SOCA (2008) 2:93–110
DOI 10.1007/s11761-008-0023-6

SPECIAL ISSUE PAPER

Log-based mining techniques applied to Web service composition reengineering

Walid Gaaloul · Karim Baïna · Claude Godart

Received: 31 October 2007 / Accepted: 24 March 2008 / Published online: 8 May 2008

Abstract Web service compositions are becoming more and more complex, involving numerous interacting ad-hoc services. These services are often implemented as business processes themselves. By analysing such complex web service compositions one is able to better understand, control and eventually re-design them. Our contribution to this problem is a mining algorithm, based on a statistical technique to discover composite web service patterns from execution logs. Our approach is characterised by a "local" pattern's discovery that covers partial results through a dynamic programming algorithm. Those locally discovered patterns are then composed iteratively until the composite Web service is discovered. The analysis of the disparities between the discovered model and the initial ad-hoc composite model (delta-analysis) enables initial design gaps to be detected and thus to re-engineer the initial Web service composition.

Keywords Composite service mining · Service intelligence · Service analysis · Service validation · Service reengineering frameworks · Model driven reengineering · Workflow patterns

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

W. Gaaloul (✉)
DERI-NUIG, IDA Business Park, Galway, Ireland
e-mail: walid.gaaloul@deri.org

K. Baïna
ENSIAS, Université Mohammed V-Souissi,
BP 713, Agdal-Rabat, Morocco
e-mail: baina@ensias.ma

C. Godart
LORIA-INRIA-UMR 7503, BP 239,
54506 Vandœuvre-les-Nancy Cedex, France
e-mail: godart@loria.fr

1 Introduction

Service Oriented Architectures (SOAs) are suffering from their own success: a lack of an explicit process model, difficult maintainability, and poor monitoring facilities. Our paper is a contribution to this problem through our composite service (CS) mining algorithm. CS mining supports business process rediscovery based on a log analysis and can be used to retroactively (re)design models to better understand the actual process execution reality. Our approach aims to support *composite web service continuous evolution* by analysing composite web service execution logs, discovering effective web service compositions and helping to improve and to correct the initially designed process models.

The main idea of our algorithm is that a set of web services interact in an ad-hoc manner with their execution logic implicit to the implementation (i.e., no explicit process model). This ad-hoc web service interactions can be better abstracted and formalised as an orchestration process. Our objective, through structural web service mining, is (1) to discover the implicit orchestration protocol behind a set of web services; interactions, and (2) to explore to which extent this implicit protocol can be mapped to an explicit orchestration protocol (e.g., a BPEL process) either: (a) to be well managed and controlled, or (b) to be well analysed and understood, or (c) to be verified from either a structural or a behavioural point of view. Our algorithm starts by collecting log information from CS execution instances. Then, through statistical techniques, a graphical intermediary representation is built to model service elementary dependencies. These dependencies are then refined to discover control flow patterns. The

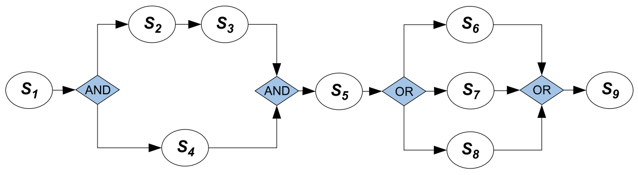

Fig. 1 CS running example: an ad-hoc composite web service orchestration

discovered results are used thereafter in re-engineering and analysis phase.

Motivating example

In this article, we motivate our approach with an example of a composite (web) service supposedly implemented as an ad-hoc composite web service orchestration. This CS represents a car rental application (see Fig. 1). It acts as a broker offering to its customers a set of car choices made from their requirements expressed in the (web) service S_1. S_4 checks the customer ID while S_2 checks the car availability and S_3 provides further information about available cars and the respective car rental supplier. Afterwards, the customer makes his choice and agrees on rental terms in S_5 service. The customer is requested to pay either by credit card (S_6), by cash (S_7), or by check (S_8) or by combining the payment by credit card and by cash. Finally, the bill is sent to the customer by S_9.

Previous approaches primarily developed a set of techniques to analyse and check the composition model, based on a specific modelling formalisms. Although powerful these approaches may fail to ensure CS reliable executions in some cases, even if they formally validate the CS model. This is because properties specified in the studied composition models remain assumptions that may not coincide with the reality. In fact, the users can express different needs from the initial ad-hoc CS model during execution, by choosing a different payment process or by removing a concurrent behaviour. Formal approaches cannot report these dynamic requirements.

Overview

In this article, we describe a set of mining techniques and algorithms for a CS patterns discovery which we have specified, proved and implemented. Our approach allows one to detect and correct design errors caused by omissions or errors at the initial (ad-hoc) design phase and caters for CS evolutions observed at run time. As such, we can rediscover the effective or "real" service interactions from a CS log. Our approach can be summarised as follows:

- **Collecting execution history** The purpose of this phase is to keep track of the composite service execution by capturing the relevant generated events. The number of instances and the log structure should be sufficiently rich to enable CS mining.
- **Analysing the execution history** The purpose of this phase is to mine the effective control flow of a composite service. In particular, we proceed in two steps (see Fig. 2) to discover the CS model based on its log only. First, we perform statistical analyses to extract a statistical dependencies table. Second, a statistical specification of the control flow properties is extracted with discovery rules which are applied on the statistical dependencies tables.
- **Improving the composition model** Based on the execution history analysis and the mined results we enable a re-engineering phase to improve the composite service model.

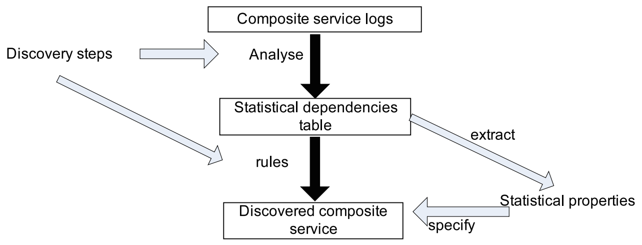

Fig. 2 Patterns mining steps

To describe our algorithm, Sect. 2 explains our CS log model. Section 3 details our structural control flow patterns mining algorithm. Validation and reengineering elements of the initial designed process model with the discovered process model are given in Sect. 4. Implementation aspects are discussed in Sect. 5. Finally, Sect. 6 discusses related work, before we conclude in the same section.

2 Composite service log

In the following, we are interested in CS log-related issues. First, we describe techniques to collect WS logs, followed by our CS log model in Sect. 2.2. Thereafter, we propose in Sect. 2.3 the log structure and the minimal conditions, which WS logs have to fulfill, to be able to apply our control flow mining algorithm.

2.1 Collecting Web service logs

Following a common requirement in the area of business process and service management, we expect composite services to be traceable, meaning that the system should in one way or another keep track of ongoing and past executions. Several research projects deal with the technical solutions necessary for collecting and logging of Web service's execution logs [1–3]. In the following, we examine the common logging possibilities in service-oriented architectures.

2.1.1 Traditional Web logging solutions

There are two main data sources for Web log collecting, corresponding to the client–server computing paradigm. The existing techniques are commonly achieved by enabling the respective Web server's logging facilities. There are many investigations and proposals on Web server log and associated analysis techniques. A survey on Web Usage Mining [4] describes the most well-known means of Web log collection. Basically, server logs are either stored in the *Common Log Format*[1] or the more recent *Combined Log Format*.[2] They primarily consist of various types of logs generated by the Web server. Most of the Web servers support as default option the *Common Log Format*, which is a fairly basic form of Web server logging.

However, the emerging paradigm of Web services requires richer information in order to fully capture business interactions. Since the Web server log is derived from requests resulting from users accessing pages, it is not tailored to capture service composition or orchestration. In the following, we describe a set of advanced logging techniques that allows to record the additional information to mine more advanced behaviour.

2.1.2 Process-based logging solutions

CS mining requires choreography or orchestration identifier and instance (case) identifier in the log record. Such information is not available in conventional Web server logs. In the following, we describe advanced solution to collect this information in choreography or orchestration execution.

A known method for debugging, is to insert logging statements into the source code of each service in order to call another service or component, responsible for logging. However, this solution has a main disadvantage: we do not have ownership over third partie's code and we cannot guarantee they are willing to change it on someone else, behalf. Furthermore, modifying existing applications may be time consuming and error prone (solution 1). Since all interactions between Web Services happen through the exchange of SOAP message, an other alternative is to use SOAP headers that can provide additional information on the message's content concerning the executed choreography. Basically, we modify SOAP headers to include and gather the additional needed information capturing choreography-ID and its instance-ID. We use SOAP intermediaries [5] which are applications, located between a client and a service provider. These intermediaries are capable of both receiving and forwarding SOAP messages. They are located on web services provider and they intercept SOAP request messages from a Web service sender or capture SOAP response messages from a Web service provider. On Web service client-side, this remote agent can be implemented to intercept those messages and extract the needed information. The implementation of client-side data collection methods requires user cooperation, either in enabling the functionality of the remote agent, or to voluntarily use and process the modified SOAP headers, but without changing the Web service implementation itself (the disadvantage of solution 1).

For orchestration log collecting, since most web service orchestration are using a WSBPEL engine, which coordinates the various orchestration's web services, interprets and executes the grammar describing the control logic, we can extend this engine with a sniffer that captures orchestration information, i.e., the orchestration-ID and its instance-ID. This solution is centralized, but less constrained than the previous one which collects choreography information.

Using these advanced logging facilities, we aim at taking into account web services' neighbors in the mining process. The term neighbors refers to other Web services that the examined Web Service interacts with. The concerned levels deal with mining web service choreography interface (abstract process) through which it communicates with other web services to accomplish a choreography, or discovering the set of interactions exchanged within the context of a given choreography or composition.

The exact structure of the web logs or the event collector depends on the used execution engine. In our experiments, we used the bpws4j[3] engine which itself uses log4j[4] to generate logging events. Log4j is an OpenSource logging API developed under the Jakarta Apache project. It provides a robust, reliable, fully configurable, easily extendible, and easy to implement framework for logging Java applications for debugging and monitoring purposes. The event collector (which is implemented as a remote log4j server) sets some log4j properties of the bpws4j engine to specify level of event reporting (INFO, DEBUG, etc.), and the destination details of the logged events. At runtime bpws4j generates events according to the log4j properties set by the event collector.

2.2 Composite Service log structure

Definition 1 defines our CS log model converted from the event collector described in the previous section to select only the required information. A CSLog is composed of a set

1 http://httpd.apache.org/docs/logs.html.

2 http://www.w3.org/TR/WD-logfile.html.

3 http://alphaworks.ibm.com/tech/bpws4j.

4 http://logging.apache.org/log4j.

Definition 1 (CSLog)
An event reports a service terminated state and its related execution time, and is defined as a tuple: Event= (serviceId, TimeStamp). An EventStream represents the history of a CS instance events as a tuple EventStream= (begin, end, sequenceLog, SOccurrence) where:
✓(begin:TimeStamp) and (end:TimeStamp) are the instance beginning and end time;
✓sequenceLog: Event*; is an ordered Event set reporting service executions;
✓SOccurrence: int; is the instance ID.
A CSLog is a set of EventStreams. CSLog=(ID, {EventStream_i, $0 \leq i <$ number of CS instances}) where EventStream_i is the event stream of the i^{th} CS instance.
Let T the set of services belonging to a CS. We note $\sigma \in \mathcal{T}^*$ a simplified EventStream format by omitting TimeStamp from Event as Events are ordered according to their occurrence time.

of EventStreams. Each EventStream traces the execution of one case (instance). It consists of a set of events (Event) that captures services execution. WS logging functionalities might collect external events that capture the service life cycle (such as activated, aborted, failed, and terminated). However, existing logging solutions can propose different levels of granularity and collect only one part of the set of event states. Moreover, the nomenclatures of these states are generally different from one system to another. As a solution, we can choose to filter them or/and designate them as a default state. In our case (i.e., we aim to only mine the control flow), it is more practical to simply consider ordered service atomic terminated state events, which omit execution times and other intermediate service states (simplified EventStream).

Although the combination of activated and terminated states can be very useful to detect concurrent behaviour implicitly from logs, our log structure reports only the event of successful termination to simplify and to minimize the constraints of log collecting. This "minimalist" feature enables us to exploit "poor" logs which contain only information concerning the successfully executed services sequence without collecting for example services execution intermediate states or execution occurrence times.

2.3 Sufficient and minimal number of CS instances

To enable correctly the mining process, the CS logs must be "complete" by respecting the *log completeness conditions* [6]. These conditions are depicted as follows:

- **Condition 1** if a service precedes another in the control flow then there should be one instance log, at least, reporting two respective related events keeping the same order. Particularly, if the execution of one service depends **directly** on the termination of another service, then the event related to the first service must *directly* (immediately, without intercalated events between them) follow at least once the event related to the second service in an instance log. For instance, in our motivating example the execution of S_8 directly depends on the termination of S_5. Thereafter, the related CS log in order to be complete should contain an EventStream where S_8 follows directly S_5 (as shown in instance 3 in Table 1).
- **Condition 2** To discover the parallel behaviour of two concurrent services with a lack of indication related to the services' begin and end execution time, we require that the events of the two parallel services should appear at least in two EventStreams without order of precedence. Basically, two parallel services must *directly* follow each other in two instances in different order to indicate that each service can finish its execution before the other. For instance, in our motivating example S_2 and S_4 are two parallel services. Thereafter, the related CS log in order to be complete should contain two EventStreams where S_4 follows S_2 in the first one, and S_2 follows S_4 in the second one (as shown in instances 4 and 5 in Table 1).

Lemma 1 (Number of instances for a complete log) *The sufficient number of different* EventStream*s to discover a CS's control flow is computed as follows:*

1. *The minimal number is equal to 1. For example, a CS containing only one sequential services flow without concurrent or conditional behaviour reports always the same sequence of services;*
2. *A conditional behaviour between n services before a "*join*" point or after a "*fork*" point requires $n-1$ different additional* EventStream*s.*
3. *A concurrent behaviour between n control flows, each flow i; $0 < i < n+1$ contains na_i; $na_i \leq na_{i+1}$ services, requires $(\Pi_{i=1..n}(na_i + i - 1)) - 1 = na_1 * (na_2 + 1) * (na_3 + 2) * (na_4 + 3) * \ldots.. * (na_n + n - 1) - 1$ different additional* EventStream*s.*

Based on these two conditions, we have specified "complete" log features describing the properties required by our control flow mining approach from logs. Concretely, we have specified "minimalist" conditions on a log structure and "sufficient" conditions on log quantity (i.e., number of logged instances) to be "complete". We deducted, for a given CS, the sufficient number of different instances logs for a "complete" CS log (Lemma 1). This lemma indicates for each behaviour, the necessary number of instance logs to enable our control flow mining approach. This feature defines a "local specification" on the sufficient number of instance logs for a "complete" CS log. Accordingly, we do not require to collect all possible instances logs to satisfy the *log completeness conditions*. For example, for a CS containing n concurrent services followed by m concurrent services, the number of possible scenarios (number of instances logs) is equal to $n! * m!$. But, by applying our lemma we need only $n! + m!$ instances logs.

Table 1 Six simplified EventStreams of our motivating example

Instance ID	EventStream
Instance1	$S_1\ S_2\ S_3\ S_4\ S_5\ S_7\ S_9$
Instance2	$S_1\ S_2\ S_3\ S_4\ S_5\ S_6\ S_9$
Instance3	$S_1\ S_2\ S_3\ S_4\ S_5\ S_8\ S_9$
Instance4	$S_1\ S_2\ S_4\ S_3\ S_5\ S_7\ S_9$
Instance5	$S_1\ S_4\ S_2\ S_3\ S_5\ S_6\ S_7\ S_9$
Instance6	$S_1\ S_4\ S_2\ S_3\ S_5\ S_7\ S_6\ S_9$

Proof (*Proof of Lemma 1*):

1. If CS is a simple sequence of services $\{S_i\ , 0 < i < n{+}1\}$ without concurrent or conditional behaviour then CS is a combination of *sequence* patterns. Consequently, only **Condition 1** is concerned.
   The EventStream "$S_1 S_2 S_3 \ldots S_n$" satisfies this point.
2. A conditional behaviour between n services $\{S_i\ , 0 < i < n + 1\}$ and a service B exists after a "fork" point (*OR-split* and *XOR-split* patterns) or before a "join" point (*OR-join* and *M-out-of-N* patterns). Consequently, only **Condition 1** is concerned. We are interested in the following on "join" patterns, the proof for the "fork" patterns can be done symmetrically.
   The n EventStreams "...$S_i B$...."; $0 < i < n + 1$ satisfy this point.
3. For the third point proof, firstly we are interested on the concurrent behaviour between n control flows each containing only one service. This behaviour is described by a set of $\{S_i\ , 0 < i < n + 1\}$. Consequently, only **Condition 2** is concerned.

The set of permutation of size n computed from $\{S_i\ ,\ 0 < i < n + 1\}$ is $n! = \Pi_{i=1\ldots n}(i)$ EventStream satisfies this condition.

We suppose now that the control flow i contains more that one service and $\{\ S_{k,j}\ , 0 < j < m + 1\ \}$ is the set of these services, then it is enough to compute permutations of size n from $\{\ S_{k,j}, S_i\ ; 0 < i < k, k < i < n + 1\ \}$ for each service $S_{k,j}$ creating $\Pi_{0<i<k,k<i<n+1}(i) * (k-1+m)$ EventStreams. By extending this reasoning to the other control flows we find the lemma's formula $(\Pi_{i=1..n}(na_i + i - 1)) - 1 = na_1 * (na_2 + 1) * (na_3 + 2) * (na_4 + 3) * \cdots * (na_n + n - 1) - 1$, where each flow i; $0 < i < n + 1$ contains na_i services. □

Table 2 represents the execution of six instances of our running example. This table contains the sufficient information which we assume to be present to correctly apply our control flow mining approach. Indeed, our CS example contains a conditional behaviour between three services (S_6, S_7, and S_8), which implies 3 EventStreams $1 + (3 - 1) = 3$ by applying the second point of our lemma and by adding the necessary instance to satisfy the first point. In addition, the

Table 2 Initial statistical dependencies table ($P(x/y)$) and service frequencies (#)

$P(x/y)$	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
S_1	0	0	0	0	0	0	0	0	0
S_2	**0.54**	0	0	$\underline{0.46}$	0	0	0	0	0
S_3	0	**0.69**	0	$\underline{0.31}$	0	0	0	0	0
S_4	**0.46**	$\underline{0.31}$	$\underline{0.23}$	0	0	0	0	0	0
S_5	0	0	**0.77**	**0.23**	0	0	0	0	0
S_6	0	0	0	0	1	0	0	0	0
S_7	0	0	0	0	1	0	0	0	0
S_8	0	0	0	0	0	0	0	0	0
S_9	0	0	0	0	0	0.38	0.62	0	0

$\#S_1 = \#S_2 = \#S_3 = \#S_4 = \#S_5 = \#S_9 = 100$,
$\#S_6 = 38, \#S_7 = 62, \#S_8 = 0$

two concurrent flows, containing, respectively, S_2, S_3, and S_4 imply $(1 * (2 + 1)) - 1 = 2$ additional EventStreams by applying the third point of our lemma. Finally, the parallel behaviour that can be observed between S_6 and S_7 if the user decides to pay using credit card and cash implies $(1 * (1 + 1)) - 1 = 1$ additional EventStreams. The total of sufficient EventStreams equals $3 + 2 + 1 = 6$, shown in Table 1.

In this table, the EventStreams 1 and 4 describe the case where a user chooses to pay by cash. Although these different EventStreams (1 and 4) describe the same scenario, the concurrent services execution scenario is not the same (i.e., services S_2, S_3 and S_3 do not have the same order). These different EventStreams, (in the same way for the EventStreams 5 and 6) allow to describe the various possible choices of the processing as well as the various possible combinations of concurrent services' execution in these choices.

3 Mining structural control flow patterns

The control flow (or skeleton) of a CS specifies the partial ordering of component services activations. We use (workflow-like) patterns to define a composite service skeleton. As defined in [7], a pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts". A workflow pattern [8] can be seen as an abstract description of a recurrent class of interactions based on (primitive) activation dependency. In the following, we describe a bottom-up approach, as illustrated in Fig. 3, to discover these patterns:

1. **Discovering activation dependencies** First, we specify dependencies linking the component services during execution. These dependencies are of two kinds: causal and non-causal. A causal dependency between two services expresses that the occurrence of a service event involves

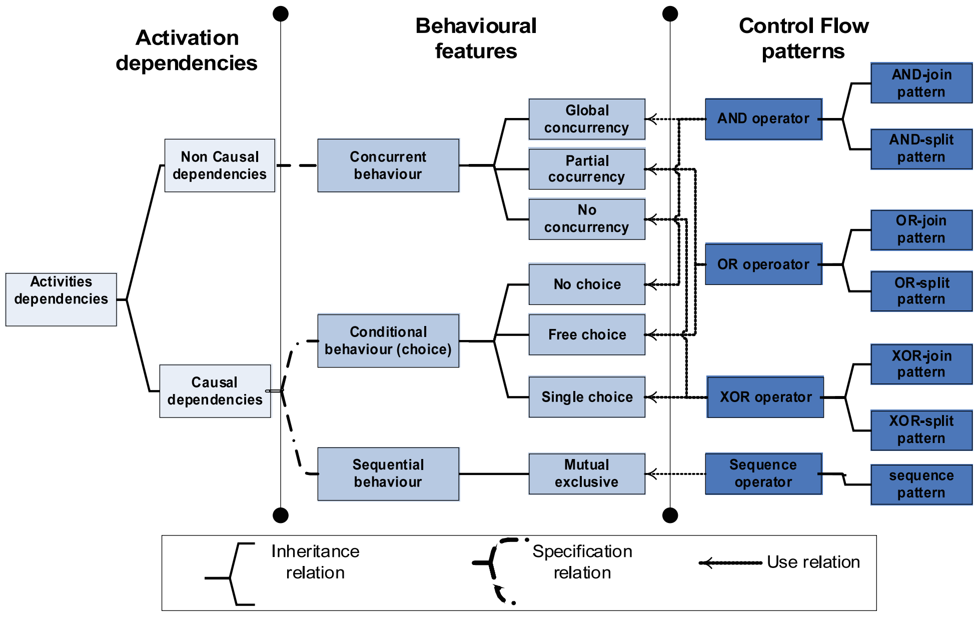

Fig. 3 Hierarchical view of our patterns mining approach

the activation of another service event. While a non-causal dependency specifies any other services behavioural dependency (concurrent behaviour, for instance).

2. **Computing statistical behavioral properties** Second, we compute the statistical behavioural properties from log. These properties tailor the main behaviour features of the discovered patterns. These properties are of three types: "sequential", "concurrent" and "choice". The "sequential" and "choice" properties inherit from causal dependency. The first expresses an exclusive causal dependency between two services. While the second specifies a causal dependency between a service on the one hand and one or many services belonging to a set of services, on the other hand. The "concurrent" property inherits from non-causal dependency and characterises the concurrent behaviour of a set of services.
3. **Discovering control flow patterns** Finally, we use a First of rules to discover a set of the most useful patterns. These rules are expressed using the statistical properties and could be expressed as a first order logic predicate, for instance. In this work, we have chosen to discover the most useful patterns: sequence, xor-split, and-split, or-split, xor-join , and-join and M-out-of-N-Join. However, the adopted approach allows to enrich this set of patterns by specifying new statistical dependencies and their associated properties by using the existing properties in new combinations.

It has to be noted that the only input of our mining approach is a CS log. In the following, we suppose, after sufficient execution cases, that the CS log collecting cannot report new different cases, and the collected log should be complete for the *discovered* CS. However, if this log is not complete for the initially designed ad-hoc CS model we cannot faithfully mine this model (see Sect. 4 for more details).

3.1 Discovering activation dependencies

The aim of this section is to explain our algorithm for discovering activation dependencies among a CSLog and build an intermediary model representing these dependencies: the statistical dependency table (SDT).

3.1.1 Discovering direct dependencies

A direct dependency is an "immediate" dependency linking two services in the sense that the termination of the first causes *directly* the activation of the last. Thus, the event of termination of the first service is considered as the pre-condition of the activation of the last and reciprocally the activation of the last is considered as a post-condition of the termination of the first service. In order to discover direct dependencies from a CSLog, we need an intermediary representation of this CSLog through a statistical analysis. We call this intermediary representation: SDT, which is based on a notion of frequency table [9].

Basically, SDT is built through statistical calculus that extracts event direct dependencies. For each service S, we extract from CSLog the following information in the statistical dependency table (SDT): (a) The overall occurrence number of this service (denoted $\#S$) and (b) The elementary dependencies to previous services S_i (denoted $P(S/S_i)$). The size of SDT is $n * n$, where n is the number of component services. The (m,n) table entry is the frequency of the nth service immediately preceding the mth service. Based on this, Algorithm 1 computes the "initial" SDT (Table 2) of

Algorithm 1 Computing initial SDT

```
1: procedure COMPUTINGSDT($SDT$, $MSDT$)
2: input: CSLog: CSLog
3: output: #: service occurrence table ; $SDT$: Statistical dependency
   table;
4: Variable: $stream_{size}$: int; $TDS_{size}$: int;
5: #: int[]; depFreq: int[][];                        ▷ initialized to 0;
6:
7:   for every stream: EventStream in CSLog do
8:     $stream_{size} \leftarrow$ stream.size();    ▷ size returns the number of
   services in a stream
9:     for int i=1; $i < stream_{size}$; i++; do
10:      #[stream.get(i)]++;  ▷ get returns the service whose index
11:      depFreq[stream.get(i)][stream.get(i-1)]++;
12:      is i
13:    end for
14:  end for
15:  $SDT_{size}$ = Size-tab(#); /*return the size of #*/
16:  for int j=0; $j < SDT_{size}$; j++; do
17:    for int k=0; $k < SDT_{size}$; k++; do
18:      $SDT$[j, k] $\leftarrow$ depFreq[j][k]/ #[j];
19:    end for
20:  end for
21: end procedure
```

our motivating example given in Fig. 1. For instance, in this table $P(S_3/S_2) = 0.69$ expresses that we have 69% of chance that S_2 occurs directly before S_3 in the log. This table was computed using 100 EventStreams captured after executing 100 instances (cases) of our motivating example.[5]

We demonstrated a correlation between the service activation dependencies and the log statistics expressed in SDT (see Theorem 1). Each dependency between two services is expressed by a positive value in the corresponding SDT entry. This expresses a relation of equivalence between the positive entries in SDT and the dependencies between the related services.

Theorem 1 (Correlation between SDT and services dependencies) *Let wft be a CS whose control flow is composed using the set of 7 described patterns and does not contain short loops. $\forall a, b \in wft$ where a precedes $b \Leftrightarrow P(b/a) > 0 \wedge P(a/b) = 0$*

Proof (*Proof of Theorem 1*): Proofing first right implication "$\Rightarrow$"

Let L the CSLog capturing a and b execution. By applying the *log completeness conditions* specified in section 2.3 we can deduce that:

$$\exists\sigma_1 = t_1t_2t_3\ldots t_{n-1} \in L \wedge \exists 0 < i < n, \quad |t_i = a,\ t_{i+1} = b$$

And through SDT building definition and based on instance log, we can deduce that:

$$a \prec b \Rightarrow P(b/a) > 0$$

Furthermore, supposing now (proof by contradiction) that $P(a/b) > 0$ this implies:

$$\exists\sigma_1 = t_1t_2t_3\ldots t_{n-1} \in L \wedge \exists 0 < i < n, \quad |t_i = b, \quad t_{i+1} = a$$

and as we have yet $P(b/a) > 0$ this implies:

$$\exists\sigma_1 = t_1t_2t_3\ldots t_{n-1} \in L \wedge \exists 0 < i < n, \quad |t_i = a, \quad t_{i+1} = b$$

However, this is meaningless because this case arises only in a short[6] loop or if we have concurrent services. Thus we have:

$$a \prec b \Rightarrow P(b/a) > 0 \wedge P(a/b) = 0$$

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

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

$$\begin{aligned}&\exists\sigma_1 = t_1t_2t_3\ldots t_{n-1} \in L \wedge \exists 0 < i < n, \quad |t_i = a,\ t_{i+1} = b\\ &\wedge\nexists\sigma_1 = t_1t_2t_3\ldots t_{n-1} \in L \wedge \exists 0 < i < n, \quad |t_i = a,\ t_{i+1} = b\end{aligned} \tag{1}$$

And as a and b belong to the set of the seven described patterns, two sub cases happen if they are causally independent:

1. The two services a and b belong to two different separated patterns. This is meaningless based on instance log (1) that shows that the two services happen one after the other.
2. The two services a and b are in concurrence. This is meaningless based on instance log (1) and the *log completeness conditions*.

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

$$a \prec b \Leftarrow P(b/a) > 0 \wedge P(a/b) = 0$$

□

But as it was calculated, SDT presents some problems to express "correctly" and "completely" service 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.

3.1.2 Discarding erroneous dependencies

If we assume that each EventStream from CSLog comes from a sequential (i.e., no concurrent behaviour) CS, a zero entry in SDT represents a causal independence and symmetrically a non-zero entry means a causal dependency relation (i.e., sequential or conditional relation). But, in case of

[5] Table 1 is not used to compute this table.

[6] Contain one or two services.

concurrent behaviour, as we can see in patterns (like and-split, and-join, etc.), the EventStreams may contain interleaved events sequences from concurrent threads. As consequence, some entries in initial SDT can indicate non-zero entries that do not correspond to causal dependencies. For instance, the EventStream 4 in Table 1 "suggests" erroneous causal dependencies between S_2 and S_4 in one side, and between S_4 and S_3 in another side. Indeed, S_2 comes immediately before S_4 and S_4 comes immediately before S_3 in this EventStream. These erroneous entries are reported by $P(S_4/S_2)$ and $P(S_3/S_4)$ in initial SDT which are different to zero. These entries are erroneous because there are no causal dependencies between these services as suggested (i.e., noisy SDT). Underlined values in Table 2 report this behaviour for other similar cases.

Algorithm 2 Marking concurrent services in SDT

```
1: procedure MARKINGSDT($SDT$, $MSDT$)
2: input: $SDT$ Statistical dependencies table
3: output: $MSDT$ Marked Statistical dependencies table
4: Variable: $MSDT_{size}$: int;
5:
6:    $MSDT \leftarrow SDT$;
7:    $MSDT_{size} \leftarrow$ 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 $\wedge$ $SDT$[j][i]>0 then
11:             MSDT[i][j] $\leftarrow$ -1;
12:             MSDT[j][i] $\leftarrow$ -1;
13:          end if
14:       end for
15:    end for
16: end procedure
```

Formally, based on the *log completeness conditions*, we can easily deduce that from a complete CS log that two services A and B are in concurrence iff $P(A/B)$ and $P(B/A)$ entries in SDT are non-zero entries in SDT. Based on this, we propose an algorithm to discover services 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 services dependencies by changing their values to (-1). For instance, we can deduce from Table 2 that S_2 and S_4 services are in concurrence (i.e., $P(S_2/S_4) \neq 0 \wedge P(S_4/S_2) \neq 0$), so after applying our algorithm $P(S_2/S_4)$ and $P(S_4/S_2)$ the result will be equal to -1 in the final table. The algorithm's output is an intermediary table that we called marked SDT (MSDT).

3.1.3 Discovering indirect dependencies

For concurrency reasons, a service might not depend on its immediate predecessor in the EventStream, but it might depend on another "indirectly" preceding services. As an example of this behaviour, S_4 is logged between S_2 and S_3 in the EventStream 4 in the Table 1. As consequence, S_2 does not occur always immediately before S_3 in the CS log. Thus, we have only $P(S_3/S_2) = 0.69$ that is an underestimated dependency frequency. In fact, the right value is 1 because the execution of S_3 depends exclusively on S_2. Similarly, values in bold in the initial SDT report this behaviour for other cases.

Definition 2 (Concurrent window) Formally, a concurrent window defines a triplet **window**(bWin, eWin, wLog, SerID) as a log slide over an EventStream S: EventStream (bStream, eStream, sLog, CSocc) where:

- bStream $\leq$ bWin $\wedge$ eWin $\leq$ eStream
- wLog $\subset$ sLog and $\forall$ e: **event** $\in$ S.sLog where bWin $\leq$ e.TimeStamp $\leq$ eWin $\Rightarrow$ e $\in$ wLog.

We define the function *width*(**window**) which returns the number of services in the **window**.

To discover these indirect dependencies, we introduce the notion of *concurrent window* (Definition 2) that defines a set of contiguous Event interval over an EventStream. A *concurrent window* (CW) is related to the service of its last event (SerID) covering its causal preceding services. Using this window, we will not only consider for a concurrent service the immediate previous event but also the previous events covered by the interval. For instance, in our motivating example S_3 and S_4 are two parallel services. Due to this concurrent behaviour, S_2 does not occur always immediately before S_3 in CS log because S_4 can be logged between S_2 and S_3. Therefore, the concurrent window of S_3, as shown in Fig. 4, covers S_2 in addition to S_4. Thus, S_2 is considered as a preceding service when SDT is computed.

Initially, the width of CW of a service is equal to 2. Each time the service is in concurrence with another service we add 1 to this width. If this service is not in concurrence with

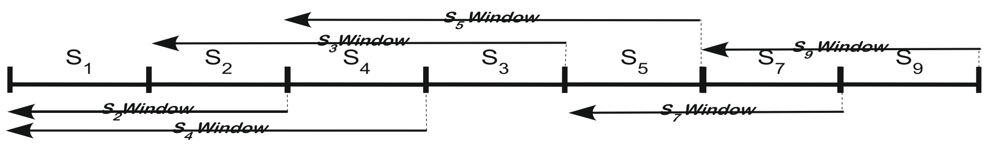

Fig. 4 EventStream partition

other services and has preceding concurrent services, then we add their number to CW width. For example, when S_3 is in concurrence with S_4 the width of its CW is equal to 3. Based on this, the Algorithm 3 computes the *concurrent window* of each service grouped in the CW table. This algorithm scans the "marked" SDT calculated in the last section and updates the CW table in consequence.

Algorithm 3 Calculating concurrent window size

```
procedure WINDOWWIDTH($MSDT$, $ACWT$)
input: $MSDT$: Marked Statistical dependencies table
output: $ACWT$: CW size table
Variable: $MSDT_{size}$: int;

    $MSDT_{size} \leftarrow$ Size-tab($MSDT$);        ▷ calculates MSDT size
    for int i=0; i< $MSDT_{size}$; i++; do
        ACWT[i]=2;
    end for
    for int i=0; i< $MSDT_{size}$; i++; do
        for int j=0; j < $MSDT_{size}$; j++; do
            if MSDT[i][j] =-1 then
                ACWT[i]++;
                ACWT[j]++;
            end if
            for int k=0; k< $MSDT_{size}$; k++; do
                if MSDT[k][i] >0 then
                    ACWT[k]++;
                end if
            end for
        end for
    end for
end procedure
```

After that, we proceed through an EventStream partition (Definition 3) that builds a set of partially overlapping windows over the EventStreams using the CW table. Definition 3 specifies that each window shares the set of its elements with the window which precedes it except the last event which contains the reference service of the window.

Definition 3 (Partition) A **Partition** builds a set of partially overlapping windows over an EventStream.
Partition: CSLog → (Window)*
S: EventStream(bStream, eStream, sLog, CSocc) → $\{w_i$:Window; $0 \le i < n\}$:

- w_1.bWin = bStream and w_n.eWin = eStream,
- $\forall w$: *window* $\in$ *partition*, e:Event= the last event in w, *width*(w)= ACWT[e.ServiceID],
- $\forall$ $0 \le i < n$; w_{i+1}.wLog - {the last event e:Event in w_{i+1}.wLog} $\subset$ w_i.wLog $\wedge$ w_{i+1}.wLog $\neq$ w_i.wLog.

In Figure 4 we have applied a partition over the EventStream of the running example presented in the EventStream 4 in Table 1. For example, the size of the CW of S_5 is equal to 3 because this service has two concurrent services S_3 and S_4 that precede it. We note that for each service in this EventStream its CW enables it to cover only all its causal preceding services.

Finally, Algorithm 4 computes the final SDT. For each *concurrent window*, it computes for its reference (last) service the frequencies of its preceding services. The final SDT will be found by dividing each row entry by the frequency of the row's service.

Algorithm 4 Calculating final SDT

```
procedure FINALSDT($Wlog$, #, $MSDT$)
input: Wlog: CSLog, #: Service Frequencies Table; $MSDT$: Marked Statistical Dependencies Table;
output: $FSDT$:Final Statistical Dependencies Table
Variable: $S_{reference}$: int; $S_{preceding}$: int; fWin: window; depFreq: int[][]; freq: int;

    $MSDT_{size} \leftarrow$ Size-tab($MSDT$);        ▷ returns MSDT size
    for all win:window in partition(Wlog) do
        $S_{reference}$ = last-service(win);      ▷ returns the last service's event
        fwin = preceding-events(win);      ▷ returns "win" without the last event
        for all e:event in fwin.wLog do
            $S_{preceding}$= e.serviceId;
            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
            $FSDT[t_{ref}, t_{pr}]$= depFreq[$t_{ref}$, $t_{pr}$]/#$t_{ref}$;
        end for
    end for
end procedure
```

Now by applying these algorithms, we can compute the final SDT (FSDT) which will be used in the next section to discover the patterns (Table 3). Note that, our approach

Table 3 Final statistical dependencies table (FSDT)

$P(x/y)$	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
S_1	0	0	0		0	0	0	0	0
S_2	**1**	0	0	$\underline{-1}$	0	0	0	0	0
S_3	0	**1**	0	$\underline{-1}$	0	0	0	0	0
S_4	**1**	$\underline{-1}$	$\underline{-1}$	0	0	0	0	0	0
S_5	0	0	**1**	**1**	0	0	0	0	0
S_6	0	0	0	0	1	0	0	0	0
S_7	0	0	0	0	1	0	0	0	0
S_8	0	0	0	0	0	0	0	0	0
S_9	0	0	0	0	0	0.38	0.62	0	0

$\#S_1 = \#S_2 = \#S_3 = \#S_4 = \#S_5 = \#S_9 = 100$,
$\#S_6 = 38, \#S_7 = 62, \#S_8 = 0$

adjust **dynamically**, through the width of CW, the process of calculating the services' dependencies. Indeed, this width is meaningfull to concurrent behaviour: it increases in case of concurrence and is "neutral" (equal to 2) in case absence of concurrent behaviour. Thus, our algorithm adapts its behaviour to the "concurrent" context. This strategy allows the improvement of the algorithm's complexity and runtime execution comparing to an analog pattern's discovery [10] which uses an invariable concurrent window width. Indeed, the use of an invariable width could apply a width superior to 2 for nonconcurrent services or simply a non-optimal width and then involve unnecessary computations increasing simply the algorithm's complexity.

3.2 Control flow statistical properties

We have identified three kinds of behaviour: sequential exclusive, conditional, and concurrent, which specify the patterns that we aim to discover. We have described these behavioural features by statistical properties using SDT (see Fig. 3). We use these properties to separately identify patterns from CS logs. These properties bind a correlation link between log statistics represented in the SDT and patterns' main behaviour using a set of corollaries deduced from Theorem 1.

We begin with the statistical exclusive dependency property (Corollary 1) which characterises a single sequential flow. The behaviour of the *mutual* exclusive dependency between two services specifies that the execution of one of the two services depends only on the end of the execution of the other, and the end of the execution of the first service starts only the execution of the second.

Corollary 1 **(*P1*: Mutually exclusive dependency property)** *Let S_i and S_j be two services. S_i and S_j describe a* mutually *exclusive dependency property* (P1) *from S_i to S_j iff in terms of:*

- *services frequencies: $\#S_i = \#S_j$*
- *services dependencies : $P(S_i/S_j) = 1 \wedge \forall(0 \le k, l < n;\ k \ne j; l \ne i;\ P(S_i/S_k) = 0 \wedge P(S_l/S_j) = 0)$.*

The parallel behaviour (Corollary 2) inherits from a non-causal relation. It specifies, in terms of concurrence, the execution of a set of services. This set of services is located after a "fork" or before a "join" operator. We distinguish three types of parallel behaviour:

1. **P2.1: Global concurrency** where in the same instantiation the services are performed simultaneously;
2. **P2.2: Partial concurrency** where in the same instantiation we have at least a partial concurrent execution between the services;
3. **P2.3: No concurrency** where there is no concurrency between the services.

Corollary 2 **(*P2*: Concurrency property)**
Let $\{S_i, 0 \le i < n\}$ a set of services that describes a:

1. ***P2.1: Global concurrency*** *property iff $\forall i, j; 0 \le i < j < n;$ $\#S_i = \#S_j \wedge P(S_i/S_j) = -1$*
2. ***P2.2: Partial concurrency*** *property iff $\exists i, j; 0 \le i < j < n;$ $P(S_i/S_j) = -1$*
3. ***P2.3: No concurrency*** *property iff $\forall i, j; 0 \le i < j < n; \wedge P(S_i/S_j) \ne -1$*

Corollary 3 **(*P3*: Choice property)** *Let A be a service and $\{S_i, 0 \le i < n\}$ a group of services forming the operands of a "*`fork`*" operator or a "*`join`*" operator. A and $\{S_i, 0 \le i < n\}$ describe a:*

- ***P3.1: Free choice*** *property iff in terms of services frequencies we have $(\#A \le \Sigma_{i=0}^{n-1}(\#S_i)) \wedge (\#S_i \le \#A)$ and in terms of services dependencies we have:*
  - *In "*`fork`*" operator : $\forall 0 \le i < n; P(S_i/A) = 1$ (S_i occurs certainly after occurrence A)*
  - *In "*`join`*" operator : $1 < \Sigma_{i=0}^{n-1} P(A/S_i) < n$ (A occurs certainly after some S_i occurrences "$1 <$", but not always after all S_i "$< n$")*
- ***P3.2: Single choice*** *property iff in terms of services frequencies we have $(\#A = \Sigma_{i=0}^{n-1}(\#S_i)))$ and in terms of services dependencies we have:*
  (a) *In "*`fork`*" operator : $\forall 0 \le i < n; P(S_i/A) = 1$ (S_i occurs certainly after A occurrence)*
  (b) *In "*`join`*" operator : $\Sigma_{i=0}^{n-1} P(A/S_i) = 1$ (A occurs certainly after only one of S_i occurrences)*
- ***P3.3: No choice*** *property iff in terms of services frequencies we have $\forall 0 \le i < n, \#A = \#S_i$ and in terms of services dependencies we have:*
  - *In "*`fork`*" operator : $\forall 0 \le i < n; P(S_i/A) = 1$ (S_i occurs certainly after A occurrence)*
  - *In "*`join`*" operator : $\forall 0 \le i < n; P(A/S_i) = 1$ (A occurs certainly after all S_i occurrences)*

The conditional behaviour (Corollary 3) specifies how the activation choice is carried out among a group of services after a "fork" operator or before a "join" operator. It defines a causal relation between a service and a group of services forming the operands of the "fork" operator or the "join" operator. We distinguish three types of conditional behaviour:

- **P3.1: Free choice** where a part of the group of services is executed according to the constraints and parameters of each instantiation;
- **P3.2: Single choice** where only one service is executed. The choice of this service depends on the constraints and parameters on the executed instance;
- **P3.3: No choice** where all services are executed for each instantiation.

3.3 Patterns discovering rules

Using the previous statistical properties, the last step is the patterns discovery through a set of rules. Our approach

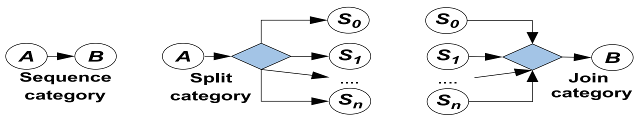

Fig. 5 Patterns categories

provides a dynamic algorithm that builds iteratively a global solution (i.e., global CS) based on local solutions (i.e., CS patterns). Each pattern has its own statistical rules which abstract statistically its causal and non-causal dependencies, and identifies it in an unique manner. Our control flow mining rules are characterised by a "local" patterns discovery. Indeed, these rules proceed through a **local log analysis** that allows to **recover partial results** of mining patterns. To discover a particular pattern we only need events relating to pattern's elements. Thus, even using only fractions of CS logs, we can discover correctly corresponding patterns (with their events belonging to these fractions).

We divided the CSs patterns in three categories (cf. Fig. 5) : sequence, split and join patterns. In the following, we present rules to discover the most useful patterns belonging to these three categories.

3.3.1 Discovering sequence pattern

In this category, we only find the sequence pattern (Table 4). In this pattern, the enactment of B depends only on the completion of service A. By using the statistically exclusive dependency property (Corollary 1) we can ensure this relation linking B to A.

For instance, by applying the rules on this pattern over Table 3, we discover a sequence pattern linking S_2 and S_3. Indeed, $(\#S_2 = \#S_3)$ and $(P(S_2/S_3) = 1)$ and $\forall S_{0 \le i < n} \neq S_2;\ P(S_3/S_i) \le 0$ and $\forall S_{0 \le j < n} \neq S_2;\ P(S_j/S_3) \le 0$.

3.3.2 Discovering split *patterns*

This category (Table 5) has a "fork" operator where a single thread of control splits into multiple threads of control which can be, according to the used pattern, either executed or not. The dependency between services A and $\{S_i; 0 \le i \le n\}$ before and after "fork" operator differs in the three patterns of this category: xor-split, and-split, and or-split. These dependencies are characterised by the statistical choice properties (Corollary 3). The xor-split pattern, where one of the flows after the "fork" operator is chosen, adopts the single choice property (P3.2). And-split and xor-split patterns differentiate themselves through the no choice (P3.3) and free choice (P3.1) properties. Ultimately, only a part of the services is executed in an or-split pattern after "fork" operator, while all S_i are executed in and-split patterns. The non-parallelism between S_i, in xor-split patterns is ensured by the no concurrency property P2.3, while the partial and the global parallelism in or-split and and-split patterns is identified through the application of the statistical partial and global concurrency properties P2.1 and P2.2. For instance, Table 3 indicates that we have an and-split pattern linking S_1, S_2 and S_4. In fact, there is a global parallelism between S_2 and S_4 $((P(S_4/S_2) = -1 \wedge P(S_2/S_4) = -1))$ and these services depend exclusively on S_1 $((P(S_4/S_1) = 1 \wedge P(S_2/S_1) = 1))$.

3.3.3 Discovering join *patterns*

This category (Table 5) has a "join" operator where multiple threads of control merge in a single thread of control. The

Table 4 Rules of sequence pattern

Sequence	Rules
	$(\#B = \#A) \wedge$
	$(P(B/A) = 1) \wedge \forall S_{0 \le i < n} \neq A;\ P(B/S_i) \le 0 \wedge \forall S_{0 \le j < n} \neq B;\ P(S_j/A) \le 0$

Table 5 Rules of split and join patterns

Split	Rules	Join	Rules
	$(\Sigma_{i=0}^{n-1}\ (\#S_i)=\#A) \wedge$		$(\Sigma_{i=0}^{n-1}\ (\#S_i)=\#B) \wedge$
(xor)	$(\forall 0 \le i < n;\ P(S_i/A) = 1) \wedge$	(xor)	$(\Sigma_{i=0}^{n-1}\ P(B/S_i)=1) \wedge$
	$(\forall 0 \le i \neq j < n;\ P(S_i/S_j) = 0)$		$\forall 0 \le i \neq j < n;\ P(S_i/S_j) = 0$
	$((\forall 0 \le i < n;\ \#S_i=\#A) \wedge$		$(\forall 0 \le i < n;\ \#S_i=\#B) \wedge$
(and)	$(\forall 0 \le i < n;\ P(S_i/A) = 1) \wedge$	(and)	$(\forall 0 \le i < n;\ P(B/S_i) = 1) \wedge$
	$(\forall 0 \le i \neq j < n\ P(S_i/S_j) = -1)$		$(\forall 0 \le i \neq j < n\ P(S_i/S_j) = -1)$
	$(\#A \le \Sigma_{i=0}^{n-1}\ (\#S_i)) \wedge$		$(m * \#B \le \Sigma_{i=0}^{n-1}\ (\#S_i))$
(or)	$(\forall 0 \le i < n;\ \#S_i \le \#A)$	(M-out	$\wedge\ (\forall 0 \le i < n;\ \#S_i \le \#B)$
	$(\forall 0 \le i < n;\ P(S_i/A) = 1) \wedge$	-of-N)	$(m \le \Sigma_{i=0}^{n-1}\ P(B/S_i) \le n)$
	$(\exists 0 \le i \neq j < n;\ P(S_i/S_j) = -1)$		$\wedge\ (\exists 0 \le i \neq j < n;\ P(S_i/S_j) = -1)$

Table 6 Rewriting rules defining a coherent pattern composition grammar

RR1	$sequence(a, b),\{p_i\} \longrightarrow \mathcal{A}(a, b),\{p_i\}$
RR2	$and\text{-}split(a, b_1, b_2, ..., b_n),\{p_i\} \longrightarrow \mathcal{B}\ (a, b_1, b_2, ..., b_n),\{p_i\}$
RR3	$or\text{-}split(a, b_1, b_2, ..., b_n),\{p_i\} \longrightarrow \mathcal{C}\ (a, b_1, b_2, ..., b_n),\{p_i\}$
RR4	$xor\text{-}split(a, b_1, b_2, ..., b_n),\{p_i\} \longrightarrow \mathcal{D}\ (a, b_1, b_2, ..., b_n),\{p_i\}$
RR5	$and\text{-}join(a_1, a_2, ..., a_n, b),\{p_i\} \longrightarrow \mathcal{E}\ (a_1, a_2, ..., a_n, b),\{p_i\}$
RR6	$M\text{-}out\text{-}of\text{-}N(a_1, a_2, ..., a_n, b),\{p_i\} \longrightarrow \mathcal{F}\ \{a_1, a_2, ..., a_n, b),\{p_i\}$
RR7	$xor\text{-}join(a_1, a_2, ..., a_n, b),\{p_i\} \longrightarrow \mathcal{G}\ (a_1, a_2, ..., a_n, b),\{p_i\}$
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	$\mathcal{B}\ (a, b_1, b_2, ..., b_n),\ \mathcal{F}\ (b_1, b_2, ..., b_n, c),\ \{p_i\} \longrightarrow \mathcal{A}(a, c),\ \{p_i\}$
RR15	$\mathcal{B}\ (a, b_1, b_2, ..., b_n),\ \mathcal{G}\ (b_1, b_2, ..., b_n, c),\ \{p_i\} \longrightarrow \mathcal{A}(a, c),\ \{p_i\}$
RR16	$\mathcal{C}\ (a, b_1, b_2, ..., b_n),\ \mathcal{F}\ (b_1, b_2, ..., b_n, c),\ \{p_i\} \longrightarrow \mathcal{A}(a, c),\ \{p_i\}$
RR17	$\mathcal{C}\ (a, b_1, b_2, ..., b_n),\ \mathcal{G}\ (b_1, b_2, ..., b_n, c),\ \{p_i\} \longrightarrow \mathcal{A}(a, c),\ \{p_i\}$
RR18	$\mathcal{D}\ (a, b_1, b_2, ..., b_n),\ \mathcal{G}\ (b_1, b_2, ..., b_n, c),\ \{p_i\} \longrightarrow \mathcal{A}(a, c),\ \{p_i\}$
RR19	$\mathcal{A}(a, b),\ \varepsilon \longrightarrow \mathcal{W}orkflow$

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

number of necessary branches for the activation of service B after the "join" operator depends on the used pattern. To identify the three patterns of this category: xor-join, and-join and M-out-of-N-Join, we have analysed dependencies between S_i and B before and after the "join" operator. Thus, the single choice (P3.2) and the no concurrency (P2.3) properties are used to identify xor-join pattern where two or more alternative branches come together without synchronisation and none of the alternative branches is ever executed in parallel. As for and-join pattern where multiple parallel services converge into one single thread of control, the no choice (P3.3) and the global concurrency (P2.3) are both used to discover this pattern. Contrary to the M-out-of-N-Join pattern, where we need only the termination of M services from the incoming n parallel paths to enact B, the concurrency between S_i is partial (P2.2) and the choice is free (P3.1). For instance, using a FSDT table we mine an xor-join pattern linking S_6, S_7 and S_9. In fact, FSDT's entries of these services indicate a non-concurrent behaviour between S_6 and S_7 ($P(S_6/S_7) = P(S_6/S_7) \neq -1$) and the execution of S_9 depends on the termination of S_6 or S_7 ($P(S_9/S_6)$+$P(S_9/S_7)$= 1).

3.4 Coherent composition of the discovered patterns

The construction of the CS complete graph is made by linking one by one the discovered patterns. Indeed, in our approach we define the control flow as a union of patterns. We use rewriting rules (illustrated in Table 6) to bind the discovered patterns (terminals). We consider a composite service as a word that has patterns as terminals (laterals). These terminals can be associative and commutative in the word constituting the composite service when applying the rewriting rules.

Discovering a pattern-oriented model ensures a sound and well-formed mined CS model. Therefore, by using this kind of model, we are sure that the discovered CS 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 CS 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 not have senseless unions (disjoined patterns) or incoherent flows (deadlocks, liveness, etc.), the grammar of this rewriting rules system 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 RR19 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, RR17, RR18, RR19).
- 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, M-out-of-N, xor-join or sequence (rewriting rules RR10, RR13, RR14, RR15).

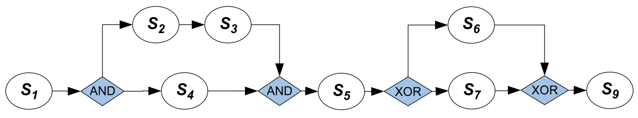

Fig. 6 CS discovered example

- An or-split pattern should be followed by one of these patterns M-out-of-N, xor-join or sequence (rewriting rules RR10, RR16, RR17).
- An xor-split pattern should only be followed by an xor-join pattern or a sequence pattern (rewriting rules RR10, RR18).

By applying the discovering rules (Tables 4, 5) over the final SDT (Table 3) we discovered the composite service illustrated in Fig. 6. We built the control flow as a pattern composition over this pattern word:

and-split(S_1,S_2,S_4), sequence(S_2, S_3), and-join(S_3,S_4,S_5), xor-split(S_5, S_6, S_7), xor-join(S_6, S_7, S_9).

Concretely, by applying the rewriting rules (Table 6) 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 composite service:

and-split(S_1,S_2,S_4), sequence(S_2, S_3), and-join(S_3, S_4,S_5), xor-split(S_5, S_6, S_7), xor-join(S_6, S_7, S_9).
$\longrightarrow_{RR1,RR2,RR3,RR4,RR5,RR6,RR7}$ $\mathcal{B}(S_1,S_2,S_4)$, $\mathcal{D}(S_5, S_6, S_7)$, $\mathcal{G}(S_6, S_7, S_9)$, $\mathcal{A}(S_2, S_3)$, $\mathcal{E}(S_3,S_4,S_5)$ $\longrightarrow_{RR11}$ $\mathcal{B}(S_1,S_2, S_4)$, $\mathcal{D}(S_5, S_6, S_7)$, $\mathcal{G}(S_6, S_7, S_9)$, $\mathcal{E}(S_2, S_4, S_5)$ $\longrightarrow_{RR18}$ $\mathcal{B}(S_1, S_2, S_4)$, $\mathcal{A}(S_5, S_9)$, $\mathcal{E}(S_2, S_4, S_5)$ $\longrightarrow_{RR13}$ $\mathcal{A}(S_1,S_5)$, $\mathcal{A}(S_5, S_9)$ $\longrightarrow_{RR8}$ $\mathcal{A}(S_1,S_9)$ $\longrightarrow_{RR19}$ $Workflow$

4 Composite service validation and re-engineering

There are two important process validation questions [11] : (1) *"Does our model reflect what we actually do ?"* and (2) *"Do we follow our model" ?*. Within the context of a business process re-engineering, we address question (2) to propose a set of improvement and correction tools based on the CS discovery results. We are interested, in particular, on the correction and the improvement of the CS's control flow. The goal is to provide an assistance tool to correct the CS design by applying semantic (adding services, suppressing services, and/or modifying pattern type) corrections. Our aim is to be close to the business and conceptual choices of CS designers and the evolution needs of CS users expressed during runtime and reported by the CS discovery results.

We will use the process discovery for a delta analysis (cf. Sect. 4.1), i.e., to compare the real operational process, represented by the discovered CS, with the initially designed CS model (for example, an ad-hoc composite web service orchestration). By comparing the initial CS with the discovered CS, the discrepancies between the two models can be

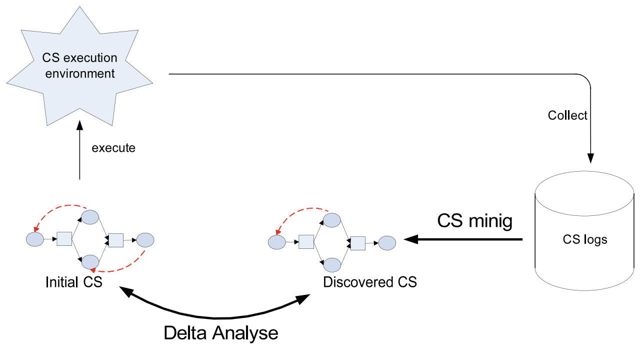

Fig. 7 Delta analyse for CS re-engineering

detected and used to improve, in particular, the control flow. With this intention, we propose, thereafter, a set of actions which allow to correct or to remove, if necessary, any erroneous or useless flow and optimize the process execution (c.f., Sect. 4.2). By erroneous or useless behaviour, we mean any initially designed flow which is not necessary or which does not coincide with the execution reality expressing new users' needs or conceptual choice errors made in the initial design phase.

4.1 Delta analysis

Concretely, we can use the CS discovery for Delta analysis, i.e., compare the "real" operational process, represented by the discovered CS, with the initially designed CS (for example, an ad-hoc composite web service orchestration). By comparing the initial CS with the discovered CS, we aim to detect discrepancies between the two models. Indeed, at run time users can deviate from the initially designed CS. Delta Analysis (Fig. 7) between the initially designed and the discovered CS allow to monitor these deviations.

Delta analysis uses comparison techniques to compare between the two models. Although the comparison techniques of process models do not constitute the core of our work but rather a tool, we present, in the following, an overview of the existing solutions. Judging by the great number of equivalence concepts [12] this task is far from being insignificant. Most of business process comparison approach propose a node-mapping technique rather than behaviour-mapping technique, i.e., the interest goes on the syntactic differences rather than on the semantic differences.

However, from a theoretical point of view, there are at least two approaches which also include a behavioural comparison. The first approach defined in [13,14] uses the "behaviour legacy". The second one is based on the process "changing regions". Based on the "behaviour legacy" concept, Van der Aalst et al. developed the concept of the largest common divider and the smaller common multiple of two processes [14] as comparison tool. While the "changing regions"

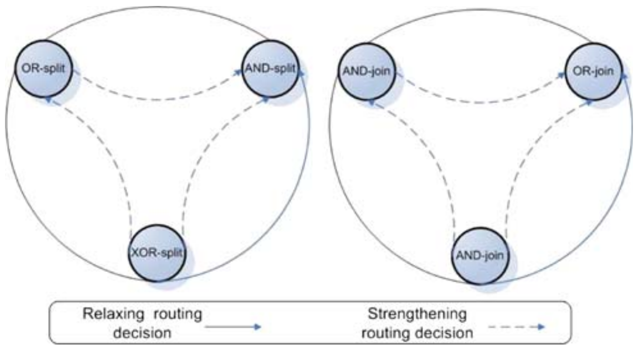

Fig. 8 Operators evolution

processing [15,16] is obtained by comparing the two processe's models and by extending the areas which were changed directly by the parts of the processes which are also affected by the change coming from the other process, i.e., the syntactically affected parts are extended with the semantically affected parts to produce the "changing regions".

4.2 Improving and correcting the control flow

Independently of the chosen comparison technique, a delta analysis process aims to detect the discrepancies between the discovered and the initial models. These discrepancies express the possible process model deviations. The analysis of these deviations is fundamental for a new re-engineering phase. Indeed, deviations 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 fact, some of these deviations become a current practice rather than to be a rare exception. In the following, we propose a set of tools to correct and improve the control flow according to option 2.

Thus, the discrepancies detected thanks to the delta analysis are used to improve, correct or remove, if necessary, all "erroneous or useless designs". By "erroneous or senseless designs", we mean an initially designed flow which is not necessary or that does not coincide with the reality of execution and express evolution needs from errors made at the initial design phase. In this case, the accuracy and the reliability of the initially designed process are uncertain and a re-engineering phase based on the discrepancies between the two models is required.

The correction and improvement actions related to the re-engineering phase depend on the discovered discrepancies. These actions should respect designers' and users' business needs. We distinguish between two kinds of discrepancies related to the flows and operators nature which allow:

- to suppress erroneous flows containing useless services. These useless services are not reported in logs and their related rows and columns are empty in SDT. Thereafter, the discovered CS does not contain these services. Keeping these services in the CS can not only be merely expensive but also can be a source of errors. For instance, by comparing the initial CS (Fig. 1) with the discovered CS (Fig. 6), S_8 does not exist any more in the discovered CS. This indicates that the payment by check is never executed and can be removed from the initial CS model because it does not represent a "used" payment choice (or alternative). Indeed, the flow containing this service is not necessary for an optimal CS processing and does not coincide with the reality of execution and its maintenance can cause a additional costs.
- to correct or improve the operators nature expressing routing decisions. Indeed, you can detect discrepancies between the operator (xor, and, or) in the discovered and initially designed CSs. These discrepancies can express either a relaxation or a strengthener of the parameters related to the routing decisions specifying the choice performed over the set of services after the operator in split patterns or before the operator in join patterns (see Fig. 8). The routing decision relaxation (for instance, from and-join to or-join) expresses that the executed services will be wider after the operator in the split pattern and limited before the operator in the join pattern than in the initially designed CS model. This could be a result of relaxed decision constraints due to the needs of new users that require to remove these constraints in a flexible and dynamic process execution.
  For instance, we discover a xor-split pattern linking S_5, S_6 and S_7 (Fig. 6), instead of an or-split in the initially designed CS (Fig. 1). This discrepancy indicates that the users do not combine the payment by cash and the payment by credit card and use exclusively one of them. This restriction of choice is induced by specific user's evolution needs expressed through the CS execution and captured by the CS logs that we use to discover the "real" behaviour. The transformation of the or-split pattern in the initially designed CS to the xor-split pattern yields a more efficient and accurate CS that reflects the users' behaviour and avoids implementing unnecessary payment means (the combination of the payment by credit card and cash) that can imply additional operational costs.

5 Implementation

Our approach, thought initially to discover process structure from their linear event streams, have been implemented within our prototype WorkflowMiner.[7] Since the single

[7] WorkflowMiner demonstration can be downloaded at http://workflowminer.drivehq.com/workflowminer.avi.

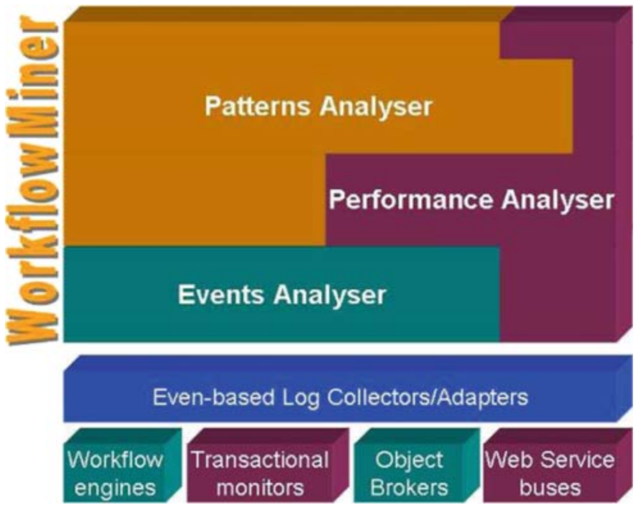

Fig. 9 WorkflowMiner applicative and technical Architecture

assumption of our approach is *the availability of execution logs* containing minimalist information (processes identifiers, process instances identifiers, services identifies, and time stamp for service instance event termination), application to web service re-engineering is only conditioned by the collection of web service execution logs.

Figure 9 shows the general architecture of WorkflowMiner upon four main components: (1) *Event-based Log Collectors/Adapters*, (2) *Events Analyser*, (3) *Pattern Analyser*, and (3) *Performance Analyser*. The WorkflowMiner techniques inherit from first order logic predicate-based reasoning, multidimensional database-based business intelligence, and rich visual reporting. The WorkflowMiner components are built on a panel of libraries and packages which the authors have either developed or integrated into WorkflowMiner. The Data flow between WorkflowMiner components is described in the Fig. 10. Starting from executions of composite web services, (1) event streams are gathered into an XML log. In order to be processed, (2) these log events are wrapped into a first order logic format, compliant with Definition 1. (3) Mining rules are applied on resulting first order log events to discover structural patterns. We use a Prolog-based presentation for log events, and mining rules. (4) Discovered patterns are given to a web service designer to assist him/her in the analysis of the web service to restructure or redesign them either manually or semi-automatically.

Web service execution log collectors/adapters Once intercepted and logged, web service interaction events (textual log lines, exchanged network messages), need to be adapted to be homogeneous, and usable by WorkflowMiner. Event adapters translate, in an Event-Transform-Load (ETL) style, those non-structured Web service events into a WorkflowMiner compliant XML structures, and then into first order logic predicate form. WorkflowMiner event-based log collectors/adapters are developed using java xml parser, ad-hoc

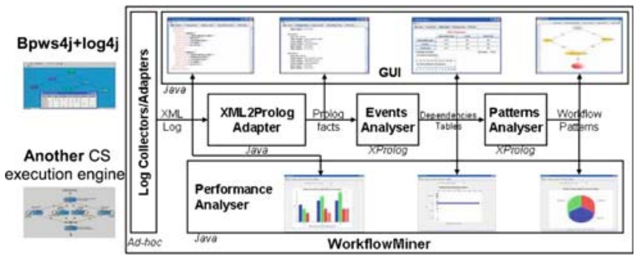

Fig. 10 WorkflowMiner Pipes and Filters Data Flow

adapters, and XProlog.[8] WorkflowMiner inputs are now ready to be processed within events, pattern, and performance analysers respecting the single assumption mentioned above.

Web service event analyser Through statistical techniques developed in this paper, *final statistical dependencies* are inferred iteratively over event-based log. Accordingly, the WorkflowMiner events analyser discovers basic web service interaction protocol is based on intercepted web service interaction events. The WorkflowMiner events analyser is developed using java xml parsers, and XProlog.

Web service patterns analyser First order logic predicates rules are used to discover a set of the most useful patterns which are divided into three categories: sequence patterns, split patterns (xor-split, and-split, or-split patterns) and join patterns (xor-join, and-join and M-out-of-N-Join patterns). Each structural pattern is expressed using statistical properties over the FSDT. As such, the WorkflowMiner pattern analyser discovers advanced web service interaction protocol that refines the basic web service interaction protocol. This advanced web service interaction protocol can be serialised to a linear execution language (e.g., BPEL, BPML, etc.) replacing the ad-hoc composite web service orchestration. WorkflowMiner pattern analyser is developed using XProlog, and JGraph.[9]

Web service performance analyser WorkflowMiner *Performance Analyser* uses adapted event-based logs and the discovered causal dependencies and structural patterns to measure composite web service performance metrics (aka key performance indicators, KPI). The theoretical, and experimental discussion of web service KPI is out of the scope of this paper.

In addition to WorkflowMiner, our approach has been implemented within the ProM framework [17], as a plug-in [18]. ProM is a "plug-in" environment for process mining. The ProM framework is flexible with respect to the input and output formats, and is also open enough to allow for the easy reuse of code during the implementation of new process mining ideas. This plug-in [18] helps us to provide detailed comparison [19] of our approach to other implemented

[8] http://www.iro.umontreal.ca/~vaucher/XProlog/.

[9] http://www.jgraph.com/.

Table 7 Comparing process mining tools

Process mining tools	EMiT [20]	Little thumb [21]	InWoLvE [22]	Process miner [23]	WorkflowMiner [24]
Structure	Graph	Graph	Graph	Block	Patterns
Local discovery	No	No	No	No	Yes
Parallelism	Yes	Yes	Yes	Yes	Yes
Non-free choice	No	No	No	No	Yes
Basic loops	Yes	Yes	Yes	Yes	Yes
Short loops	Yes	Yes	No	No	No
Noise	No	Yes	Yes	No	No
Time	Yes	No	No	No	No

mining tools. Table 7 compares our WorkflowMiner prototype to workflow mining tools representing previous approaches. Some of these approaches are also implemented in ProM. There are others process mining tools (concerning for instance, Social network mining, workflow analysis frameworks), but they are out of the scope of this paper. We focus on seven aspects: **structure** of the target discovering language, **local discovery** dealing with incomplete parts of logs (opposed to global and complete log analysis), **parallelism** (a fork path beginning with and-split and ending with and-join), **non-free choice** (NFC processes mix synchronisation and choice in one construct), **loops** (basics cyclic workflow transitions, or paths), **short loops** (mono- or bi- activity(ies) loops), **noise** (situation where log is incomplete or contains errors or non-representative exceptional instances), and **time** (event time stamp information used to calculate performance indicators such as waiting/synchronisation times, flow times, load/utilisation rate, etc.).

WorkflowMiner can be distinguished by supporting local discovery through a set of control flow mining rules that are characterised by a "local" pattern discovery enabling partial results. It recovers partial results from log fractions. Moreover, even if the non-free choice (NFC) construct is mentioned as an example of a pattern that is difficult to mine, WorkflowMiner discovers M-out-of-N-Join pattern which can be seen as a generalisation of the useful Discriminator pattern that was proven to be inherently non free-choice. Recently [37,41] propose a complete solution that can deal with such constructs. In previous process mining works, the discovery of short loops and the handling of log noise are generally done in a separate log pre-processing step. We choose not to include it in this stage of our approach in order to not reduce our algorithm efficiency. We currently research within a more original method and use less heavy approach to answer to these two points.

6 Discussion

Generally, previous formal approaches developed based on their respective modelling formalisms, a set of techniques to analyse the composition model and check its properties. Bultan et al. [25] proposes a formal framework for modelling, specifying and analysing the global behaviour of Web services compositions. This approach models web services by mealy machines (finite state machines with input an output). Based on this formal framework, the authors illustrate the unexpected nature of the interplay between local and global composite Web services. Hamadi and Benatallah [26] propose a Petri net-based algebra for composing Web services. This formal model allows the verification of properties and the detection of inconsistencies both between and within services. Although powerful, the above formal approaches may fail, in some cases, to ensure anoptimum CS model even if they formally validate their composition models. This is because properties specified in the studied composition models may not coincide with the reality (i.e., effective CSs executions).

To the best of our knowledge, there are practically no approaches to web service composition's correction based on event-based logs. Prior art in this field is limited to estimating deadline expirations and exceptions prediction. References [27,28] 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. van der Aalst et al. [29] check and quantify how much the actual behaviour of a service, as recorded in message logs, conforms to the expected behaviour as specified in a process model. Our approach differs from the above: using our pattern of mining approach, we discover and prevent web service interactions anomalies. We start from CS logs and analyse them in order to re-engineer the CS model.

Obvious applications of process mining exist in model-driven business process software engineering, both for bottom-up approaches used in business process alignment [30,31], and for top-down approaches used in workflow generation [32]. A number of research efforts in the area of workflow management have been directed for mining workflows 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 [33]. This work proposes methods for automatically deriving a formal model of a process from a log of events related to its executions and is based on workflow graphs. Cook and Wolf [34] investigated similar issues in the context of software engineering processes. They extended their work initially limited to sequential processes, to concurrent processes in [35]. van der Aalst et al. [36] present an exhaustive survey of preceding process mining research works. Both areas of Process Mining and Business Process Reengineering are actively researched and considered a "hot" area in current business process management research activities. Process Mining covers different perspectives: the control flow perspective relates to the "How?" question, the organizational perspective to the "Who?" question, and the case perspective to the "What?" question. New issues in control flow perspective has been recently addressed by [37] that propose genetic algorithms to tackle log noise problem or non-trivial constructs using a global search technique. Bergenthum et al. [38] uses region based synthesis methods and compares their efficiency and usefulness. While the organizational perspective, Ref. [39] discovers information related to the social network in a process. And in the case perspective, Ref. [40] deals with the performance characteristics and business rules based on the case-related information about a Process.

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, M-out-of-N-Join, and M-out-of-N-Join patterns). We provided rules to discover the seven most used patterns. But the adopted approach allows to enrich this set of patterns by specifying new statistical dependencies and their associated properties or by using the existing properties in new combinations. Further, our approach deals better with concurrency through the introduction of the "*concurrent window*" that proceeds dynamically with concurrence. Indeed, the size of the "*concurrent window*" is not static or fixed, but variable from one service to another according to their 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, the algorithms which we described 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.

Our current work is about discovering complex patterns by using more metrics (e.g., entropy, periodicity, etc.) and by enriching the CS log. We are also interested in discovering more complex transactional characteristics of cooperative Composite service [42,43]. In [44], we proposed a set of mining techniques to discover CS transactional flow in order to improve CS recovery mechanisms. Our work in [3] uses web services logs to enable the verification of behavioural properties in web service composition. The main focus has not been on discovery, but on verification. This means that given an event log and a formal property, we check whether the observed behaviour matches the (un)expected/ (un)desirable behaviour. Recently, we have specified in [45] a combined approach that describes a formal framework, based on Event Calculus to check the transactional behaviour of CS before and after execution. Our approach provides a logical foundation to ensure transactional behaviour consistency at design time and reports recovery mechanisms deviations after runtime.

References

1. Gombotz R, Baïna K, Dustdar S (2005) Towards web services interaction mining architecture for e-commerce applications analysis. In: International conference on e-business and e-learning (EBEL'05), Amman, Jordan
2. Fauvet MC, Dumas M, Benatallah B (2002) Collecting and querying distributed traces of composite service executions. In: On the move to meaningful Internet systems, 2002—DOA/CoopIS/ODBASE 2002 Confederated international conferences DOA, CoopIS and ODBASE 2002, Springer, Heidelberg, pp 373–390
3. Rouached M, Gaaloul W, van der Aalst WMP, Bhiri S, Godart C (2006) Web service mining and verification of properties: an approach based on event calculus. In: Meersman R, Tari Z (eds) OTM conferences (1). Lecture Notes in Computer Science, vol 4275. Springer, Heidelberg, pp 408–425
4. Punin J, Krishnamoorthy M, Zaki M (2001) Web usage mining: Languages and algorithms. In: Studies in classification, data analysis, and knowledge organization. Springer, Heidelberg
5. Baglioni M, Ferrara U, Romei A, Ruggieri S, Turini F (2002) Use soap-based intermediaries to build chains of web service functionality
6. van der Aalst WMP, Weijters T, Maruster L (2004) Workflow mining: discovering process models from event logs. IEEE Trans Knowl Data Eng 16(9):1128–1142
7. Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns, elements of reusable object-oriented software. Addison-Wesley, MA
8. vander Aalst WMP, Ter Hofstede AHM, Kiepuszewski B, Barros AP (2003) Workflow patterns. Distrib Parallel Datab 14(1):5–51
9. Cook JE, Wolf AL (1998) Event-based detection of concurrency. In: sixth ACM SIGSOFT international symposium on foundations of software engineering. ACM Press, New York
10. Mannila H, Toivonen H, Verkamo AI (1997) Discovery of frequent episodes in event sequences. Data Min Knowl Discov 1(3):259–289
11. Cook JE, Wolf AL (1999) Software process validation: quantitatively measuring the correspondence of a process to a model. ACM Trans Softw Eng Methodol (TOSEM) 8(2):147–176
12. v Glabbeek RJ, Weijland WP (1996) Branching time and abstraction in bisimulation semantics. J ACM 43(3):555–600
13. Basten T, van der Aalst WMP (2001) Inheritance of behavior. J Log Algebr Program 47(2):47–145
14. van der Aalst WMP, Basten T (2001) Identifying commonalities and differences in object life cycles using behavioral inheritance. In: ICATPN '01 Proceedings of the 22nd international conference on application and theory of petri nets. Springer, London, pp 32–52
15. van der Aalst WMP (2001) Exterminating the dynamic change bug: a concrete approach to support workflow change. Info Syst Front 3(3):297–317

16. Ellis CA, Keddara K, Rozenberg G (1995) Dynamic change within workflow systems. In: COOCS, pp 10–21
17. van der Aalst WMP, van Dongen BF, Günther CW, Mans RS, de Medeiros AKA, Rozinat A, Rubin V, Song M, Verbeek HMWE, Weijters AJMM (2007) Prom 4.0: Comprehensive support for *eal* process analysis. In: Kleijn J, Yakovlev A (eds) ICATPN. Lecture Notes in Computer Science, vol 4546. Springer, Heidelberg, pp 484–494
18. Gaaloul W, Godart C (2006) A workflow mining tool based on logs statistical analysis. In: Zhang K, Spanoudakis G, Visaggio G (eds) SEKE, pp 595–600
19. Baïna K, Gaaloul W, Khattabi RE, Mouhou A (2006) Workflowminer: a new workflow patterns and performance analysis tool. In: 18th international conference on advanced information systems engineering (CAiSE'06) forum, Luxembourg, Grand-Duchy of Luxembourg
20. van der Aalst WMP, van Dongen BF (2002) Discovering workflow performance models from timed logs. In: First international conference on engineering and deployment of cooperative information systems. Springer, Heidelberg, pp 45–63
21. Weijters AJMM, van der Aalst WMP (2002) Workflow mining: discovering workflow models from event-based data. In: ECAI workshop on knowledge discovery and spatial Data, pp 78–84
22. Herbst J, Karagiannis D (2004) Workflow mining with inwolve. Comput Ind 53(3):245–264
23. Schimm G (2002) Process miner—a tool for mining process schemes from event-based Data. In: European conference on logics in AI. Springer, Heidelberg, pp 525–528
24. Gaaloul W, Baïna K, Godart C (2005) Towards mining structural workflow patterns. In: Andersen KV, Debenham JK, Wagner R (eds) DEXA. LNCS, vol 3588. Springer, Heidelberg, pp 24–33
25. Bultan T, Fu X, Hull R, Su J (2003) Conversation specification: a new approach to design and analysis of e-service composition. In: Proceedings of the twelfth international conference on World Wide Web. ACM Press, NewYork, pp 403–410
26. Hamadi R, Benatallah B (2003) A petri net-based model for web service composition. In: Proceedings of the Fourteenth Australasian database conference on database technologies 2003. Australian Computer Society, Inc., pp 191–200
27. Sayal M, Casati F, Shan M, Dayal U (2002) Business process cockpit. In: Proceedings of 28th international conference on very large data Bases (VLDB'02), pp 880–883
28. Grigori D, Casati F, Castellanos M, Dayal U, Sayal M, Shan MC (2004) Business process intelligence. Comput Ind 53(3): 321–343
29. van der Aalst W, Dumas M, Ouyang C, Rozinat A, Verbeek H (2007) Conformance checking of service behavior. ACM Trans Internet Technol (TOIT). Special issue on Middleware for Service-Oriented Computing
30. van der Aalst WMP (2004) Business alignment: Using process mining as a tool for delta analysis. In: CAiSE Workshops, vol 2, pp 138–145
31. Benatallah B, Casati F, Toumani F (2004) Analysis and management of web service protocols. In: ER, pp 524–541
32. Baïna K, Benatallah B, Casati F, Toumani F (2004) Model-driven web service development. In: CAiSE, pp 290–306
33. Agrawal R, Gunopulos D, Leymann F (1998) Mining process models from workflow logs. Lect Notes Comput Sci 1377: 469–498
34. Cook JE, Wolf AL (1998) Discovering models of software processes from event-based data. ACM Trans Softw Eng Methodol (TOSEM) 7(3):215–249
35. Cook JE, Wolf AL (1998) Event-based detection of concurrency. In: Proceedings of the 6th ACM SIGSOFT international symposium on foundations of software engineering, ACM Press, NewYork, pp 35–45
36. van der Aalst WMP, Dongen BF , van Herbst J, Maruster L, Schimm G, Weijters AJMM (2003) Workflow mining: a survey of issues and approaches. Data Knowl Eng 47(2):237–267
37. de Medeiros AKA, Weijters AJMM, van der Aalst WMP (2007) Genetic process mining: an experimental evaluation. Data Min Knowl Discov 14(2):245–304
38. Bergenthum R, Desel J, Lorenz R, Mauser S (2007) Process mining based on regions of languages. In: Alonso G, Dadam P, Rosemann M (eds) BPM. Lecture Notes in Computer Science, vol 4714. Springer, Heidelberg, pp 375–383
39. van der Aalst WMP, Reijers HA, Song M (2005) Discovering social networks from event logs. Comput Support Coop Work 14(6):549–593
40. van der Aalst WMP, de Beer HT, van Dongen BF (2005) Process mining and verification of properties: An approach based on temporal logic. In: Meersman R, Tari Z, Hacid MS, Mylopoulos J, Pernici B, Babaoglu Ö, Jacobsen HA, Loyall JP, Kifer M, Spaccapietra S (eds) OTM conferences (1). Lecture Notes in Computer Science, vol 3760. Springer, Heidelberg, pp 130–147
41. Wen L, van der Aalst WMP, Wang J, Sun J (2007) Mining process models with non-free-choice constructs. Data Min Knowl Discov 15(2):145–180
42. Gaaloul W, Bhiri S, Godart C (2004) Discovering workflow transactional behaviour event-based log. In: 12th International conference on cooperative information systems (CoopIS'04). LNCS, Larnaca, Cyprus. Springer, Heidelberg
43. Gaaloul W, Godart C (2005) Mining workflow recovery from event based logs. In: Bus process manage 3649:169–185
44. Bhiri S, Gaaloul W, Godart C (2006) Discovering and improving recovery mechanisms of compositeweb services. In: ICWS. IEEE Computer Society, NewYork, pp 99–110
45. Gaaloul W, Hauswirth M, Rouached M, Godart C (2007) Verifying composite service recovery mechanisms: a transactional approach based on event calculus. In: 15th International conference on cooperative information systems CoopIS07