

# Towards Mining Structural Workflow Patterns

Walid Gaaloul<sup>1</sup>, Karim Baïna<sup>2</sup>, and Claude Godart<sup>1</sup>

<sup>1</sup> LORIA - INRIA - CNRS - UMR 7503,  
BP 239, F-54506 Vandœuvre-lès-Nancy Cedex, France

<sup>2</sup> ENSIAS, Université Mohammed V - Souissi,  
BP 713 Agdal - Rabat, Morocco  
baina@ensias.ma, {gaaloul, godart}@loria.fr

**Abstract.** Collaborative information systems are becoming more and more complex, involving numerous interacting business objects within considerable processes. Analysing the interaction structure of those complex systems will enable them to be well understood and controlled. The work described in this paper is a contribution to these problems for workflow based process applications. In fact, we discover workflow patterns from traces of workflow events based on a workflow mining technique. Workflow mining proposes techniques to acquire a workflow model from a workflow log. Mining of workflow patterns is done by a statistical analysis of log-based event. Our approach is characterised by a "local" workflow patterns discovery that allows to cover partial results and a dynamic technique dealing with concurrency.

**Keywords:** workflow patterns, workflow mining, business process reengineering.

## 1 Introduction

With the technological improvements and the continuous increasing market pressures and requirements, collaborative information systems are becoming more and more complex, involving numerous interacting business objects. Analysing interactions of those complex systems will enable them to be well understood and controlled. Our paper is a contribution to this problem in a particular context : workflow application analysis and control by mining techniques (a.k.a. "reversing processes" [1]).

In our approach, we start by collecting log information from workflow processes instances as they took place. Then we build, through statistical techniques, a graphical intermediary representation modelling elementary dependencies over workflow activities executions. These dependencies are then refined to discover workflow patterns [2]. This paper is structured as follows. Section 2 explains our workflow log model. Section 3, we detail our structural workflow patterns mining algorithm. Section 4 discusses related work, and concludes.

## 2 Workflow Log Model

As shown in the UML class diagram in figure 1, WorkflowLog is composed of a set of EventStreams (definition 1). Each EventStream traces the execution of one case (instance). It consists of a set of events (Event) that captures the activities life cycle

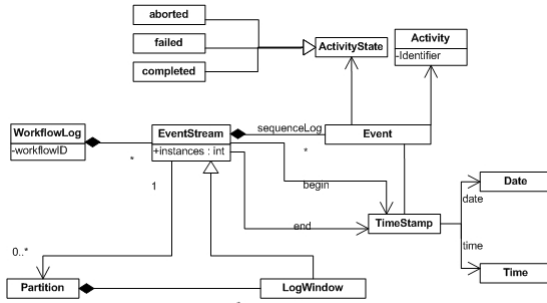


Fig. 1. Workflow Log Model

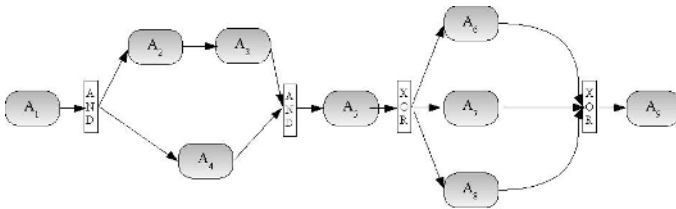


Fig. 2. Running example of workflow

performed in a particular workflow instance. An Event is described by the activity identifier that it concerns, the current activity state (aborted, failed and completed) and the time when it occurs (TimeStamp). A Window defines a set of Events over an EventStream. Finally, a Partition builds a set of partially overlapping Windows partition over an EventStream.

**Definition 1.** (EventStream)

An EventStream represents the history of a workflow instance events as a tuple stream=(begin, end, sequenceLog, instances) where:

- ✓(begin:TimeStamp) and (end:TimeStamp) are the log beginning and end time;
- ✓sequenceLog : Event\* is an ordered Event set belonging to a workflow instance;
- ✓instances : int is the instance number.

A WorkflowLog is a set of EventStreams. WorkflowLog=(workflowID, {EventStream<sub>i</sub>, 0 ≤ i ≤ number of workflow instances}) where EventStream<sub>i</sub> is the event stream of the i<sup>th</sup> workflow instance.

Here is an example of an EventStream extracted from the workflow example of figure 2 in its 5<sup>th</sup> instantiation :

**L** = EventStream((13/5,5:42:12), (14/5, 14:01:54), [Event( Event(" A<sub>1</sub>", completed, (13/5, 5:42:12)), Event(" A<sub>2</sub>", completed, (13/5,11:11:12)), Event(" A<sub>4</sub>", completed, (13/5,14:01:54)), Event(" A<sub>3</sub>", completed, (14/5, 00:01:54)), Event(" A<sub>5</sub>", completed, (14/5,5:45:54)), Event(" A<sub>6</sub>", aborted, (14/5,10:32:55)), Event(" A<sub>7</sub>", completed, (14/5,10:32:55)), Event(" A<sub>9</sub>", completed, (14/5,14:01:54))],5)

### 3 Mining Structural Workflow Patterns

As we state before, we start by collecting WorkflowLog from workflow instances as they took place. Then we build, through statistical techniques, a graphical intermediary representation modelling **elementary dependencies** over workflow logs (see section 3.1). These dependencies are then refined by **advanced structural workflow patterns** (see section 3.2).

#### 3.1 Discovering Elementary Dependencies

In order to discover direct dependencies from a WorkflowLog, we need an intermediary representation of this WorkflowLog through a statistical analysis. We call this intermediary representation : statistical dependency table (or SDT). SDT is built through a statistical calculus that extracts elementary dependencies between activities of a WorkflowLog that are executed without "exceptions" (*i.e.* they reached successfully their completed state). Then, we need to filter the analysed WorkflowLog and take only EventStreams of instances executed "correctly". We denote by  $\text{WorkflowLog}_{\text{completed}}$  this workflow log projection. Thus, the unique necessary condition to discover elementary dependencies is to have workflow logs containing at least the completed event states. These features allow us to mine control flow from "poor" logs which contain only completed event state. By the way, any information system using transactional systems or workflow management systems offer this information in some form [1].

For each activity  $A$ , we extract from  $\text{workflowLog}_{\text{completed}}$  the following information in the statistical dependency table (SDT): (i) The overall occurrence number of this activity (denoted  $\#A$ ) and (ii) The elementary dependencies to previous activities  $B_i$  (denoted  $P(A/B_i)$ ). The size of SDT is  $N * N$ , where  $N$  is the number of workflow activities. The  $(m,n)$  table entry (notation  $P(m/n)$ ) is the frequency of the  $n^{\text{th}}$  activity immediately preceding the  $m^{\text{th}}$  activity. The initial SDT in table 1 represents a fraction of the SDT of our workflow example given in figure 2. For instance, in this table  $P(A_3/A_2)=0.69$  expresses that if  $A_3$  occurs then we have 69% of chance that  $A_2$  occurs directly before  $A_3$  in the workflow log. As it was calculated SDT presents some problems to express correctly activities dependencies relating to concurrent behaviour. In the following, we detail these issues and propose solutions to correct them.

**Discarding erroneous dependencies :** If we assume that each EventStream from WorkflowLog comes from a sequential (*i.e.* no concurrent behaviour) workflow, a zero entry in SDT represents a causal independence and a non-zero entry means a causal dependency relation (*i.e.* sequential or conditional relation). But, in case of concurrent behaviour, as we can see in workflow patterns (like and-split, and-join, or-join, etc.) the EventStreams may contain interleaved events sequences from concurrent threads. As a consequence, some entries in initial SDT can indicate non-zero entries that do not correspond to dependencies. For example the events stream given in section 2 "suggests" erroneous causal dependencies between  $A_2$  and  $A_4$  in one side and  $A_4$  and  $A_3$  in another side. Indeed,  $A_2$  comes immediately before  $A_4$  and  $A_4$  comes immediately before  $A_3$  in this events stream. These erroneous entries are reported by  $P(A_4/A_2)$  and  $P(A_3/A_4)$  in initial SDT which are different to zero. These entries are erroneous be-

**Table 1.** Fraction of Statistical Dependencies Table ( $P(x/y)$ ) and activities Frequencies (#)

Initial SDT							Final SDT						
$P(x/y)$	$A_1$	$A_2$	$A_3$	$A_4$	$A_5$	$A_6$	$P(x/y)$	$A_1$	$A_2$	$A_3$	$A_4$	$A_5$	$A_6$
$A_1$	0	0	0	0	0	0	$A_1$	0	0	0	0	0	0
$A_2$	<b>0.54</b>	0	0	<u>0.46</u>	0	0	$A_2$	<b>1</b>	0	0	<u>-1</u>	0	0
$A_3$	0	<b>0.69</b>	0	<u>0.31</u>	0	0	$A_3$	0	<b>1</b>	0	<u>-1</u>	0	0
$A_4$	<b>0.46</b>	<u>0.31</u>	<u>0.23</u>	0	0	0	$A_4$	<b>1</b>	<u>-1</u>	<u>-1</u>	0	0	0
$A_5$	0	0	<b>0.77</b>	<b>0.23</b>	0	0	$A_5$	0	0	<b>1</b>	<b>1</b>	0	0
$A_6$	0	0	0	0	1	0	$A_6$	0	0	0	0	1	0

$\#A_1 = \#A_2 = \#A_3 = \#A_4 = \#A_5 = \#A_9 = 100,$   
 $\#A_6 = 23, \#A_7 = 42, \#A_8 = 35$

cause there is no causal dependencies between these activities as suggested (i.e. noisy SDT). Underlined values in initial SDT report this behaviour for other similar cases.

Formally, two activities  $A$  and  $B$  are in concurrence *iff*  $P(A/B)$  and  $P(B/A)$  entries in SDT are different from zero with the assumption that WorkflowLog is complete. Indeed, a WorkflowLog is complete if it covers all possible cases (i.e. if a specific routing element can appear in the mined workflow model, the log should contain an example of this behaviour in at least one case). Based on this definition, we propose an algorithm to discover activities parallelism and then mark the erroneous entries in SDT. Through this marking, we can eliminate the confusion caused by the concurrence behaviour producing these erroneous non-zero entries. Our algorithm scans the initial **SDT** and marks concurrent activities dependencies by changing their values to  $(-1)$ .

**Discovering indirect dependencies:** For concurrency reasons, an activity might not depend on its immediate predecessor in the events stream, but it might depend on another "indirectly" preceding activity. As an example of this behaviour,  $A_4$  is logged between  $A_2$  and  $A_3$  in the events stream given in section 2. As consequence,  $A_2$  does not occur always immediately before  $A_3$  in the workflow log. Thus we have only  $P(A_3/A_2) = 0.69$  that is an under evaluated dependency frequency. In fact, the right value is 1 because the execution of  $A_3$  depends exclusively on  $A_2$ . Similarly, values in bold in initial SDT report this behaviour for other cases.

### Definition 2. Window

A log window defines a log slide over an events stream  $S$ : **stream** ( $bStream, eStream, sLog, workflowocc$ ). Formally, we define a log window as a triplet **window**( $wLog, bWin, eWin$ ):

$\checkmark (bWin : TimeStamp)$  and  $(eWin : TimeStamp)$  are the moment of the window beginning and end (with  $bStream \leq bWin$  and  $eWin \leq eStream$ )

$\checkmark wLog \subset sLog$  and  $\forall e: event \in S.sLog$  where  $bWin \leq e.TimeStamp \leq eWin \Rightarrow e \in wLog$ .

To discover these indirect dependencies, we introduce the notion of *activity concurrent window* (definition 2). An *activity concurrent window* (ACW) is related to the activity of its last event covering its directly and indirectly preceding activities. Initially, the width of ACW of an activity is equal to 2. Every time this activity is in concurrence with an other activity we add 1 to this width. If this activity is not in concurrence with

other activities and has preceding concurrent activities, then we add their number to ACW width. For example the activity  $A_4$  is in concurrence with  $A_2$  and  $A_3$  the width of its ACW is equal to 4. Based on this, we propose an algorithm that calculates for each activity the activity concurrent width regrouped in the ACW table. This algorithm scans the "marked" SDT calculated in last section and updates the ACW table.

**Definition 3.** Partition

A *partition* builds a set of partially overlapping Windows partition over an events stream.  $Partition : WorkflowLog \rightarrow (Window)^*$

$Partition(S : EventStream(bStr, eStr, sLog : (Evt_i \ 1 \leq i \leq n), wocc)) = \{w_i : Window; 1 \leq i \leq n\}$  where :  $Evt_i =$  the last event in  $w_i \wedge width(w_i) = ACWT[Evt_i.ActivityID]$ .

After that, we proceed through an EventStreams partition (definition 3) that builds a set of partially overlapping windows over the EventStreams using the ACW table. Finally, we compute the final SDT. For each ACW, we compute for its last activity the frequencies of its preceding activities. The final SDT will be found by dividing each row entry by the frequency of the row's activity. Note that, our approach adjusts **dynamically**, through the width of ACW, the process calculating activities dependencies. Indeed, this width is sensible to concurrent behaviour : it increases in case of concurrence and is "neutral" in case of concurrent behaviour absence. Now, we can compute the final SDT (table 1) which will be used to discover workflow patterns.

### 3.2 Discovering Advanced Dependencies: Workflow Patterns

We have identified three kinds of statistical properties (sequential, conditional and concurrent) which describe the main behaviours of workflow patterns. Then, we have specified these properties using SDT's statistics. We use these properties to identify separately workflow patterns from workflow logs. We begin with the statistic exclusive dependency property which characterises, for instance, the sequence pattern.

**Property 1. Mutual exclusive dependency property:** A *mutual exclusive dependency relation* between an activity  $A_i$  and its immediately preceding previous activity  $A_j$  specifies that the enactment of the activity  $A_i$  depends only on the completion of activity  $A_j$  and the completion of  $A_j$  enacts only the execution of  $A_i$ . It is expressed in terms of:

$\checkmark$  activities frequencies :  $\#A_i = \#A_j$

$\checkmark$  activities dependencies :  $P(A_i/A_j) = 1 \wedge \forall k \neq j; P(A_i/A_k) = 0 \wedge \forall l \neq i; P(A_l/A_j) = 0$ .

The next two statistic properties: concurrency property (property 2) and choice property (property 3) are used to insulate statistical patterns behaviour in terms of concurrence and choice after a "fork" or before a "join" point.

**Property 2. Concurrency property:** A *concurrency relation* between a set of activities  $\{A_i, 0 \leq i \leq n\}$  belonging to the same workflow specifies how, in terms of concurrency, the execution of these activities is performed. This set of activities is commonly found after a "fork" point or before a "join" point. We have distinguished three activities concurrency behaviours:

✓ *Global concurrency where in the same instantiation the whole activities are performed simultaneously* :  $\forall 0 \leq i, j \leq n; \#A_i = \#A_j \wedge P(A_i/A_j) = -1$

✓ *Partial concurrency where in the same instantiation we have at least a partial concurrent execution of activities* :  $\exists 0 \leq i, j \leq n; P(A_i/A_j) = -1$

✓ *No concurrency where there is no concurrency between activities*:  $\forall (0 \leq i, j \leq n; P(A_i/A_j) \geq 0)$

**Property 3. Choice property:** A **choice** relation specifies which activities are executed after a "fork" point or before a "join" point. The two actors of a "fork" point (respectively a "join" point) perform this relation are : (**actor 1**) an activity  $A$  from which comes (respectively to which) a single thread of control which splits (respectively converges) into (respectively from) (**actor 2**) multiple activities  $\{A_i, 1 \leq i \leq n\}$ . We have distinguished three activities choice behaviours :

✓ *Free choice where a part of activities from the second actor are chosen. Expressed statistically, we have in terms of activities frequencies ( $\#A \leq \sum_{i=1}^n (\#A_i)$ )  $\wedge (\forall (1 \leq i, j \leq n; \#A_i \leq \#A))$  and in terms of activities dependencies we have :*

✓ *In "fork" point* :  $\forall 1 \leq i \leq n; P(A_i/A) = 1$

✓ *In "join" point* :  $1 < \sum_{i=1}^n P(A/A_i) < n$

✓ *Single choice where only one activity is chosen from the second actor. Expressed statistically, we have in terms of activities frequencies ( $\#A = \sum_{i=1}^n (\#A_i)$ ) and in terms of activities dependencies we have :*

✓ *In "fork" point* :  $\forall 1 \leq i \leq n; P(A_i/A) = 1$

✓ *In "join" point* :  $\sum_{i=1}^n P(A/A_i) = 1$

✓ *No choice where all activities in the second actor are executed. Expressed statistically, we have in terms of activities frequencies  $\forall 1 \leq i \leq n \#A = \#A_i$  and in terms of activities dependencies we have :*

✓ *In "fork" point* :  $\forall 1 \leq i \leq n; P(A/A_i) = 1$

✓ *In "join" point* :  $\forall 1 \leq i \leq n; P(A_i/A) = 1$

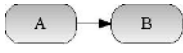
Using these statistical specifications of sequential, conditional and concurrent properties, the last step is the identification of workflow patterns through a set of rules. In fact, each pattern has its own statistical features which abstract statistically its causal dependencies, and represent its unique identifier. These rules allow, if workflow log is completed, to mine the whole workflow patterns hidden in this workflow.

Our control flow mining rules are characterised by a "local" workflow patterns discovery. Indeed, these rules are context-free, they proceed through a **local log analysing** that allows us to **recover partial results** of mining workflow patterns. In fact, to discover a particular workflow pattern we need only events relating to pattern's elements. Thus, even using only fractions of workflow log, we can discover correctly corresponding workflow patterns (which their events belong to these fractions).

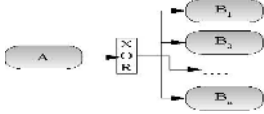
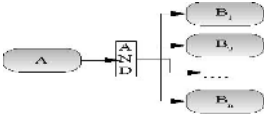
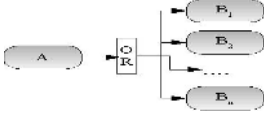
We divided the workflows patterns in three categories : sequence, fork and join patterns. In the following we present rules to discover the most interesting workflow patterns belonging to these three categories. Note that the rules formulas noted by : (P1) finger the Statistic exclusive dependency property, (P2) finger statistic concurrency property and (P3) finger statistic choice property.

**Discovering sequence pattern:** In this category we find only the sequence pattern (table 2). In this pattern, the enactment of the activity  $B$  depends only on the completion

**Table 2.** Rules of sequence workflow pattern

Rules	workflow patterns
(P1) ( $\#B = \#A$ )	Sequence pattern
(P1) ( $P(B/A) = 1$ )	

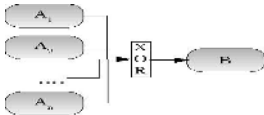
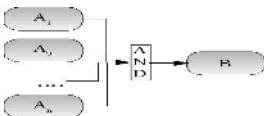
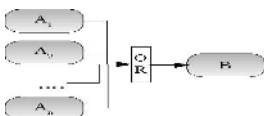
**Table 3.** Rules of fork workflow patterns

Rules	workflow patterns
(P3)( $\sum_{i=0}^n (\#B_i) \neq \#A$ )	xor-split pattern
(P3)( $\forall 0 \leq i \leq n; P(B_i/A) = 1$ ) $\wedge$ (P2)( $\forall 0 \leq i, j \leq n; P(B_i/B_j) = 0$ )	
(P3)( $\forall 0 \leq i \leq n; \#B_i = \#A$ )	and-split pattern
(P3)( $\forall 0 \leq i \leq n; P(B_i/A) = 1$ ) $\wedge$ (P2)( $\forall 0 \leq i, j \leq n; P(B_i/B_j) = -1$ )	
(P3)( $\#A \leq \sum_{i=0}^n (\#B_i)$ ) $\wedge$ ( $\forall 0 \leq i \leq n; \#B_i \leq \#A$ )	or-split pattern
(P3)( $\forall 0 \leq i \leq n; P(B_i/A) = 1$ ) $\wedge$ (P2)( $\exists 0 \leq i, j \leq n; P(B_i/B_j) = -1$ )	

of activity  $A$ . So we have used the statistical exclusive dependency property to ensure this relation linking  $B$  to  $A$ .

**Discovering fork patterns:** This category (table 3) has a "fork" point where a single thread of control splits into multiple threads of control which can be, according to the used pattern, executed or not. The dependency between the activities  $A$  and  $B_i$  before and after "fork" point differs in the three patterns of this category: and-split, or-split, xor-split. These dependencies are characterised by the statistic choice properties. The xor-split pattern, where one of several branches is chosen after "fork" point, adopts the single choice property. and-split and or-split patterns differentiate themselves through the no choice and free choice properties. Effectively, only a part of activities are executed in the or-split pattern after a "fork" point, while all the  $B_i$  activities are executed in the and-split pattern. The non-parallelism between  $B_i$ , in the xor-split pattern are ensured by the no concurrency property while the partial and the global parallelism in or-split and and-split is identified through the application of the statistical partial and global concurrency properties.

**Table 4.** Rules of join workflow patterns

Rules	workflow patterns
$(P3)(\sum_{i=0}^n (\#A_i)=\#B)$  $(P3)(\sum_{i=0}^n P(B/A_i)=1) \wedge$ $(P2)(\forall 0 \leq i, j \leq n; P(A_i/A_j) = 0)$	xor-join pattern  
$(P3)(\forall 0 \leq i \leq n; \#A_i=\#B)$  $(P3)(\forall 0 \leq i \leq n; P(B/A_i) = 1) \wedge$ $(P2)(\forall 0 \leq i, j \leq n P(A_i/A_j) = -1)$	and-join pattern  
$(P3)(m * \#B \leq \sum_{i=0}^n (\#A_i))$ $\wedge (\forall 0 \leq i \leq n; \#A_i \leq \#B)$ $(P3)(m \leq \sum_{i=0}^n P(B/A_i) \leq n)$ $\wedge (P2)(\exists 0 \leq i, j \leq n; P(A_i/A_j) = -1)$	M-out-of-N-Join pattern  

**Discovering join patterns:** This category (table 4) has a "join" point where multiple threads of control merge in a single thread of control. The number of necessary branches for the causal of the activity  $B$  after the "join" point depends on the used pattern.

To identify the three patterns of this category: and-join pattern, xor-join pattern and M-out-of-N-Join pattern we have analysed dependencies between the activities  $A_i$  and  $B$  before and after "join". Thus the single choice and the no concurrency properties are used to identify the xor-join pattern where two or more alternative branches come together without synchronisation and none of the alternative branches is ever executed in parallel. As for the and-join pattern where multiple parallel activities converge into one single thread of control, the no choice and the global concurrency are both used to discover this pattern. In contrary of the M-out-of-N-Join pattern, where we need only the termination of  $M$  activities from the incoming  $N$  parallel paths to enact the  $B$  activity, The concurrency between  $A_i$  would be partial and the choice is free.

## 4 Discussion

The idea of applying process mining in the context of workflow management was first introduced in [3]. This work proposes methods for automatically deriving a formal model of a process from a log of events related to its executions and is based on workflow graphs. Cook and Wolf [4] investigated similar issues in the context of software engineering processes. They extended their work limited initially to sequential processes, to concurrent processes [5]. Herbst [6,7] presents an inductive learning component used to support the acquisition and adaptation of sequential process models, generalising execution traces from different workflow instances to a workflow model covering all traces. Starting from the same kind of process logs, van der Aalst et al. explore also proposes techniques to discover workflow models based on Petri nets. Beside analysing



**Table 5.** Comparing Process Mining Tools

	EMiT [12]	Little Thumb [13]	InWoLvE [14]	Process Miner [15]	WorkflowMiner
<i>Structure</i>	Graph	Graph	Graph	Block	<b>Patterns</b>
<i>Local discovery</i>	No	No	No	No	<b>Yes</b>
<i>Parallelism</i>	Yes	Yes	Yes	Yes	<b>Yes</b>
<i>Non-free choice</i>	No	No	No	No	<b>Yes</b>
<i>Loops</i>	Yes	Yes	Yes	Yes	<b>No</b>
<i>Noise</i>	No	Yes	Yes	No	<b>No</b>
<i>Time</i>	Yes	No	No	No	<b>No</b>

process structure, there exist related works dealing with process behaviour reporting, such as [8,9,10] that describe tools and case studies that discuss several features, such as analysing deadline expirations, predicting exceptions, process instances monitoring.

We have implemented our presented workflow patterns mining algorithms within our prototype WorkflowMiner [11]. WorkflowMiner is written in Java and based on Bonita Workflow Management System<sup>1</sup> and XProlog Java Prolog API<sup>2</sup>. Starting from executions of a workflow, (1) events streams are gathered into an XML log. In order to be processed, (2) these workflow log events are wrapped into a 1<sup>st</sup> order logic format, compliant with UML class diagrams shown in figure 1. (3) Mining rules are applied on resulted 1<sup>st</sup> order log events to discover workflow patterns. We use a Prolog-based presentation for log events, and mining rules. (4) Discovered patterns are given to the workflow designer so he/she will have a look on the analysis of his/her deployed workflow to restructure or redesign it either manually or semi-automatically.

Table 5 compares our WorkflowMiner prototype to workflow mining tools representing previous studied approaches. We focus on seven aspects: **structure** of the target discovering language, **local discovery** dealing with incomplete parts of logs (opposed to global and complete log analysis), **parallelism** (a fork path beginning with and-split and ending with and-join), **non-free choice** (NFC processes mix synchronisation and choice in one construct), **loops** (cyclic workflow transitions, or paths), **noise** (situation where log is incomplete or contains errors or non-representative exceptional instances), and **time** (event time stamp information used to calculate performance indicators such as waiting/synchronisation times, flow times, load/utilisation rate, etc.).

WorkflowMiner can be distinguished by supporting **local discovery** through a set of control flow mining rules that are characterised by a "local" workflow patterns discovery enabling **partial results** to be discovered correctly. Moreover, even if non-free choice (NFC) construct is mentioned as an example of a workflow pattern that is difficult to mine, WorkflowMiner discovers M-out-of-N-Join pattern which can be seen as a generalisation of the basic Discriminator pattern that were proven to be inherently non free-choice. None of related works can deal with such constructs.

In our future works, we aim to discover more complex patterns by enriching our workflow log, and by using more metrics (*e.g.* entropy, periodicity, etc.). We are also interested in the modeling and the discovery of more complex transactional characteristics of cooperative workflows [16].

<sup>1</sup> Bonita, [bonita.objectweb.org](http://bonita.objectweb.org)

<sup>2</sup> XProlog, [www.iro.umontreal.ca/~vaucher/XProlog/](http://www.iro.umontreal.ca/~vaucher/XProlog/)

## References

1. W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: a survey of issues and approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003.
2. W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.
3. Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. Mining process models from workflow logs. *Lecture Notes in Computer Science*, 1377:469–498, 1998.
4. Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):215–249, 1998.
5. Jonathan E. Cook and Alexander L. Wolf. Event-based detection of concurrency. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45. ACM Press, 1998.
6. Joachim Herbst. A machine learning approach to workflow management. In *Machine Learning: ECML 2000, 11th European Conference on Machine Learning, Barcelona, Catalonia, Spain*, volume 1810, pages 183–194. Springer, Berlin, May 2000.
7. Joachim Herbst and Dimitris Karagiannis. Integrating machine learning and workflow management to support acquisition and adaptation of workflow models. In *DEXA '98: Proceedings of the 9th International Workshop on Database and Expert Systems Applications*, page 745. IEEE Computer Society, 1998.
8. M. Sayal, F. Casati, M.C. Shan, and U. Dayal. Business process cockpit. *Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 880–883, 2002.
9. Daniela Grigori, Fabio Casati, Malu Castellanos, Umeshwar Dayal, Mehmet Sayal, and Ming-Chien Shan. Business process intelligence. *Comput. Ind.*, 53(3):321–343, 2004.
10. K. Baïna, I. Berrada, and L. Kjiri. A Balanced Scoreboard Experiment for Business Process Performance Monitoring : Case study. In *1st International E-Business Conference (IEBC'05)*, Tunis, Tunisia, June 24-25, 2005.
11. W. Gaaloul, S. Alaoui, K. Baïna, and C. Godart. Mining Workflow Patterns through Event-data Analysis. In *The IEEE/IPSJ International Symposium on Applications and the Internet (SAINT'05). Workshop 6 Teamware: supporting scalable virtual teams in multi-organizational settings*. IEEE Computer Society Press, 2005.
12. Wil M. P. van der Aalst and B. F. van Dongen. Discovering workflow performance models from timed logs. In *Proceedings of the First International Conference on Engineering and Deployment of Cooperative Information Systems*, pages 45–63. Springer-Verlag, 2002.
13. A. J. M. M. Weijters and W. M. P. van der Aalst. Workflow mining: Discovering workflow models from event-based data. In Dousson, C., Hppner, F., and Quiniou, R., editors, *Proceedings of the ECAI Workshop on Knowledge Discovery and Spatial Data*, pages 78–84, 2002.
14. Joachim Herbst and Dimitris Karagiannis. Workflow mining with involve. *Comput. Ind.*, 53(3):245–264, 2004.
15. Guido Schimm. Process Miner - A Tool for Mining Process Schemes from Event-Based Data. In *Proceedings of the European Conference on Logics in Artificial Intelligence*, pages 525–528. Springer-Verlag, 2002.
16. W. Gaaloul, S. Bhiri, and C. Godart. Discovering workflow transactional behaviour event-based log. In *12th International Conference on Cooperative Information Systems (CoopIS'04)*, LNCS, Larnaca, Cyprus, October 25-29, 2004. Springer-Verlag.