

Mining Program Workflow from Interleaved Traces

Jian-Guang Lou¹ Qiang Fu¹ Shengqi Yang² Jiang Li¹ Bin Wu²

¹Microsoft Research Asia
Beijing, P. R. China

{jlou, qifu, jiangli}@microsoft.com

²Dept. of Computer Science
Beijing Univ. of Posts and Telecom
wubin@bupt.edu.cn

ABSTRACT

Successful software maintenance is becoming increasingly critical due to the increasing dependence of our society and economy on software systems. One key problem of software maintenance is the difficulty in understanding the evolving software systems. Program workflows can help system operators and administrators to understand system behaviors and verify system executions so as to greatly facilitate system maintenance. In this paper, we propose an algorithm to automatically discover program workflows from event traces that record system events during system execution. Different from existing workflow mining algorithms, our approach can construct concurrent workflows from traces of interleaved events. Our workflow mining approach is a three-step coarse-to-fine algorithm. At first, we mine temporal dependencies for each pair of events. Then, based on the mined pair-wise temporal dependencies, we construct a basic workflow model by a breadth-first path pruning algorithm. After that, we refine the workflow by verifying it with all training event traces. The refinement algorithm tries to find out a workflow that can interpret all event traces with minimal state transitions and threads. The results of both simulation data and real program data show that our algorithm is highly effective.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications—*Data Mining*; D.2.1 [Software Engineering]: Requirements/ Specifications—*Tools*

General Terms

Algorithms, Experimentation

Keywords

Workflow mining, graphical behavior models, temporal properties

1 INTRODUCTION

With the increasing dependence on software at all levels of our society and economy, software maintenance is becoming increasingly critical. A key technical issue of software maintenance is the difficulty in understanding what happens to systems (and software) over time [18], e.g. about 50% of the maintenance cost is due to comprehending or understanding the code base. Program workflow can help operators and developers to understand system behaviors, to verify the system execution, and to validate the running results. For example, Mariani et al. [6] identify incompati-

bilities and illegal interactions of a reused component by comparing its interaction workflow in the new context with that in the original system. Unfortunately, the knowledge of program workflow is not always available to hand. This is because software documents and specifications are often not complete, accurate and updated due to bad development management and tight schedule (e.g. hard product shipping deadlines and “short-time-to-market”). Therefore, there is a great demand of building automatic tools for mining workflows for programs.

Our motivating application is in the area of program dynamic analysis where program event traces are analyzed to mine program workflows. There are a set of existing research efforts [1, 2, 4, 5, 6, 8, 10] on mining program workflows. Most of them are variants of the classical k-Tails algorithm [2]. They learn a finite state automaton (FSA) from event traces. However, these k-Tails based algorithms cannot perform well in a complex system, in which multiple independent threads/processes generate traces of interleaved events. Because the events can be interleaved in many different ways, and a k-Tails based algorithm tries to interpret a huge number of event sequencing patterns, the resulting FSA model often becomes very complex. Because of this, the k-Tails based algorithms assume that the traces contain thread IDs and can be analyzed thread by thread [6]. Although most software programs produce traces with thread IDs, there are still some existing systems that do not record thread IDs in their logs or traces by default. More important, most advanced software programs are designed and implemented based on event driven architecture, in which the flow of the program is determined by events [11]. In most event driven systems, a task’s workflow is divided into several stages or subtasks, and each subtask is related to an event which is handled by an event handler. In these systems, a thread usually handles many different events from multiple concurrent tasks, and these events often interleave with each other. Therefore, even given the thread IDs, we still cannot learn a task workflow from the traces of such an event driven system by a k-Tails based algorithm.

In this paper, we present an algorithm to discover a program workflow model from traces of interleaved events by a three-step coarse-to-fine algorithm. At first, we mine temporal dependencies for each event pair from event traces. The dependencies that we mined are always valid under any event interleaving patterns. Rather than directly constructing the workflow from the input event traces, we construct a basic workflow model from the mined dependencies. Then, based on the different properties between a loop/shortcut structure and a thread spawn/sync structure, we design an algorithm to refine the basic workflow. The refinement algorithm can find the simplest workflow with a minimal number of thread types to interpret all event traces. Our approach works quite well on traces of interleaved events. To the best of our knowledge, the paper is the first work to learn workflows from traces of interleaved events produced by concurrent programs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

KDD’10, July 25-28, 2010, Washington, DC, USA.

Copyright 2010 ACM 978-1-4503-0055-1/10/07 ...\$10.00.

The rest of the paper is organized as follows. In section 2, we briefly introduce the previous work that is closely related to ours. Section 3 provides the basic ideas and concepts of our approach. In section 4, we describe an algorithm to mine temporal dependencies from event traces. The detailed algorithm on constructing a workflow model based on mined temporal dependences is presented in section 5. In section 7, we validate the approach with some experimental results on both simulations and real event traces. Finally, we conclude the paper in section 8.

2 RELATED WORK

There are a set of existing research efforts [1, 2, 4, 5, 6, 8, 10,] on learning program workflow models from execution traces for software testing and debugging. In these algorithms, Finite State Automata (FSA) are used as workflow models. These algorithms are mostly extended from the popular k-Tails algorithm proposed by Biermann and Feldman [9]. At first, the algorithm constructs an initial FSA from input traces, and then it progressively refines the FSA by merging equivalent states until no further merge operation is possible. Different algorithms [6, 10] use different equivalent criteria to derive different FSA depending on different desired degree of generalization. For example, in the k-strings algorithm [10], if two states s_a and s_b generate the same k-strings (i.e. symbol sequences of length up to k that can be generated from the state), they are merged. The learner modifies the k-Tails algorithm by comparing how likely two states are to generate the same k-strings. L. Mariani et al [6] also proposed an incremental FSA learner, namely k-Learner, by modifying the k-Tails with a new merge operation. The algorithm identifies subsequences of a new trace in the current FSA, and augments the FSA to include the new trace. Besides the state equivalence conditions, some recent algorithms also introduce other pre-conditions for the merge operation. In [2], two equivalent states s_a and s_b can only be merged when the merged FSA does not break a set of temporal patterns that are automatically mined from the training traces. A similar steering idea is also proposed by Walkinshaw et al [8], in which user defined Linear Temporal Logic (LTL) rules are used to determine whether two equivalent states can be merged. All these k-Tails based algorithms assume that the event traces are sequential traces [6]. However, many workflows exhibit concurrent behavior, where a single event trace comprises a set of events produced by more than one thread of control. The sequential state machine model cannot capture the concurrent behavior of a system, and the k-Tails based algorithms will create a very complex model as they try to interpret all event sequencing patterns generated by the interleaved events. Unlike these, our algorithm learns workflow models from traces of interleaved events. Some researches [26, 27, 28, 29, 30, 31] on mining temporal patterns from program traces are highly related to our work too. For example, Javert [28] learns simple generic patterns, such as the alternating pattern $((ab)^*)$ and the resource usage pattern $((ab^*c)^*)$, and composes them using two simple rules to construct large, complex specifications. In this paper, we construct a basic workflow model by composing simple temporal dependency patterns. The similar concept can also be found in [27].

Another set of research efforts [3, 7, 12, 13, 14, 15, 16, 17] from the area of Business Intelligence [12] are also related to our work. Rather than software system traces, they try to mine business processes (or workflows) from business activity logs. In business workflow mining algorithms, different from FSA models, a process is often represented by a graph in which nodes correspond to the activities to be performed, and arcs describe the precedence

or dependent relationships among the activities. For example, the authors of [14] use a directed acyclic graph (DAG) to model a workflow process. In [3] and [7], a constrained DAG, namely an AND/OR workflow graph (AO graph), is proposed to model a workflow process. Aalst et al [15] use a special class of Petri-net model, namely *workflow nets (WF-net)*, to model a workflow process. Among these models, the WF-net is the most powerful model that can be simplified to other models by adding some constraints. Although the business process mining and software workflow mining algorithms have different application backgrounds and models, they share the same basic idea, i.e. constructing constrained graphic activity models from traces [13], and they can leverage each other [17]. For example, we can construct a DFA to mimic the behaviors of a DAG based model. Almost all business workflow mining algorithms consider the parallel property of different tasks by introducing AND split/join nodes [3, 7, 13, 14], fork/join places [12, 15, 17], or Parallel operators [16]. In order to handle the concurrent task problem, similar to our approach, these algorithms reconstruct the workflow graph by utilizing the dependent or causal relationships mined from workflow activity logs. For example, in [15], Aalst et al only use the ordering relation “ $>_w$ ” that describes whether one is directly followed by the other in a trace, to discover a WF-net. Cook et al [17] use more information, such as event type counts, event entropy, and event periodicity, and combine the information into a ranking score to discover the concurrent behavior in event streams. However, all of these approaches assume that an input log sequence is produced by a single case in isolation [15, 17], and they cannot handle an interleaved log sequence generated by multiple concurrent executions of the same workflow (e.g. an event trace is produced by a WF-net workflow if there are two or more tokens in the start place of a WF-net). In addition, their models also do not allow that multiple concurrent threads run the same sub-workflow [15, 17]. However, in a paralleled program, a job may often be divided into several concurrent sub-tasks, with each sub-task having the same workflow. For example, a MapReduce job is often split into many Map tasks, and each Map task follows the same execution logic. Our algorithm handles these problems by a coarse-to-fine approach.

3 PRELIMINARY

This section introduces the basic concepts and techniques used in our algorithm including our view of events, our workflow model, different types of dependency relationships involved in our algorithm, and the assumptions of our mining algorithm.

3.1 EVENT TRACE

Similar to other methods, we view the event logs in a log file as a trace of events being produced by a black-box system. Here, events are used to characterize the dynamic behavior of a software program. Each event trace is generated and collected during a complete execution of the program on a batch of use cases. During an execution, the program may have simultaneous executing threads of control, with each of them producing events that form a resulting event trace. Thus, each event trace may contain interleaved events from all the concurrent threads (namely a *trace of interleaved events*). Note here that although the term “thread” means a sequential execution control path within the workflow, it may not directly map to a process/thread in the operating system in an event-driven system. We can collect many event traces by running the program many times. In software systems, events may be recorded with attributes, such as timestamp, related resources

and data. Although in some systems, event attributes can help to distinguish the events from different control threads of the workflow, in this paper, we present a general approach that only utilizes the ordering of events.

Formally, let Σ be a set of distinct events that appear in all event traces. An *event trace* is a sequence of events, denoted as $\langle e_1, e_2, \dots, e_m \rangle$, where e_i is an event, i.e. $e_i \in \Sigma$ for $1 \leq i \leq m$. For brevity, we can also write it as e_1, e_2, \dots, e_m . Given an event trace $l = e_1, e_2, \dots, e_m$, for an event e_i , $1 \leq i \leq m$, we call the subsequence $e_{i+1}, e_{i+2}, \dots, e_m$ as the *postfix* of e_i in l , denoted as $post(e_i)$, and subsequence e_1, e_2, \dots, e_{i-1} as the *prefix* of e_i in l , denoted as $pre(e_i)$. If $i = 1$, then $pre(e_i)$ is a null sequence, denoted as \emptyset . Similarly, when $i = m$, we also have $post(e_i) = \emptyset$.

Program event traces may contain some noise that is usually caused by some execution anomalies. For event traces that may contain noise, we can use the noise filtering method [19] or anomaly detection method [20] to filter out the noise before we learn workflow models. In addition, we assume that the temporal ordering patterns of different events can be as diverse as possible. This is reasonable for event logs produced by independent concurrency. Some concurrent systems that may not display much randomness are not the targets of our method. As with most data driven approaches, we also assume that we have enough event logs to make our analysis meaningful. Because our method is based on the temporal orders of events in event traces, enough log data means that we have enough instances of any possible ordering relationship of every event pair to make the mined dependencies statistically meaningful.

3.2 WORKFLOW MODEL

In this paper, our algorithm outputs a transition-labeled finite state machine to represent a workflow. In order to model the concurrent properties of workflows, we introduce a vector of active states instead of just a single active state so that the state machine can have multiple, concurrent threads. To create and destroy these threads, we define four new types of states to mimic the concurrent behavior of a Petri net: *split/merge* states and *fork/join* states. Similar to the AND split/merge nodes in papers [3, 7], a *split* state has multiple out transitions (i.e. transitions leaving from the state), and all these transitions are taken concurrently. A merge state has multiple entering transitions, and it can transit to another state only if all its entering transitions have occurred. Informally, split states represent the points where the system will spawn one thread for each leaving transition. As a counterpart, merge states are used to represent points of synchronization. That is, before the merge state can transit to another state, it has to wait for the completion of threads from all its entering transitions. Unlike a split state, a fork state only has one leaving transition. At a fork state, the system will fork several threads to execute a sub-workflow starting from the state. Similar to a merge state, a join state is also a point of synchronization, and it is specially designed to absorb all threads created by a fork state. In general, a fork state and a join state will exist in pairs. To facilitate the description, we call the ordinary states in a state machine as a *switch* state. Fig. 1 shows some sample workflows, where a square box “□” is used to represent a split/merge state node, a diamond box “◊” is used to represent a fork/join state node, and a circle “○” is used to represent a *switch* state node. By introducing the split/merge and fork/join nodes, we can model the concurrent properties in a workflow. For example, a single run of the workflow of Fig.1(a) can generate event traces of “ABCD” or “ACBD” because B and

C are two concurrent events. Furthermore, two concurrent runs of the workflow can generate many kinds of traces, such as “AB-ACDCBD”, “AACCBDD”, and so on. The fork node in Fig.1(c) can also generate concurrent events, because the sub-workflow from the fork node to the join node will be run with multiple threads. Formally, a workflow model is defined as:

Definition 3.1 A workflow model is a tuple $W = (\Sigma, S, s_0, \delta, f)$, where:

- Σ is the set of event types.
- S is a finite and non-empty set of states including switch states, split/merge states and fork/join states. s_0 is the initial state. f is the end state.
- δ is the state-transition function that represents the transition from one state to other states, $\delta: S \times \Sigma \rightarrow S$. Each transition is labeled by an event type, which means the transition can generate an event of that type, and a transition can be uniquely determined by its event label. In this paper, we use these two terms interchangeably.
- We also define a special symbol $\epsilon \in \Sigma$, which means an *empty* event. $\delta(q_1, \epsilon) = q_2$ means that the transition from state q_1 to state q_2 does not generate any event. We call such a transition a *shortcut*.

Definition 3.2 Given a transition from q_1 to q_2 , we call q_1 and q_2 as *the starting state* and *the ending state* of the transition respectively. If one transition’s ending state is the starting state of another transition, we call the two transitions as *neighboring transitions*. Given a state q , we define all transitions that start from q as q ’s *out transitions* (denoted as $OutSet(q)$), and the transitions that ending at q as q ’s *in transitions* (denoted as $InSet(q)$).

For a realistic workflow, all states and transitions should be reachable from the initial state. It means that, for every state s , there is at least one transition path from the initial state to s . At the same time, each state also has a path to the end state.

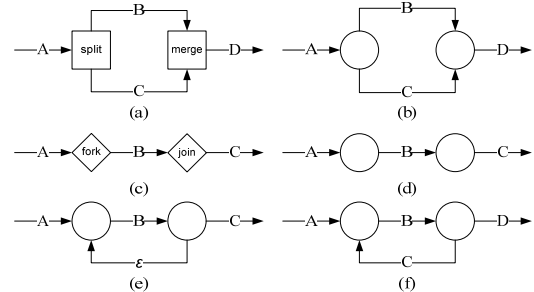


Figure 1. Some simple workflows.

3.3 EVENT DEPENDENCIES

In the context of workflow traces, the occurrence of an event may be dependent on the occurrence of another event in the trace. Our workflow model reconstruction depends on mining the temporal dependencies among events. Dependence means that the occurrence of one event type depends on the occurrence of another event type. In this paper, we define four types of temporal dependencies. *Forward dependency* (FD) describes that “Whenever an event A occurs, another event B must eventually occurs after A ’s occurrence”. Forward dependency is denoted as $A \rightarrow_f B$. *Backward dependency* (BD) describes that “Whenever an event B occurs, another event A must have occurred before B ’s occurrence”.

Backward dependency is denoted as $A \rightarrow_b B$. The third dependency we defined is *strict forward dependency* (SFD), which means “For each occurrence of event A , there must be at least one occurrence of event B after A ’s occurrence.” We denote it as $A \rightarrow_{sf} B$. Different from $A \rightarrow_f B$, which means one or more occurrences of event A will eventually cause the occurrence of event B , $A \rightarrow_{sf} B$ means that each occurrence of event A will cause at least one occurrence of event B . Similarly, we also define *strict backward dependency* (SBD) with “For each occurrence of event B , there must be at least one occurrence of event A before B ’s occurrence.”, which is denoted as $A \rightarrow_{sb} B$. Unlike the dependencies defined in [15, 17], our dependencies do not require that one event is directly followed by another event in an event trace, and they are not influenced by various interleaving patterns. We use $A \parallel B$ to denote two events A and B without any defined temporal dependencies.

Among these four types of dependencies, FD and BD focus on the global temporal relationship between two event types, and SFD and SBD not only look at the temporal relationship but also take the event count information into account. It is obvious that: $A \rightarrow_{sf} B \Rightarrow A \rightarrow_f B$, and $A \rightarrow_{sb} B \Rightarrow A \rightarrow_b B$. Thus, for simplicity, if two events have a strict dependency relationship, we will not list the corresponding non-strict dependency.

For each pair of events, the temporal dependency is determined by the structure of the workflow. For example, for event A and B in Fig.1(a), we have $A \rightarrow_{sf} B$ and $A \rightarrow_{sb} B$ because these two dependencies are always true in the traces with any potential event interleaving pattern. For the workflow in Fig.1(e), we always have $A \rightarrow_{sf} B$ and $A \rightarrow_b B$. For all simple workflows in Fig.1, we list all the dependencies between event type A and B in Table 1. Such dependencies are always true no matter how event logs interleave together. We can see that different local logical structures in a workflow often have different types of dependencies.

Table 1. Temporal dependencies of workflows in Fig.1

Workflow	Valid Temporal Dependencies between A and B
Fig. 1(a)	$A \rightarrow_{sf} B, A \rightarrow_{sb} B$
Fig. 1(b)	$A \rightarrow_{sb} B$
Fig. 1(c)	$A \rightarrow_{sf} B, A \rightarrow_b B$
Fig. 1(d)	$A \rightarrow_{sf} B, A \rightarrow_{sb} B$
Fig. 1(e)	$A \rightarrow_{sf} B, A \rightarrow_b B$
Fig. 1(f)	$A \rightarrow_{sf} B, A \rightarrow_b B$

Pair-wise temporal dependencies describe the causal relationships of each event pair. They can provide information for workflow reconstruction. Our basic workflow model constructing algorithm is based on the properties of the mined temporal dependencies. Obviously, the dependencies have the following property:

Property 3.1 Let $W = (\Sigma, S, s_0, \delta, f)$ be a workflow. For any $A, B \in \Sigma$, if $A \rightarrow_f B$ or $A \rightarrow_b B$, there must be a transition path from A to B (denoted as $A \rightarrow B$) in the workflow.

4 MINING TEMPORAL DEPENDENCY

In this section, we provide the details about the method of mining temporal dependencies. As with most classical algorithms of sequence pattern mining, we measure the significance of a temporal dependency between two events by computing the statistical metrics of support and confidence. For event types A and B , when we mine the relationships of $A \rightarrow_{sf} B$, the support of $A \rightarrow_{sf} B$ is

defined as the number of times that event A appears in the traces that satisfy $|\{e|e = A, e \in post(e_i)\}| < |\{e|e = B, e \in post(e_i)\}|$ (see the numerator of equation (3)). In contrast to the support of $A \rightarrow_{sf} B$, the support of $A \rightarrow_f B$ is computed as the number of traces that contain event A and B and $B \in post(A)$. As a counterpart, the support of $A \rightarrow_{sb} B$ and $A \rightarrow_b B$ can also be calculated similarly. The confidence values of the dependencies are defined by the corresponding conditional probabilities. For example, the confidence of $A \rightarrow_f B$ is calculated by

$$conf(A \rightarrow_f B) = \frac{\text{No. of traces that have } B \text{ after the last } A}{\text{No. of traces that contain } A} \quad (1)$$

Note, according to the definition of $A \rightarrow_f B$, we only have to compute the confidence by investigating the events for traces after their last appearance of event A . As the size of traces scales up, this trick can significantly reduce the computational cost. Similarly, the confidence of $A \rightarrow_b B$ can also be calculated by

$$conf(A \rightarrow_b B) = \frac{\text{No. of traces that have } A \text{ before the first } B}{\text{No. of traces that contain } B} \quad (2)$$

The computing of $A \rightarrow_{sf} B$ and $A \rightarrow_{sb} B$ ’s confidence values is a little bit complex. We take $A \rightarrow_{sf} B$ as an example to describe the computing procedure. For each event trace l , we find all occurrences of event e_i that satisfy $e_i = A$ and $|\{e|e = A, e \in post(e_i)\}| < |\{e|e = B, e \in post(e_i)\}|$ (i.e. the number of B is larger than the number of A in $post(e_i)$). Denoting the total number of such events e_i in all traces as $|\{e|e \in (A \rightarrow_{sf} B)\}|$, we can calculate the confidence by

$$conf(A \rightarrow_{sf} B) = \frac{|\{e|e \in (A \rightarrow_{sf} B)\}|}{\text{No. of } A \text{ in all traces}} \quad (3)$$

As a counterpart, the dependency of $A \rightarrow_{sb} B$ is

$$conf(A \rightarrow_{sb} B) = \frac{|\{e|e \in (A \rightarrow_{sb} B)\}|}{\text{No. of } A \text{ in all traces}} \quad (4)$$

where $|\{e|e \in (A \rightarrow_{sb} B)\}|$ is the number of events that satisfy $e_i = B$ and $|\{e|e = B, e \in pre(e_i)\}| < |\{e|e = A, e \in pre(e_i)\}|$ (i.e. the number of B is larger than the number of A in $pre(e_i)$).

By scanning the event traces, we can obtain the support numbers and confidence values of these dependencies for each pair of event (the pseudo code can be found in [25]). The time complexity of the algorithm is $O(Nd)$, where N is the cumulative length of all event traces. Generally, $d = |\Sigma|$ (i.e. the number of distinct event types) is constant for a program, and is always significantly smaller than N . Thus, the algorithm possesses linear complexity with respect to N .

Unlike the scenarios of traditional sequence (or frequent item set) mining, where some meaningless patterns can happen by chance, in our context any occurrence of an event ordering in event traces is meaningful and reflects an aspect of the execution behavior of the software system. In this paper, we set the support threshold as 5 (The number of observations should be at least five to make the analysis results statistically meaningful [17]), and all our events in our experiments can meet this requirement. In addition, a dependency relationship is valid only if it has a perfect confidence ($conf=100\%$).

5 WORKFLOW RECONSTRUCTION

In this section, we provide our main algorithm of constructing

workflow from mined temporal dependencies. We first construct an initial workflow by recovering all *connections* (defined in section 5.1) based on the mined temporal dependencies. The learned basic workflow does not contain some workflow structures. In order to recover these missing structures, we refine the workflow by verifying with event traces. The aim of refinement is to construct the simplest workflow based on the basic workflow to interpret all training event traces.

5.1 CONSTRUCT A BASIC MODEL

From section 3, we can see that, given dependencies $A \rightarrow_f B$ or $A \rightarrow_b B$, we can conclude that there is a path from event A to event B (denoted as $A \rightarrow B$). In addition, for two neighboring events A and B , if $A \rightarrow B$, we can determine a *connection* between A and B in the original workflow, i.e. the ending state of A is the starting state of B . In this paper, we call the dependency between two neighboring events as a *direct dependency*. Furthermore, supposing that we have a pair-wise dependency for each pair of neighboring events, we can recover all *connection* relationships.

Although a mined temporal dependency from event traces shows that there is a path between two events, we cannot directly establish a *connection* between them because many dependencies are not direct dependencies (i.e. they are indirect dependencies). An indirect dependence does not correspond to a *connection* between two events. For example, in Fig.1 (d), we have a temporal dependency of $A \rightarrow_f C$. However, there is no *connection* between A and C . Here, the path from A to C is composed by a path from A to B and a path from B to C . In order to handle such problems, we try to construct a compact basic workflow in which there is at most one transition path between every two events. We use a pruning strategy to remove indirect dependencies during the basic workflow construction. For each event pair (α, β) that satisfies $\alpha \rightarrow_f \beta$ or $\alpha \rightarrow_b \beta$, we denote β as α 's successor, and α as β 's predecessor. For the simplicity of implementation, we first use a graph data structure to store the obtained paths, in which each event has a predecessor list and a successor list. The algorithm starts from the events that do not have any preceding event. Then, we add events into the graph and construct preceding/succeeding relations according to the mined dependencies. For any pair of events A and C where A is a predecessor of C , if a successor event of A (e.g. B) is also a predecessor of C , we remove C from A 's successor list. In the resulting graph, all indirect dependencies are removed. By converting the remaining preceding/succeeding relations to event *connections*, we can construct a transition-labeled workflow, namely *basic workflow*. The algorithm is shown in Algorithm 1. In the algorithm, the function $Find_Root(NV)$ returns a set of events in which each event does not have any predecessor in the set NV .

The following theorem shows that the remaining transitional paths obtained by the above algorithm must exist in the original workflow under a certain condition. In other words, our algorithm can obtain a basic workflow skeleton. The proof of the theorem [25] is straightforward, and we ignore it in this paper.

Theorem 5.1. Let $W = (\Sigma, S, s_0, \delta, f)$ be a workflow, with at least one temporal dependence between every two neighboring events. For any $A, B \in \Sigma$ that $A \rightarrow_f B$ or $A \rightarrow_b B$, if $\{C | A \rightarrow C \wedge C \rightarrow B\} = \emptyset$, there must be a *connection* from A to B in the original workflow.

The above algorithm does not consider a special case where two

events have dependencies with different directions. For example, from the event traces generated by the workflow in Fig.1(f), we can learn both dependencies of $B \rightarrow_b C$ and $C \rightarrow_f B$ at the same time. We call it a bidirectional dependence, denoted as $C \leftrightarrow B$. If we directly run the basic workflow construction algorithm on such dependencies, the algorithm will run into an endless loop. In order to overcome this problem, we first check whether there are bidirectional dependencies in the mined dependencies. If there is a bidirectional dependence, e.g. $C \leftrightarrow B$, we create a new virtual event type B' to replace the events of type B in all forward dependencies. Then, we run our basic construction algorithm to reconstruct the basic workflow. After that, we merge the virtual events (e.g. B') with their corresponding events (e.g. B) in the basic workflow.

Algorithm 1. Pseudo Code of Basic Workflow Construction

Inputs:

L : the set of all event traces
 D : the set of dependencies

Output:

T : learned basic workflow

1. $N = NV =$ the set of all log keys;
 2. $Q =$ an empty FIFO queue;
 3. **while** NV is not empty
 4. $S = Find_Root(NV)$;
 5. Add S into T ;
 6. $Push_back(Q, S)$;
 7. **while** Q is not empty:
 8. $i = Pop_front(Q)$;
 9. **if** i is not in NV
 10. continue;
 11. **for each** j in N that satisfies $i \rightarrow_f j$ or $i \rightarrow_b j$
 12. **if** j has predecessor in T
 13. flag = false;
 14. **for each** k in j 's predecessors:
 15. **if** $(i \parallel k)$
 16. add j to i 's successor list;
 17. **else if** $k \rightarrow_f i$ or $k \rightarrow_b i$
 18. remove j from the successor list of k
 19. add j to i 's successor list;
 20. **else**
 21. flag = true;
 22. **if** (flag)
 23. remove j from the successor list of i
 24. **else**
 25. add j to i 's successor list;
 26. **if** j is in NV
 27. $Push_back(Q, j)$;
 28. remove i from NV ;
 29. Covert T to a transition-labeled workflow;
 30. **return** T
-

Adding the initial&end state: Each workflow contains an initial state and an end state. Thus, we need to add an initial state and an end state into the basic workflow. Obviously, the first event and the last event of each event trace are potentially an initial event and an end event of the workflow respectively. In this paper, we find out all events that have appeared as the first event in event traces. If the support number of an event appearing as the first event in event traces is larger than a certain level (we use 5% in

experiments because we assume the noise level is less than 5%), we add a shortcut transition from the initial state of the workflow to the starting state of the event. Similarly, if the support number of an event appearing as the last event in event traces is larger than a certain level, we add a shortcut from the ending state of the event to the end state of the workflow.

Determining the state types: According to the definition of the workflow model in section 3.2, there are five types of states. In the above basic workflow construction algorithm, we do not identify the type of each state. Given an event type that has several event types following it, we have to make a decision on whether the program behavior at this point is a sequential selection (i.e. a switch state) or a concurrent splitting (i.e. a split/fork state). In this subsection, we determine the state types by utilizing the information of event type counts. As studied in our previous work [20], the linear relationships between the occurrence times of different event types can also provide cues for the workflow structure. For example, for a switch state q , it is always true in every event trace that:

$$\sum_{A \in InSet(q)} Occur(A) = \sum_{B \in OutSet(q)} Occur(B) \quad (5)$$

On the other hand, if q is a split state, then for any $A \in OutSet(q)$ and $B \in OutSet(q)$,

$$Occur(A) = Occur(B) \quad (6)$$

Similarly, a merge state also has a property that $Occur(A) = Occur(B)$ for any $A \in InSet(q)$ and $B \in InSet(q)$. Here, $Occur(A)$ and $Occur(B)$ denote the occurrence numbers of event A and event B in a trace respectively. However, fork and join states do not have such regular properties on the counts of event types. If a state satisfies equation (5), it must be a switch state. In this subsection, we first find out the split/merge states by verifying whether a state satisfies equation (6), and then find out the switch states that can be identified by equation (5). Because our model allows *shortcut* transitions, some switch states cannot be easily identified by equation (5). For all remaining states with their state types undetermined, we will determine their state types in the next sub-section. The default state type is *switch*.

5.2 REFINE THE WORKFLOW MODEL

The basic workflow obtained by Algorithm 1 does not contain any shortcut transitions or loop structures, because we only keep one transition path between every two dependent events. However, in a real workflow, there may be some shortcut transitions and loop structures. In addition, Algorithm 1 also cannot identify the fork/join state types. Therefore, the basic workflow only contains a part of the real workflow, and it often cannot interpret all training event traces. In fact, a meaningful mined workflow should be able to interpret the traces. In this subsection, based on the learned basic workflow, we build a workflow to interpret all event traces by identifying fork/join states and recovering loop structures and shortcut transitions.

We recover loop structures or shortcut transitions based on the statistical properties of these structures. Here, we use a simple example to describe the basic idea behind our algorithm. Fig.2(a) presents a simple program workflow containing a loop structure. We generated event traces by running a concurrent program with different interleaving patterns in which each thread runs along the workflow in Fig.2(a). Two typical sample event traces are shown in Fig. 2(b). Our basic workflow construction algorithm can only

construct a basic workflow without a loop (see Fig.2(c)). When we use the basic workflow in Fig.2(c) to interpret the first event trace in Fig. 2(b), we find that the first five events of the event trace are generated by two threads (denoted as T_1 and T_2) running along the basic workflow. When the 6th event of the trace (i.e. B) is being verified, the active states of T_1 and T_2 are s_3 and s_2 respectively, and both threads cannot produce event B from their active states (i.e. this event B is an un-interpretable event by the basic workflow.). The reason why some events cannot be interpreted is that some transitions are missing in the basic workflow. Specifically, event B is a part of the recurrence of the missing loop in Fig.2(a). However, we do not have knowledge about the loop structure and the original workflow. In order to interpret the first event trace, we now have two possible solutions: the event is either generated by T_1 or generated by T_2 . If it is generated by T_1 , then there is a loop from s_3 to s_1 in the workflow, which is the workflow in Fig.2(a). If the event is generated by T_2 , then there is a loop from s_2 to s_1 in the workflow, which is the workflow in Fig.2(d). Similarly, when we try to interpret the 8th event of the second event trace in Fig.2(b), the active states of the threads are s_3 and s_4 . One can interpret the second event trace either by the workflow in Fig.2(a) or by that in Fig.2(e). Here, we observe that, for both event traces, when we try to interpret an event B that is a part of the recurrence of the loop, there is a thread at state s_3 . In general, for any training event trace with a different interleaving pattern, when we verify an un-interpretable event of type B , which is a part of the recurrence of the loop, there is always at least one thread whose active state is s_3 . On the contrary, there is a thread with active states of s_2 or s_4 only by chance. Therefore, if we vote for threads' active states over all event traces once we encounter an un-interpretable event B , we will find that s_3 has the highest vote value. This property can help us to detect the loop structures, a loopback transition from s_3 to s_1 . Although the example in Fig.2 is a simple case, this statistical property is widely valid for workflows with loop or shortcut structures. For example, Fig.3 shows the vote value histogram that is counted for the un-interpretable events of B_2 in $WS(b)$ during the experiment. In the figure, there is a high peak value at s_2 , which indicates a loopback transition from s_2 to s_1 . Our algorithm utilizes this statistical property to detect and recover loop structures and shortcuts.

Unlike loop and shortcut structures, fork/join states do not expose any unique statistical property. We cannot use the above statistical method to identify a fork/join structure. If we perform the above method forcibly on the event traces generated by a fork/join structure, then the resulting workflow is often very complex, which is caused by various event interleaving patterns. On the other hand, all event traces generated by a loop structure can always be interpreted by a fork/join structure. For example, all event traces produced by the workflow in Fig.2(a) can always be interpreted by the workflow in Fig.2(f). Formally, the observation is described as:

Property 5.1 Event traces that can be interpreted by a workflow W_1 with loop structures can also be interpreted by a workflow W_2 which is created based on W_1 by replacing the loop structures with fork/join structures, and $cp(W_2) \leq cp(W_1)$, but not vice versa.

Here, $cp(W)$ is the complexity of a workflow W , which is defined as the sum of transition number and the number of thread types. Each thread type is defined as a pair of thread starting and ending points. Given the constant states of the basic workflow (we only add transitions or mark thread spawn/sync states in the refinement algorithm), $cp(W)$ is a good description of a workflow

model's complexity. Based on property 5.1, we introduce a loop favorite rule in our algorithm. For two workflows that can interpret the event traces, we prefer the workflow with less complexity. If event traces can be interpreted by either a workflow with a loop structure or a workflow with a fork/join structure, and both of them have the same complexity, we prefer the former. In summary, starting from the learned basic workflow, we try to construct the simplest workflow with a minimal number of threads to interpret all event traces.

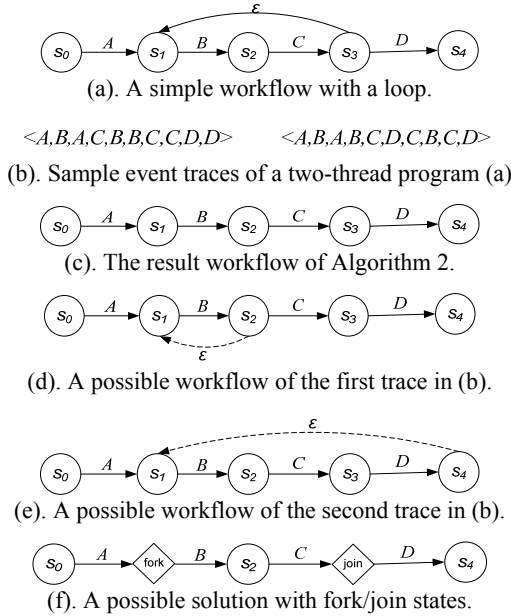


Figure 2. An example for the depiction of our refinement idea.

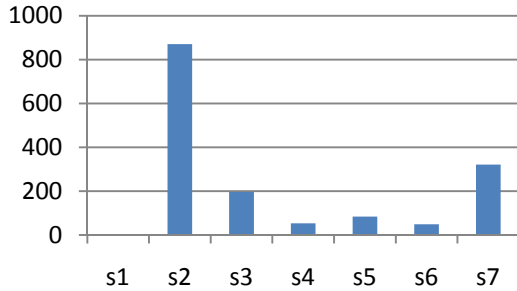


Figure 3. The distribution of vote values counted when we encounter the un-interpretable events of B_2 in $WS(b)$.

Because we have no information about when a new thread starts, an un-interpretable event can be interpreted as an event log produced by either a missing workflow structure component (i.e. shortcut or loop) or a newly started thread (i.e. fork/join states). In the algorithm, we have to make a decision to select one structure between them (i.e. loop decision or fork decision) whenever a new un-interpretable event is encountered. A workflow has a Markov property that states that the current state is what determines the next step in its behavior. Thus, an early decision will influence the later decisions, but the converse is not true. At each decision point,

we first create two temporary workflows. One (denoted as W_1 in the algorithm) is constructed by a procedure in which we make a loop decision at the current decision point and make fork decision at all following decision points. The other (denoted as W_2 in the algorithm) is constructed through a procedure with all fork decisions. Then, we select a decision at the current decision point based on the loop favorite rule. Similarly, we also make the next decision with the same procedure. Note: here, the temporary workflows are only constructed for decision making, and they are not output as the results of the algorithm. The detailed algorithm is presented in APPENDIX A. In the algorithm, we do not count the active states with their neighboring events having strict forward dependencies, because such a state neither has an out-shortcut transition nor is a join state. For example, we do not have $B \rightarrow_{sf} C$ in Fig. 1(c) and (f). The following theorem shows that the workflow learned by our refinement algorithm is optimal in the sense of complexity defined above. The proof of the theorem can be found in [25].

Theorem 5.2. The refinement algorithm finds out the workflow with a minimal complexity to interpret all event traces.

6 EMPIRICAL EVALUATION

To validate and evaluate our proposed workflow algorithm, we performed a set of experiments on simulated event traces and case studies on real event traces generated by some open source programs (Hadoop and JBoss). We use open source programs because they are publicly available for download. The results on them are easy to be verified and reproduced by third parties. The results demonstrate the usefulness of our workflow mining technique in recovering underlying workflows from event traces. Due to space limitation, the results on JBoss are available at [25]. The code of the simulator and our algorithm will also be available soon (after the legal review according to the code release policy of Microsoft) at <http://research.microsoft.com/apps/pubs/default.aspx?id=118640>.

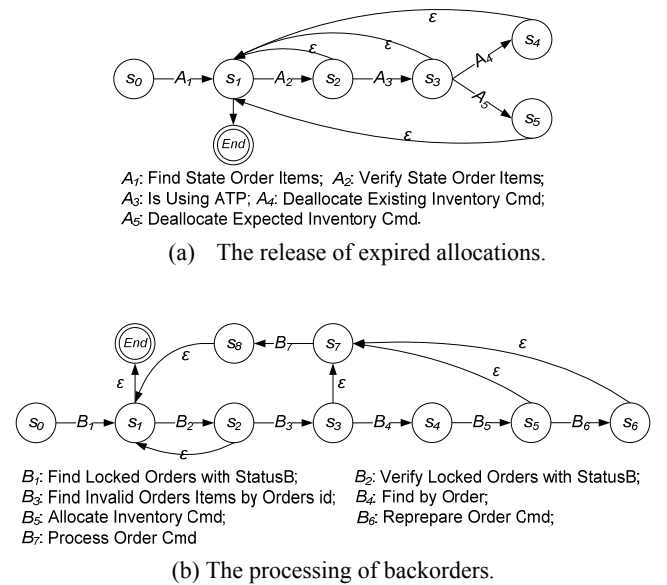


Figure 4. WebSphere® Commerce Processes.

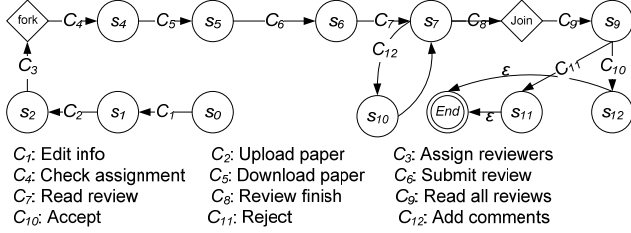


Figure 5. A workflow of reviewing a conference paper.

6.1 SIMULATION

To construct a controlled experimental environment, we designed a simulator that can generate synthetic program event traces according to a software workflow model. The design of the simulator follows the principles proposed in QUARK [21] including the guarantee of “code and branch coverage” and locality of reference, and so on. Unlike QUARK, our simulator can generate traces of interleaved events based on a concurrent workflow model. In this experiment, we measure the performance of our workflow miner by discovering workflows from the synthetic program event traces.

Simulation Models: In our simulation experiments, several real application models are used to generate the event traces. In [23], Zou et al. present two workflows in the form of automata: the releases of expired allocations (Fig.4(a)) and the processing of backorders (Fig.4(b)). In [17], the authors use a paper reviewing process (Fig.5) to demonstrate their workflow mining algorithm. The models are shown in Fig.4 and Fig.5, and are referred to as $WS(a)$, $WS(b)$, and Rev respectively. By using these models, users can evaluate and compare our algorithm with other algorithms in [17] [23]. In addition, these typical real application workflows are complex enough to demonstrate the capability of our algorithm: the models in Fig.4 contain several nested loops and many shortcut transitions, and the model in Fig.5 has a loop embraced by a fork/join structure.

Evaluation Metric: In order to carry out a quantitative evaluation of the workflow miner, we adopt two metrics to measure the similarity from the mined workflow X and the simulator model Y in terms of their *generated traces*. The first metric is known as *recall*, the percentage of event traces generated by workflow Y that can be interpreted by workflow X . The second metric is *precision*, the percentage of event traces produced by workflow X that can be interpreted by workflow Y .

Results: We run these models with several threads (we randomly start 1-3 threads) in our simulator to generate traces of interleaved events. From the generated traces (2000 event traces for each), we learn workflow models through the algorithm provided in the above sections. We compare the effectiveness of the k-Learner algorithm [6] and our algorithm by measuring the precision and recall of the resulting state machines. We repeat each experiment 10 times with 10 different set of traces, and computing the average that shown in Table 2. Here, we round the results to keep three numbers after the decimal point. For these three workflows, our algorithm can exactly rediscover the original workflow model, thus, both the recall and precision are 100%. However, the precisions of the models produced with k-Learner are very poor (2 models with a precision less than 0.1). This indicates that k-Learner cannot perform well when events are interleaved.

Computational cost: Our algorithm is efficient, which only uses 9.9, 22.3 and 72.0 seconds (with a CPU of 2.33GHz, the code is

not fully optimized) to learn the models of $WS(a)$, $WS(b)$, and Rev from 2000 event traces respectively. Table 3 shows that the computing time of each model grows almost linearly as the number of input event traces increases.

Table 2. Empirical Results: Precision and Recall

	Simulation Models					
	$WS(a)$		$WS(b)$		Rev	
	Precs.	Recall	Precs.	Recall	Precs.	Recall
k-Learner (k=1)	0.511	1.000	0.069	1.000	0.000	1.000
k-Learner (k=2)	0.255	1.000	0.080	1.000	0.001	1.000
Our Algorithm	1.000	1.000	1.000	1.000	1.000	1.000

Table 3. Computing Time (in seconds) vs. Event Trace Number

Trace No.	Simulation Models		
	$WS(a)$	$WS(b)$	Rev
2000	9.9	22.3	72.0
4000	19.3	41.1	154.4
6000	29.4	62.5	228.6

6.2 CASE STUDY: HADOOP

Hadoop [22] (our version: 0.19) is a well-known open-source implementation of Google’s Map-Reduce computing framework and distributed file system (HDFS). It enables distributed computing of large scale, data-intensive and stage-based parallel applications. The logs produced by Hadoop are not sequential log message sequences. Even in the log messages of the same Map task, some messages (e.g. messages about data shuffling) are also interleaving.

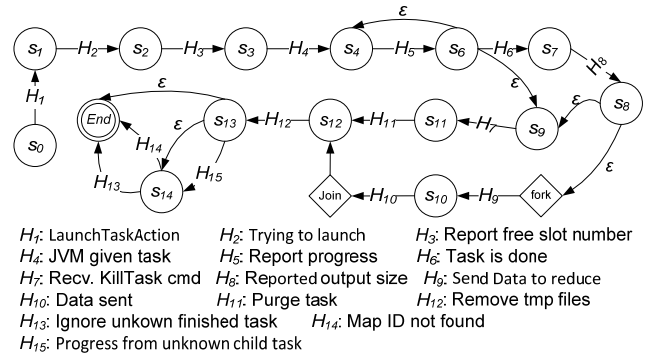


Figure 6. Learned workflow of Hadoop MapTask

Trace collection and preprocessing: We run different Hadoop jobs of some sample applications, such as WordCount and Sort, and collect log data after each job is finished (Note: we enable the logging tool at the info level). At first, we use the log preprocessing method presented in [20, 24] to parse event log messages and to group log messages according to log parameters. For example, all log messages that contain a parameter called MapTask ID are grouped into an event trace. Because several MapTasks are running simultaneously, these events are highly interleaved with each other. Then, we use the error detection algorithm in [20] to filter out event traces that contain errors. Finally, we obtain 1767 event traces. Each trace contains about 15 concurrent map tasks and about 234 events in average.

Results: After that, we learn workflow from event traces with our proposed algorithm. Fig.6 is an example of the resulting workflow that is learned from the event traces related to the parameter MapTask ID. By carefully checking with Hadoop source code and

documents, we find that the workflow can reflect the real process of MapTask with a high precision. A task is launched and then it processes the input data read from HDFS. After the task is done, it sends the resulting data to many Reducers concurrently (This is represented by a fork/join structure in the workflow), and finally cleans up the resources by removing temporary files. Hadoop has a scheduling strategy to run a small number of speculative tasks. Therefore, there may be two running instances of the same task at the same time. If one task instance finishes, the other task instance will be killed no matter which stage the task instance is running at. Some killed tasks can also report their status due to thread race conditions, therefore, events of H_{13} , H_{14} and H_{15} are triggered.

6.3 DISCUSSION

As with the results of all other workflow mining algorithms [2][17][21], some resulting workflow models of our algorithm are over-generalized (i.e. having more possible routes than the real workflow). For example, in the case in section 6.2, there is a path from s_{10} to s_{14} , which does not exist in the real program. There are two main reasons for the over-generalization problem. At first, we only consider the first-order of event dependencies (i.e. the dependencies between neighboring events) in our current algorithm. In some real programs, there are some high-order dependencies, e.g. the occurrences of H_{13} , H_{14} and H_{15} depend on the occurrence of H_7 in Fig.6. Second, our approach assumes that there is at most one transition referring to an event type in a workflow model. In some real systems, one event type may appear at multiple positions in a workflow. For example, from the event traces generated by the workflow in Fig.7(a) (This is the workflow of X11 [19]), a workflow in Fig.7(b) is learned through our algorithm which is more general than the original one. The workflow in Fig.7(b) can generate event traces such as $\langle B, E, \dots, E \rangle$ and $\langle A, D, E, \dots, E \rangle$ that cannot be generated by the original workflow Fig.7(a). We will leave it for future work to deal with these problems.

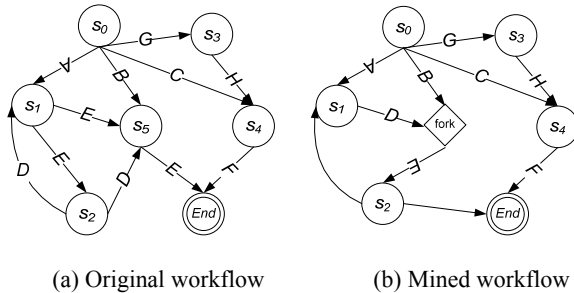


Figure 7. An example of the over-generalization problem

The proposed method only finds out the workflow that minimizes the defined complexity metric, and it does not provide any means to tune the precision and recall of the learned workflow. In addition, the workflow with minimal complexity may not be exactly the same as the real workflow or the one a human operator wants to see.

7 CONCLUSION

Most existing techniques for mining program workflow models can only learn models from sequential event traces. They cannot be applied to traces of interleaved events which are prevalent in distributed or parallel programs (or some event driven programs). In this paper, we proposed an approach to automatically discover program execution workflows from traces of interleaved events.

We extend the traditional state machine to support concurrency by introducing *split/merge* states and *fork/join* states. Our mining approach is based on the statistical inference of temporal dependency relations from event traces. We then use such dependency relations to construct a basic workflow by building the *connections* among neighboring events. After that, we further refine the workflow by validating it with event traces. During the validation procedure, we add the shortcut transitions, loop structure, and fork/join states into the workflow model to make sure that all event traces can be interpreted by the workflow model. The experimental results on both simulated event traces and real program traces demonstrate that our approach can learn the workflow with a high precision.

Although our work is motivated by the purpose of software comprehension, workflow mining is a basic research topic that has a wide range of application fields other than software engineering. We believe our approach can be widely applied in many applications, such as business intelligence. Future research directions include integrating high order temporal dependencies, incorporating domain or existing knowledge about a program, allowing for a workflow model having an event type at multiple points.

8 REFERENCES

- [1]. G. Ammons, R. Bodik, and J. R. Larus, “Mining Specifications”, in Proc. of *POPL’02*, Jan. 2002, Portland, USA.
- [2]. David Lo, L. Mariani, and M. Pezzè, “Automatic Steering of Behavioral Model Inference”, in Proc. of the *7th ESEC&FSE*, Aug. 2009, The Netherlands.
- [3]. R. Silva, J. Zhang, and J. G. Shanahan, “Probabilistic Workflow Mining”, in Proc. of *ACM SIGKDD’05*, Aug. 2005, Chicago, USA.
- [4]. D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore. “Investigation of Failure Causes in Work-load Driven Reliability Testing”, In Proc. of the *4th SOQA*, Sep. 2007.
- [5]. D. Lorenzoli, L. Mariani, and M. Pezzè. “Automatic Generation of Software Behavioral Models”, In Proc. of *ICSE*, 2008.
- [6]. L. Mariani, M. Pezzè. “Dynamic Detection of COTS Components Incompatibility”, *IEEE Software*, 24(5):76–85, Sep./Oct. 2007.
- [7]. G. Greco, A. Guzzo, L. Pontieri, and D. Sacca. “Mining Expressive Process Models by Clustering Workflow Traces”, in Proc. of *PAKDD*, 2004.
- [8]. N. Walkinshaw, K. Bogdanov, “Inferring Finite-state Models with Temporal Constraints”, In Proc. of the *23rd ASE*, 2008.
- [9]. A. Biermann, J. Feldman, “On the Synthesis of Finite-state Machines from Samples of Their Behavior”, *IEEE Transactions on Computers*, 21:591–597, 1972.
- [10]. Anand V. Raman, Jon D. Patrick, “The SK-strings Method for Inferring PFSA”, In Proc. of the *workshop on automata induction, grammatical inference and language acquisition*, 1997.
- [11]. S. Ferg, “Event-Driven Programming: Introduction, Tutorial, History”, <http://eventdrivenpgm.sourceforge.net/>, Jan. 2006.
- [12]. W. van der Aalst, A. Wejters, “Process mining: a research agenda”, *Computers and Industry*, vol. 53, pp.231–244, 2004.
- [13]. G. Greco, A. Guzzo, G. Manco, L. Pontieri, and D. Saccà, “Mining Constrained Graphs: The Case of Workflow Systems”, *European Workshop on Inductive Database and Constraint Based Mining 2004*.
- [14]. R. Agrawal, D. Gunopulos, and F. Leymann, “Mining Process Models from Workflow Logs”, in Proc. of the *6th EDBT*, pp. 469–483, 1998.

- [15]. W. van der Aalst, A. J. M. M. Weijters and L. Maruster, “Workflow Mining: Discovering Process Models from Event Logs”, *IEEE Trans. on KDE*, vol. 16, 2004.
- [16]. G. Schimm, “Mining Exact Models of Concurrent Workflows”, *Computers and Industry*, vol. 53, pp.265–281, 2004.
- [17]. J. E. Cook, Z. Du, C. Liu, and A. L. Wolf, “Discovering Models of Behavior for Concurrent Workflows”, *Computers and Industry*, vol. 53, pp.297–319, 2004.
- [18]. P. Grubb, A. Takang, “Software Maintenance”, *World Scientific Publishing*, 2003, ISBN 9789812384256.
- [19]. David Lo, S. C. Khoo, “SMaRTIC: Towards Building an Accurate, Robust and Scalable Specification Miner”, in *Proc. of FSE*, Nov. 2006.
- [20]. J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, “Mining Invariants from Console Logs for System Problem Detection”, In *proc. of USENIX Annual Technical Conference*, Boston, Jun. 2010.
- [21]. David Lo, S. C. Khoo, “QUARK: Empirical Assessment of Automaton-based Specification Miners”, In *Proc. WCRE*, 2006.
- [22]. Hadoop. <http://hadoop.apache.org/core>. 2009.
- [23]. Y. Zhou, T. Lau, K. Kontogiannis, T. Tong, and R. McKegney, “Model-driven Business Processes Recovery”, in *Proc. WCRE*, 2004.
- [24]. Q. Fu, J.-G. Lou, Y. Wang, and J. LI, “Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis”, In *Proc. of ICDM*, Florida, Dec. 2009.
- [25]. J.-G. Lou, Q. Fu, S. Yang, and J. Li, “Mining Program Workflow from Interleaved Logs”, Technical Report of Microsoft Research, MSR-TR-2010-6, 2010.
- [26]. M. Pradel, T. R. Gross, “Automatic Generation of Object Usage Specifications from Large Method Traces”, In *Proc. of ASE*, pp. 371-382, 2009.
- [27]. M. Acharya, T. Xie, J. Pei, and J. Xu, “Mining API Patterns as Partial Orders from Source Code: from Usage Scenarios to Specifications”, In *Proc. of SIGSOFT FSE*, pp. 25-34, 2007.
- [28]. M. Gabel, Z. Su, “Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces”, In *Proc. of SIGSOFT FSE*, pp.339-349, 2008.
- [29]. J. YANG, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, “Perracotta: Mining Temporal API Rules from Imperfect Traces”, In *Proc. of ICSE*, pp.282-291, 2006.
- [30]. David Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani, “Mining Quantified Temporal Rules: Formalism, Algorithm, and Evaluation”, In *Proc. of WCRE*, pp. 62-71, 2009.
- [31]. A. Wasylkowski, A. Zeller, “Mining Temporal Specifications from Object Usage”, In *Proc. of ASE*, pp. 295-306, 2009.

9 APPENDIX A

In this appendix, we present the detail of our refinement algorithm. For every event type a , and state q , we define two integer values $N[a]$ and $C[a, q]$. Here, $N[a]$ denotes the number of times that event type a in the training event traces cannot be inferred by the workflow during refinement. $C[a, q]$ records the number of workflow threads whose current state is q when an uninterpretable event of type a is encountered. f_1 and f_2 are two flags used to indicate whether current workflow model \mathbf{W} should be refined by a loop decision or a fork decision. Their initial values are both false. \mathbf{W}_1 and \mathbf{W}_2 are two temporary workflow models (refer to section 5.2). The whole refinement process contains the following steps.

Step 0. Let $C[a, q] = 0$, $K[q] = \text{an empty set}$, and $N[a] = 0$ for any $a \in (\Sigma \setminus \emptyset)$ and $q \in S$; $p_{\text{ass}} = \text{true}$;

Step 1. For an input training event trace L , we use the method presented in [25] to interpret the trace. Once there is an event l in L that cannot be interpreted, we increase $N[l]$ by 1, and set $p_{\text{ass}} = \text{false}$. At the same time, for each workflow thread W_k , we increase $C[l, q_k]$ by 1 where q_k is the active state of W_k and q_k is not a connection between two events having a strict forward dependence.

Step 2. For each event trace in the training set, we carry out the process of **Step 1**.

Step 3. For each event a with a non-zero value of $N[a]$, we find out a state q_t that satisfies $C[a, q_t] = \max_{q_i \in S} (C[a, q_i])$, and add a to $K[q_t]$.

Step 4. If $p_{\text{ass}} = \text{false}$, we find an arbitrary element $q \in \text{Find_Root}(\{p \mid p \in S \cap K[p] \neq \emptyset\})$ (refer to section 5.1), and find event a that does not have any predecessor in $K[q]$, then goto **Step 5**. Otherwise, goto **Step 6**.

Step 5. We denote q' as the preceding state of a in the current workflow model \mathbf{W} . If $f_1 = \text{false}$, we set $\mathbf{W}' = \mathbf{W}$, $f_1 = \text{true}$, and update \mathbf{W} by adding a shortcut transition from q to q' , else update \mathbf{W} by setting q' as a fork state. After that, back to **Step 0**.

Step 6. If $p_{\text{ass}} = \text{True}$ and $f_1 = \text{false}$, we mark all join states and terminate the execution of the algorithm. Otherwise, if $f_2 = \text{false}$, we set $\mathbf{W}_1 = \mathbf{W}$ and goto **Step 7**; else $\mathbf{W}_2 = \mathbf{W}$ and go to **Step 8**.

Step 7. Set $\mathbf{W} = \mathbf{W}'$, $f_2 = \text{true}$, and then back to **Step 0**.

Step 8. Set $f_1 = f_2 = \text{false}$ and $\mathbf{W} = \mathbf{W}'$. If $cp(\mathbf{W}_1) \leq cp(\mathbf{W}_2)$, we update \mathbf{W} by adding a shortcut transition from q to q' ; else update \mathbf{W} by setting q' as a fork state. After that, back to **Step 0**.