Noname manuscript No.
(will be inserted by the editor)

Log-based transactional workflow mining

Walid Gaaloul · Khaled Gaaloul · Sami Bhiri · Armin Haller · Manfred Hauswirth

Received: date / Accepted: date

Abstract A continuous evolution of business process parameters, constraints and needs, hardly foreseeable initially, requires a continuous design from the business process management systems. In this article we are interested in developing a reactive design through process log analysis ensuring process re-engineering and execution reliability. We propose to analyse workflow logs to discover workflow transactional behaviour and to subsequently improve and correct related recovery mechanisms. Our approach starts by collecting workflow logs. Then, we build, by statistical analysis techniques, an intermediate representation specifying elementary dependencies between activities. These dependencies are refined to mine the transactional workflow model. The analysis of the discrepancies between the discovered model and the initially designed model enables us to detect design gaps, concerning particularly the recovery mechanisms. Thus, based on this mining step, we apply a set of rules on the initially designed workflow to improve workflow reliability.

Keywords workflow mining · transactional workflow · workflow patterns · workflow logs · process mining · business process analysis · business process intelligence · process reengineering · execution reliability · correction.

The work presented in this paper was partially supported by the EU under the SUPER project (FP6-026850) and by the Lion project supported by Science Foundation Ireland under Grant No. SFI/02/CE1/I131

Walid Gaaloul
GET/INT (Institut National des Telecommunications)
9, Rue Charles Fourier, 91011 Evry, France
E-mail: walid.gaaloul@it-sudparis.eu

Sami Bhiri, Armin Haller, Manfred Hauswirth
DERI-NUIG
IDA Business Park, Galway, Ireland
E-mail: firstname.lastname@deri.org

Khaled Gaaloul
SAP CEC Karlsruhe
Vincenz-Priessnitz-Strasse 1, 76131 Karlsruhe, Germany
E-mail: khaled.gaaloul@sap.com

Khaled Gaaloul
LORIA-INRIA-UMR 7503
BP 239, F-54506 Vandœuvre-les-Nancy Cedex, France
E-mail: firstname.lastname@loria.fr

1 Introduction

The increasing use of Workflow Management Systems (WfMS) in companies expresses their undeniable importance to improve the efficiency of their processes and their execution costs. However, 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. Consequently, in spite of its obvious potential, WfMS show some limitations to ensure a correct and reliable execution. Due to the complex design process and the initially unforeseeable character of other parameters which appear after the execution phase (users' evolution needs, unexpected execution exception, etc.), it is impossible to easily foresee and initially realize all necessary parameters for a "perfect" design.

A great diversification of company services and products lead to a continuous process evolution. New requirements emerge and existing processes change ("the only constant is change"). Consequently, the alignment of a process in regard to its observed evolution requires a permanent attention and reaction during the process life cycle. To maintain this alignment it is important to detect changes, i.e. the deviations of the described or prescribed behaviour. Analysing interactions of those complex systems will enable them to be well understood, controlled, and redesigned. It is obvious that the discovery, and the analysis of workflow interactions at runtime, would enable the designer to be alerted of design gaps, to better understand the model and to correct the recovery mechanisms. Indeed, this kind of analysis is very useful in showing cause effect relationships and to analyse the discrepancies between the discovered model and the initially designed model. These discrepancies can be used to detect initial design gaps which may be used in a re-engineering process.

Most previous approaches develop a set of techniques to analyse and check model correctness in their respective workflow model [1–4]. Although powerful, these approaches may fail to ensure reliable workflow execution in some cases. Indeed, it is neither possible nor intended by workflow designers to model all failures. As such the process description will become complex very soon [5]. Furthermore, workflow errors and exceptions are commonly not detected until the workflow model is executed. In this article, we present an original approach to ensure reliable workflow transactional behaviour. Different from previous works, our approach allows to address process evolution requirements and to correct potential design errors after runtime using workflow logs. Previous works only use specification properties which only reflect the designer's assumptions.

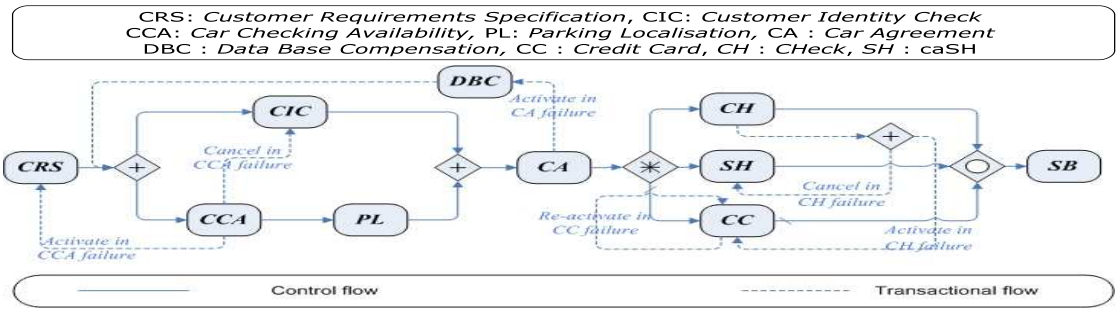
Basically, we describe a set of mining techniques which we have specified, proved and implemented in order to discover and improve workflow transactional behaviour. These workflow mining techniques are suitable tools to detect process changes during the execution which can reflect process evolution. Our approach aims at detecting and correcting design errors due to omissions in the initial design or workflow schema evolutions observed at runtime. With that aim, we discover firstly the "real" transactional workflow based on its execution logs. Thereafter, we use a set of rules to improve and correct its transactional behaviour. To the best of our knowledge, there are practically no approaches addressing transactional workflow re-engineering using mined results from logs. A set of tools were implemented in order to validate the different steps of our work.

1.1 Motivating Example

In this article, we will illustrate our ideas using a running example (see Figure 1). We consider a car rental scenario where one party acts as a broker offering to its customers a set of services based on the customer's earlier choice in the Customer Requirements Specification (CRS) activity. The Customer Identity Check (CIC) activity checks the customer ID while the Car Checking Availability (CCA) activity and the Parking Localisation (PL) provide information about available cars and the respective car rental companies supplier. Afterwards, the customer makes his choice and agrees on rental terms in the Car Agreement (CA) activity. Then, the customer is requested to pay either by Credit Card (CC), by CHeck (CH), or by caSH (SH). The customer can combine the payment by check and by cash. Finally, the bill is sent to the customer by the Send Bill (SB) activity.

To deal with workflow failures and to ensure a reliable execution, designers specify additional transactional interactions (dotted arrows). In our example, it was specified that if CA fails then the compensation activity (DBC) compensates already executed activities and the car rental discovery process (CIC, CCA and PL) should

Figure 1 Workflow Example in BPMN



be restarted. To ensure the car rental payment, the payment by credit card was specified as an alternative if the payment by check fails, and the payment by cash is simultaneously cancelled if it was combined with the failed payment by check. Besides, CC has the capability to be (re-)executed until success in case of failure. As for the failures of CCA (the workflow instance does not find any car propositions), the workflow instance cancels the CIC execution and restarts the CCA execution to (re-)enter the client requirements. The workflow designer did not provide failure handling mechanisms for the other activities and assumes that these activities never fail, or are not critical[1].

Let us suppose now that in reality (by observation of sufficient execution cases[2] [6]) CCA never fails but CIC can fail. This means there is no need to specify a recovery mechanism for CCA, and as CIC can fail we should also provide a recovery mechanism for it to resume the workflow execution. Starting from workflow logs, we propose, in this article, a set of workflow mining techniques that detect these transactional design gaps and provide help to correct them. Indeed, such incorrect specifications of the transactional behavior can result in unpredictable behavior, which, in turn, can lead to unavailability of resources, information integrity problems and global workflow execution failure.

Basically, there are two main approaches that deal with failures in existing BPMSs [7]: Ad Hoc and Run Time. The former specifies the exception handling logic within the normal behavior of the workflow. This makes the design of the workflow complicated for the designer. The raising of expected exceptions is typically unpredictable. Thus, it is often impossible to represent all exceptions of the business process model at design time. To overcome the limits of the first approach, the second one deals with exceptions at and after run time, meaning that there must be a business process expert who decides which changes have to be made to the business process logic in order to handle exceptions. In general, the first approach offers extreme and expensive solutions, and therefore some activity failures may deserve an a posteriori handling through a re-engineering process. In this case, activity failures are discovered after run time, and can be then re-designed. Our work can be considered as the first attempt to use mining techniques to ensure workflow execution reliability and workflow re-engineering.

1.2 Overview

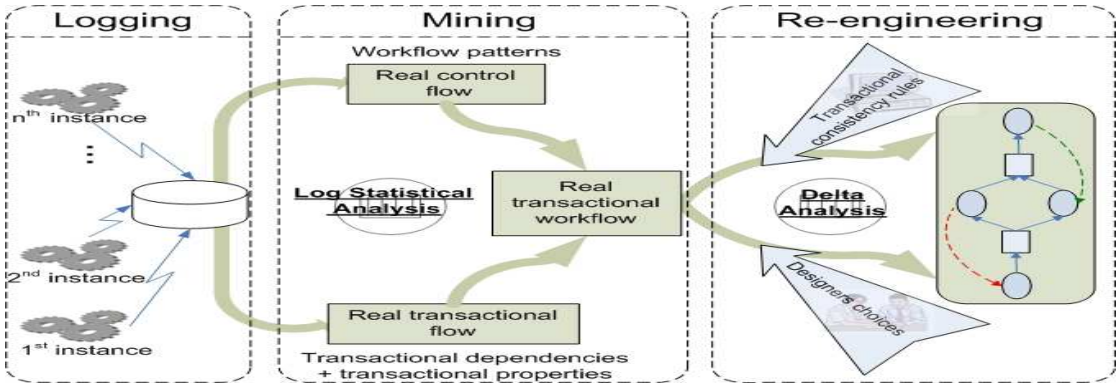
Figure 2 depicts an overview of our approach through the following steps :

– **Collecting workflow logs:** The purpose of this phase is to keep track of the workflow execution by capturing the relevant generated events. The number of instances and the log structure should be sufficiently rich to enable transactional mining.

[1] Their failures do not affect the execution of the workflow instance

[2] A sufficient execution cases is the number of workflow instances that are considered enough by the workflow designers to reproduce all possible workflow behaviours and to enable the workflow re-engineering phase

Figure 2 Overview



– **Mining the "real" transactional workflow:** The purpose of this phase is to mine the effective control flow (workflow pattern) [8,9] and transactional flow (activity transactional properties and dependencies) [10,11] based on its log only. We proceed in two steps. First, we perform statistical analysis to extract statistical dependency tables from workflow log. Then, a statistical specification of the control flow and the transactional flow is extracted using a set of rules applied on the statistical dependency tables.
– **Re-engineering the transactional behaviour:** Based on the mined results obtained from the previous step, we use a set of rules providing suggestions to correct and improve the workflow transactional behaviour and thereafter the workflow execution reliability. These rules depend on transactional consistency rules specified by the transactional workflow model. The designers decide based on the business process semantics and requirements on the set of suggestions to apply.

The remainder of this article is structured as follows. First, some distinctive concepts and prerequisites are detailed in section 2 to describe the adopted workflow transactional model. Therefore, we present in section 3 the structure of workflow event logs. Afterwards, we detail our transactional workflow mining techniques: section 4 computes statistical analysis upon logs, and section 5 discovers the transactional workflow model based on a statistical specification of its behaviour. Based on these mined results, we use a set of rules to improve workflow failure handling and recovery, and consequently process reliability (section 6). We illustrate, in section 7, the implementation efforts done to validate our approach. Finally, section 8 discusses the related works, before concluding in section 9.

2 Transactional Workflow Model

WfMS are expected to recognize and handle errors to support reliable and consistent execution of workflows. Since business processes contain activities that access shared and persistent data resources, they have to be subject to transactional semantics [12,13], and must span multiple transaction models and protocols native to the underlying legacy systems upon which the workflows are dependent. As [14] pointed out, the introduction of some kind of transactions in WfMSs is unavoidable to guarantee reliable and consistent workflow executions. Traditional transaction processing models [15] are inadequate for processing specialized transactions including workflow applications, which have long-duration transactions, and a set of activities that must be properly coordinated to ensure correct behavior. To address this shortcoming, researchers have proposed various advanced transaction processing models [7,16–19] that are suitable for executing such specialized applications. These advanced transaction processing models coordinate their activities using different kinds of dependencies.

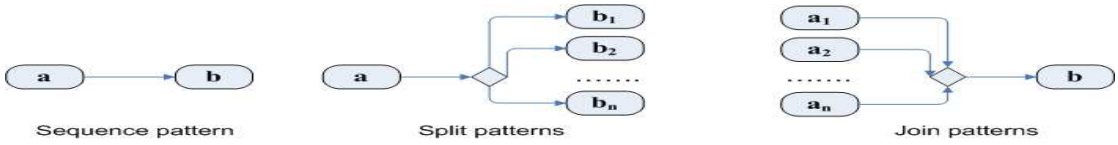
Transactional workflows propose to combine workflow flexibility and transactional reliability in order to deal with these issues. They have been introduced in [20] to clearly recognize the relevance of transactions

in the context of workflows. In the following, we present a synthetic description of a common and extensible transactional workflow model which was specified in our previous works [21,22]. Basically, within transactional workflow models, we distinguish between the control flow (continuous arrows in the motivating example in Figure 1) and the transactional behavior (dotted arrows in the motivating example in Figure 1).

2.1 Control flow

The control flow (or skeleton) of a workflow specifies the partial ordering of component activity activations. Within a workflow instance where all activities are executed without failure or cancellation, the control flow defines activity dependencies. In order to enhance reusability and common comprehension, we use the workflow patterns [23] as an abstract description of a reoccurring class of dependencies to describe the control flow as a pattern composition.

Figure 3 Patterns categories



We emphasize on the following seven patterns *sequence*, *AND-split*, *OR-split*, *XOR-split*, *AND-join*, *OR-join*, *XOR-join* and *m-out-of-n*[3] to explain and illustrate our approach. We divided the workflow patterns in three categories (Figure 3) : *sequence*, *split* and *join* patterns. For instance, *AND-join*$(a_1, ..a_n, b)$, which belongs to the *join* category, describes the convergence of n (two or more) branches $\{a_1, ..a_n\}$ into a single subsequent branch b such that the thread of control is passed to the subsequent branch when all input branches have been enabled.

2.2 Transactional Behaviour

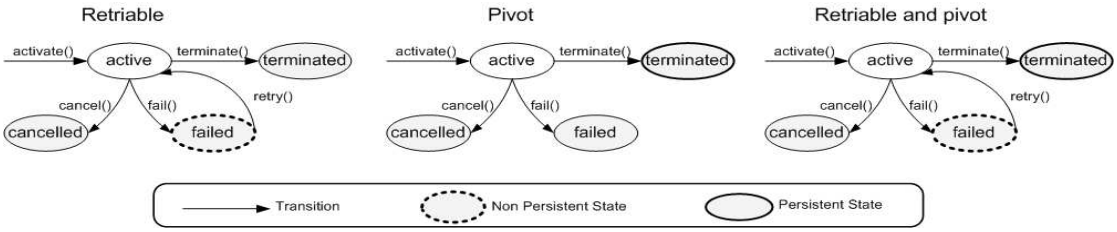
The transactional behaviour is observed in case of activity failures and defines recovery mechanisms supporting the automation of failure handling during runtime. Recovery mechanisms are used to ensure workflow fault-tolerance which is defined as the property allowing a business process to respond gracefully to expected but unusual situations and failures (also called exceptions). Basically, the transactional behaviour is described through the activity's transactional properties and the transactional flow depicting respectively the intra and the inter activity *transactional dependencies* after activity failures. The *transactional dependencies* play an important role in coordinating and executing a workflow instance. These dependencies define the relationships that may exist between the events reporting activity execution state (cancelled, failed, and terminated) of one or two activities specifying respectively the intra-activity state dependencies (i.e activity transactional properties) or the inter-activity state dependencies (i.e transactional flow).

2.2.1 Activity transactional properties

Every activity can be associated to a life cycle statechart that models the possible states through which the executions of this activity can go, and the possible transitions between these states. These transitions describe the intra-activity state dependencies on which depend the activity transactional properties.

[3] The used *m-out-of-n* pattern inherits from the canceling discriminator, i.e. triggering the OR gateway in this pattern, cancels the execution of all of the other incoming parallel branches

Figure 4 Activity transactional properties



The main transactional properties that we are considering are *retriable* and *pivot* [24] (Figure 4). An activity a is said to be retriable (a^r) *iff* it is sure to complete successfully. Even if it fails, a^r is reactivated until successful execution. For instance, in our motivating example (see Figure 1) CC is retriable. a is said to be pivot (a^p) *iff* once the activity successfully completes, its effects remain and cannot be semantically undone or reactivated. A pivot activity cannot be compensated. For instance, in our motivating example (see Figure 1) CIC, CCA, and PL are not pivot because they can be compensated by DBC.

2.2.2 Transactional flow

The transactional flow specifies the inter-activity state dependencies after activity failure. We distinguish two different transactional dependencies in the transactional flow: alternative (transactional)[4] dependencies, and cancellation dependencies. After the failure of an activity, a recovery process can be initialized by an alternative dependency that activates another activity, which is located through a related coherent point, to resume instance execution. An activity failure can cause a cancellation (non-regular or abnormal end) of one or more active activities which is described through a cancellation dependency. This dependency can be enacted within the recovery mechanism in order to regain a consistent state. Indeed, the cancellation of running activities, after the activity failure, can avoid their undesired terminations.

The transactional behaviour depends on the control flow. Consistency relations between the control flow (i.e workflow patterns) and the transactional behaviour (i.e transactional properties and transactional dependencies) will be described later in this article (see section 6.1)

2.2.3 Recovery mechanisms

This section outlines the transactional recovery rules that govern the recovery mechanisms after activity failures. During the execution of a transactional workflow, it has to be decided, after an activity execution failure (i.e observation of failed state), whether an inconsistent state was reached. According to this decision either a recovery procedure has to be started or the process execution continues according to the control flow (if the activity failure does not affect the execution of the workflow instance). Recovery mechanisms which are specified by transactional dependencies allow failed workflow instances to rollback to a coherent state if an inconsistent state was reached after activity failures. The goal is to recover the execution from a semantically acceptable state. Thus, the incoherent failure state can be corrected and the execution can be restarted from a coherent point thanks to the recovery mechanism. A coherent point is an execution step of the workflow (equivalent to a save point in database transactions) which represents an acceptable intermediate execution state. It is also a decision point where certain actions can be taken to either solve the problem that caused the failure or to choose an alternative execution path to avoid this problem [25]. Designers define according to their business needs for each failed activity the localisation of the coherent point. As such, exception handling is part of the

[4] In the following, we omit the term “transactional”

business logic of a workflow and may dominate its normal behaviour [26]. In summary, after an activity "a" fails, we have the following possibilities :

- The activity failure does not affect the execution of the workflow instance. The workflow instance can continue without any specific recovery mechanism.
- An inconsistent state is reached after the activity failure. The workflow instance execution cannot continue without a recovery mechanism. The activity is critical and should be recoverable. We distinguish between three different recovery mechanisms:
  - The activity is *retriable*: It is reactivated automatically until success. Such a mechanism is generally used if the failed activity is idempotent or neutral (noted a^n). An idempotent activity can be executed one or more times without interfering with the other activity executions. For instance, CC is retriable in our motivating example (see Figure 1), because a failure of CC has no side-effect on the other activities.
  - Forward recovery: Another activity, which is situated after the failed activity or in an alternative path, is enacted to correctly terminate the workflow execution. Thus, the workflow will enforce the regular workflow execution, probably along another execution path. This recovery mechanism is applied in case where the failed activity can be substituted by another activity without side-effects. For instance, a forward recovery was specified for CH to CC in our motivating example (see Figure 1), because the payment by credit card can completely substitute the payment by check.
  - Backward recovery: The instance execution is resumed from a consistent state situated before the failed activity. This happens when the successful execution of the failed activity is necessary for the instance execution. It may also be necessary to start a compensation activity which removes inconsistent side effects and semantically "undoes" the effect of the corresponding failed activity [27]. For instance a backward recovery was specified for CA in our motivating example (see Figure 1). The instance execution is resumed from the coherent point situated after CRS in order to restart the car rental discovery process (CIC, CCA, and PL), and DBC is executed as a compensation activity. The designers assumed that DBC never fail. Consequently, they did not provide a recovery mechanism for this activity.

The described transactional behaviour of this workflow model is common to existing transactional workflows model. It is specified by a set of activities, the dependencies between these activities, and the associated recovery mechanisms. The basic ideas of: (i) distinguishing between inter and intra activity transactional dependencies, (ii) attaching compensation activities, and cancellation dependencies to the activities of the workflow, (iii) declaring some of the activities to be critical, and (iv) defining coherent points in the process up to which a rollback occurs in case of failure, are common to many of the transactional models proposed earlier, (e.g. SARN [7], WAMO [28], WIDE [29], COO [30] and CREW [31]). These models typically differ in how much flexibility the business process designer has in specifying the backward and forward execution process. In this article, we propose a generic "basic" transactional workflow model. Adding new policies may provide support and flexibility, but it also makes the business process model more complex. Complexity severely compromises the usability and adoption. Therefore, our proposed transactional workflow model tries to remain in a striking balance between expressive power and simplicity. As a consequence, the goal that guided this model is that "right-sizing" the model, while providing room for it to evolve as the need arises. To achieve this goal, we have determined a minimal set of recovery policies that are useful and needed in practice to adequately model and handle most of the exceptions. We focus on recovery policies that are commonly used in practice. The list of recovery policies is, however, not exhaustive. Indeed, new (customized) recovery policies can be added. Thus, our model can be easily adapted or extended to any specific workflow exception handling approach. It should be noted that our focus in this article is not on specifying a new transactional workflow model, but on a comprehensive description of activity-based recovery policies that is used in the following as a support model for our workflow mining and re-engineering techniques.

3 Workflow Log

Information systems might not be concerned with the details of the internal processing of their process' activity executions. However, most process-aware management systems, such as ERP, CRM, SCM, log the external

events that capture the activities life cycle (such as activated, suspended, aborted, failed, and terminated). It is possible to record events such as (i) each event is referred to one activity (i.e, a well defined step in the workflow), (ii) each event refers to one case (i.e, a workflow instance), and (iii) events are completely ordered [32]. Thus, we expect that events are detectable and collectable, i.e. WfMS are required to capture and keep a workflow log. Any information system such as ERP, CRM, or WfMS using transactional systems offers this information in a certain form [32]. Data warehouses storing these workflow logs were proposed in the literature [33, 34]. These data warehouses simplify and accelerate workflow mining techniques' requests.

3.1 Workflow Log structure

Workflow logs can contain hundreds, even thousands of instances logs. Each instance log captures atomic events representing the changing of its activities state. Events in the same instance are ordered. The order between the events within the same instance is important because it is a matter of activity interaction causality. There is no bond of causality, which can influence our workflow mining approach, between the different instances. Thus, the different instances are not ordered while their events are. Basically, we can represent workflow logs as a set of distinct sequences (see Definition 1), such as each sequence represents an instance log. As shown in the UML class diagram in Figure 5, a WorkflowLog is composed of a set of EventStreams. Each EventStream traces the execution of one case (instance). It consists of a set of events (Event) that captures the activities life cycle performed in a particular workflow instance. An Event is described by the activity identifier and the activity execution result state (canceled, failed and terminated).

Definition 1 (WorkflowLog)

- A WorkflowLog is a set of EventStreams. WorkflowLog=(workflowID, {EventStream$_i$, $0 \leq$ i $\leq$ number of workflow instances}) where EventStream$_i$ is the event stream of the i^{th} workflow instance.
- An EventStream represents the history of a worflow instance event as a tuple stream (sequenceLog, SOccurrence) where:
  - SOccurrence : int is the instance number.
  - sequenceLog : Event* is an ordered event set belonging to a workflow instance
- Event is defined as a tuple Event= (activityId, state), where state = (cancelled=c) $\vee$ (failed= f) $\vee$ (terminated= t)

Log collecting facilities can offer different levels of granularity and collects only parts of the set of event states. The nomenclatures of these states are generally different from one system to another. As in our case we aim to mine only the transactional workflow behaviour described through termination events (i.e. "final" life cycle activity events). As such, it is more practical to simply consider the following atomic events (cancelled, failed, and terminated). These states are included in the standard activity states specified by the W*f*MC[5] [35]. Thus, we can simply choose to filter out the other states.

Although the combination of activated and terminated states can be very useful to implicitly detect the concurrent behaviour from logs, our log structure reports only "final" life cycle activity events (cancelled, failed, and terminated). This approach omits execution times and reports only termination events greatly simplifying the constraints of log collecting. This "minimalist" feature enables us to exploit "poor" logs which contain only information concerning the "final" life cycle activity events without collecting, for example, intermediate states (such as activated, suspended)) or execution occurrence times.

It is recommended to make sure that WorkflowLogs and EventStreams are described by a single identifier. Each activity ID should also be unique in the related workflow. If there are two activities or more in a workflow having the same name, we assume that they refer to the same activity. Although activities usually have a unique name, they can be present several times in the same instance: what we can observe for example in a loop.

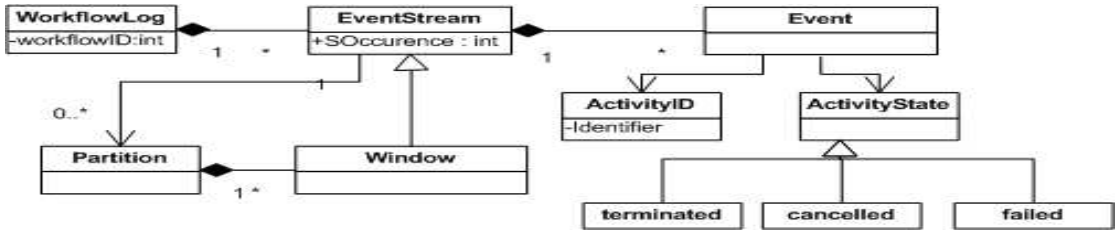
[5] Workflow Management Coalition

In the following we present the EventStream extracted from the workflow example of Figure 1 representing the 5^{th} and the 7^{th} workflow instance. The 5^{th} EventStream reports a forward recovery specified by an alternative dependency from CH to CC and a cancellation dependency from CH to SH. After the failure of the payment by cheque, the payment by credit card was executed as an alternative and the combined payment by cash was cancelled. The 7^{th} EventStream describes the retriable transactional property of the CC activity in this instance. Indeed, the payment by credit card was executed three times until success (the first two times the payment failed). Mining algorithms to discover this transactional behaviour from logs are specified in section 5.3.1.

L = EventStream([**Event**(CRS, t), **Event**(CCA, t), **Event**(CIC, t), **Event**(PL, t), **Event**(CA, t), **Event**(CH, f), **Event**(SH, c) **Event**(CC, t), **Event**(SB, t)], 5)

L = EventStream([**Event**(CRS, t), **Event**(CIC, t), **Event**(CCA, t), **Event**(PL, t), **Event**(CA, t), **Event**(CC, f), **Event**(CC, f), **Event**(CC, t), **Event**(SB, t)], 7)

Figure 5 Structure of a workflow event Logs



3.2 Log completeness conditions

To enable a correct mining process, the workflow logs must be "complete" by respecting the *log completeness conditions* [36]. These conditions are as follows:

- **Condition 1:** if an activity precedes another in the control flow then there exists at least one instance log that reports two respective related events with the same order. Particularly, if the execution of one activity depends **directly** on the termination of another activity then the event related to the first activity must *directly* (immediately, without intercalated events between them) follow at least once the event related to the second activity in an instance log. For instance, in our motivating example (see Figure 1) the execution of CCA directly depends on the termination of CRS. Thereafter, in order to be complete the related workflow log should contain an EventStream where CCA directly follows CRS (as shown in the 5^{th} EventStream above).
- **Condition 2:** To discover the parallel behaviour of two concurrent activities where the activities' begin and end execution times is unknown, we require that the events of the two parallel activities should appear at least in two EventStreams without order of precedence. Basically, two parallel activities must *directly* follow each other in two instances in different order to indicate that each activity can finish its execution before the other. For instance, in our motivating example (see Figure 1) CIC and CCA are two parallel activities. Thereafter, the related workflow log should contain two EventStreams where CIC follows CCA in the first one, and CCA follows CIC in the second one (as shown in the 5^{th} and 7^{th} EventStreams).

Table 2 represents the execution of six instances of our running example (see Figure 1) without any activity failures. This table contains the sufficient information which we assume to be present to mine the associated

control flow. Indeed, our motivating example contains conditional behavior between three activities (CH, SH, and CC), which implies three EventStreams according to the first condition. Additionnaly, the two concurrent flows, containing respectively CIC, CCA, and PL imply two additional EventStreams by applying the second point. The parallel behavior that can be observed between CC and SH if the user decides to pay using credit card and cash implies one additional EventStreams. The total of sufficient EventStreams equals 3+2+1 = 6, shown in Table 2. In this table, the EventStreams 1 and 4 describe the case where a user pays by cash. Although these different EventStreams (1 and 4) describe the same scenario, the concurrent activities execution differs (i.e CIC, CCA, and PL). These different EventStreams (in the same way for the EventStreams 5 and 6) allow one to describe the various possible choices of the processing as well as the various possible combinations of concurrent activities' execution in these choices.

Table 1 Six EventStreams of our motivating example

Instance ID	EventStream
Instance1	(CRS,t) (CCA,t) (PL,t) (CIC,t) (CA,t) (SH,t) (SB,t)
Instance2	(CRS,t) (CCA,t) (PL,t) (CIC,t) (CA,t) (CH,t) (SB,t)
Instance3	(CRS,t) (CCA,t) (PL,t) (CIC,t) (CA,t) (CC,t) (SB,t)
Instance4	(CRS,t) (CCA,t) (CIC,t) (PL,t) (CA,t) (SH,t) (SB,t)
Instance5	(CRS,t) (CIC,t) (CCA,t) (PL,t) (CA,t) (CH,t) (SH,t) (SB,t)
Instance6	(CRS,t) (CIC,t) (CCA,t) (PL,t) (CA,t) (SH,t) (CH,t) (SB,t)

In this section, we have focused on defining a meta model for the workflow log structure by presenting an independent log format which is compatible with the majority of WfMSs collected logs. This format is used as the single and only input for our workflow mining approach that we detail in the following sections. As a matter of principle, any system which collects events related to the execution of its activities can use this independent format to store and exchange its logs. The intention of this format is to reduce log processing efforts and to provide support for other workflow discovery tools. Besides, it is simple to translate into the MXML format suggested and used by many of the current workflow mining tool [37]. We note that more details about *log completeness conditions* can be found in our previous work [38] where we defined and proved the sufficient and minimal number of workflow instances to achieve these *log completeness conditions*.

4 Workflow log statistical analysis

Attie et al. [39] discuss how workflows can be represented as a set of dependencies. The activities in a workflow are described in terms of significant *event dependencies*. The aim of this section is to detail our log statistical analysis for discovering *event dependencies* among workflow activities and build an intermediary model that represents these dependencies: the statistical dependency tables (SDT).

4.1 Statistical dependency table

We propose to build Statistical Dependency Tables (SDT) reporting log elementary event dependencies by statistically analysing the WorkflowLog. An elementary event dependency is a relation linking an Event e_i to another Event e_j which expresses that there is an EventStream where the event e_i precedes directly the occurrence of the event e_j. These tables, which are based on frequency tables [40], report for each event a the following information: (i) the overall frequency of this event (denoted $\#a$) and (ii) the causal dependencies to previous events b_i (denoted $SDT(a, b_i)$). The size of SDT is N*N, where N is the number of workflow events. The (*m,n*) table entry is the frequency of the n^{th} event immediately preceding the m^{th} event (see algorithm 1).

Table 2 represents fragments of the SDT of our motivating workflow example concerning only non-failed workflow instances (i.e. instances containing failed activities are discarded from this table). For instance, in

Algorithm 1 Computing initial SDT

```
1: procedure COMPUTINGSDT(Workflow Log)
2: input: WorkflowLog: WorkflowLog
3: output: #: event occurrence table ; SDT: Statistical dependency table;
4: Variable: stream_size: int; SDT_size: int;
5: #: int[]; depFreq: int[][];                                   ▷ initialized to 0;
6:
7:     for every stream: EventStream in WorkflowLog do
8:         stream_size = stream.size();                          ▷ size returns the number of events in a stream
9:         for int i=1; i < stream_size; i++; do
10:            #[stream.get(i)]++;                               ▷ get returns the event whose index is i
11:            depFreq[stream.get(i)][stream.get(i-1)]++;
12:        end for
13:    end for
14:    SDT_size = Size-tab(#); /*return the size of #*/
15:    for int j=0; j < SDT_size; j++; do
16:        for int k=0; k < SDT_size; k++; do
17:            SDT[j, k] = depFreq[j][k]/ #[j];
18:        end for
19:    end for
20: end procedure
```

this table SDT$((SB,t),(SH,t))$=0.22 expresses that if the event (SB, t) occurs then we have a 22% chance that the event (SH, t) occurs directly before the event (SB, t) in the workflow logs.

Table 2 Fractions of initial SDT

SCT((x,y)	CRS,t	CIC,t	CCA,t	PL,t	CA,t	CC,t	CH,t	SH,t	SB,t
CRS,t	0	0	0	0	0	0	0	0	0
CIC,t	0.46	0	**0.33**	**0.21**	0	0	0	0	0
CCA,t	0.54	**0.46**	0	0	0	0	0	0	0
PL,t	0	**0.33**	0.67	0	0	0	0	0	0
CA,t	0	0.21	0	0.79	0	0	0	0	0
CC,t	0	0	0	0	1	0	0	0	0
CH,t	0	0	0	0	1	0	0	0	0
SH,t	0	0	0	0	1	0	0	0	0
SB,t	0	0	0	0	0	0.43	0.35	0.22	0

$$\#CRS,t = \#CIC,t = \#CCA,t = \#PL,t = \#CA,t = \#SB,t = 100$$
$$\#CC,t = 43\#CH,t = 35\#SH,t = 22$$

We demonstrated a correlation between the workflow dependencies and the log statistical dependencies expressed in SDT (Theorem 1). This theorem expresses that a relation of equivalence between activity dependencies and positive entries in SDT: an activity "a" immediately precedes [6] another activity "b" if and only if SDT reports a zero value between the successive events and a positive value in the opposite entry.

Proof (Proof of Theorem 1:)
Proving first right implication "⇒"

Let L be the WorkflowLog capturing the execution of a and b. By applying **Condition 1** of the *log completeness conditions* specified in section 3 we can deduce that:

[6] i.e. the only process model elements between activities "a" and "b" are gateways, meaning that there is no path in the process model between "a" and "b" that contains another activity

Theorem 1 (Correlation between SDT and activity dependencies)
*Let wft be a workflow whose control flow is composed using the set of 7 described patterns (*sequence*,* AND-split*,* OR-split*,* XOR-split*,* AND-join*,* OR-join*,* XOR-join *and* m-out-of-n*) and does not contain short loop spanning only two activities. $\forall a, b \in wft$*

$$a \textit{ immediately precedes } b \Leftrightarrow SDT((b,t),(a,t)) > 0 \wedge SDT((a,t),(b,t)) = 0$$

$$\exists \sigma_1 = t_1 t_2 t_3 ....t_{n-1} \in L \wedge \exists 0 < i < n, |t_i = (a,t), t_{i+1} = (b,t)$$

Using the SDT building definition and the preceding instance log, we can deduce that:

$$a \prec b \,^{7} \Rightarrow SDT((b,t),(a,t))$$

Let us *suppose* now (proof by contradiction) that $SDT((a,t),(b,t)) > 0$. This implies:

$$\exists \sigma_1 = t_1 t_2 t_3 ....t_{n-1} \in L \wedge \exists 0 < i < n, |t_i = (b,t), t_{i+1} = (a,t)$$

and $SDT((b,t),(a,t)) > 0$ implies:

$$\exists \sigma_1 = t_1 t_2 t_3 ....t_{n-1} \in L \wedge \exists 0 < i < n, |t_i = (a,t), t_{i+1} = (b,t)$$

However, this case only applies if we have a short loop (By applying **Condition 1** of the *log completeness conditions*), or a and b are concurrent activities (By applying **Condition 2** of the *log completeness conditions*). Thus we have:

$$a \prec b \Rightarrow SDT((b,t),(a,t)) > 0 \wedge SDT((a,t),(b,t)) = 0$$

Proving second left implication "$\Leftarrow$" (proof by contradiction)

We have $SDT((b,t),(a,t)) > 0 \wedge SDT((a,t),(b,t)) = 0$, thus based on the SDT building definition we can deduce:

$$\textbf{(1)}\ \exists \sigma_1 = t_1 t_2 t_3 ....t_{n-1} \in L \wedge \exists 0 < i < n, |t_i = (a,t), t_{i+1} = (b,t)$$
$$\wedge \nexists \sigma_1 = t_1 t_2 t_3 ....t_{n-1} \in L \wedge \exists 0 < i < n, |t_i = (b,t), t_{i+1} = (a,t)$$

Since a and b belong to the set of the 7 described patterns, two sub cases happen if *we suppose* that they are causally independent:

1. The two activities a and b belong to two different separate patterns. However, this is impossible because our instance log **(1)** shows that the two activities happen one after the other.
2. The two activities a and b are concurrent. This is impossible based on instance log **(1)** and the **Condition 2** of the *log completeness conditions*.

Thus a precedes b. Indeed, the case b precedes a is trivially meaningless by applying "$\Rightarrow$". In conclusion, we have:

$$a \prec b \Leftarrow SDT((b,t),(a,t)) > 0 \wedge SDT((a,t),(b,t)) = 0$$

As shown this "initial" SDT has problems to "correctly" and "completely" express event dependencies related to the *concurrent* and the *conditional* behaviour. Indeed, these entries are not able to identify the *conditional* behaviour and to report the *concurrent* behaviour pertinently. In the following, we detail these problems and we propose solutions to correct and complete these statistics.

[7] a immediately precedes b

4.2 Discarding erroneous dependencies

If we assume that each EventStream from a WorkflowLog stems from a sequential (i.e no concurrent behaviour) workflow, a zero entry in SDT represents a causal independence and a non-zero entry a causal dependency (*i.e.* sequential or conditional dependency). However, in case of concurrent behaviour EventStreams may contain interleaved event sequences from concurrent threads. As a consequence, some entries in a SDT can indicate non-zero entries that do not correspond to dependencies. For example, the 5^{th} EventStream given in section 3 erroneously "suggests" causal dependencies between (CCA and CIC), and (CIC and PL). Indeed, CIC comes just after CCA and CIC comes immediately before PL in this EventStream. These erroneous entries are reported by SDT((CIC, t), (CCA, t)) and SDT((PL, t), (CIC, t)) in the initial SDT which are different to zero. These entries are erroneous because there are no causal dependencies between these events as the initial SDT suggests. Bold values in Table 2 report this behaviour for other similar cases.

Formally, based on **condition 2** of the *log completeness conditions*, we can easily deduce that two activities A and B are in concurrence *iff* SDT((A, t), (B, t)) and SDT((B, t), (A, t)) entries in SDT are non-zero entries in SDT. Based on this, we propose an algorithm to discover activity parallelism and then mark the erroneous entries in SDT. Through this marking, we can eliminate the confusion caused by the concurrent behaviour producing these erroneous non-zero entries. The algorithm 2 scans the initial SDT and marks concurrent activities dependencies by changing their values to (-1). For instance, we can deduce from Table 2 that CIC and CCA activities are in concurrence (i.e $SDT((CCA, t), (CIC, t)) \neq 0 \wedge SDT((CIC, t), (CCA, t)) \neq 0$). After applying our algorithm, the entries SDT((CCA, t), (CIC, t)) and SDT((CIC, t), (CCA, t)) are changed to -1. The algorithm's output is an intermediary table that we call marked SDT (MSDT).

Algorithm 2 Marking concurrent activities in SDT

```
1: procedure MARKINGSDT(SDT, MSDT)
2: input: SDT Statistical dependency table
3: output: MSDT Marked Statistical dependency table
4: Variable: MSDT_size: int;
5:
6:     MSDT = SDT;
7:     MSDT_size = Size-tab(MSDT);                    ▷ calculates MSDT size
8:     for int i=0; i< MSDT_size; i++; do
9:         for int j=0; j<i; j++; do
10:            if SDT[i][j] > 0 ∧ SDT[j][i]>0 then
11:                MSDT[i][j] = -1;
12:                MSDT[j][i] = -1;
13:            end if
14:        end for
15:    end for
16: end procedure
```

4.3 Discovering indirect dependencies

An activity might not depend on its immediate predecessor in the EventStream, but it might depend on another "indirectly" preceding activities. As an example of this behaviour, CIC is logged between CCA and PL in the 5^{th} EventStream given in section 3. As a consequence, CCA does not occur always immediately before PL in the workflow log. Thus we will have $SDT((PL, t), (CCA, t))) < 1$ which is under-evaluated. In fact, the right value is 1 because the execution of PL depends exclusively on CCA. Similarly, underlined values in Table 3 report this behaviour for other cases.

To discover these indirect dependencies, we introduce the notion of a *concurrent window* (Definition 2) that defines a set of contiguous Event intervals over an EventStream. A *concurrent window* is related to the activity (the reference activity) of its last event (ActID) covering its causal preceding activities. Using this window,

Definition 2 (Concurrent window)
Formally, a concurrent window defines an uplet **window**(wLog, ActID) as a continuous log slide (interval) over an EventStream S(sLog, SOcc) where wLog $\subset$ sLog and ActID is the last event in this log slide.
We define the function *width*(**window**) which returns the number of activities in the **window**.

we do not only consider the immediate previous event for a concurrent activity but also the previous events covered by the interval. For instance, in our motivating example (see Figure 1) PL and CIC are two parallel activities. Due to this concurrent behaviour, CCA does not occur always immediately before PL in workflow log because CIC can be logged between PL and CCA. Therefore, the concurrent window of PL, as shown in Figure 6, covers CCA in addition to CIC. Thus, thanks to PL's concurrent window CCA will be considered as a preceding activity of PL when SDT will be computed using the concurrent window correction.

Initially, the width of the *concurrent window* of an activity is equal to 2. Each time the activity is in concurrence with another 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 the *concurrent window*'s width. For example, CIC is only in concurrence with PL, so the width of its *concurrent window* is equal to $3 = 2 + 1$. Based on this, the algorithm 3 computes the *concurrent window* of each activity grouped in the *concurrent window* table. This algorithm scans the "marked" SDT calculated in the last section and updates the *concurrent window* table consequently.

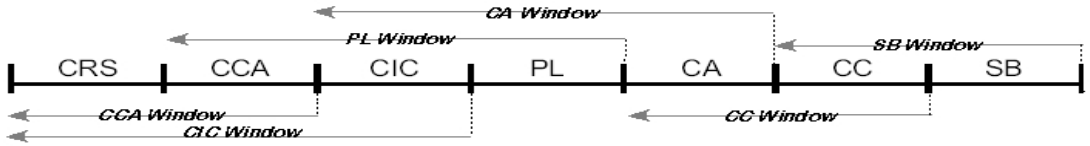
Algorithm 3 Calculating concurrent window size

```
1: procedure WindowWidth(MSDT, ACWT)
2: input: MSDT: Marked Statistical dependency table
3: output: ACWT: CW size table
4: Variable: MSDT_size: int;
5:
6:     MSDT_size = Size-tab(MSDT);                    ▷ calculates MSDT size
7:     for int i=0; i< MSDT_size; i++; do
8:         ACWT[i]=2;
9:     end for
10:    for int i=0; i< MSDT_size; i++; do
11:        for int j=0; j < MSDT_size; j++; do
12:            if MSDT[i][j] ==-1 then
13:                ACWT[i]++;
14:            end if
15:            for int k=0; k< MSDT_size; k++; do
16:                if MSDT[k][i] >0 then
17:                    ACWT[k]++;
18:                end if
19:            end for
20:        end for
21:    end for
22: end procedure
```

After that, we proceed through an EventStream partition that builds a set of partially overlapping windows over the EventStreams using the *concurrent window* table computed in algorithm 3. In an EventStream partition, each window shares the set of its elements with the preceding window except from the last event which contains the reference activity of the window (ActID in Definition 2). In Figure 6 we have applied a partition over the EventStream of the running example presented in the 5^{th} EventStream in section 3. For example, the size of the *concurrent window* of CA is equal to 3 because this activity has two concurrent activities PL and CIC that precede it. We note that for each activity in this EventStream its *concurrent window* enables it to cover only its causal preceding activities.

Finally, Algorithm 4 computes a new "corrected" SDT (final SDT). This algorithm takes as input the workflow log (Wlog). The function (partition(Wlog)) builds a set of partially overlapping windows over the

Figure 6 EventStream partition



EventStreams of Wlog using the *concurrent window* table computed in algorithm 3. Then, for each *concurrent window* (fwin), it computes for its reference (last) activity ($S_{reference}$) the frequencies of its preceding activities ($S_{preceding}$). Using MSDT table, only activities dealing with the concurrent behavior are updated in the initial SDT. Eventually, the final SDT is calculated by dividing each row entry by the frequency of the row's activity.

Algorithm 4 Calculating final SDT

procedure FINALSDT($Wlog, \#, MSDT$)
input: Wlog: WorkflowLog, #: event Frequencies Table; $MSDT$: Marked Statistical Dependency Table;
output: $CSDT$:Corrected Statistical Dependency Table
Variable: $S_{reference}$: int; $S_{preceding}$: int; fWin: window; depFreq: int[][]; freq: int;

    $MSDT_{size}$ = *Size-tab*($MSDT$); ▷ returns MSDT size
    for *all* win:window *in* partition(Wlog) **do**
        $S_{reference}$ = *last-activity*(win); ▷ returns the last activity's event
        fwin = *preceding-events*(win); ▷ returns "win" without the last event
        for *all* e:event *in* fwin.wLog **do**
            $S_{preceding}$= e.activityId;
            if MSDT[$S_{reference}$, $S_{preceding}$]>0 **then**
                depFreq[$S_{reference}$, $S_{preceding}$]++;
            end if
        end for
    end for
    for int t_{ref}=0; $t_{ref} < MSDT_{size}$; t_{ref}++; **do**
        for int t_{pr}=0; $t_{pr} < MSDT_{size}$; t_{pr}++ **do**
            $CSDT$[t_{ref}, t_{pr}]= depFreq[t_{ref}, t_{pr}]/#t_{ref};
        end for
    end for
end procedure

Now by applying these algorithms, we can compute the corrected SDT (Table 3) which will be used in the next section to discover the workflow patterns. Note that, our approach **dynamically** adjusts, through the width of *concurrent window*, the process of calculating the events' dependencies. Indeed, this width is tightly related to the concurrent behaviour: it increases in case of a concurrence and remains "neutral" (equal to 2) in case of a concurrent behaviour absence. Thus, our algorithm adapts its behaviour to the "concurrent" context. This strategy allows the improvement of the algorithm's complexity and runtime execution compared to an analog patterns discovery algorithm [41] which uses an invariable concurrent window width. Indeed, the use of an invariable width could apply a width superior to 2 for non-concurrent activities or a non optimal width and then involve unnecessary computations increasing the algorithm's complexity.

5 Transactional workflow mining

In the following, we describe a set of techniques and algorithms for transactional workflow mining based on log statistics analysis. We proceed in three steps by discovering: (i) the workflow patterns composing the control flow (section 5.1), the transactional flow (section 5.3.1) and the transactional properties (section 5.3.2) including the recovery mechanisms.

Table 3 Final SDT (SCfT) and activity Frequencies (#)

SCfT((x,y)	CRS,t	CIC,t	CCA,t	PL,t	CA,t	CC,t	CH,t	SH,t	SB,t
CRS,t	0	0	0	0	0	0	0	0	0
CIC,t	1	0	**-1**	**-1**	0	0	0	0	0
CCA,t	1	**-1**	0	0	0	0	0	0	0
PL,t	0	**-1**	1	0	0	0	0	0	0
CA,t	0	1	0	1	0	0	0	0	0
CC,t	0	0	0	0	1	0	0	0	0
CH,t	0	0	0	0	1	0	0	0	0
SH,t	0	0	0	0	1	0	0	0	0
SB,t	0	0	0	0	0	0.43	0.35	0.22	0

$$\#CRS, t = \#CIC, t = \#CCA, t = \#PL, t = \#CA, t = \#SB, t = 100$$
$$\#CC, t = 43 \#CH, t = 35 \#SH, t = 22$$

5.1 Mining workflow patterns

In this section we focus on discovering "elementary" routing workflow patterns: *Sequence*, *AND-split*, *OR-split*, *XOR-split*, *AND-join*, *OR-join*, and *M-out-of-N* patterns composing the control flow. As we have mentioned, these patterns describe the control flow interactions for activities executed without "exceptions" (*i.e.* they successfully reach their terminated state). Thus, there is no need to use the event dependencies relating to (failed or cancelled) states which concern only workflow transactional behaviour. As such we can filter the workflow logs and only consider EventStreams executed without failures or cancellations. Thus, the minimal condition to discover workflow patterns is a workflow log containing at least the terminated event state. Thus, we build a control flow SDT (noted SCfT) that captures only event dependencies with a terminated state in successful executions (Table 3). Therefore we do not differentiate between activity dependencies and event dependencies (i.e. we do not need to capture event states as we are only interested in terminated states).

The identification of workflow patterns is archived through a set of rules. Each pattern has its own features which statistically abstract its causal dependencies, and represent its unique identifier. These rules allow to discover the set of workflow patterns included in the mined workflow. Our control flow mining rules are characterized by a "local" workflow mining approach. Indeed, these rules proceed through a local log analysis that allows us to recover partial results of mining workflow patterns. To discover a particular workflow pattern we only need Events related to the activities of this pattern. Thus, even using only fractions of workflow logs, we can correctly discover the corresponding workflow patterns.

5.1.1 Discovering sequence pattern

In this category, we discover sequence patterns (Table 4). In this pattern the enactment of the activity b depends only on the completion of the activity a which implies only the enactment of b.

For instance, by applying the rules of this pattern over SCfT (*i.e.* Table 3), we discover a sequence pattern linking CCA and PL. Indeed, $(\#CCA = \#PL)$ and $(SCfT(PL, CCA) = 1)$ and $\forall A_{1 \leq i \leq n} \neq CCA; SCfT(PL, A_i) \leq 0$ and $\forall A_{1 \leq j \leq n} \neq PL; SCfT(A_j, CCA) \leq 0$.

Table 4 Mining rules of *sequence* workflow pattern

Pattern	rules
sequence	$(\#b = \#a) \wedge (SCfT(b,a) = 1) \wedge (\forall c \neq a; SCfT(b,c) \leq 0) \wedge (\forall d \neq b; SCfT(d,a) \leq 0)$

5.1.2 Discovering split *patterns*

Table 5 Mining rules of split workflow patterns

Pattern	rules
XOR-split	$(\Sigma_{i=1}^{n} (\#b_i)$=#a$) \wedge (\forall 1 \leq i \leq n; SCfT(b_i,a) = 1) \wedge (1 \leq i,j \leq n; SCfT(b_i,b_j) = 0)$
AND-split	$((\forall 1 \leq i \leq n; \#b_i$=#a$) \wedge (\forall 1 \leq i \leq n; SCfT(b_i,a) = 1) \wedge (\forall 1 \leq i \neq j \leq n\ SCfT(b_i,b_j) = -1)$
OR-split	$(\#a \leq \Sigma_{i=1}^{n} (\#b_i)) \wedge (\forall 1 \leq i \leq n; \#b_i \leq \#a)\ (\forall 1 \leq i \leq n; SCfT(b_i,a) = 1) \wedge (\exists 1 \leq i \neq j \leq n; SCfT(b_i,b_j) = -1)$

This category (see Figure 3) is characterised by "fork" operator where a single thread of control splits into multiple threads of control which can be, according to the used pattern, executed or not. To identify the three patterns of this category: XOR-split pattern, AND-split pattern and OR-split pattern, we have analysed dependencies between the activities a and b_i before and after the "fork" operator (see Table 5). The three patterns share the causality between activities a and b_i; that means $(\forall 1 \leq i \leq n; SCfT(b_i,a) = 1)$. In the xor-split pattern, the non concurrent behaviour between b_i is identified by the statistical property $(\forall 1 \leq i,j \leq n; SCfT(b_i,b_j) = 0)$. The difference between the or-split and the and-split patterns is the frequency relation between activities a and b_i . Effectively, in the or-split pattern only a part of these activities are executed after the "fork" point. However, all the b_i activities are executed in the and-split pattern. For instance, using Table 3 we mine that an *AND-split* pattern links CRS, CCA and CIC. In fact, the SCfT's entries of these activities indicate a concurrent behaviour between CCA and CIC ($SCfT(CCA,CIC)$ = $SCfT(CIC,CCA)$ = -1), and CCA and CIC executions depend on the termination of CRS ($SCfT(CCA,CRS) = SCfT(CIC,CRS) = 1$).

5.1.3 Discovering join *patterns*

This category (see Figure 3) has a "join" operator where multiple threads of control merge in a single thread of control. The number of necessary branches for the activation of the activity b after the "join" operator depends on the used pattern. To identify the three patterns of this category: xor-join pattern, and-join pattern and M-out-of-N-Join pattern, we have analysed dependencies between the activities a_i and b before and after the "join" operator (see Table 6). The three patterns differ in the number of necessary branches to activate b. The and-join pattern requires the execution of all the a_i activities; which can be identified by $(\forall 1 \leq i \leq n; SCfT(b,a_i) = 1)$. The M-out-of-N-Join pattern supports a "partial" parallelism between a_i activities; that means $(\exists 1 \leq i,j \leq n; SCfT(a_i,a_j) = -1)$. Finally, the non concurrent behaviour between a_i in the xor-join pattern can be catched by $((\forall 1 \leq i,j \leq n; SCfT(a_i,a_j) = 0))$. For instance, using Table 3 we mine an xor-join pattern linking CC, CH, SH and SB. In fact, the FSDT's entries of these activities indicate a non concurrent behaviour between CC, CH, and SH ($SCfT(CH,SH)$=$SCfT(SH,CH)$=$SCfT(SH,CC) \neq$ -1) and the execution of SB depends on the termination of CC, CH, and SH: ($SCfT(SB,CC)$+$SCfT(SB,CH)$+$SCfT(SB,SH)$= 1). We notice here that the payment by cash and check is never combined. Indeed, those two activities are never executed together ($SCfT(CH,SH) \neq$ -1). Keeping this payment facilities can be costly. This new discovered behaviour should be used in the re-engineering phase by advising the designers to modify the concurrent behaviour linking the activities CH and SH (see green flows in Figure 8).

Table 6 Mining rules of join workflow patterns

Pattern	rules
XOR-join	$(\Sigma_{i=1}^{n}(\#a_i)$=#b$) \wedge (\Sigma_{i=1}^{n}$ SCfT(b,a_i)=1$) \wedge (\forall 1 \leq i \neq j \leq n; SCfT(a_i, a_j) = 0)$
AND-join	$(\forall 1 \leq i \leq n; \#a_i$=#b$) \wedge (\forall 1 \leq i \leq n; SCfT(b, a_i) = 1) \wedge (\forall 1 \leq i \neq j \leq n\ SCfT(a_i, a_j) = -1)$
M-out-of-N	$(m * \#b \leq \Sigma_{i=1}^{n}(\#a_i)) \wedge (\forall 1 \leq i \leq n; \#a_i \leq \#b)\ (m \leq \Sigma_{i=1}^{n} SCfT(b, a_i) \leq n) \wedge (\exists 1 \leq i \neq j \leq n; SCfT(a_i, a_j) = -1)$

5.2 Coherent composition of the discovered patterns

The construction of the discovered workflow is made by linking the discovered patterns one by one. Indeed, in our approach we define the control flow as a union of patterns. We use rewriting rules (illustrated in Table 7) to bind the discovered patterns (terminals). We consider a workflow as a word that has patterns as terminals (laterals). These terminals can be associative and commutative in the word constituting the discovered workflow when applying the rewriting rules.

Table 7 Rewriting rules defining a coherent pattern composition grammar

RR1: *sequence*(a,b),$\{p_i\} \longrightarrow \mathcal{A}(a,b)$,$\{p_i\}$
RR2: *AND-split*$(a,b_1,b_2,...,b_n)$,$\{p_i\} \longrightarrow \mathcal{B}(a,b_1,b_2,...,b_n)$,$\{p_i\}$
RR3: *OR-split*$(a,b_1,b_2,...,b_n)$,$\{p_i\} \longrightarrow \mathcal{C}(a,b_1,b_2,...,b_n)$,$\{p_i\}$
RR4: *XOR-split*$(a,b_1,b_2,...,b_n)$,$\{p_i\} \longrightarrow \mathcal{D}(a,b_1,b_2,...,b_n)$,$\{p_i\}$
RR5: *AND-join*$(a_1,a_2,...,a_n,b)$,$\{p_i\} \longrightarrow \mathcal{E}(a_1,a_2,...,a_n,b)$,$\{p_i\}$
RR6: *M-out-of-N*$(a_1,a_2,...,a_n,b)$,$\{p_i\} \longrightarrow \mathcal{F}\{a_1,a_2,...,a_n,b)$,$\{p_i\}$
RR7: *XOR-join*$(a_1,a_2,...,a_n,b)$,$\{p_i\} \longrightarrow \mathcal{G}(a_1,a_2,...,a_n,b)$,$\{p_i\}$
RR8: $\mathcal{A}(a,b), \mathcal{A}(b,c)$,$\{p_i\} \longrightarrow \mathcal{A}(a,c)$,$\{p_i\}$
RR9: $\mathcal{A}(x,a), \mathcal{F}sn(a,b_1,b_2,...,b_n)$, $\{p_i\} \longrightarrow \mathcal{F}sn(x,b_1,b_2,...,b_n)$,$\{p_i\}$
RR10: $\mathcal{F}sn(a,b_1,...,b_n)$, $\mathcal{A}(b_i,x)$, $\{p_i\} \longrightarrow \mathcal{F}sn(a_0,b_1,...,b_{i-1},x,b_{i+1},...,b_n)$,$\{p_i\}$
RR11: $\mathcal{A}(x,a_i)$, $\mathcal{J}nt(a_1,...,a_n,b)$, $\{p_i\} \longrightarrow \mathcal{J}nt(a_1,...,a_{i-1},x,a_{i+1},...,a_n,b)$,$\{p_i\}$
RR12: $\mathcal{J}nt(a_1,a_2,...,a_n,b)$, $\mathcal{A}(b,x)$, $\{p_i\} \longrightarrow \mathcal{J}nt(a_1,a_2,...,a_n,x)\{p_i\}$
RR13: $\mathcal{B}(a,b_1,b_2,...,b_n)$, $\mathcal{E}(b_1,b_2,...,b_n,c)$, $\{p_i\} \longrightarrow \mathcal{A}(a,c)$, $\{p_i\}$
RR14: $\mathcal{C}(a,b_1,b_2,...,b_n)$, $\mathcal{F}(b_1,b_2,...,b_n,c)$, $\{p_i\} \longrightarrow \mathcal{A}(a,c)$, $\{p_i\}$
RR15: $\mathcal{D}(a,b_1,b_2,...,b_n)$, $\mathcal{G}(b_1,b_2,...,b_n,c)$, $\{p_i\} \longrightarrow \mathcal{A}(a,c)$, $\{p_i\}$
RR16: $\mathcal{A}(a,b), \varepsilon \longrightarrow \mathcal{W}orkflow$

$\{p_i\}$ remaining terminal set
$\mathcal{F}sn = \mathcal{B} \vee \mathcal{C} \vee \mathcal{D}$
$\mathcal{J}nt = \mathcal{E} \vee \mathcal{F} \vee \mathcal{G}$

Discovering a pattern-oriented model ensures a sound and well-formed mined workflow model. By using such as model we are sure that the discovered workflow model does not contain any deadlocks or other flow anomalies. Indeed, this set of rules allows to discover a coherent and well-formed pattern-oriented workflow model. Consequently, by using these rewriting rules we are sure that the discovered patterns do not contain any incoherent flow. In fact, in order to have meaningful unions (disjoined patterns) or incoherent flows (deadlocks, liveness, etc.), this grammar does not allow, for instance, the combination of *XOR-split* and *AND-join* patterns which implies a deadlock or a combination of *AND-split* and *XOR-join* which can induce a meaningless union where some executed flows are unneeded. The grammar of the rewriting rules defines a language of coherent unions that reduces the discovered patterns to the final word $\mathcal{W}orkflow$. Concretely, rules RR1 to RR7 rewrite the discovered in federated expressions that are reduced thereafter in rules RR8 to RR16 in the final word $\mathcal{W}orkflow$. Thus, a discovered control flow is coherent ***iff*** the union of the corresponding discovered patterns is a word generated by this grammar. Concretely this grammar, which was specified for the set of the seven studied patterns, postulates that:

– A control flow should start with one of these patterns: *sequence*, *AND-split*, *OR-split* or *XOR-split* (rewriting rules RR8, RR13, RR14, RR15, RR16).

- All patterns can be followed or preceded by the *sequence* pattern (rewriting rules RR8, RR9, RR10, RR11, RR12).
- An *AND-split* pattern should be followed by one of these patterns: *AND-join* or *sequence* (rewriting rules RR10, RR13).
- An *OR-split* pattern should be followed by one of these patterns *M-out-of-N* or *sequence* (rewriting rules RR10, RR14).
- An *XOR-split* pattern should be followed by an *XOR-join* pattern or a *sequence* pattern (rewriting rules RR10, RR15).

By applying the rules (Tables 4, 5 and 6) over SCfT table (Table 3) we discovered the workflow illustrated in Figure 8. We built the control flow as a pattern composition over this pattern word:

and-split(CRS,CIC,CCA), *sequence*(CCA,PL), *and-join*(CIC,PL,CA), *xor-split*(CA,CH,SH,CC), *xor-join*(CH,SH,CC,SB).

Concretely by applying the rewriting rules (Table 7) to this word, we can combine these discovered patterns, by binding them in a coherent structure to rebuild and analyze the coherence of our discovered workflow:

and-split(CRS,CIC,CCA), *sequence*(CCA,PL), *and-join*(CIC,PL,CA), *xor-split*(CA,CH,SH,CC), *xor-join*(CH,SH,CC,SB).
$\longrightarrow_{RR1,RR2,RR3,RR4,RR5,RR6,RR7}$ $\mathcal{B}$(CRS,CIC,CC), $\mathcal{D}$(CIC,PL,CA), $\mathcal{G}$(CA,CH,SH,CC), $\mathcal{A}$(CCA,PL), $\mathcal{E}$(CH,SH,CC,SB)
$\longrightarrow_{RR11}$ $\mathcal{B}$(CRS,CIC,PL), $\mathcal{D}$(CIC,PL,CA), $\mathcal{G}$(CA,CH,SH,CC), $\mathcal{E}$(CH,SH,CC,SB) $\longrightarrow_{RR15}$ $\mathcal{B}$(CRS,CIC,PL),
$\mathcal{D}$(CIC,PL,CA), $\mathcal{A}$(CA,SB)$\longrightarrow_{RR13}$ $\mathcal{A}$(CRS,CA), $\mathcal{A}$(CA,SB)$\longrightarrow_{RR8}$ $\mathcal{A}$(CRS,SB) $\longrightarrow_{RR16}$ $\mathcal{W}orkflow$

5.3 Mining transactional behaviour

5.3.1 Mining transactional flow

Similar to the discovery of workflow patterns we build statistical transactional dependency tables (STrD). However, these tables report only event dependencies captured after activity failures. These dependencies allow us to specify and reason about the workflow transactional behaviour expressed in terms of transactional properties and transactional flow. To calculate these dependencies we use the same definition, except that we capture only event dependencies after activity failures. In practical terms, each STrD is related to an activity "act" and statistically captures statistically workflow behaviour after "act" fails. ITR_{act} is a STrD, built to capture transactional inter-activity dependencies after the failure of "act" (see definition 3). We note that act's dependencies before the failure are not reported in ITR_{act}.

Definition 3 (Inter-activity transactional dependency table)
We denote by ITR_{act} the inter-activity transactional dependency table that reports event dependencies after the failures of "act" . Let e_{act} be the failure event of "act" (i.e. e_{act}.activity = $act \wedge e_{act}$.state= f), each entry in $ITR_{act}(e_1, e_2)$ is an event dependency where :

- $(e_1 = e_{act} \vee e_2 = e_{act}) \vee$
- $\exists Evt : Event |$ ($(Evt = e_1 \vee Evt = e_2) \wedge e_{act} < Evt$);

To build ITR_{act}, we only use logs where "act" fails, keeping only the event dependencies after its failure. The dependencies located before "act" fails report the control flow. We distinguish two types of inter-activities transactional dependencies in ITR_{act} (see definition 3). The first category reports the event dependencies where we can find the failed event of "act" (e.state= f $\wedge$ e.activity = act). The second category reports any dependency where there is an activity executed after the failure of "act". In this set of activities, we can find activities which are not in the discovered control flow. Indeed, the recovery mechanisms can require "new" compensation activities which semantically undo "act"'s failure side-effects. Table 8.a represents a fraction of ITR_{CA} of our motivating workflow example after CA fails.

Table 9 describes the statistical log rules to discover the alternative dependencies related to an activity A_i using ITR_{A_i}. These dependencies are deduced if we observe a positive entry between the event reporting

Table 8 Fractions of Statistical Transactional Dependency tables of CA

(a) ITR_{CA} table

ITR_{CA}	CA,f	DBC,t	CIC,t	CCA,t	PL,t
CA,f	0	0	0	1	1
DBC,t	1	0	0	0	0
CIC,t	0	1			
CCA,t	0	1			
PL,t	0	0			

(b)ATR_{CA} tables

ATR_{CA}^{CIC}	t	f	a
t	1	0	0
f	0	0	0
c	0	0	0

ATR_{CA}^{CCA}	t	f	a
t	1	0	0
f	0	0	0
c	0	0	0

ATR_{CA}^{PL}	t	f	a
t	1	0	0
f	0	0	0
c	0	0	0

t=terminated, f=failed, c=cancelled

the failure of A_i and the event reporting the execution of A_j in ITR_{A_i}. According to the localisation of A_j we identify two types of alternative dependencies: a forward (respectively backward) alternative if A_j is after (respectively before) A_i in the discovered control flow. If A_j is not in the discovered control flow then we mine an alternative forward (respectively backward) if there is an activity A_k in the control flow such that A_j is executed before A_k in ITR_{A_i} and A_k is after (respectively before) A_i in the control flow. For instance in our motivating example (see Figure 1), we can deduce from Table 8.a that we have a backward alternative dependency from CA to DBC (ITR_{CA} ((DBC, t), (CA, f)) = 1 $\wedge$ ITR_{CA} ((CIC, t), (DBC, t)) = 1 $\wedge$ $SCfT(CA, CIC) > 0$). Besides, we can observe the execution of a "new" compensation activity (*i.e*, DBC) which does not exist in the discovered control flow.

Table 9 Statistical log properties of inter-activity transactional dependencies

Dependencies	Rules
$depAlt(A_i, A_j)$	ITR_{A_i} ((A_j, t), (A_i, f)) $\neq 0$ – **Backward:** $SCfT(A_i, A_j) > 0 \vee \exists A_k \mid SCfT(A_i, A_k) > 0 \wedge ITR_{Ai}$ ((A_k, t), (A_j, f))> 0 – **Forward:** $SCfT(A_j, A_i) > 0 \vee \exists A_k \mid SCfT(A_k, A_i) > 0 \wedge ITR_{Ai}$ ((A_k, t), (A_j, f))> 0
$depCnl(A_i, A_j)$	ITR_{A_i} ((A_j, c), (A_i, f))> 0

Table 9 describes statistical log features that specify the rules to discover the cancellation dependencies of an activity A_i based on ITR_{A_i}. These dependencies are deduced if we observe a positive entry between the event reporting the failure of A_i and the event reporting the cancellation of A_j in ITR_{A_i}.

5.3.2 Mining activity transactional properties

Transactional properties are discovered through the intra-activity transactional dependency table ATR (*c.f.* definition 4). Indeed, if an activity "act" fails, a table ATR_{act}^{A} is built for each activity "A" executed after the failure of "act" to capture the internal state dependencies between the events of "A". Thus, ATR_{act} extracts from instances logs where the failure of "act" is observed the dependencies between the internal events of act's following activities. Table 8.b captures the intra-activities transactional dependencies of CIC, CCA and PL after the failure of CA.

An activity a is retriable (a^r) if it always finishes successfully after a finite number of activations. Table 10 describes the statistical log rules to discover the retriable transactional property of a based on ITR_a. This property is mined if we observe an entry in ATR_a^a equal to 1 between the event reporting the state failed of a and the event reporting the state terminated of a and we also observe in ITR_a an entry equal to 1 between the events failed and terminated of a.

An activity a is pivot (a^p) if it cannot be re-executed once it terminates without failure, thus its execution effects are persistent. Table 10 describes the statistical log rules to mine the transactional property pivot of

Definition 4 (Intra-activities transactional dependency table)
We denote by ATR^{A}_{act} the Intra-activities transactional dependency table that reports "A"'s intra-event dependencies after "act" failures. These dependencies are extracted from a workflow log projection taking only "A" events from instances related to act failures. Let e_{act} be the failure event of "act" (i.e. e_{act}.activity = $act \wedge e_{act}$.state= f), each entry in $ITR_{act}(e_1, e_2)$ is an event dependency where :

- (e_1.activity = $A \wedge e_2$.activity = A) $\wedge$
- $\exists Evt : Event |\ Evt$.activity = $A \wedge e_{act} < Evt$;

Table 10 Statistical log properties of intra-activities transactional dependencies

properties	rules
a^r	$ATR^{a^r}_{a^r}(t,f)$=1 $\wedge$ $ITR_{a^r}((a^r,t),(a^r,f)) = 1$
a^p	$\nexists\, act |(\ act \neq a^p \wedge ATR^{a^p}_{act}(\text{“}x\text{”}, t) \neq 0 \wedge (\text{“}x\text{”}=\mathsf{t} \vee \text{“}x\text{”}=\mathsf{f} \vee \text{“}x\text{”}=\mathsf{c}))$

a based on ATR^a. This property is mined if we do not observe any ATR^a tables reporting a positive entry between a's terminated state and a's terminated, failed or cancelled state.

By using these statistical specifications we can discover the activity's transactional properties from StrD tables. For instance, we can deduce from Table 8.b that the CIC, CCA and PL activities are not pivot. Indeed, these activities are re-executed after reaching a terminated state when CA fails: (ATR^{CIC}_{CA}(t,t)= ATR^{CCA}_{CA}(t,t)= ATR^{PL}_{CA}(t,t)= 1).

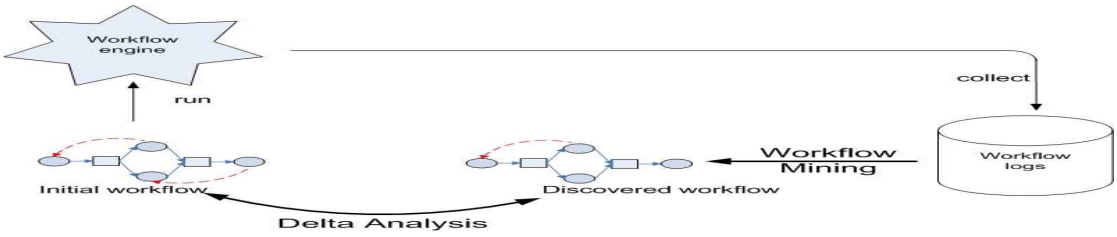
6 Transactional behaviour re-engineering

[42] reports two important process validation questions: (1) *"Does our model reflect what we actually do ?"* and (2) *"Do we follow our model?"*. Within a business process re-engineering context, we address these questions to propose a set of improvement and correction tools based on the process discovery results. We are interested, in particular, on the correction and the improvement of the workflow transactional flow. In some cases, a recovery mechanism initially designed and a-priori verified can generate execution errors due to unpredictable external factors (e.g. failures in the execution engine or system). Using a-posteriori verification, our work attempts to apply process log-based analysis to provide knowledge about discrepancies between the initially designed process model and its execution instances. Such a verification is necessary and useful since some interactions between process' activities may be dynamically specified at runtime, causing unpredictable interactions with other activities. Classical verification methods performed at design time are insufficient in such cases, as they only take static aspects into account.

The rational of this a-posteriori verification approach is to compare the "effective" process discovered after execution to the initially designed process to monitor whether it is coherent with the initial design and to detect potential discrepancies that can express incoherences or potential "new" process evolution requirements. As such, Delta Analysis (see Figure 7) is used to compare the real process, represented by the discovered workflow, to the initially designed process. In a nutshell, at run time users can deviate from the initially designed workflow. Delta Analysis between the initially designed and the discovered process allows us to monitor these deviations. Indeed, a deviation can become a current practice rather than to be a rare exception.

Independently of the chosen comparison technique [43,44], a delta analysis process aims to detect the discrepancies between the discovered and the initial models. These discrepancies can be exploited: (option 1) to motivate the process users to be closer to the initially designed process if the discrepancies do not express a real evolution or (option 2) to correct and improve the process model to be as close as possible to the "execution" reality. In the following, we propose a set of rules to improve the transactional behaviour according to option 2. These rules allow to correct or to remove, if necessary, any erroneous or omittable transactional behaviour and thus to reduce the recovery efforts. By erroneous or impractical transactional behaviours, we mean initially designed transactional flows which are not necessary or do not coincide with the execution. These transactional

Figure 7 Delta analyse for workflow re-engineering



behaviours can simply be expensive or a cause for additional errors. The correction and improvement rules (section 6.2) depend on transactional consistency semantics (section 6.1). The transactional behaviour specifications must respect these consistency rules linking the control to the transactional flow. These rules respect the semantic relations between the transactional flow and the control flow.

6.1 Transactional consistency rules

The specification of a recovery mechanisms defined through the transactional behaviour should respect rules which are partially dependent on the control flow to ensure reliable execution. Indeed, the recovery mechanisms (defined by the transactional flow) depends on the process execution logic (defined by the control flow). For example, regarding our motivating example (see Figure 1), it was possible to define CC as an alternative to CH because (according to the XOR-split control flow operator) they are defined as exclusive branches. Similarly, it was possible to define a cancellation dependency from CCA to CIC because (according to the AND-join control flow operator) the failure of CCA requires the cancellation of any activity executed in parallel before recovering the workflow execution.

Concretely, these consistency relations are inspired by [21] where "*potential*" transactional dependencies of workflow patterns are specified and proven. They determine the set of impossible transactional flows and then induce the set of "*potential*" transactional dependencies that can be defined according to a given pattern. From this set the designer chooses the effective transactional behaviour of the pattern that will be implemented and executed as the recovery mechanism in case of the failure of one of the pattern's activities. In Rule 1, we formally specify these relations. The first rule expresses the fact that a cancellation dependency can occur only between concurrent activities; whereas the second rule specifies the localisation of the coherent point[8] for some patterns. The third rule specifies which activities should be recoverable. The last rule is a consequence of the persistent feature of the pivot activity.

Rule 1 (Transactional consistency relations)
- ✓*R1* : For the *XOR-split*, *XOR-join* and *sequence* patterns: *No cancellation dependency in these patterns*;
- ✓*R2* For the *AND-join* and the *AND-split* patterns: *No coherent point located in parallel activities*;
- ✓*R3* : Except parallel flow outside *AND* or *XOR* patterns compositions: *All activities should be recoverable in case of failure*;
- ✓*R4* :For the *sequence*, *AND-split*, *XOR-split*, *OR-split*, *XOR-join*: *No backward recovery for pivot activities.*

For instance, we detail below how we can deduce the "potential" transactional behaviour for the *AND-join* pattern by interpreting the above consistency rules. *R2* can be considered for the AND-join pattern since

[8] A coherent point c of an activity a is noted as the predicate $Coh(a, c)$

this pattern does not consider alternatives. Indeed, *R2* indicates that coherent points can be located in patterns containing parallel flows where their activities are partially concurrent or not concurrent. So no alternative dependency is allowed in the *AND-join* pattern as the the activities are concurrent. However, the failure of any concurrent activity can cause the cancellation of those activities, according to *R1*. According to *R3*, all component activities should be recoverable if this pattern is involved in an AND composition. While *R4* only allows forward alternative dependencies from failed activities in case one of them is pivot. These relations are common and can be enriched to include other rules dependent on business semantics of the transactional workflow model. Indeed, further transactional properties or transactional dependencies can be added which consequently will restructure these semantic relations to integrate these modifications.

6.2 Correcting and improving the transactional behaviour

The correction rules (Rule 2) allow to remove the initially designed recovery mechanisms which are not in the discovered workflow. These erroneous mechanisms can be expensive for the WfMS which should provide the necessary means to support them. The transactional behaviour of the activity a at the initial design phase is noted as a_{nt}. Activity a_{dcvr} indicates the discovered transactional behaviour of a. While a_{cr} indicates the corrected or improved transactional behaviour of a. The first three rules derived by relation *R3* express that if we discover that an activity never fails then any recovery mechanism initially designed (for instance, retriable property, alternative or cancellation dependencies) is not necessary and should be removed. The last rule *S4*, inspired by relation *R1*, indicates that cancellation dependencies can exist only between parallel activities. If there is a cancellation dependency between two (ore more) activities and we discover that they are not concurrent (e.g. these activities belong to the parallel flows of the *M-out-of-N* or *OR-split* patterns and we discover that they are never executed concurrently) then this transactional behaviour should be suppressed.

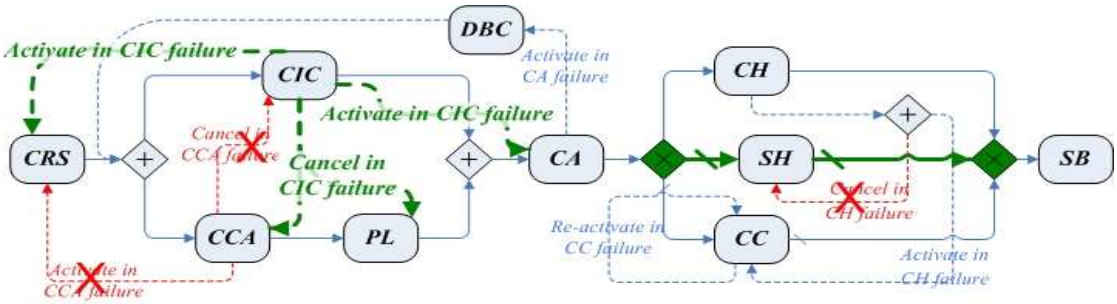
Rule 2 (Suppressing omitable (extraneous) transactional behaviours)
✓*S1* : $\nexists ITR_{a_{dc}} \wedge a^r_{nt} \Rightarrow \neg a^r_{cr}$;
✓*S2* : $\nexists ITR_{a_{dc}} \wedge depAlt(a_{nt}, b_{nt}) \Rightarrow \neg depAlt(a_{cr}, b_{cr})$
✓*S3* : $\nexists ITR_{a_{dc}} \wedge depCnl(a_{nt}, b_{nt}) \Rightarrow \neg depCnl(a_{cr}, b_{cr})$
✓*S4* : $SCfT(b_{dc}, a_{dc}) \geq 0 \wedge depCnl(a_{nt}, b_{nt}) \Rightarrow \neg depCnl(a_{cr}, b_{cr})$

For instance, we have discovered in section 5.1.3 that the payment by cash is never combined with the payment by check. As such, CH and SH are never executed in parallel. Consequently, by applying *S4* the cancellation dependency between these activities, belonging to the recovery mechanism of CH, should be removed. We also discover in our motivating example that CCA never fails. This implies a discrepancy between the discovered model and the initially designed model. Then we can conclude: (i) that there is no need to specify CCA as recoverable and thus, we suppress, by applying *S1*, the alternative transactional dependency between CCA and CRS; (ii) that there is no need to cancel CIC and thus, by applying *S2*, we suppress the cancellation dependency between CCA and CIC. Figure 8 shows these corrections red crossed out on our motivating example.

We also define rules proposing suggestions of recovery mechanisms for discovered failed activities without an initially designed recovery mechanisms (Rule 3). However, not every failed activity does necessarily require a recovery mechanism. The choice of specifying a recovery mechanism depends on a designers' business choice. However, we have to guarantee that if we discover that an initially designed activity can fail and induce the instance failure then a recovery mechanism for this activity has to be defined to respect *R3* in Rule 1.

A1.1, *A1.2* and *A1.3* suggest three different propositions specifying recovery mechanisms for a discovered failed activity. *A1.1* proposes to specify the retriable property as recovery mechanism if the activity is neutral. *A1.2* suggests a forward alternative dependency in case we have an activity representing a coherent point located after the failed activity. *A1.2* respects *R2* that indicates that the coherent point should not be in concurrence with the failed activity. *A1.3* suggests an alternative backward dependency in case we have an activity

Figure 8 Correcting and improving transactional flow



Rule 3 (Suggesting additional recovery mechanisms)
For each activity a, not initially recoverable but discovered to fail, we propose the following suggestions as recovery mechanisms:
✓*A1.1* : $a^n \Rightarrow a^r_{cr}$
✓*A1.2* : $Coh(a,b) \wedge SCfT(b_{dc}, a_{dc}) > 0 \Rightarrow depAlt(a_{cr}, b_{cr})$
✓*A1.3* : $Coh(a,b) \wedge SCfT(a_{dc}, b_{dc}) \geq 0 \Rightarrow depAlt(a_{cr}, b_{cr}) \wedge b^{p'}_{cr}$
✓*A2* : $(SCfT(a_{dc}, b_{dc}) = -1) \Rightarrow depAnl(a_{cr}, b_{cr})$

representing a coherent point located before the failed activity. *A1.3* respects *R2* that indicates that there should be no pivot activities between the coherent point and the failed activity. We note that the specification of the coherent points and their localisations depends on the designer's choices. Finally, *A2*, w.r.t to *R1*, suggests cancellation dependencies from discovered failed activities to its concurrent activities.

For instance, if we discover in our workflow example the fact that CIC can fail, we can induce a discrepancy between the discovered model and the initially designed model which indicates that CIC never fails. If the designer considers that the failure of CIC has no side-effects on the continuation of the instance execution, we can suggest: (i) by applying *A1.2*, a forward alternative recovery where the coherent point is located on CA. (ii) otherwise by applying *A1.3*, we can suggest to specify a backward recovery to CRS, if the concurrent activities CCA and PL are not pivot. Finally, by applying *A2*, we can suggest to specify cancellation dependencies between CIC and (CCA and PL). Figure 8 shows these suggestions on our motivating example in bold green. These suggestions could be entirely or partially applied w.r.t to a designer's business choices.

As we have shown through this example, monitoring the "effective" transactional behaviour allows us to detect design gaps and to improve the application reliability. Some deviations from the expected behaviour may be highly desirable to detect process evolution and execution anomalies showing initially hardly foreseeable process parameters, constraints and needs.

7 Validation

In this section we have developed a set of tools to validate our proposition for a reliable and correct workflow execution by log analysis. First, in section 7.1, we present log collecting tools. Thereafter, we have implemented (section 7.2) our mining workflow algorithms in a prototype that we have called Workflowminer [9], and as a plug-in [45] within the ProM framework. Finally, we have developed (section 7.3) a workflow reengineering tool based on the Event Calculus formalism to check, correct, and improve the workflow transactional behaviour.

7.1 Log collecting

In this section, we propose different means to validate the workflow log collecting issue. We have used two complementary tools: collecting real execution logs and generating simulated logs. The first objective was achieved by implementing an API which was grafted into a WfMS to collect log instances from already designed and executed workflow processes. To satisfy the second objective, we have used CPN [46] Petri nets simulation tools to generate simulated logs.

7.1.1 Real logs

In order to achieve the implementation of the first proposition, we have used Bonita as a workflow management system support. Bonita [47] is a third generation cooperative WfMS, open source and downloadable under the LGPL[9] license. It enables to specify, execute, and control cooperative process. It provides a complete set of integrated GUI tools to design, execute, and monitor workflow processes. It ensures user/system interaction and includes also a navigator enabling to manage and to control the process execution in an interactive way. Bonita is implemented using J2EE Enterprise Java Beans [48] that provide a flexible and portable environment. Integrating transactional behaviour in Bonita was the subject of a work done recently in our research team within the frame of an application to composite transactional web services. Using "transactional" plug-ins [48], Bonita was extended to be able to support the described transactional workflow model in section 2.

Although Bonita's persistence layer report and record the already executed project list, this initially collected data lacks several details which are important for our workflow mining algorithm. Among the services that Bonita integrates to control the cooperative aspects is JMS[10]. It is an J2EE[11] framework that enables to build and exchange messages. Each interaction between the user and the application (such as, project creation, suppression, activity activation or termination, etc.) is recorded as an event. As such, the JMS messaging services provide all the events which have just occurred from a workflow activity execution in Bonita. Based on this, we built our log collecting API that can be considered as a spy in Bonita sniffing and collecting all the transactions that can be produced and exchanged by the JMS messaging services during workflow instance execution. Each event reports the name of the project, the name of the activity carried out, its state, the date of its execution, etc. Afterwards, we have used the XML parser JDOM [49] to filter and read the data. JDOM performs a lexical analysis for each XML file produced by the log collecting tool and translates it to the adopted XML format conform to Definition 1.

7.1.2 Simulated logs

To evaluate the scalability of our approach, we have occasionally used simulated logs. Indeed, getting real logs from big size workflow examples that are enough various turns out to be a difficult task. The advantage in using simulated logs is that it is easier to fix and vary external factors ensuring a better diversity of the examples and a better and more accurate validation. The scaling issue of our validation test is consequently better dealt with simulated logs that enable us to cover a qualitatively and quantitatively various set.

In order to do this, we use a log simulating tool [50] which creates random XML logs by simulating already designed workflow processes based on CPN tools [12]. These tools support the modelling, the execution and the analysis of colored Petri nets [46]. They enable to create simulated logs conforming with the XML structure proposed in [32]. Modifications were brought to these tools to call predefined functions that create logs for each executed workflow instance. This stage implies modifications in the modelling level of CPN workflow declarations, particulary in the actions and the transition input/output levels. These functions indicate

[9] Lesser General Public License

[10] Java Messaging System

[11] Java 2 Platform, Enterprise Edition

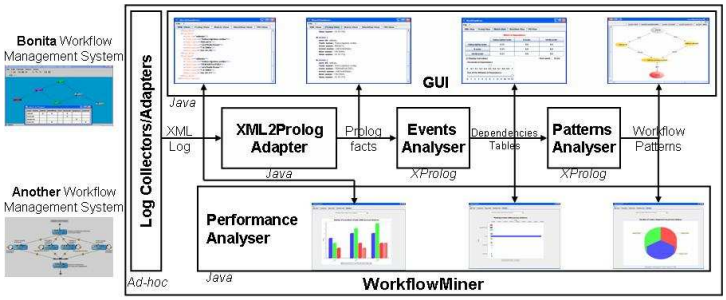
[12] http://wiki.daimi.au.dk/cpntools/cpntools.wiki

in particular the place, the prefix and the extension of the XML files that CPN tools create for each executed workflow instance. Thereafter, we have used ProMimport[13] to group or gather the simulated workflow logs.

7.2 Implementing workflow mining techniques

In this section, we describe the tools that we have implemented to validate the workflow mining techniques described in sections 4 and 5. We have implemented our presented workflow patterns mining algorithms within our prototype WorkflowMiner [8]. WorkflowMiner is written in Java and XProlog Java Prolog API[14]. WorkflowMiner, as seen in Figure 9, is composed of (a) an Events Analyser component dealing with the causal dependency analysis (producing different SDT tables), (b) the Patterns Analyser component using causal dependencies to discover workflow patterns and related transactional behaviour, and (c) the Performance Analyser exploiting brute event-based log, discovered causal dependencies, and discovered partial and global workflow patterns to measure performance metrics (more details in [51]). WorkflowMiner was also integrated to Bonita [52] as a workflow re-engineering component. Starting from workflow instantiations, (1) event streams are gathered in an XML log. In order to be processed, (2) these workflow log events are wrapped into a 1^{st} order logic format, compliant with the WorkflowLog format in the Definition 1. (3) Mining rules are applied on resulted 1^{st} order log events to discover workflow patterns. We use a Prolog-based presentation for log events, and mining rules.

Figure 9 WorkflowMiner Pipes and Filters Data Flow



In addition to WorkflowMiner, our workflow patterns mining technique has been implemented within the ProM framework [53] as a plug-in called "Workflow patterns miner" [52]. ProM is a plug-in environment for process mining. The ProM framework is flexible with respect to the input and output format, and is also open enough to allow for the easy reuse of code during the implementation of new process mining ideas. This plug-in helps us to provide detailed comparison of our approach to other implemented mining tools. An evaluation of our workflow mining technique has been done through sample and representative workflow applications [52]. Simulated logs for these examples were generated using the tool described in section 7.1.

7.3 Transactional checking and reengineering tool

We use Event Calculus (EC) formulas in order to check workflow transactional consistency and improve workflow execution reliability as it was described in section 6. Mukherjee et al. [54] discuss the relative merits and demerits of analyzing workflows using temporal logic, event algebra, concurrent transactional (computational tree logic), and Event Calculus (EC) logic. The concurrent transaction and the Event Calculus (EC) logics are the most suitable for expressing workflows because they can handle generalized constraints and can represent

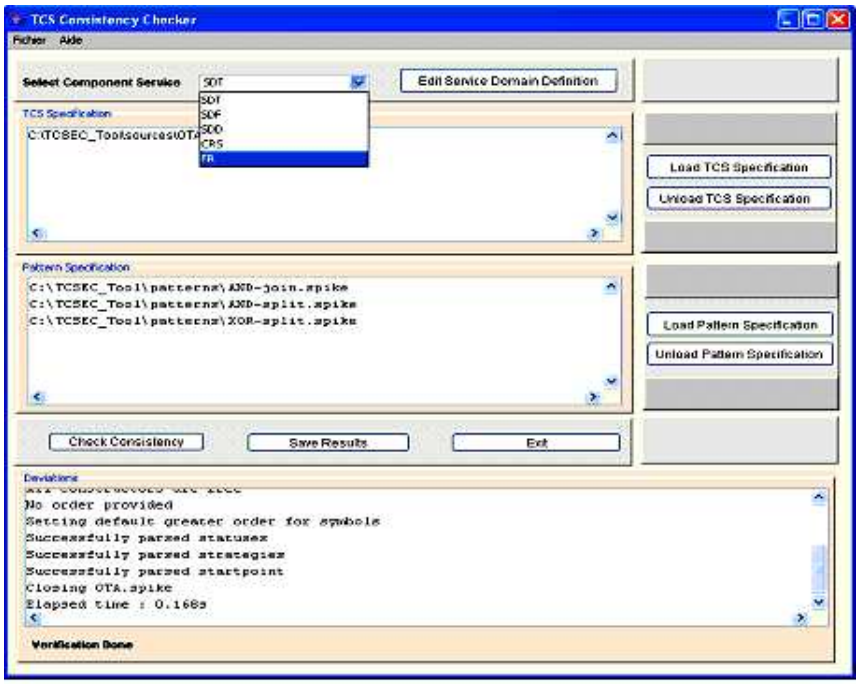
[13] www.promimport.sourceforge.net

[14] XProlog, www.iro.umontreal.ca/∼vaucher/XProlog/

control flow graphs with transition conditions on the arcs. Our approach uses the Discrete EC proposed by Mueller [55] as an extension of the classic EC, to declaratively model event based requirements specifications. We adopt an EC reasoning to check the initially designed transactional behaviour against the discovered transactional behaviour. This approach is defined by the specification of the axioms describing the transitions carried out and their effects on activities states during workflow instance executions. Compared to other formalisms, the choice of EC is motivated by both practical and formal needs, and has several advantages. First, in contrast to pure state-transition representations, the EC ontology includes an explicit time structure. This helps managing event-based systems where a number of input events may occur simultaneously. Second, the EC ontology is close enough to popular WfMC standards and provide an automatic support into the logical representation. Third, the semantics of non-functional requirements can be represented in EC, so that verification is once again straightforward. Due the lack of space, we refer to [56] for the complete formal description of our transactional workflow model.

The transformation of a process model to its EC specification is built as a parser that can automatically transform a given set of transactional workflow patterns into EC formulas according to the definitions and dependencies explained so far. In our tool, a pattern editor offers to the designer the different types of events and fluent initiation predicates that have been identified in the workflow and supports the specification of rules as logical combinations of these event and fluent initiation predicates. Designers may also use the editor to define additional fluents or transitions to represent activities, activity states, and relevant initiation and holding predicates through a graphical user interface that supports the specification of the process policies. Subsequently, the user specifies the predicates and rules of each pattern.

Figure 10 The graphical interface of the process consistency checker



When the workflow is specified, the editor can check its syntactic correctness. Afterwards, a transactional consistency checker proposes to verify the correctness of the transactional behaviour according to the above described rules. The user must load the process specification before the check consistency button can be enabled.

Then he has to choose and select the patterns to be used. Following this, he can select component activities and edit the domain definition of each one (transactional properties). When both the process specification and the patterns specifications are loaded, the verification process is ready to be executed. The result of the process verification can be saved in a file and the deviations specifications can be analyzed and used to create queries to locate activities that could substitute malfunctioning or unavailable activities.

To enable the re-engineering phase, we use logical predicates of the initially designed model, but we compare these predicates with the discovered process (effective workflow) which is the output of the workflow mining tool and the input of this re-engineering tool. When one or several consistency predicates are unsatisfied, this means that we have a wrong transactional behaviour in the execution. Thus, it is possible to exactly determine what happened. Eventually, a discovered pattern and its related suggestions and corrections are given to the workflow designer. Our tool displays the deviations from the specifications. A screenshot of the graphical interface of the process consistency checker is shown in Figure 10. At the verification back-end, we have used the induction-based theorem prover SPIKE [57]. SPIKE was chosen for (i) its high automation degree, (ii) its ability on case analysis, (iii) its refutational completeness (to find counter-examples), and (iv) its incorporation of decision procedures (to automatically eliminate arithmetic tautologies produced during the proof attempt).

8 Related works

Some studies investigated the issues related to the exception handling and recovery from activity failures in workflow management systems. A first discussion on workflow recovery issues has been presented in [58]. The necessity of workflow recovery concepts was partly addressed in [14]. Especially, the concept of business transactions, introduced in [59], deals with some basic workflow recovery. A significant amount of work applies the concepts introduced in transaction management to business process environments. Hamadi et al. [7] offer an exhaustive survey. Besides, they describe a framework for the specification of high-level recovery policies that are incorporated either with a single task or a set of tasks, called Recovery Regions. They propose an extended Petri net model for specifying exceptional behavior in business processes and adapt the mechanisms of the underlying Petri net at run time to handle exceptions while keeping the Petri net design simple and easy. Recently, Indraksy et al. [60] focus on how to correctly specify dependencies in advanced transactions. The authors enumerate the different kinds of dependencies that may be present in advanced transactions and classify them in two broad categories: event ordering and event enforcement dependencies. Different event ordering and event enforcement dependencies in an advanced transaction often interact in subtle ways resulting in conflicts and redundancies. They describe different types of conflicts that can arise due to the presence of multiple dependencies and describe how one can detect such conflicts. Zongwei et al. [61] identify the exception handling techniques that support conflict resolution in cross-organizational settings. In particular, they propose a novel,"bundled" exception-handling approach, which supports (1) knowledge sharing exception specifications and handling experiences, (2) coordinated exception handling, and (3) intelligent problem solving using case based reasoning to reuse exception handling experiences.

In this article, we have presented a transactional workflow model that should be considered as a support model to our workflow mining and re-engineering techniques which are the article's main contribution. This model is flexible enough to include additional recovery mechanisms. It can be customized for different kinds of transaction models by restricting or extending the type of dependencies that can exist between the activities. This customization is done by limiting or extending the types of the inter or intra activity dependencies. Thus, the transactional dependencies can be enriched and modified to include further transactional behaviour relying on specific or generic business semantics. Thanks to this flexibility, the extension or the adaptation of the above described transactional workflow model can be easily deployed with minor changes to the workflow mining approach.

There is only few research activities on process re-engineering based on process log analysis. Prior works in this field is limited to estimating deadline expirations and exceptions prediction. In [62], van der Aalst et al. check and quantify how much the actual behavior of a process, as recorded in message logs, conforms to the expected behavior as specified in a process model. In [63,64], Sayal et al. describe a tool set on top of

HPs Process Manager including a "BPI Process Mining Engine" to support business and IT users in managing process execution quality by providing several features, such as analysis, prediction, monitoring, control, and optimization. We also need to mention the work of van der Aalst et al. [65] for retrospective checks of security violations which shows how a specific mining algorithm can be used to support intrusion detection. In a similar context, Rozinat et al. [66] use log analysis techniques to measure process alignments, i.e. to compare the execution behaviour with the intended behaviour. However, this work does not describe how to use the results of this process alignment to improve or correct the process.

Obvious applications of process mining exist in model driven business process software engineering, both for bottom up approaches used in business process alignment [67,68], and for top down approaches used in workflow generation [69]. A number of research efforts in the area of workflow management have been directed for mining workflow models. This issue is close to what we propose in terms of discovery. The idea of applying process mining in the context of process management was first introduced in [70]. This work proposes methods for automatically deriving a formal process model from a log of events related to its executions based on workflow graphs. Cook and Wolf [71] investigated similar issues in the context of software engineering processes. They extended their work initially limited to sequential processes to concurrent processes in [72]. Van der Aalst et al. propose an exhaustive survey in [32]. However, previous works in workflow mining seem to focus on control flow mining perspectives.

To the best of our knowledge there are practically no approaches in workflow mining that address the issue of failure handling and recovery except our works [10,45] which propose techniques for discovering workflow transactional behaviour. This article may be seen as a first step in this area. Recently, new issues in control flow mining have been addressed by [73] that propose genetic algorithms to tackle log noise problem or non-trivial constructs using a global search technique. Bergenthum et al. use region based synthesis methods and compare their efficiency and usefulness [74] . Our process mining approach can be distinguished by supporting local discovery through a set of control flow mining rules that are characterized by a "local" pattern discovery enabling partial results to be discovered. It recovers partial results from log fractions. Moreover, even though the non-free choice (NFC) construct is mentioned as an example of a pattern that is difficult to mine [32], our approach discovers an M-out-of-N-Join pattern which can be seen as a generalisation of the Discriminator pattern that were proven to be inherently non free-choice. Recently, [75,73] propose a complete solution that can deal with such constructs. Besides, our mining approach discovers more complex features with a better specification of the "fork" operator (*and-split*, *or-split*, *xor-split* patterns) and the "join" operator (*and-join*, *XOR-Join*, and *M-out-of-N-Join* patterns). We provided rules to discover seven most used patterns, but this set of patterns can be easily enriched by specifying new statistical dependencies and their associated properties or by using the existing properties in new combinations. Our approach deals better with concurrency through the introduction of the "*concurrent window*" that proceeds dynamically with concurrency. Indeed, the size of the "*concurrent window*" is not static or fixed, but variable according to the activity's concurrent behaviour without increasing the computing complexity. It is trivial to establish that the algorithms describing our approach are of polynomial-complexity not exceeding the quadratic order $O(n^2)$. Indeed, these algorithms do not contain recursive calls and contain no more than two overlapping loops whose length is equal to the number of Events within an Eventstream. Due to the lack of space we can refer to [45,52] for a thorough description of the WorkflowMiner's architecture and a comparison to existing implemented workflow mining tools. At the end of this paper, we present an annex describing some experimentations performed in our plug-in. These experimentations show different workflow examples with different levels of complexity discovered in our plug-in.

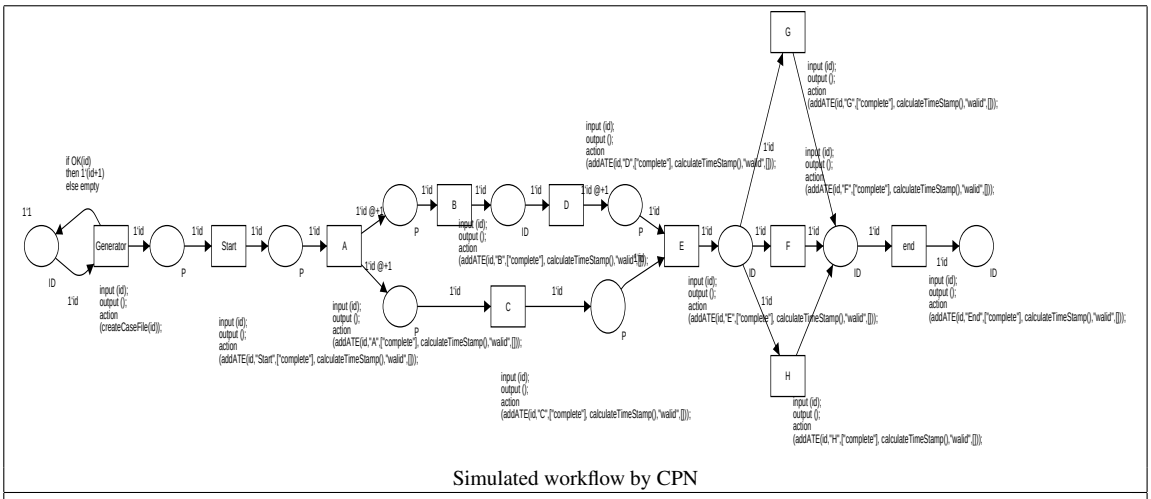
9 Conclusion

Workflow management systems are increasingly deployed to deliver reliable e-business transactions across organizational boundaries. To ensure a high service quality in such transactions, failure handling re-engineering for reliable executions is needed. However, failed instance executions may arise because it is hard to predict workflow transactional behavior that can cause an unexpected activity failure. In this article we have presented an original approach to ensure reliable workflow transactional behaviour. Our work attempts to apply process

log-based analysis to provide knowledge about discrepancies between the initially designed process model and its execution instances represented by the log which can subsequently be used in a workflow re-engineering process. Our approach starts from workflow logs and uses a set of mining techniques to discover the workflow control flow and the workflow transactional behaviour. Then, based on a Delta Analysis of the initially designed model and the discovered workflow, a set of rules is used to improve the recovery mechanisms. Our aim is to be as close as possible to the business and conceptual choices of a workflow designer and the evolution needs of workflow users expressed during runtime and reported by the workflow mining results.

Currently we are working on discovering additional patterns by using more metrics (*e.g.* entropy, periodicity, etc.) and by enriching the workflow log. We are also trying to enhance workflow recovery mining techniques by enriching workflow logs and extracting data flow dependencies. We are also interested in applying process mining techniques in composite Web services. In [76], we proposed a set of mining techniques to discover composite (web) service transactional flow in order to improve composite service recovery mechanisms. Our work in [77] uses web services logs to enable the verification of behavioral properties in web service composition. The main focus of this work is not on on discovery, but on verification. This means that given an event log and a formal property, we check whether the observed behavior matches the (un)expected/(un)desirable behavior. Recently, we have specified in [56] a combined approach that describes a formal framework, based on Event Calculus to check the transactional behavior of composite service before and after execution. Our approach provides a logical foundation to ensure transactional behavior consistency at design time and report recovery mechanisms deviations after runtime.

Table 11 Four Different "fork" and "join" operators



10 Annexe : Examples of workflow discovered "Workflow patterns miner" plug-in

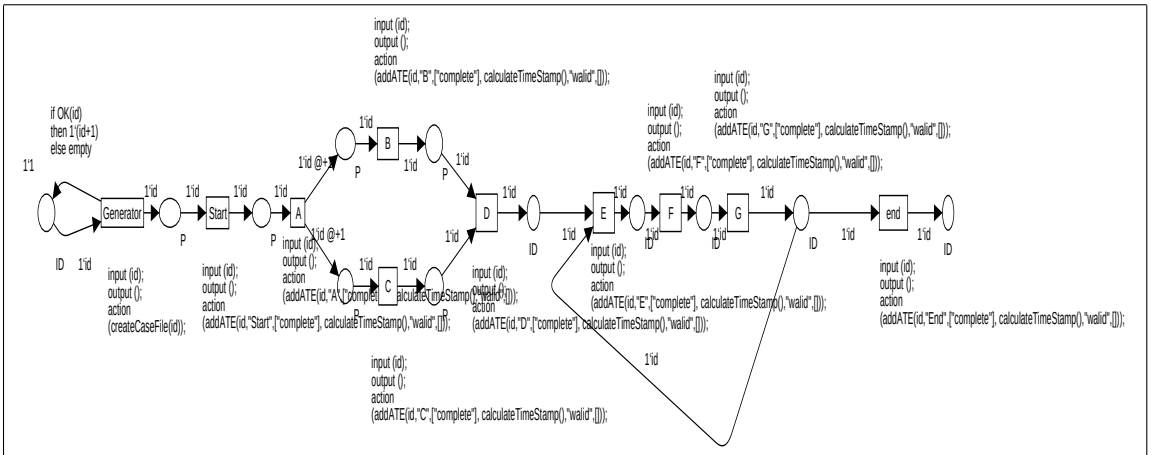
In this section, we present the discovered workflow results of some experiments executed within "Workflow patterns miner" plug-in using CPN simulated tools. Namely, we used five different workflow examples with different levels of complexity that cover various kind of "join" and "fork" operators describing some interesting pattern combinations, concurrent flows and loops. Our aim is to discover these five workflow examples from their logs. We note that the simulated workflows are represented using the Petri nets and the discovered workflows are modeled using workflow patterns. The two models have an equivalent behavior.

10.1 Four different "fork" and "join" operators

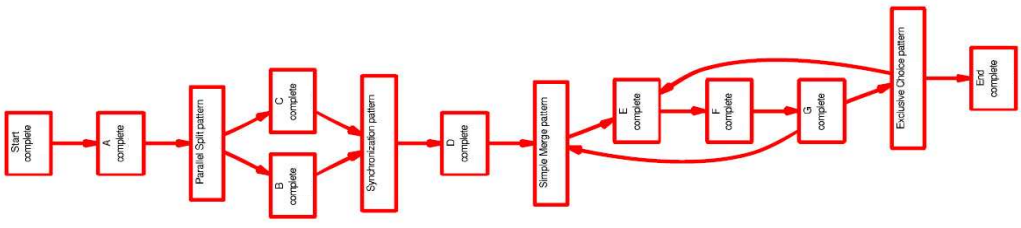
Table 11 describes a workflow that contains 2 "fork" operators and 2 "join" operators describing four different types of patters : *AND-join*, *AND-split*, *XOR-join*, *XOR-split*.

10.2 Ordinary loop

Table 12 Ordinary loop example



Simulated workflow by CPN



Discovered workflow by "Workflow patterns miner"

Table 12 describes a workflow that contains an 3-activity length ordinary loop (i.e. E, F, G). This loop is designed in the discovered workflow using the patterns : *XOR-split* and *XOR-join*.

10.3 Short loop

Table 13 One activity short loop workflow example

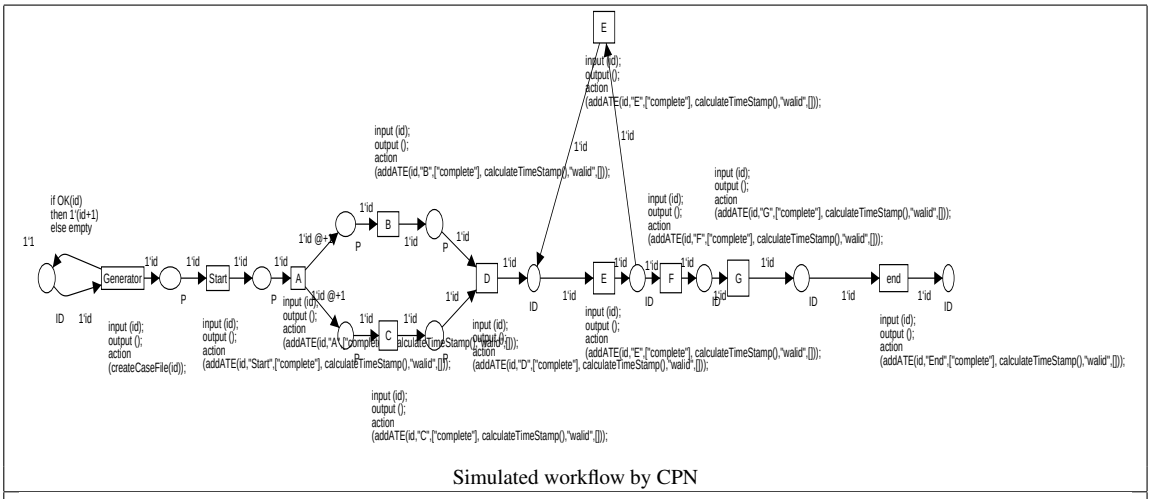
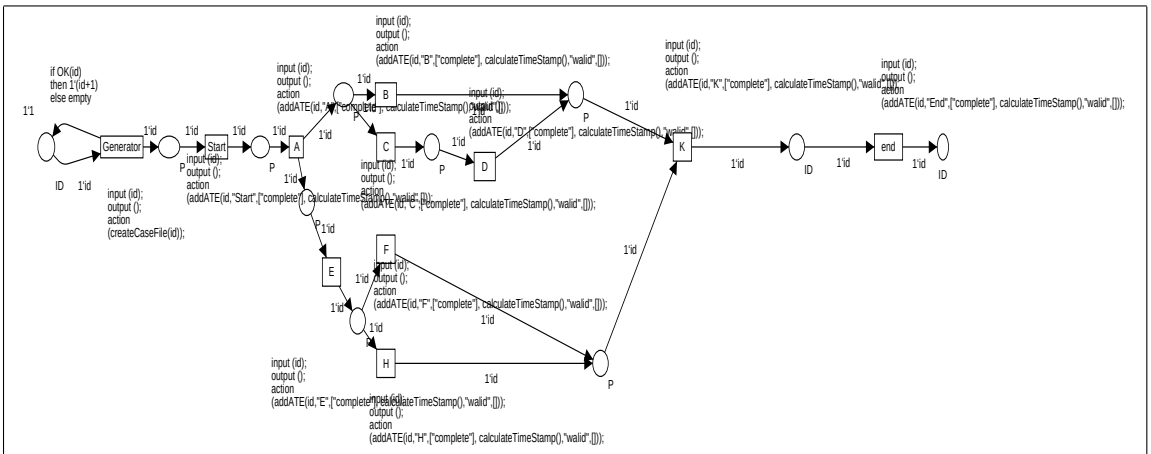


Table 13 describes a workflow that contains a short loop that involves one activity (i.e. E activity). As table 12, this loop is designed in the discovered workflow using the patterns : *XOR-split* and *XOR-join*. We note that our mining algorithm can discover any kind of loop except the 2-activity length activity that needs a specific log preprocessing (out of the scope of this paper). This preprocessing is able to distinguish between the concurrent behavior and this special kind of loops.

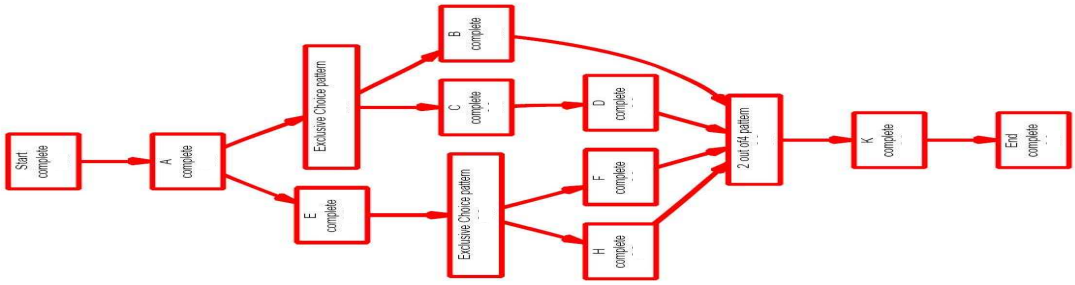
10.4 Non free choice

Table 14 describes a workflow that contains the particular *M-out-of-N-Join*. This pattern describes the non free choice special behaviors, a mix between the synchronization and the choice in the same pattern. The *M-out-of-N-Join* pattern represents an implementation of this behavior [23].

Table 14 A workflow example containing the *M-out-of-N-Join* pattern



Simulated workflow by CPN



Discovered workflow by "Workflow patterns miner"

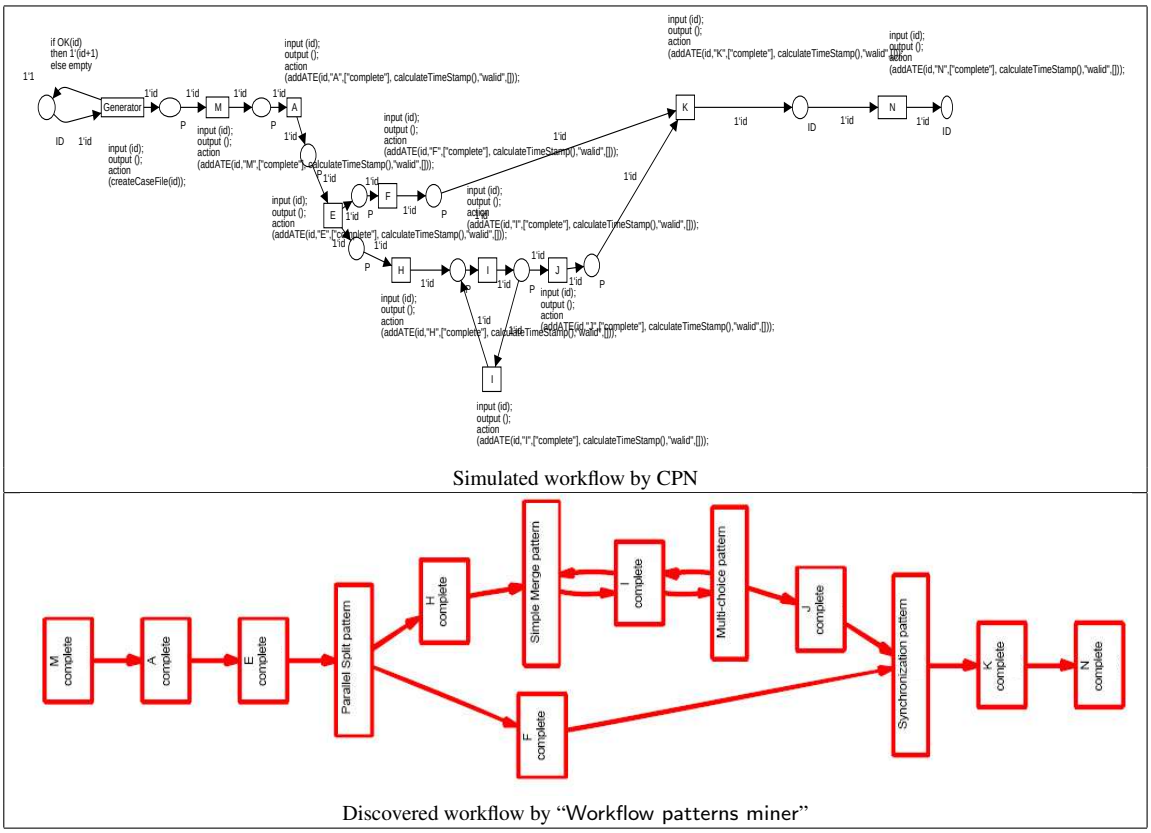
10.5 Concurrent behavior and loop

Table 15 describes a workflow that contains one-activity length loop (i.e activity I) concurrent with the flow containing the activity F. The originality of this workflow comes from the complexity or the difficulty to distinguish between these two types of behavior.

References

1. A. H. M. ter Hofstede, M. E. Orlowska, and J. Rajapakse. Verification problems in conceptual workflow specifications. *Data Knowl. Eng.*, 24(3):239–256, 1998.
2. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
3. N. R. Adam, V. Atluri, and W.-K. Huang. Modeling and analysis of workflows using petri nets. *J. Intell. Inf. Syst.*, 10(2):131–158, 1998.
4. B. F. van Dongen, R. M. Dijkman, and J. Mendling. Measuring similarity between business process models. In Zohra Bellahsene and Michel Léonard, editors, *CAiSE*, volume 5074 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2008.
5. J. Eder and W. Liebhart. Workflow recovery. In *Conference on Cooperative Information Systems*, pages 124–134, 1996.
6. B. C. Glasson, I. Hawryszkiewycz, A. Underwood, and R. Weber. *Business Process Re-Engineering*, volume A-54 of *IFIP Transactions*. Elsevier, 1994.
7. R. Hamadi, B. Benatallah, and B. Medjahed. Self-adapting recovery nets for policy-driven exception handling in business processes. *Distributed and Parallel Databases*, 23(1):1–44, 2008.

Table 15 Concurrent behavior and loop example



Simulated workflow by CPN

Discovered workflow by “Workflow patterns miner”

8. W. Gaaloul, K. Baïna, and C. Godart. Towards mining structural workflow patterns. In K. V. Andersen, J. K. Debenham, and R. Wagner, editors, *DEXA*, volume 3588 of *LNCS*, pages 24–33. Springer, 2005.
9. W. Gaaloul, K. Baïna, and C. Godart. A bottom-up workflow mining approach for workflow applications analysis. In *The 2nd International Workshop on Data Engineering Issues in E-Commerce and Services*, LNCS, San Francisco, California, USA, June 26 2006. Springer-Verlag.
10. 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.
11. W. Gaaloul and C. Godart. Mining workflow recovery from event based logs. In *Business Process Management*, pages 169–185, 2005.
12. J. Veijalainen, F. Eliassen, and B. Holtkamp. The S-transaction Model. In A.K. Elmagarmid, editor, *Database transaction models for advanced applications*. Morgan Kauffman, 1990.
13. U. Dayal, M. Hsu, and R. Ladin. Business process coordination: State of the art, trends, and open issues. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB*, pages 3–13. Morgan Kaufmann, 2001.
14. D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases*, 3(2):119–153, 1995.
15. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
16. M. Ansari, L. Ness, M. Rusinkiewicz, and A. P. Sheth. Using flexible transactions to support multi-system telecommunication applications. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 65–76. Morgan Kaufmann Publishers Inc., 1992.
17. P. K. Chrysanthis and K.Ramamritham. Synthesis of Extended Transaction Models. *ACM Transactions on Database Systems*, 19(3):451–491, september 1994.
18. H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM Press, 1987.

19. M. Rusinkiewicz and A. Sheth. Specification and Execution of Transactional Workflows. In Won Kim, editor, *Modern Database Systems, The Object Model Interoperability and beyond*, pages 592–620. Addison Wesley, ACM Press, 1995.
20. A. Sheth and M. Rusinkiewicz. On transactional workflows. *Special Issue on Workflow and Extended Transaction Systems IEEE Computer Society , Washington DC*, 1993.
21. S. Bhiri, O. Perrin, and C. Godart. Extending workflow patterns with transactional dependencies to define reliable composite web services. In *AICT/ICIW*, page 145. IEEE Computer Society, 2006.
22. W. Gaaloul, S. Bhiri, and A. Haller. Mining and re-engineering transactional workflows for reliable executions. In Christine Parent, Klaus-Dieter Schewe, Veda C. Storey, and Bernhard Thalheim, editors, *ER*, volume 4801 of *Lecture Notes in Computer Science*, pages 485–501. Springer, 2007.
23. W. M. P. van der Aalst, A. P. Barros, A. H. M. ter Hofstede, and B. Kiepuszewski. Advanced Workflow Patterns. In O. Etzion and Peter Scheuermann, editors, *5th IFCIS Int. Conf. on Cooperative Information Systems (CoopIS'00)*, number 1901 in LNCS, pages 18–29, Eilat, Israel, September 6-8, 2000. Springer-Verlag.
24. A. Elmagarmid, Y. Leu, W. Litwin, and Marek Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the sixteenth international conference on Very large databases*, pages 507–518. Morgan Kaufmann Publishers Inc., 1990.
25. W. Du, J. Davis, and M.-C. Shan. Flexible specification of workflow compensation scopes. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work : the integration challenge*, pages 309–316. ACM Press, 1997.
26. J. Moss. Nested transactions and reliable distributed computing. In *Proceedings Of The 2nd Symposium on Reliability in Distributed Software and database Systems*. IEEE CS Press, 1982.
27. B. Kiepuszewski, R. Muhlberger, and M. E. Orlowska. Flowback: providing backward recovery for workflow management systems. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 555–557. ACM Press, 1998.
28. J. Eder and W. Liebhart. The workflow activity model wamo. In *CoopIS*, pages 87–98, 1995.
29. P. W. P. J. Grefen, J. Vonk, E. Boertjes, and P. M. G. Apers. Two-layer transaction management for workflow management applications. In Abdelkader Hameurlain and A. Min Tjoa, editors, *DEXA*, volume 1308 of *Lecture Notes in Computer Science*, pages 430–439. Springer, 1997.
30. G. Canals, C. Godart, F. Charoy, P. Molli, and H. Skaf-Molli. Coo approach to support cooperation in software developments. *IEE Proceedings - Software*, 145(2-3):79–84, 1998.
31. M. Kamath and K. Ramamritham. Failure handling and coordinated execution of concurrent workflows. In *ICDE*, pages 334–341. IEEE Computer Society, 1998.
32. W. M. P. van der Aalst and B. F. van Dongen. Workflow mining: A survey of issues and approaches. In *Data and Knowledge Engineering*, 2003.
33. J. Eder, G. E. Olivotto, and W. Gruber. A data warehouse for workflow logs. In *Proceedings of the First International Conference on Engineering and Deployment of Cooperative Information Systems*, pages 1–15. Springer-Verlag, 2002.
34. M. zur Muehlen. Process-driven management information systems - combining data warehouses and workflow technology. In Bezalel Gavish, editor, *Proceedings of the 4th International Conference on Electronic Commerce Research (ICECR-4)*, pages 550–566, Dallas (TX), 2001. Southern Methodist University.
35. WorkFlow Management Coalition. Terminology and glossary. technical report wfms-tc-1011. Technical report, Workflow Management Coalition Brussels - Belgium, 1996.
36. W. M. P. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.*, 16(9):1128–1142, 2004.
37. B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. The prom framework: A new era in process mining tool support. In Gianfranco Ciardo and Philippe Darondeau, editors, *ICATPN*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer, 2005.
38. W. Gaaloul. *La Dcouverte de Workflow Transactionnel pour la Fiabilisation des Excutions*. Phd thesis, Université Henri Poincaré - Nancy 1, LORIA, November 3, 2006.
39. P. C. Attie, M. P. Singh, A. P. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 134–145. Morgan Kaufmann, 1993.
40. J. E. Cook and A. 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.
41. H. Mannila, H. Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.
42. J. E. Cook and A. L. Wolf. Software process validation: quantitatively measuring the correspondence of a process to a model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 8(2):147–176, 1999.
43. W. M. P. van der Aalst. Exterminating the dynamic change bug: A concrete approach to support workflow change. *Information Systems Frontiers*, 3(3):297–317, 2001.
44. T. Basten and W. M. P. van der Aalst. Inheritance of behavior. *J. Log. Algebr. Program.*, 47(2):47–145, 2001.
45. W. Gaaloul and C. Godart. A workflow mining tool based on logs statistical analysis. In Frank Maurer and Günther Ruhe, editors, *SEKE*, pages 37–44, 2006.
46. K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 2*. Springer-Verlag, London, UK, 1995.
47. V. Miguel and F. Charoy. Bonita: Workflow cooperative system. http://bonita.objectweb.org, 2003.
48. Sun. *Enterprise JavaBeansTM Specification, Version 2.1*. Sun Microsystems, August 2002.
49. E. R. Harold. *Processing XML with Java: a guide to SAX, DOM, JDOM, JAXP, and TrAX*. 2003.

50. A. de Medeiros and C. Gunther. Process mining: Using cpn tools to create test logs for mining algorithms, 2005.
51. 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.
52. K. Baïna, W. Gaaloul, R. El Khattabi, and A. Mouhou. Workflowminer: a new workflow patterns and performance analysis tool. In Nacer Boudjlida, Dong Cheng, and Nicolas Guelfi, editors, *CAiSE Forum*, volume 231 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.
53. W. M. P. van der Aalst, B. F. van Dongen, C. W. Günther, R. S. Mans, A. K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H. M. W. Verbeek, and A. J. M. M. Weijters. Prom 4.0: Comprehensive support for *eal* process analysis. In Jetty Kleijn and Alex Yakovlev, editors, *ICATPN*, volume 4546 of *Lecture Notes in Computer Science*, pages 484–494. Springer, 2007.
54. S. Mukherjee, H. Davulcu, M. Kifer, P. Senkul, and G. Yang. Logic based approaches to workflow modeling and verification. In Jan Chomicki, Ron van der Meyden, and Gunter Saake, editors, *Logics for Emerging Applications of Databases*, pages 167–202. Springer, 2003.
55. E. T. Mueller. Event calculus reasoning through satisfiability. *J. Log. and Comput.*, 14(5):703–730, 2004.
56. W. Gaaloul, M. Hauswirth, M. Rouached, and C. Godart. Verifying composite service recovery mechanisms: A transactional approach based on event calculus. In *15th International Conference on Cooperative Information Systems CoopIS07*, November, 2007.
57. S. Stratulat. A general framework to build contextual cover set induction provers. *Journal of Symbolic Computation*, 32(4):403–445, 2001.
58. W. Woody Jin, M. Rusinkiewicz, L. Ness, and A. Sheth. Concurrency control and recovery of multidatabase work flows in telecommunication applications. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 456–459. ACM Press, 1993.
59. F. Leymann. Supporting business transactions via partial backward recovery in workflow management systems. In *Proceedings of BTW95*, pages 51–70. Springer, 1995.
60. I. Ray and T. Xin. Analysis of dependencies in advanced transaction models. *Distributed and Parallel Databases*, 20(1):5–27, 2006.
61. Z. Luo, A. P. Sheth, K. Kochut, and I. B. Arpinar. Exception handling for conflict resolution in cross-organizational workflows. *Distributed and Parallel Databases*, 13(3):271–306, 2003.
62. W.M.P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and H.M.W. Verbeek. Conformance checking of service behavior. *ACM Transactions on Internet Technology (TOIT), Special issue on Middleware for Service-Oriented Computing*, 2007.
63. 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.
64. D. Grigori, F. Casati, M. Castellanos, U. Dayal, M. Sayal, and M.-C. Shan. Business process intelligence. *Comput. Ind.*, 53(3):321–343, 2004.
65. W. M. P. van der Aalst and A. Karla A. de Medeiros. Process mining and security: Detecting anomalous process executions and checking process conformance. *Electr. Notes Theor. Comput. Sci.*, 121:3–21, 2005.
66. A. Rozinat and W. M. P. van der Aalst. Conformance testing: Measuring the fit and appropriateness of event logs and process models. In *Business Process Management Workshops*, pages 163–176, 2005.
67. W. M. P. van der Aalst. Business alignment: Using process mining as a tool for delta analysis. In *CAiSE Workshops (2)*, pages 138–145, 2004.
68. B. Benatallah, F. Casati, and F. Toumani. Analysis and management of web service protocols. In *ER*, pages 524–541, 2004.
69. K. Baïna, B. Benatallah, F. Casati, and F. Toumani. Model-driven web service development. In *CAiSE*, pages 290–306, 2004.
70. R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. *Lecture Notes in Computer Science*, 1377:469–498, 1998.
71. J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):215–249, 1998.
72. J. E. Cook and A. 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.
73. A. K. A. de Medeiros, A. J. M. M. Weijters, and W. M. P. van der Aalst. Genetic process mining: an experimental evaluation. *Data Min. Knowl. Discov.*, 14(2):245–304, 2007.
74. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Process mining based on regions of languages. In *BPM*, pages 375–383, 2007.
75. L. Wen, W. M. P. van der Aalst, J. Wang, and J. Sun. Mining process models with non-free-choice constructs. *Data Min. Knowl. Discov.*, 15(2):145–180, 2007.
76. S. Bhiri, W. Gaaloul, and C. Godart. Mining and improving composite web services recovery mechanisms. *Int. J. Web Service Res.*, 5(2):23–48, 2008.
77. M. Rouached, W. Gaaloul, W. M. P. van der Aalst, S. Bhiri, and C. Godart. Web service mining and verification of properties: An approach based on event calculus. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4275 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 2006.