the neural network approach suffers from its inability to produce interpretable knowledge [26]. The discrimination analysis is a math-based method, however, it is not easy to fulfill all conditions in real situations [26]. Hence, these two methods were not considered in this research. The decision tree or decision rule approach is capable of generating interpretable knowledge in the form of decision tree or rules. According to the experimental results conducted by Clark and Boswell [6], CN2 (a decision rule approach) significantly outperformed C4.5 (a decision tree approach) in predictive accuracy at the 95% confidence level. Thus, considering the research's purpose and generalization, CN2 was adopted as the classification technique in this research. In some cases, certainly, a practitioner could employ another classification technique based on the concrete situations of data. Table 3 shows the classification result by applying CN2 to the example data. It clearly indicates that the condition pertaining to the edge $A \rightarrow B$ is indeed $V1 > 60$.

The previous description of our approach is based on the assumption that the condition pertaining to an edge is determined solely by the output of the source activity. While we find this assumption valid in most cases, our proposed approach can be applied to a more general case. At this time, when it comes to determine the condition pertaining to an edge, all the variables

Table 3
Classification result by CN2

```
**EXAMPLE FILE**          *-----------------------*
%V1 Cond_B                | UN-ORDERED RULE LIST  |
 95   Y;                  *-----------------------*
 85   Y;
 75   Y;                  IF   v1 > 60.00
 65   Y;                  THEN  condition = Y  [4 0]
 55   N;
 45   N;                  IF   v1 < 60.00
 35   N;                  THEN  condition = N  [0 6]
 25   N;
 15   N;             (DEFAULT) condition = N  [4 6]
 5    N;
```

that are written by the activities executed before the source have to be taken into account. The price to pay, of course, is the longer running time.

4. Performance evaluation

To make a sensible performance comparison, we considered only algorithms that adopt the same process model, i.e., directed graph. Specifically, we compared the efficiency and the quality of the output result of our algorithm with that proposed by Agrawal et al. [2]. Since Ref. [2] does not address the problem of deriving control conditions, in the following, only the directed graph derivation algorithms are compared.

In terms of efficiency, recall from Property 3 that our directed graph derivation algorithm takes O(max $(N \times m, n^2)$) running time, where N is the number of process instances, m is the maximum size of the directly followed set for each instance, and n is the total number of activities. It has been shown that the algorithm proposed in Ref. [2] takes $O(N \times n^3)$. As we discussed in Property 2, m is equal to $O(n^2)$ in the worst case. We thus conclude that our algorithm is better than that proposed in Ref. [2] in terms of efficiency.

To compare the output result of the two algorithms, we considered two measures: *precision* and *recall*.

Definition 7. The *recall* of an algorithm X is the ratio of the number of correct transitions returned by X to the total number of correct transitions. The *precision* of X is the ratio of the number of correct transitions returned by X to the total number of transitions returned by X.

These two measures can be graphically illustrated by Fig. 5, where $A \cup B$ is the set of transitions found by the target algorithm and $B \cup C$ is the set of correct transitions. In this case, precision of the algorithm is $|B|/|A \cup B|$, and its recall is $|B|/|B \cup C|$.

4.1. General experiment information

The experiments were conducted by applying synthetic datasets to both algorithms and comparing the precisions and recalls computed from their output. Table 4 summarizes the parameters and their settings used in our experiments. Three process model digraphs, as shown in Fig. 6(a)(b) and (c), were considered. These digraphs represent three different types of graph structures and were extracted from real applications (e.g., see Refs. [8,14,15]). Fig. 6(a) (*no-loop*) features a manufacture process that consists of only sequential and parallel transition structures. Fig. 6(b) (*simple-loop*) is an insurance process that involves several simple loops. Fig. 6(c) (*composite-loop*) captures a survey process that includes a composite loop (i.e., nested loop). For each process model digraph, a number of process instances were randomly generated. When an edge was enabled, the destination activity was assumed to wait for W units of time before executing, where W was uniformly distributed on (1, 20). Then it executed for X units of time, and again X was uniformly distributed on (1, 40).[2] After that, if the activity happened to be the starting activity of a parallel split, each outgoing edge had a 0.75 probability of being enabled. The same procedure was then repeated until the final activity was reached. We varied two parameters in our experiments: G and N, prefixed by "$*$" in Table 4, denoting the structure of the underlying process model digraph and the number of process instances, respectively. Varying G allowed us to examine how the two algorithms performed for different types of process structures, and N tested the algorithms' sensitivity to the number of incoming process instances. Three experiments were conducted based on these three process model digraphs. In each experiment, we

[2] Both W and X were set to be no less than 1 because we assume no two events occur at the same time.

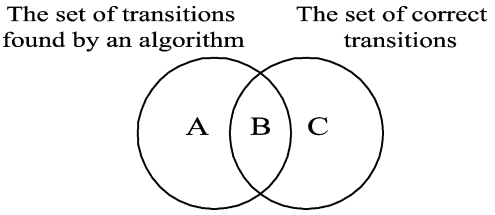

Fig. 5. A graphical representation.

varied N to test the algorithms' sensitivity to the size of instance data. Each experiment was repeated several times via different random seeds to obtain reliable results, and the following figures show only the mean values of the measures.

4.2. Experimental results

Fig. 7 shows the recalls and precisions of the two algorithms under the synthetic datasets generated from the no-loop process. As expected, recall improves as the number of instances increases. Besides, our algorithm yields better recall with fewer number of instances, but both algorithms find the correct set of transitions at approximately the same number of instances. However, in terms of precision, the curves are unstable with smaller number of instances. This is because the underlying directed graph includes a parallel split. When the number of instances is increased, more correct transitions as well as more spurious transitions will be derived. However, our algorithm is still superior in both recall and precision under such a circumstance.

Fig. 8 shows the recalls and precisions of the two algorithms under the synthetic datasets generated from the simple-loop process. In this case, both algorithms perform equally well. This is because simple-loop contains no parallel constructs at all, and thus modeling an activity execution as an interval does not provide more information than modeling it just as a single event. Besides, precision is equal to 1 even for a single instance. This is understandable because simple-loop involves only XOR Split, and thus, every induced transition must be correct.

Fig. 9 shows the recalls and precisions of the two algorithms under the synthetic datasets generated from the composite-loop process. Again, both algorithms perform equally in terms of recall. However, an interesting point is that Agrawal et al.'s algorithm [2] never achieves precision = 1 and recall = 1 at the same time, even for a large number of instances. This is because, as mentioned in Section 1, Agrawal et al.'s algorithm may not handle cycles correctly when branches occur within cycles. Our algorithm remedies this problem and performs well in this case.

Overall, we conclude that the proposed algorithm always return results of equal or better quality (in terms of both recall and precision) with the same number of process instances. Besides, unlike Agrawal et al.'s algorithm, our algorithm is able to find the correct process model digraph under all the circumstances considered by our experiments when the number of process instances is sufficiently large.

5. Noise

Until now, we assume each collected process instance conforms to the underlying process model digraph. However, this assumption may not be valid in the real world, and there is a possibility for the occurrence of noise. Noise may arise, for example, because some executed activities were not collected, the timestamps with events were mistakenly recorded, or some exceptions whose handling requires a deviation from normal processing order just happened. Therefore, we need a way to screen out the noisy data.

5.1. An approach to handling noise

For any pair of activities A and B involved in a process instance, there are four possible relationships in their execution durations: (a) A immediately followed by B, (b) B immediately followed by A, (c) A overlaps B, and (d) A and B are neither overlapped nor immediately followed. Note that activity pairs of

Table 4
Parameters and settings

Parameter	Meaning	Setting
W	activity waiting time	uniform on (1, 20)
X	activity executing time	uniform on (1, 40)
$*G$	underlying process model digraph	no-loop, simple-loop, composite-loop
$*N$	number of process instances	1... 50

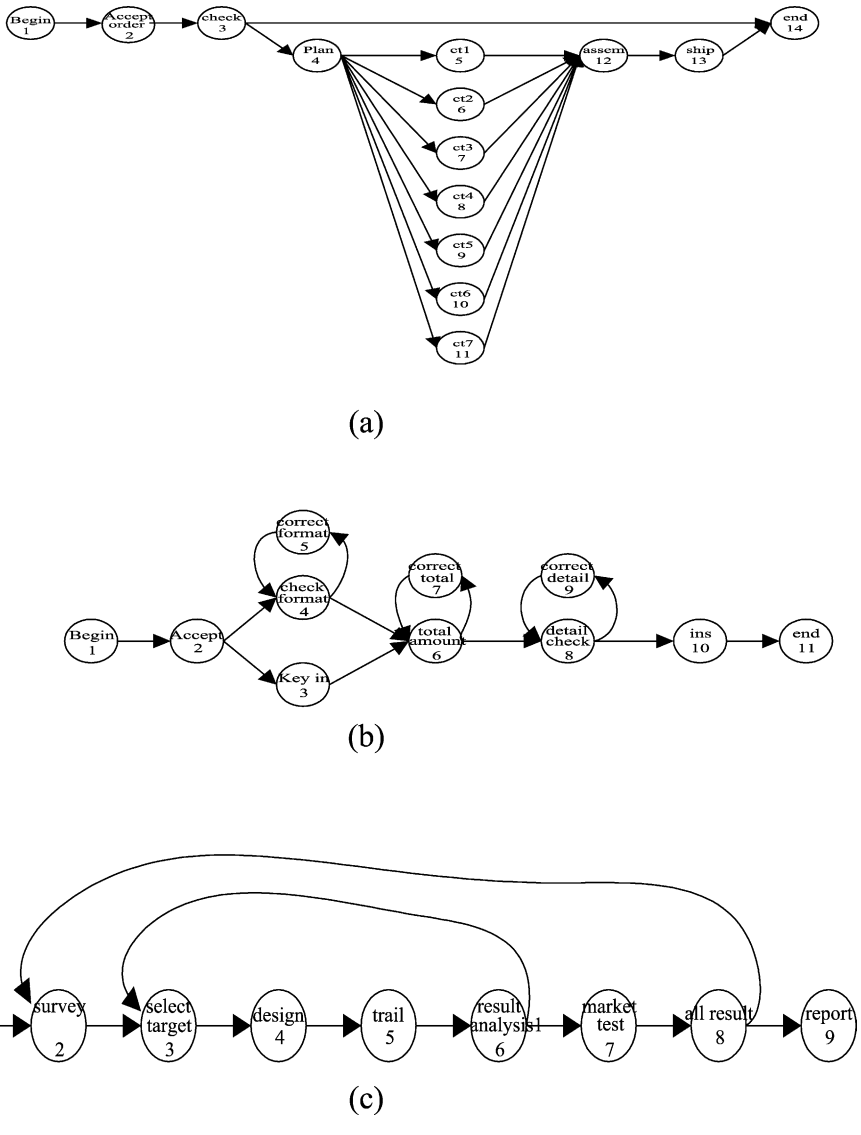

Fig. 6. (a) The no-loop directed graph, (b) the simple-loop directed graph, (c) the composite-loop directed graph.

relationships (a) and (b) must appear in *DirFollowedSet*, and activity pairs of relationship (c) are recorded in *OverlappedSet*. All activity pairs that are not recorded in either *DirFollowedSet* or *OverlappedSet* are of relationship (d). Due to the presence of noise, we can no longer conclude that transitions appeared in *DirFollowedSet* but not in *OverlappedSet* are edges in the underlying process model digraph. Further analysis is required.

For each pair of activities A and B, consider the following three cases in the underlying process model digraph:

(1) There exists an edge connecting A and B: without losing generality, let us assume the edge to be $A \rightarrow B$. In this case, (A, B)'s appear as the relationships (b), (c), and (d) are noise, and the total number of their occurrences should be less than T. Obviously, the error probability in this case is a function of T.

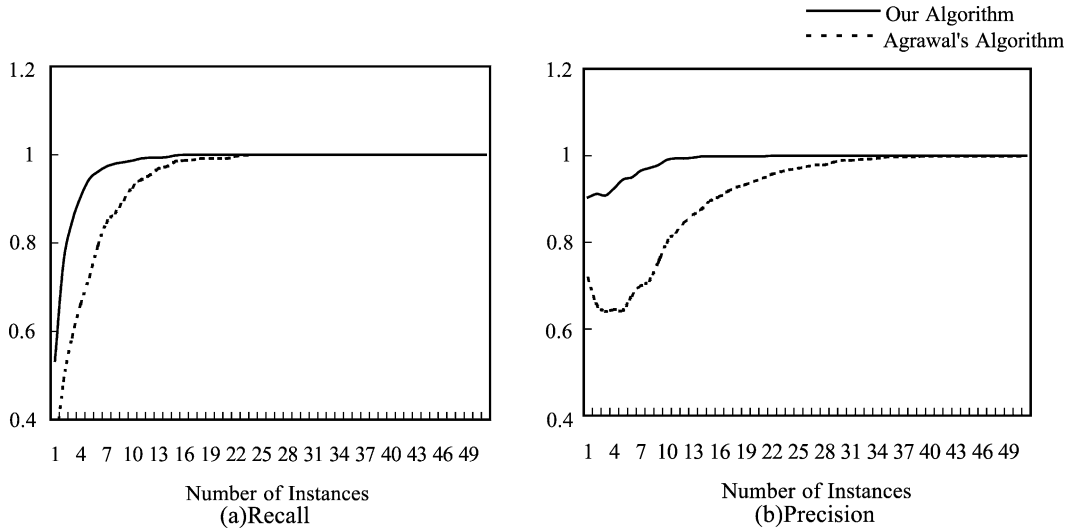

Fig. 7. (a) Recalls of the two algorithms under no-loop process, (b) precisions of the two algorithms under no-loop process.

When T is specified to be small, noisy data could remain and thus the error probability is higher. Specifically, the error probability in this case can be bounded by a value P_3 as follows:

$$P_r(A \text{ is not directly followed by } B) \leq \sum_{x=T}^{k} \binom{k}{x} \varepsilon^x (1-\varepsilon)^{k-x} \leq \binom{k}{T} \varepsilon^T = P_3$$

[10], where k is the number of instances in which activity A and activity B both exist, and ε is the noise probability of a instance.

(2) A and B are transitively ordered: in such a case, activity pairs of relationships (a), (b), and (c) are considered noise and their total number should be less than T. In other words, error occur when the number of these noisy activity pairs is no less than T. Again, the error probability in this case is also a function of T. We can calculate its upper bound P_1 as follows:

$$P_r(A \text{ is not transitively followed by } B) \leq \sum_{x=T}^{k} \binom{k}{x} \varepsilon^x (1-\varepsilon)^{k-x} \leq \binom{k}{T} \varepsilon^T = P_1.$$

(3) A and B are independent: that is, A and B are activities following the same activity through a parallel split. In such a case, all four relationships described above can possibly occur, and the approach we used in the previous two cases can no longer apply! So just as we used overlapping activity pairs to prune out the spurious adjacent activity pairs in the noise-free case, we assumed the overlapping occurrences of (A, B) must reach a significant number T in a noise-prone environment. In other words, the number of activity pairs of relationships (a), (b), and (d) should be no greater than $K - T$. Let θ denote the probability that A and B are found *not* overlapped in a process instance. The error probability can be bounded by P_2 as shown below:

$$P_r(A \text{ and } B \text{ are not overlapped}) = \sum_{x=k-T+1}^{k} \binom{k}{x} \theta^x (1-\theta)^{k-x} \leq \binom{k}{k-T} \theta^{k-T} = P_2.$$

θ can be calculated once the probability distributions of the waiting time and execution time of A and B are both known. Let W_A and W_B be the random variables of the waiting time of activity A and activity B, respectively, and let X_A and X_B, respectively be the random variables of their execution time. Furthermore, the probability density functions of W_A, W_B, X_A and X_B are denoted as f_{WA}, f_{WB}, f_{XA}, and f_{XB},

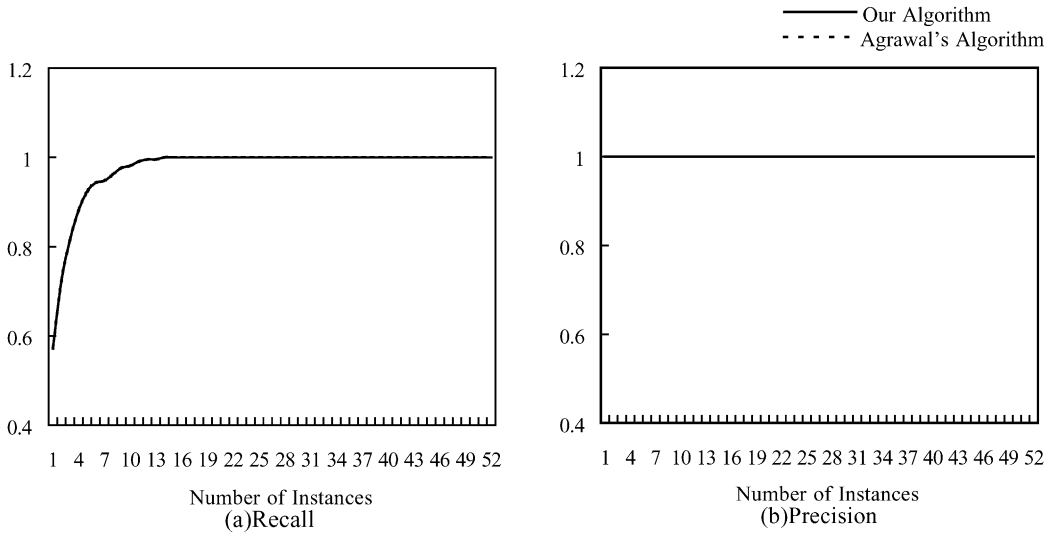

Fig. 8. (a) Recalls of the two algorithms under simple-loop process, (b) precisions of the two algorithms under simple-loop process.

respectively. Thus, $\theta = Pr(W_A + X_A > W_B) + Pr(W_B + X_B > W_A)$, and can be calculated as follows:

$$\theta = Pr(W_A + X_B < W_B) + Pr(W_B + X_B < W_A)$$
$$= 2\int_0^{\infty}\int_0^{W_B}\int_0^{W_B - W_A} f_{WA} f_{XA} f_{WB} \mathrm{d}X_A \mathrm{d}W_A \mathrm{d}W_B.$$

An ideal goal is to set the value of T so that P_1, P_2, and P_3 are all minimized at the same time. Taking the previous analysis into account, we can graphically visualize P_1, P_2 and P_3 as functions of T, as shown in Fig. 10. The best value of T, in this case, can be derived by solving the equation P_1 (or P_3) = P_2, i.e., $\varepsilon^T = \theta^{k-T}$.

Once the threshold T between two activities A and B is computed, it can be used to determine their relationship in the underlying process model digraph according to the analysis described above. In sum-

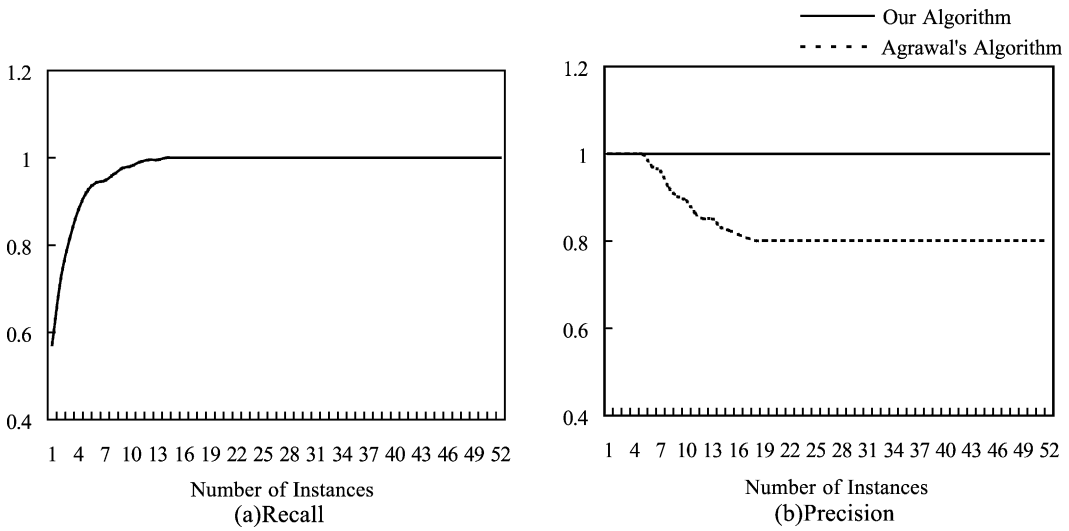

Fig. 9. (a) Recalls of the two algorithms under composite-loop process, (b) precisions of the two algorithms under composite-loop process.

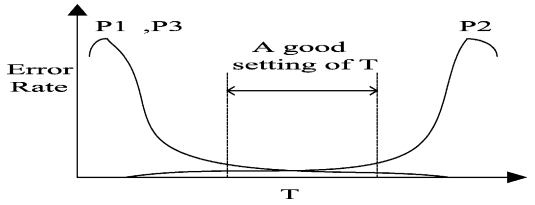

Fig. 10. Error probabilities as a function of *T*.

mary, we can determine whether each ordered activity pair (*A*, *B*) in *DirFollowedSet*, computed by the algorithm *CompDirFollowedSet*() shown in Section 3.1, is indeed an edge in the underlying process model digraph by the following rules:

(1) if *A* and *B* of relationships (b), (c), and (d) occur in no less than *T* process instances then $A \rightarrow B$ is ignored; (Case 1)
(2) if *A* and *B* of relationships (a), (b), and (c) occur in less than *T* process instances then $A \rightarrow B$ is ignored; (Case 2)
(3) if *A* and *B* of relationships (a), (b), and (d) occur in no more than $K - T$ process instances then $A \rightarrow B$ is ignored; (Case 3)
(4) otherwise, accept $A \rightarrow B$ as an edge in the underlying process model digraph.

5.2. Experimenting with noisy data

We would like to see how our algorithm, after incorporating the noise handling mechanism described in the previous subsection, works in a noise-prone environment. To do so, we applied noise to the synthetic datasets. Specifically, a parameter ε, ranging from 0% to 16%, was set to represent the probability that noise associated with a process instance. When it came to synthesizing a process instance out of a process model digraph, we first determined whether or not the process instance should incur noise. If not, the same procedure as described in Section 4.1 was followed to generate this instance. Otherwise, the execution intervals of all activities except for the beginning and ending activities were randomly generated to mimic the result of noise.

Fig. 11 shows the recalls and precisions of our algorithm under various noise probabilities generated for the no-loop process. This set of experiments was designed to see how different degrees of noise affects performance. Overall, the trends are the same for each different noise probability. Besides, as expected, the lower the noise probability, the better on both recalls and precisions. But both recalls and precisions improve as the number of instances increase.

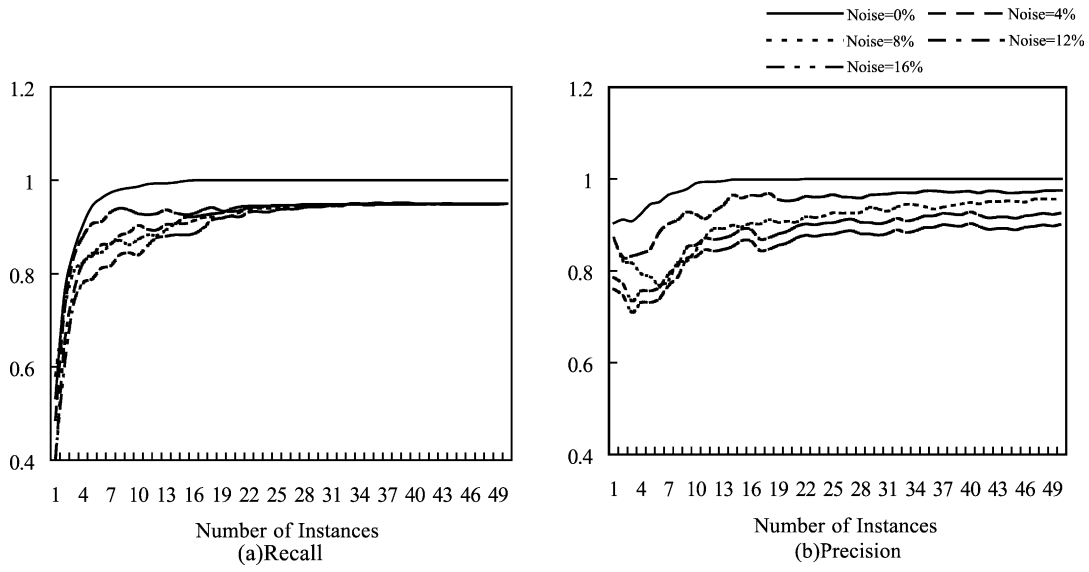

Fig. 11. (a) Recalls of our algorithm under no-loop process with different noise, (b) precisions of our algorithm under no-loop process with different noise.

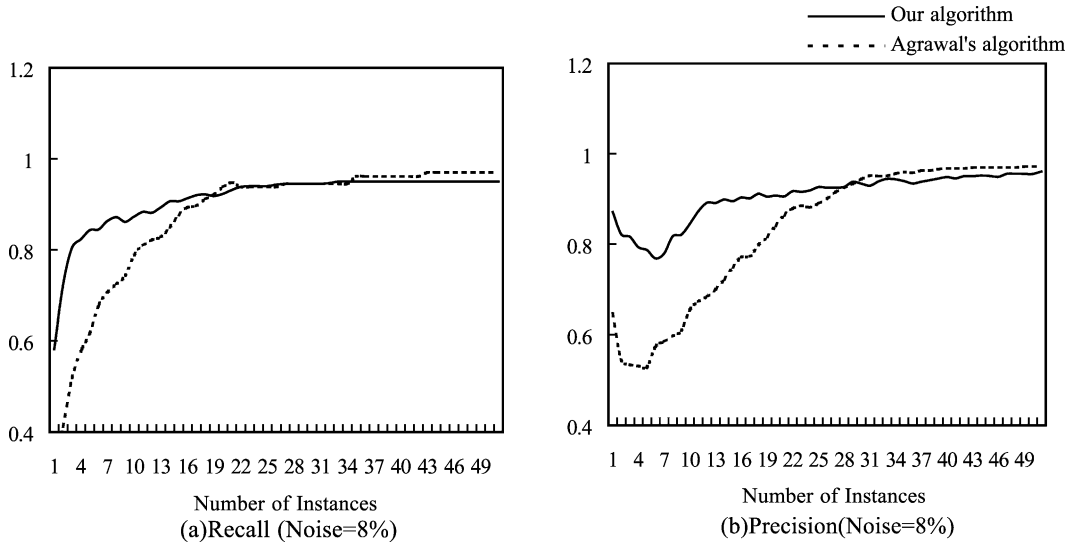

Fig. 12. (a) Recalls of the two algorithms under no-loop process with 8% noise, (b) precisions of the two algorithms under no-loop process with 8% noise.

The same experiments were conducted by using Agrawal et al.'s algorithm [2], with their noise handling mechanism incorporated, and compared with the experimental result of our algorithm. Fig. 12 shows the recall and the precision of the two algorithms under the synthetic datasets with 8% noise generated from the no-loop process. When the number of instances is large, both algorithms perform equally well, with both recall and precision approaching 1. As in the noise-free environment, our algorithm still converges more quickly under such a noise-prone circumstance.

Recall that the threshold T is computed from the estimate of the noise probability ε. But what if this estimate is distant from the real value? To see the

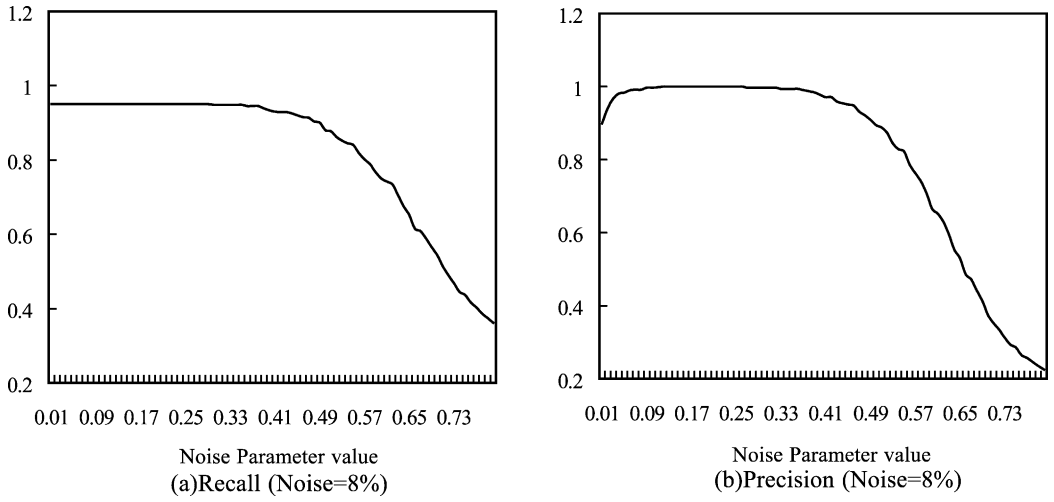

Fig. 13. (a) Recall of our algorithm under no-loop process with real noise 8%, (b) precision of our algorithm under no-loop process with real noise 8%.

effect of mis-estimation of ε, in the following experiments we set the estimate of ε to range from 0% to 80%, with the real value to be set at 8%. The purpose of this experiment set was to test the sensitivity of our algorithm to the deviation of noise probability estimate. Fig. 13 shows the experimental results. It is clear that when deviations are within 30% from the actual noise probability, our algorithm still performs well. This result indicates that the extent to which our noise handling mechanism tolerates imprecise estimates of noise probability is quite large.

6. Conclusions

We have presented a novel approach to discovering a process model from past executions. The process model includes two components: control dependencies represented by a directed graph and the conditions pertaining to these dependencies. We have shown through both theoretic analysis and experiments that our proposed algorithm for deriving the control dependencies executes faster and is able to return more accurate results. Using intervals for modeling activity executions contributes to this improvement. Besides, we also developed a noise handling mechanism to be incorporated into the control dependency derivation algorithm. Experimental results showed that the algorithm performed well under a noise-prone environment. We have also described how to induce conditions associated with dependencies via classification techniques. These two major components together make our work a complete framework for process discovery.

Many applications are eligible for generating historical information in the form of interval sets. While the framework and algorithms proposed in this paper can be applied in some of the domains, they may not fit perfectly in others, partly due to an important assumption made by virtually all process discovery research: *a unique underlying process model that encompasses all process instances is assumed to exist*. Consider the project execution in several occasions that aims to achieve a similar goal. Each project execution is characterized by a Gantt Chart, which comprises a set of intervals. Project executions carried out at different corporations, though pursuing the same objective, may take a significantly different approach. Thus, we can no longer expect the existence of a single process model to which all process instances conform. What we can probably best hope for are some partial process models, each of which represents features exhibited by a significant number of process instances. To this end, this problem becomes somewhat like those discussed in the data mining (or knowledge discovery) community, especially the sequential pattern discovery problem. However, most research work in the context of sequential pattern discovery intends to find total orders on some constituent items, whereas we are interested in finding structures like the process model digraph described in this paper. The intrinsic sophistication of the process model digraph makes this problem challenging. We have embarked on a study of this problem, and a preliminary result has been reported in Ref. [28].

References

[1] N.R. Adam, V. Atluri, W. Huang, Modeling and analysis of workflows using Petri Nets, Journal Intelligent Information Systems, 10 (Mar. 1998) 131–158.
[2] R. Agrawal, D. Gunopulos, F. Leymann, Mining process models from workflow logs, Proc. of the 6th Int'l Conf. on Extending Database Technology (EDBT), Valencia, Spain, Expanded version available as IBM Research Report, RJ 10100, 1998.
[3] P.C. Attie, M.P. Singh, A. Sheth, M. Rusinkiewicz, Specifying and enforcing intertask dependencies, Proc. of Int. Conf. on VLDB, 1993.
[4] A.W. Biermann, J.A. Fieldman, On the synthesis of finite state machines from samples of their behavior, IEEE Transactions on Computers 21 (6), (Jun. 1972) 592–597.
[5] M.G. Bradac, D.E. Perry, L.G. Votta, Prototyping a process monitoring experiment, IEEE Transactions on Software Engineering 20 (10) Oct. 1994, pp. 774–784.
[6] P. Clark, P. Boswell, Rule induction with CN2: some recent improvements, Proc. of 5th European Working Session on Learning (EWSL '91), (1991).
[7] P. Clark, T. Niblett, The CN2 induction algorithm, Machine Learning Journal 3 (4), (Dec. 1989) 261–283.
[8] L. Cohen, L. Manion, Research Methods in Education, Routledge, London, New York, (1996).
[9] J. Cook, A. Wolf, Automating process discovery through event-data analysis, Proc. 17th Intl. Conf. on Software Engineering (ICSE17), (Apr. 1995).
[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.
[11] A. Datta, Automating the discovery of AS-IS business process models: probabilistic and algorithmic approaches, Information Systems Research 9 (3) (Sep. 1998), pp. 275–301.
[12] T. Davenport, Process Innovation—Reengineering Work

through Information Technology, Harvard Business School, Boston, MA, 1993.
[13] T.F. Furey, A six step guide to process reengineering, Planning Review, Mar. 1993, 20–23.
[14] http://www.nhi.gov.tw/hospital/data/hospital_1.jpg.
[15] http://www.wfmc.org/standards/docs/tc003v11.pdf.
[16] http://www.workflowsoftware.com.
[17] Interface 1: Process definition interchange process model. Workflow Management Coalition, No. TC-1016-P, Aug. 1998.
[18] I/S Analyser, The role of IT in business reengineering, I/S Analyser 31 (8), (Feb. 1993) 1–14.
[19] R. Johnson, D. Wichern, Applied Multivariate Statistical Analysis, Prentice-Hall, Englewood Cliffs, NJ, 1992.
[20] R.L. Manganelli, M.M. Klein, The Reengineering Handbook, American Management Association, New York, 1996.
[21] R. Medina-Mora, H.K.T. Wong, P. Flores, ActionWorkflow as the enterprise integration technology, IEEE Data Engineering Bulletin, (1993) 49–52.
[22] R.S. Michalski, On the quasi-minimal solution of the general covering problem, Proc. of the 5th International Symposium on Information Processing (FCIP69), A3, Bled, Yugoslavia, 1969.
[23] M.E. Nissen, Redesigning reengineering through measurement-driven inference, MIS Quarterly 22 (4) (Dec. 1998) 509–534.
[24] J.R. Quinlan, Induction of decision trees, Machine Learning 1, (Jan. 1986) 81–106.
[25] D.E. Rumelhart, G.E. Hinton, R.J. Williams, in: D.E. Rumelhart, J.L. McClelland (Eds.), Learning Internal Representations by Error Propagation, Parallel Distributed Processing: Explorations in the Microstructures of Cognition, 1, MIT Press, Cambridge, MA, 1986, pp. 318–362.
[26] T.K. Sung, N. Chang, G. Lee, Dynamics of modeling in data mining: interpretive approach to bankruptcy prediction, Journal of Management Information Systems 16 (1) (Summer, 1999) 63–86.
[27] The Workflow Reference Model, Workflow Management Coalition, No. TC-00-1003, Nov. 1994.
[28] C.-P. Wei, S.-Y. Hwang, W.-S. Yang, Mining frequent temporal patterns in process databases, Proc. of 10'th International Workshop on Information Technologies and Systems (WITS00), Australia, 2000.
[29] D. Wodtke, G. Weikum, A formal foundation for distributed workflow execution based on state charts, Proc. of the Int. Conf. on Database Theory, Springer LNCS 1186, 1997.

San-Yih Hwang received the BS and MS degrees from National Taiwan University, Taiwan, in 1984 and 1988, respectively; and the PhD degree from the University of Minnesota, Minneapolis, in 1994, all in computer science.
He is presently an Associate Professor in the Department of Information Management, National Sun Yat-Sen University, where he initially joined in 1995. Between 1994 and 1995, he was with the Computer and Communication Laboratory, Industrial Technology Research Institute (CCL/ITRI), Taiwan. His current research interests include workflow systems, data mining, and data management aspects in mobile computing.

Wan-Shiou Yang received her MS degree in Management Information Systems from National Sun Yat-Sen University, and BS degree in Computer Education from National Taiwan Normal University. She is currently pursuing her PhD degree at National Sun Yat-Sen University. Her current research interests are workflow systems and data mining related areas.